Publication VI

Mikko Pohja. 2010. Server push for Web applications via instant messaging. Journal of Web Engineering, volume 9, number 3, pages 227-242.

SERVER PUSH FOR WEB APPLICATIONS
VIA INSTANT MESSAGING

MIKKO POHJA

Department of Media Technology, Aalto University
P.O. Box 15400, FI-00076 Aalto, Finland
mikko.pohja@hut.fi

Server Push is an essential part of modern web applications. The ability to send relevant information to users in reaction to new events enables highly interactive applications on the WWW. User interfaces of desktop applications have had a two-way communication with underlying software since their advent, but web applications are only reaching the same state now. In addition, currently, server push is usually emulated using pull technology, as HTTP protocol alone is not sufficient to realize a real push. This paper evaluates how an instant messaging protocol, namely XMPP, can complement HTTP-based web applications. We present a communication paradigm of a push system and an implementation of it. In addition, another communication paradigm is sketched for inter-widget messaging on the Web. Based on that paradigm a new research problem is defined and presented.

Keywords: Instant Messaging, Server Push, XMPP, Web Widget

1 Introduction

The communication of today's web applications is not just requests of documents and responses to that request. Modern web applications require instant communication in response to events that occur at both the client and the server-side. Examples of such applications include chats, stock tickers, news services, auction sites, etc. The common denominator for all these is the server's ability to push updates to the clients whenever they occur on the server. Often, a document on the client-side is modified partially, i.e., only the changed parts are updated. That has brought web applications closer to desktop applications. In addition, there are emerging scenarios, which differ even more from the first web applications. For instance, there are several novel web widget services, which give users a chance to create and customize services according to their personal needs. One way to realize a web widget service is a web widget environment like iGoogle,[a] Netvibes,[b] or Pageflakes.[c] They typically provide a container, where users can collect the widgets they need. The web widgets are usually run completely on the client-side and implemented with HTML, CSS, and ECMAScript. These widgets have even more complex communication requirements compared to the tradi-

---

[a]iGoogle, http://www.google.com/ig
[b]Netvibes, http://www.netvibes.com/
[c]Pageflakes, http://www.pageflakes.com/

tional web applications, because, in addition to the communication between the client and the server, an inter-widget communication is needed. An example of this kind of communication is present in an upcoming Google Wave.[d]

The study of push technology for more than a decade has led to several highly interactive web applications. However, the technology is not really sophisticated. An observation of Franklin and Zdonik [1] is still valid: most push systems are actually implemented using a periodic pull or its derivative. That is due to the fact that IP and HTTP protocols do not properly support push communication. Ajax [2] and Comet [3] frameworks are a step further, if compared to a simple polling, especially performance-wise [4, 5]. Nevertheless, they also rely on client initiated updates and HTTP. In addition, an implementation effort is remarkable with them.

In literature, it is widely suggested that web applications be implemented using both pull and push protocols [6, 7], due to their dynamic nature. The pull protocol is naturally HTTP, but the push protocol has not been settled yet even though there have been some proposals, e.g., [8, 9]. Push updates are reactions on events and they are sent to receivers as messages. Hence, a web application or a widget running on a client compares with a subscriber in a centralized Instant Messaging (IM) application. This paper aims to introduce models which show how instant messaging can complement HTTP-based web applications and web widgets. Extensible Messaging and Presence Protocol (XMPP) was used as the push protocol. XMPP was selected, because it has been successively used in many IM applications and its numerous extensions enables diverse usage scenarios.

The main aims of the paper are:

- A derivation of a set of requirements for an IM-based push system based on literature and use cases.

- A presentation of communication paradigms for the push system and web widgets.

- A demonstration of a prototype implementation of the web application model as a proof of concept.

- An evaluation of the web application model based on the derived requirements and a use case implementation.

- The introduction of a new research problem on web widget communication.

The rest of the paper is organized as follows. The next Section reviews a background of the topic. Section 3 defines the scope of the paper, use cases, and requirements. The proposed communication and a proof of concept implementation are discussed in Sections 4 and 5, respectively. A use case implementation is described in Section 6 and results are evaluated in Section 7 and discussed in Section 8. Finally, Section 9 concludes the paper.

2   Background

The objective of this paper is to find a way to complete HTTP to enable server push for web applications. The limitations and server push of HTTP are discussed in the following Subsections.

---

[d]Google Wave, http://wave.google.com/

2.1   HTTP

HTTP is an application level request/response protocol. It is a stateless protocol, so it is effective in its original use for delivering static documents. That is, a client sends a request to the server and the server sends a corresponding response back to the client. The problems arise when HTTP is used for an application, which is distributed between the client and the server. For example, a user operation on the client-side may require an action on the server-side. To initiate the action, the client must send another HTTP request to the server and wait for the response before the operation can be completed. That also means that the whole document has been downloaded and rendered again, which are both time consuming events and reduce the responsiveness of the user interface. The performance can be enhanced by updating only a part of the user interface using Ajax framework [2].

