Unified Push Messaging for Web Applications

Paul Emathinger

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2015

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, September 28, 2015

Paul Emathinger

Contents

Abstract

In contrast to native applications the web is decentralized and runs on multiple platforms. Bringing programming interfaces from native to the web is therefore not always simple. For the new Push API, which aims to deliver push messages for websites, the problem of how to handle different push services arises. In an ideal world all push services would run the same protocol, but unfortunately this is not the case. To communicate with multiple services the code has to be adjusted for each of them. This thesis takes a look at a way to tackle this problem by adding a new server that exposes a unified protocol.

Kurzfassung

Im Gegensatz zu nativen Anwendungen organisiert sich das Web dezentralisiert und läuft auf unterschiedlichen Plattformen. Neue Programmierschnittstellen Webanwendungen zur Verfügung zu stellen erweist sich daher nicht immer als einfach. Dies gilt auch für die “Push API”, welche darauf abzielt push Nachrichten für Webseiten zu ermöglichen. In einem perfekten Umfeld würden alle Push Dienstleister das gleiche Protokoll zum Versenden von Nachrichten verwenden, aber dies ist leider nicht der Fall. Um die unterschiedlichen Services verwenden zu können bedarf es einer Anpassung des Codes. In dieser Arbeit wird ein anderer Versuch gestartet dieses Problem zu lösen indem ein neuer Server zum System hinzugefügt wird, der ein neues Standard Protokoll unterstützt.

Chapter 1

Introduction

With the invention of the World Wide Web people got a great tool to communicate with other people around the world without big effort. This helped spread ideas within minutes and build friendships that last a lifetime. But as the web evolved from just displaying text to offer interactive applications the technology had to adopt as well. With the appearance of the first smart phones the demand for new capability was higher than ever but the web responded too slowly and native applications took over the mobile world. Ever since there were competition between web and native applications [5].

Browser manufacturers did not find the right way to utilize bookmarks on mobile [16] and operating systems just recently started treating individual websites more like native apps[1]. In recent years the web has evolved into a feature rich application platform by offering better performing rendering engines and a bigger set of programming interfaces to developers [8].

Most people would argue native applications load faster than web applications, but this is not true for first time visits. To open a native app for the first time the app has to be downloaded and installed. This can take several minutes, while web applications usually load within seconds. While some installed apps are used on a daily basis, others are only used occasionally. For booking a flight on an airplane website for example it makes a lot of sense to ask the user for permissions to show a notification if there are any changes for the departure time or gate rather than to install an app for this specific task. To show this notification a new set of browser APIs is currently in development.

1.1 Problem Statement

Compared to native applications web applications run on multiple platforms and do not target one specific device group. This has always been a difficult

[1]GCM introduced Chrome browsertabs in the overview screen with Android 4.0 Lollipop

problem for web and browser developers and this new set of APIs makes no exception. Sending push messages requires good knowledge of the used system and unfortunately these systems differ a lot. While for native applications the implementation of one proprietary protocol is still endurable, web applications have no theoretical limit of protocols to implement. Every browser or operating system can use a different solution. Luckily a new standardized protocol is being developed so one common way can be used to send push messages. The only problem with it is that already existing push services will not adopt this protocol on day one and web developers are therefore either forced to wait to use this feature or still implement the proprietary solution if the browser allows it.

1.2 Goals

To tackle the problems of multiple needed implementations of proprietary protocols the new standardized protocol is the only solution. The scientific hypothesis of this paper is therefore:

> How can decentralized push messaging for web applications be achieved in a standardized way without changing the currently existing push services?

The proposed solution adds a new server called *relay server* that exposes the new protocol and forwards the messages to the proprietary push services. The application server should notice no big difference sending the message to the relay server or sending it to a modern standardized push service. Furthermore, the goal is to do this with a minimum of added latency and complexity for good scalability.