Since HTTP always requires a request from the client-side, it is not possible to react immediately to events, which happens on the server-side of a distributed application. The server-side application must wait that the client contacts the server and then include the change into the response. That operation, which is missing from HTTP, is usually called server push in contrast to client pull. The server push and how to emulate it with HTTP are discussed in the next Subsection.

2.2   Server Push

At its simplest, server push on the Web can be emulated on HTTP by polling the server at a certain time interval. That can be done for instance by reloading the document periodically or with Ajax [2] by asking incremental updates. That causes a lot of additional network traffic, especially reloading the whole document, and there is always trade-off between the latency and the polling frequency.

Russell coined the term Comet for low-latency data transfer to the browser in 2006 [3]. Comet is not an explicit technology set but a term to describe server-push data streaming functionality. It can be implemented in many ways. For instance, keeping connection to a server open, e.g., via *iframe* element, or keeping a connection for XMLHttpRequest [10] object open. The downside of Comet is that the server must keep connections open to all clients it has to update. The central server must also be able to distribute the communication properly [11].

HTML5 specification [12] defines an approach called Server Sent Events (SSE). With SSE, an author can declaratively define a source from which the browser is listening for the incoming connections. The communication itself can be realized with Comet. The HTML 5 specification is however still work-in-progress at the World Wide Web Consortium (W3C) at the moment.

Deolasee et al. [6] and Hauswirth and Jazayeri [7] discuss components and theory of the push systems on the Web. Both have examined the combination of pull and push and find it suitable for the Web. Deolasee et al. argue that adjustment between push and pull depends on use case. Sometimes only one of them is needed. In Hauswirth and Jazayeri's system, the push is used to notify user agent to pull fresh information. Other push and pull systems are the mWeb presentation framework [8], which uses SRRTP and SRFDP protocols for push, and HTTP+RTP push/pull system by Trecordi and Verticale [9].

3   Research Aim and Scope

The main research aim of the paper was to design a system which can submit dynamic information for web applications. The goal was not to replace HTTP protocol, but define an additional protocol to support server push. The main focus is on a one-to-many delivery scheme, even though broadcasting and one-to-one communication can be realized with the same technologies to some degree. However, delivering user input is out of scope of this paper.

The other research aim, was to find out whether the selected protocol could be utilized in inter-widget communication on the Web. The web widgets are small programs embedded to a web page via *iframe* or *object* elements. Typically, they are implemented with HTML, CSS, and ECMAScript. There are also specific web widget environments, which contain a number of ready-made widgets from which users can choose. In addition, they provide APIs for developers, so that they can implement their own widgets. Examples of such web widget environments are iGoogle, Netvibes, and Pageflakes.

W3C is working on widgets specifications, which define APIs, events, packaging etc. for mobile widgets. The web widgets are however out of scope of their specification. Nevertheless, web widgets and mobile widgets are actually very similar. Naturally, the main difference is the running environment. For web widgets, it is a generic browser and, for mobile widgets, it is specific widget engine running on a mobile phone. Thus, it is possible to create a web widget engine, which conforms to W3C specifications. Actually, there are two research-oriented web widget engines, which conform to the W3C specifications, namely, Wookie [13] and myWiWall [14]. Wookie can provide widgets to any web site via a server-side plug-in, whereas myWiWall is a complete widget environment including a widget engine. All the existing web widget engines provide some kind of inter-widget communication.

In Wookie, the W3C widget API is extended to support inter-widget communication. The widgets can store shared data in the widget engine like how they store private data, which is via key-value pairs. The sharing is possible among different instances of a single widget, including different users, but not possible among different widget context. Google Wave protocol also has means to provide wide scale collaboration within users. That includes widgets, which can share data in different user contexts. The Google Wave protocol [15] is an extension to the XMPP protocol, while MyWiWall's approach is the other way round. Their widgets can share data within different widget contexts, but only within a single user. The communication is based on events, which widgets can fire and listen. Google's Gadget-to-Gadget Communication API [16] has also a method to share data within a same user context. The communication is realized with XMPP and is based on fixed channels which Gadgets can subscribe and publish data. Table 1 summaries the different data sharing paradigms of the widget engines. As can be seen, there is no widget engine, which supports different user and application contexts concurrently. That kind of communication is in the scope of this paper. In this scenario, web widgets reside in so-called web spaces. The spaces are social web applications, whose functionality consists mainly of the widgets. The difference between iGoogle or Netvibes and this scenario is that users share spaces in contrast to everyone possessing their own spaces. Each space has a group of users and they and their widgets can communicate through the system.

Table 1. Data sharing contexts of the widget engines.

	Same User	Different User
Same Application	N/A	Google Wave, Wookie
Different Application	Google Gadgets, myWiWall	

3.1 Use Cases

This subsection introduces three possible use cases of the communication protocol and the model.

3.1.1 On-line Auction Tool

There are several goods on sale on an auction site. Users can set bids for the goods they like to purchase and a user with a winning bid win an item. To follow the bids from other users, the application must be able update the bidding history on users' screen in real time.

3.1.2 Score Service

A soccer league offers a live score service on their web site. The service provides a live score and played minutes from each game on a match day. Each game has also its own page, which has additional information of the game in question. That includes scorers, bookings, substitutions etc. The main page provides access to these pages.

3.1.3 Location and Status Data

A person is using a widget, with which she can publish short microblog-like status messages and her current location. The location can be fetched from a device, if applicable, or she can set it by hand. This information can be utilized both by her own widgets and her friends' widgets. For example, her weather widget can always show the forecast according to her current location. In addition, her friend's map widget is able to show her on the map along with a status message.

3.2 Requirements

The requirements of the system are derived from literature and use cases. The requirements are divided to the general system requirements and to the technology requirements.

3.2.1 System Requirements

R1: Protocol. System must support both push and pull protocols. Based on literature, HTTP-based web applications benefit from an integration of a push protocol.

R2: Delivery Scheme. System must support a one-to-many delivery scheme. That is, more precise than broadcasting, but not a one-to-one dialog.

R3: Coherence. The data updates must be delivered in near real time.

R4: Flexibility. The system must support asynchronous communication between loosely coupled components.

R5: Performance. The system must outperform HTTP polling in terms of latency and amount of transferred data. Both of them are considered problems with polling.

R6: Late Binding. The connections between components should be created as late as possible, preferably on run-time.

3.2.2   Technology Requirements

R7: Ease of Authoring. The application development should not require any new programming language to ease the adoption of the system. Declarative languages are commonly considered easier to author than procedural languages and can be used even by non-programmers. An application should consist of re-usable software components.

R8: Web integration. Using existing technologies at client-side eases the adoption of a technology or a framework among the users.

4   Proposed Communication

Based on the above, a new communication model is proposed. This Section introduces the publish/subscribe (pub/sub) messaging model and the proposed communication paradigms, which utilize the pub/sub model.

4.1   Publish/Subscribe

Publish/Subscribe is an asynchronous messaging model. Publishers do not send their messages directly to subscribers but to channels. Subscribers receive messages from channels they have subscribed. That makes the paradigm loosely coupled. Actually, pub/sub can be decoupled on three dimensions: space, time, and synchronization [17]. That is to say, senders and receivers do not have to know each other, the messages can wait on a channel for subscription, and the production and consumption of events do not block other activity on either end of the system. Removing dependencies between parties makes the paradigm well adopted in distributed environments, which are asynchronous by nature [18]. Pub/sub paradigm is an example of the push system as the publishers push data to the subscribers through the channels.

Subscription of the pub/sub system can be channel-, content- or event-based [17]. In the channel-based systems, a receiver subscribes to a certain channel, from which it receives all the published messages. In the content-based pub/sub model, a subscriber defines filters for the subscription. It will receive messages, which pass the filters. The event-based systems are founded on event types, to which receivers subscribe. In addition, pub/sub system can be a combination of the above-mentioned types.

4.2   Communication Paradigms

The communication is based on a *Push-and-Pull (PaP)* [6] algorithm. A user agent fetches a document from the web server using HTTP protocol. When the web application is running on the user agent, dynamic data is pushed to the client whenever needed. In addition, the model allows for a graceful degradation if the push component fails. The system can rely on a pull based polling as a backup mechanism.

The idea is to use the pub/sub-messaging paradigm with a combined channel and content-based model. The user agents subscribe to the channels according to their current state. That is, in the case of web applications, the user agents subscribe to the channels according to the location of their current resource. In other words, there is a one-to-one match between URLs on the web server and the channels on the push server. On the other hand, the widgets utilize

the data of other widgets. Thus, each instance of a widget can have its own channel to which others can subscribe. The fundamental idea behind these two cases is the same, even though the models are slightly different. Both paradigms will be discussed in detail in the following Subsections.

4.2.1   Web Applications

There is a one-to-one match between URLs on the Web Server and the channels on the Push Server. The Web Server communicates the correct channel to the user agent within the data sent over the HTTP. Thus, it can be said that the model supports the REST paradigm on which the Web relies. There is an Event Source for every channel on the Push Server. The Event Sources track the changes on server-side, create update events according to the changes, and publish them on their channels. The Push Server distributes the updates to every user agent, which has subscribed to the corresponding channel. This is known as a multicast delivery scheme. The distributions are fine-grained adding content-based subscriptions. This way only relevant updates are filtered to each subscriber.

As a summary, the communication between components of the system is shown in Figure 1. The user agent fetches a document from the Web Server via HTTP as usual (1). In addition to the document, it receives address and subscription details of the Push Server. With that data, the user agent connects to the Push Server and subscribes to the relevant channel (2). As a respond, Push Server submits the latest message of the channel to keep the user agent up to date (3). That ensures that a possible change, which occurs between the Web Server's response and the subscription, is delivered to the client.
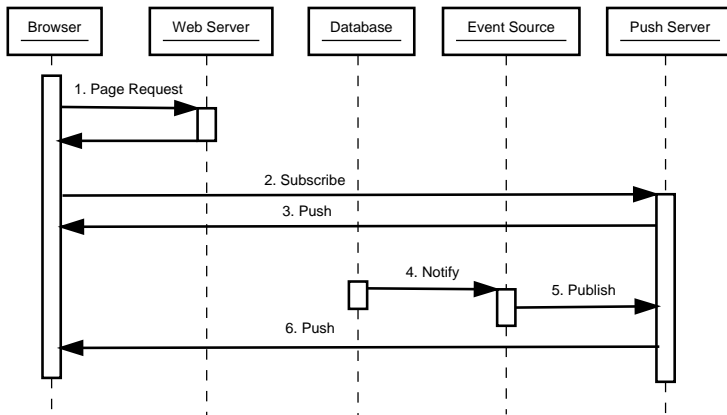

Fig. 1. The communication between components of the system.

When a data is modified for instance in the database, a trigger notifies the Event Source (4). The Event Source creates an update event and publishes it on the Push Server in the pertinent channel (5). The Push Server pushes the event to all subscribers of the channel (6).

4.2.2 Web Widgets

In the web widget's communication paradigm, the widgets themselves publish the updates on the channels. Each widget has a channel, where it can publish the updates, and interested widgets can subscribe to the channel. The paradigm does not restrict the subscribers. That is, the subscriptions can be done across any application and user contexts, even simultaneously. The paradigm is depicted in the Figure 2. When a Publisher Widget is initiated, it creates a channel for itself on the Push Server (1). The channel remains on the server even though the widget goes offline, so next time it only has to subscribe to the channel. When a widget wants to listen to another widget, it must be informed of the correct channel on the Push Server (2-3). The Publisher widget can communicate it directly, but the channel can also be communicated via other means. That is, the communication on steps 2 and 3 could also happen between a server and the Subscriber. Thus, the Publisher and the Subscriber do not necessarily have to know anything about each other, but just the channel on the server. With the channel information, the Subscriber Widget can subscribe to the Push Server (4). That can happen during initialization or later in the run-time. The Push Server delivers the latest message to the subscriber as a response. When something has changed in the Publisher Widget, it publishes that on the Push Server (6) and the update is pushed to the all subscribers (7). Any widget can be a publisher or a subscriber or both.
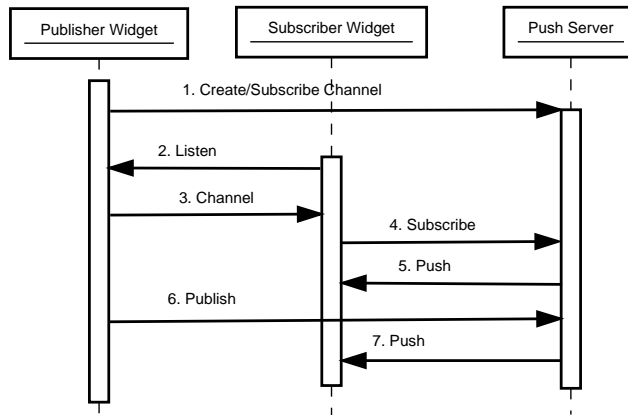

Fig. 2. The widget communication model.

Open Issues   The web widget communication paradigm represented above is not complete. It actually reveals details, which form a solid research work item. The aim of this Subsection is to formulate a research problem for future work. The following questions should still be answered regarding the widget communication model:

Q1. What is the topic of the channels?

Q2. How does a subscriber widget find a correct channel to subscribe?

Q3. What is the format of the update messages?

The topic could be tied to a single widget or to a data type. In the latter several widgets could publish on the same channel presuming they publish similar data (e.g., locations). Each user could have one or more channels for her widgets and subscribers would know them automatically, if the channels had standard names. The channel per widget option would give more freedom on the published data. The third option is to use aforementioned alternatives simultaneously.

In theory, it is possible that any widget can subscribe to listen to any other widget. It has to only know which channel to subscribe. If the same user is using both widgets, the channel can be communicated on client-side by connecting the two widgets. Otherwise, there must be a server-side system providing the information.

The requirement on loose coupling expects that two components can be implemented without knowledge of each other. Nevertheless, they must understand each other's messages. There are at least three options to solve that. The format of messages could be predefined, the messages could use data types to describe the content, or they could be tied to some ontology. It is also problematic that any of these might be updated later, which may require modifications to existing widgets.

5   Implementation

An implementation of the Push System conforming to the web application communication paradigm is discussed in this Section. The system is based on the XMPP protocol. Main reasons for selecting XMPP were its widely usage in other XML streaming applications, its pure push nature, existing implementations, and possibility to use with legacy browsers and firewalls.

5.1   XMPP

Extensible Messaging and Presence Protocol (XMPP) is a protocol for streaming Extensible Markup Language (XML) elements in near real time between any network endpoints [19]. It is initially developed as a protocol for IM applications [20]. The XMPP technology set comprises of core functionality and numerous extensions. The core defines how XML snippets are handled for instance in IM applications and the extensions utilize that base. Among the others, there is an extension for XMPP publish/subscribe [21] and HTTP binding for XMPP communications (BOSH) [22] that can be used if regular XMPP communication is blocked (e.g., by firewall).

BOSH uses Comet to mimic two-way communication between the Server and the browser. However, even if it is an emulation of push and not pure XMPP anymore, it has many benefits over plain Comet since it holds other characteristics of XMPP. These include persistent connections even if the underlying network connection is unreliable, a prescribed messaging model and format, and authentication. It can be said that XMPP BOSH provides a standardized communication model for Comet.

5.2   Components

The components of the system are depicted in Figure 3. All the software is open source software and extended as needed. The Web Server is an Apache Tomcat Servlet container[e] and the database is an eXist-db XML database.[f] In addition, there are an XMPP Server and Event Sources in the server-side. The XMPP Server is an Openfire XMPP Server.[g] It has native support for XMPP pub/sub. Event Sources are system specific components which use Smack XMPP client library[h] with su-smack[i] pub/sub extension for XMPP communication. User Agents of the system include a browser, the XMPP client, and an update handler. The XMPP Client can have native integration into the browser or it can be an ECMAScript library, which comes along with a web application. Thus, the system can be used with legacy browsers. As a proof of concept, both native and ECMAScript implementations were done. In the native implementation, the XMPP components were integrated into X-Smiles [23], which is a Java based open source browser. It uses the same libraries as the Event Sources for XMPP communication. The update handler depends on the format of the updates which will be discussed below.
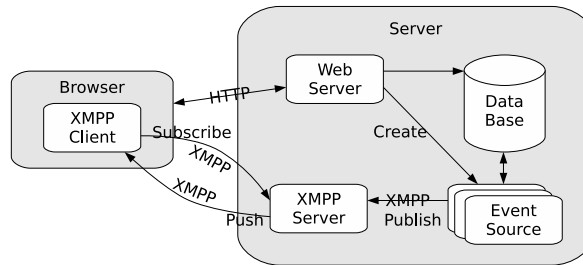

Fig. 3.   The components of the Push System.

The ECMAScript implementation uses Strophe XMPP library,[j] which was extended to support XMPP pub/sub. In addition, there is also an ECMAScript implementation of the update handler. The Strophe pub/sub implementation provides a generic interface, which creates and parses the XML stanzas used for XMPP pub/sub communication. The client must still handle the connection, but the library creates the content of the messages.

5.3   Initialization

In addition to handling the Servlets, the Web Server initializes the Event Sources automatically. Each Servlet defines which data on the database its Event Source should monitor and to which element it corresponds to the document it provides. The Event Sources are parameterized via the Servlets. An author of an application includes corresponding database reference and element pointer pairs in a Servlet. The generic Event Source can create update messages

---

[e] Apache Tomcat, http://tomcat.apache.org/
[f] eXist-db, http://exist.sourceforge.net/
[g] Openfire XMPP Server, http://www.igniterealtime.org/ projects/openfire/index.jsp
[h] Smack API, http://www.igniterealtime.org/projects/ smack/index.jsp
[i] su-smack, http://static.devel.it.su.se/su-smack/
[j] Strophe, http://code.stanziq.com/strophe/

according to the pairs. However, the author can also extend the generic Event Source to create custom messages. That might be necessary if, for instance, the author wants to modify or analyze content before submitting it to the clients through the Push Server. In addition, the author can at this point add keywords to the message to enable content-based filtering on the Push Server.

The Event Sources subscribe to the XMPP Server just like the User Agents (UAs). As pointed out in the previous Section, there is a one-to-one mapping between a web document, an Event Source, and a channel on the XMPP Server. The Event sources create their channels on the Server and publish update events on them.

UA receives the subscription details for the XMPP Server within the HTTP response of the Web Server. The subscription data is embedded in the header section of a document and is recognized by a namespace. The concept is equivalent to the aforementioned HTML 5's SSE. SSE also allows definition of the event source, but not channels and content filters, which are required for XMPP. Since XMPP messages are in XML format, the subscription data can be embedded as such in a document. The UA must be extended to handle the XMPP snippets in the documents. An alternative to the declarative subscription descriptions is to add an ECMAScript function, which subscribes to the XMPP Server using the dedicated ECMAScript library.

With XMPP, a client must first connect to an XMPP Server before it can subscribe to channels and receive messages. The system opens a new connection for every document it downloads and closes the connection whenever the user moves to a next document to avoid unused open connections.

5.4   Operation

The operation of the system is described in Figure 4. The database has a mechanism to trigger an event when a collection in a database is modified. The Event Sources have registered themselves to the relevant triggers. The triggers notify registered Event Sources when the database has been modified (1). The Event Sources publish the modified information on their channels in the XMPP Server (2), which further pushes the event to the subscribers (3). The browser handles the event and updates the relevant node in a document (4).
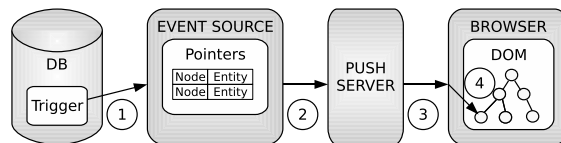

Fig. 4. Conceptual model of the operation.

5.4.1   Database Triggers

The database triggers are built on the triggers provided by the eXist-db. The eXist-db triggers do not specify exactly which data has changed on the database, but only the modified collection. To find out the exact change, we use *xmldiff*[k] to compare the collection before

---

[k]XML differencing tool, http://www.logilab.org/859

and after the modification. The *xmldiff* returns the modification in XUpdate language [24]. Effectively, the XUpdate result consists of an XPath expression, which refers to the correct element in a database and the new content. To see if the expression relates to a certain Event Source, the Event Source's XPath references are compared to the expression. The comparison is done taking the intersection of the two expressions. If the intersection is not null, the modification relates to the Event Source and it has to publish an update event on the pertinent channel in the XMPP Server. The communication between the database triggers and the Event sources is realized with Java Messaging Service (JMS) technology.

Computing the intersection against every XPath expression in every Event Source is not an optimal solution. Designing the structure of the database carefully can enhance the performance. The smaller the collections, the less frequent Event Sources will need to track changes on each of the collections. However, since XML processing methods are out of scope of this paper, it is left as future work to find out the most scalable solution.

5.4.2   Update Events

The purpose of push updates is to modify the DOM tree on the client-side. The modification methods are discussed in depth in a related research [25]. Since XMPP messages are in XML format, it is natural to use XML to describe the updates. W3C has produced a working draft of Remote Events for XML (REX), but at the present moment has finished the specification work because of patent issues.[l] In spite of this, REX represents the declarative concept to describe the update events. In this paper, that concept is called Remote DOM Events (RDE).

The Event Sources create the RDE events. An event defines a target element in the remote DOM tree and a modification on it. The target element is specified by an XPath expression and modification is obtained from a database trigger. The author of the application defines the XPath expression.

6   Use Case

The Score Service use case discussed in Section 3.1 was implemented as a proof of concept. The service provides a main page, from which a user can select which group's games he or she wants to follow. A screen capture of the main page is shown in Figure 5. It shows played minutes and scores in real time. Clicking a score on the page opens a new page, which has detailed information on the game in question.

The score service consists of pages for each group and a general page to follow a single game. If a user follows a whole group, the user agent subscribes to a pertinent channel on the XMPP Server and receives all the updates for the channel. In the case of the single game, the user agent subscribes to a general channel, which receives all the updates from all the games. The subscription also includes content-based subscription, which in this case is a game id. Hence, the user receives updates only for the game he or she has selected.

The implementation uses the generic Event Sources for the group pages, because all the updated content appears as such on the page. For the single game page, we implemented a custom Event Source, which adds the game id for each message to enable content-based filtering.

---

[l] Report of the REX PAG, http://www.w3.org/2006/rex-pag/rex-pag-report.html

Fig. 5. Screen capture of the Score Service use case.

7 Evaluation

The implementation shows that it is possible to implement a system, which pushes database modifications all the way to the web browser in near real time. Overall, the system fulfills the requirements well. XMPP with the pub/sub extension suits well in this kind of communication. Using XMPP pub/sub covers the requirements R1, R2, R4 and R6. Decoupling between the core communicators, i.e., the user agents and Event Sources, has succeeded well. They both communicate through a standard XMPP Server and do not have any dependencies between each other. In addition, the bindings between the components are formed in the run-time.

The model provides means to fulfill the coherence requirement (R3), but the current experimental implementation needs an optimization on database triggering to conform to the requirement well. The performance of push communication is similar to Comet, which has been shown to outperform legacy HTTP polling applications clearly [4, 5]. In addition, the XMPP-based system has some benefits over plain Comet as discussed above.

The push applications are easy to author with the system. An author just has to provide data reference pairs on the server-side for the Event Sources. The pairs include a pointer to the database and a corresponding element in a client-side document. In addition, if content-based filtering is required on the Push Server, an author must customize the Event Sources to provide the desired content. That requires knowledge of programming. The system specific components take care of the rest of the functionality. For client-side functionality, the author just needs to add relevant ECMAScript libraries unless the client supports the technologies natively.

The requirement R8 presumes that the legacy clients are able to use the system without additional components. That is fulfilled well in the system. As mentioned earlier, the ECMAScript libraries can handle all required client-side functionality and the XMPP BOSH

extension enables the communication even through firewalls.

The use case implementation shows that creating channels based on URL on the Push Server is a workable method. If the varying content on the page is the same for all, the channel-based subscriptions are sufficient. If the content depends on a user, the subscriptions can be optimised with content-based subscription. Nevertheless, there is only one channel per URL.

8   Discussion

Introducing a new protocol for web applications is not trouble-free. In addition to the technical issues, it has to be integrated with the existing ecosystem. Hauswirth and Jazayeri present four requirements for the widespread use of a push system [7]. The requirements relate to:

- Scalability

- Payment methods and business models

- Security and authentication

- Standardization

Hauswirth and Jazayeri observe that analyzing the scalability of a push system is not easy, because there are so many factors: number of broadcasters, receivers, and channels; amount of data on channels; frequency of updates; network latency and bandwidth; and the amount of common subscriptions to certain channels. The scalability of the proposed paradigm is not analyzed deeply because of the reason above and because the implementation is an early prototype. Nevertheless, the wide use of XMPP in instant messaging systems implies that it may scale well in large push systems, too.

Payment methods and business models as well as security issues are not in the scope of this paper, but have to be managed before large-scale deployment. The same security requirements apply to XMPP communication as well as HTTP communication. The inter-widget communication model may create a need to manage access for cross-site resources. Currently, browsers do not allow that, but W3C is working on Cross-Origin Resource Sharing (CORS) [26] specification, which would be usable in the inter-widget communication scenario. The existing XMPP servers provide authentication and authorization by default, so that part has already been resolved. Finally, XMPP and its many extensions are well standardized by the XMPP Standards Foundation.[m]

A new protocol typically requires new software both on the client and server-side. Furthermore, developers have to learn the new protocol. The requirement on new client-side software is not necessarily hard to implement for developers, but it has to be installed by most of the end-users before application developers will actually start to use it. Users' update cycles on their browsers is remarkably slow. If the same functionality can be offered via ECMAScript, it can be adopted immediately, of course, assuming that the ECMAScript support is enabled on a browser. This paper shows that the XMPP communication model can be applied right away on the Web, even though it has to be bound to HTTP. XMPP with HTTP binding could be used as a kind of standardized syntax for Comet communication. That

---

[m]XMPP Standards Foundation, http://xmpp.org/

might even help developers to create Comet-based applications. Furthermore, the application would automatically work with user agent with XMPP support.

The system requires more optimization before it could be widely deployed. The enhancement of the data base triggers was left as future research work. Furthermore, an efficient method to find the correct Event Sources must also be found. Another possible point of optimization relates to XMPP communication. Presently, the system requires that the user agent creates a new connection to the XMPP server every time it fetches a new page, and closes the connection when leaving the page. That produces communication overheads, because web applications naturally consists of several pages. The connection to the Push server could be kept open if it remains the same between the previous and new page. The logic should be implemented on the user agent or provided with ECMAScript.

9   Conclusions

Server push is an integral part of present-day web applications. It is usually emulated using HTTP-based technologies. That causes additional network load and increases implementation effort. This paper examined how to replace the current HTTP-based push technologies with instant messaging. The proposal is that HTTP communication should be complemented with a push protocol, XMPP.

Two communication paradigms of instant messaging-based push systems are presented in the paper. One is for traditional web applications and the other is for the web widgets. The paradigms are based on literature and the requirements, which were defined for the paper. In addition, a push system conforming to the communication paradigm was implemented and a use case running on the system demonstrated. The implementation shows that the paradigm is feasible and earlier research confirms that XMPP is scalable in large systems. However, the implementation requires some optimization with regard to database triggering. Otherwise, the implementation fulfills the pre-defined requirements well.

The paper also introduces a new research problem, which relates to web widgets and inter-widget communication. There are several web widgets environments, which all lack the support for data sharing within different user and application contexts. In light of the results, it can be expected that the gap be filled with the proposed model, but there are outstanding issues which have to be resolved first.

Acknowledgements

References

1. Michael Franklin and Stan Zdonik. "Data in Your Face": Push Technology in Perspective. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 516–519, New York, NY, USA, 1998. ACM.
2. Jesse James Garrett. Ajax: A New Approach to Web Applications. Technical report, Adaptive Path, 2005.
3. Alex Russell. Comet: Low Latency Data for the Browser. Weblog, March 2006. Available online: http://alex.dojotoolkit.org/?p=545.

4. Michele Angelaccio and Berta Buttarazzi. A Performance Evaluation of Asynchronous Web Interfaces for Collaborative Web Services. In *Frontiers of High Performance Computing and Networking – ISPA 2006 Workshops*, volume 4331/2006 of *Lecture Notes in Computer Science*, pages 864–872. Springer, November 2006.

5. Engin Bozdag, Ali Mesbah, and Arie van Deursen. Performance Testing of Data Delivery Techniques for AJAX Applications. Technical report, Delft University of Technology, Delft, Netherlands, 2008.

6. Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant Shenoy. Adaptive push-pull: disseminating dynamic web data. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 265–274, New York, NY, USA, 2001. ACM.

7. Manfred Hauswirth and Mehdi Jazayeri. A Component and Communication Model for Push Systems. *SIGSOFT Softw. Eng. Notes*, 24(6):20–38, 1999.

8. Peter Parnes, Mattias Mattsson, Kåre Synnes, and Dick Schefström. The mWeb presentation framework. *Comput. Netw. ISDN Syst.*, 29(8-13):1083–1090, 1997.

9. V. Trecordi and G. Verticale. An architecture for effective push/pull Web surfing. *Communications, 2000. ICC 2000. 2000 IEEE International Conference on*, 2:1159–1163 vol.2, 2000.

10. Anne van Kesteren, editor. *The XMLHttpRequest Object*. W3C Working Draft, April 2008.

11. John Resig. *Pro JavaScript^{TM} Techniques*, chapter 14, pages 287–304. Springer-Verlag, New York, NY, USA, 2006.

12. Ian Hickson and David Hyatt, editors. *HTML 5*. W3C Working Draft, January 2008.

13. Scott Wilson, Paul Sharples, and Dai Griffiths. Distributing education services to personal and institutional systems using Widgets . In *First International Workshop on Mashup Personal Learning Environments (MUPPLE08)*, pages 25–32. CEUR-WS, September 2008.

14. Stéphane Sire, Micaël Paquier, Alain Vagner, and Jérôme Bogaerts. A Messaging API for Inter-Widgets Communication. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 1115–1116, New York, NY, USA, 2009. ACM.

15. Jochen Bekmann, Daniel Berlin, Soren Lassen, and Sam Thorogood. Google Wave Federation Protocol Over XMPP. Draft protocol spec, Google Inc., June 2009. http://www.waveprotocol.org/draft-protocol-spec.

16. Gadget-to-gadget communication. Technical report, Google Inc., 2009. http://code.google.com/apis/gadgets/docs/pubsub.html.

17. Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

18. Yongqiang Huang and Hector Garcia-Molina. Publish/Subscribe in a Mobile Environment. *Wirel. Netw.*, 10(6):643–652, 2004.

19. Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. Proposed standard, IETF, October 2004. http://www.ietf.org/rfc/rfc3920.txt.

20. Peter Saint-Andre. Streaming XML with Jabber/XMPP. *IEEE Internet Computing*, 9(5):82–89, 2005.

21. Peter Millard, Peter Saint-Andre, and Ralph Meijer. Publish-Subscribe. Standards Track 0060, XMPP Standards Foundation, March 2008.

22. Ian Paterson, Dave Smith, and Peter Saint-Andre. Bidirectional-streams Over Synchronous HTTP (BOSH). Standards Track 0124, XMPP Standards Foundation, February 2007.

23. Petri Vuorimaa, Teemu Ropponen, Niklas von Knorring, and Mikko Honkala. A Java based XML browser for consumer devices. In *17th ACM Symposium on Applied Computing*, pages 1094–1099, Madrid, Spain, March 2002.

24. Andreas Laux and Lars Martin. XUpdate XML Update Language. Working draft, XML:DB Initiative, September 2000.

25. Mikko Pohja. Declarative Push on Web. In *Proceedings of the 4th International Conference on Web Information Systems and Technologies (WEBIST)*, pages 201–207. INSTICC, May 2008.

26. Anne van Kesteren, editor. *Access Control for Cross-site Requests*. W3C, February 2008. http://www.w3.org/TR/2008/WD-access-control-20080214/.