1.3 Structure

For a better overview this document is structured into several chapters. Although reading the thesis from beginning to end is recommended it is still possible to read certain chapters only as well if knowledge about the topic is present. *Chapter 2* provides an overview over the technical background needed to read the rest of the thesis. This chapter is highly recommended to read for everyone who is new to this topic but it can be also important for more experienced people because some terms are used in a certain context. *Chapter 3* lists existing push services available and provides detailed information about the new proposed standards and programming interfaces. *Chapter 4* introduces the concept of the new approach and analyzes how certain problems can be solved. Also examples of how to map some of the most common push services to the new relay server are stated. *Chapter 5* presents the prototype of the push relay server as well as a browser mockup implementation to show the usage of the new approach. *Chapter 6* evaluates the developed prototype by testing the coverage of the new protocol, ease of use for developers and performance in terms of latency time, CPU load, memory consumption, and bandwidth usage. Also unfinished parts are discussed and how the prototype could be extended. *Chapter 7* sums up discussed problems and given solution and tries to give a conclusion about the outcome of this project.

Chapter 2

Technical Background and Disambiguation

The web is a rapid changing environment with new frameworks, chart libraries, and other tools developed every day. Also JavaScript as a language is changing with new ECMAScript versions coming out each year [17]. Bringing clarity to this massive amount of new terms and features is not always an easy task, but detailed descriptions can help understand the fundamentals. However, a certain understanding and knowledge about web development and IT infrastructure in general is still needed.

2.1 Terminology

This topic may contains confusing terminology. This section will explain some of the terms to ensure to talk about the same meanings. Most of the terms have the same meaning as described in the W3C Push API [1] or the webpush protocol [14]. This thesis also uses the key words "MAY", "MUST", "MUST NOT", and "SHOULD" as decribed in RFC2119 [4] to indicate how the implementation should look like. In contrast to RFCs however this has only minor importance and is therefore not capitalized.

2.1.1 User Agent

The term *User Agent* refers to a context a web application runs in. In most cases this is a web browser like Google Chrome, Mozilla Firefox, Microsoft Edge, Safari or Opera, but it could also be other type of software. For sake of simplicity all these other devices and software that are able to receive push messages implemented with the webpush protocol are also included when using this term.

[1]w3c Push API Concepts https://w3c.github.io/push-api/#h-concepts

2.1.2 Web Application

A *Web Application* is an application running in a web browser. Used technologies for this kind of software is HTML, CSS and JavaScript. Via the Push API the developer has the option to register for Push Messaging over JavaScript. Basically any website can be called web application, because the term is not officially specified. In most cases however websites that rely on user interaction rather than on content are more likely to be called web applications. In contrast to static pages these websites use mostly JavaScript to manipulate the website to act more like native programs would.

2.1.3 Native Application

Compared to a web application a *Native Application* is developed for one particular operating system of device and is deeper integrated in the system. Operating system specific gestures and features can be used as well as a broader variety of APIs. Native applications can be downloaded in application stores or installed manually with the package file. The programming language of the application depends on the operating system requirements.

2.1.4 Application Server

An *Application Server* is a server that processes requests, stores data, and delivers the web application. This server is owned or rented by the website operator and serves as the backend of the application. This is also the server that usually requests the delivery of push messages. Many programming languages for servers are available with JavaScript, PHP, ASP.NET, Java, Perl, Phyton and C being only some of them. In this thesis code examples for the application server are however only available in JavaScript so no mixture of languages is present.

2.1.5 Push Service

A *Push Service* has the job to deliver a push message sent from the application server to the user agent in a timely fashion. The Google Cloud Messaging (GCM) or Apple Push Notification Service are examples for currently active push services. These services are highly optimized for a high number of incoming requests. GCM alone expects to deliver 25 trillion messages in the year 2015 and is currently serving 1.1 million requests every second [36].

2.1.6 Relay Service

Relay Service is the name of the developed prototype. It describes the general system in comparison to *Relay Server* that describes a specific server the relay service runs on. Although it is possible to run the relay service on multiple servers this thesis usually assumes the relay service is running on a single server with one or more instances.

2.1.7 Push Message

A *Push Message* is data sent from the application server to the web application. With the new webpush protocol the contents of the message are encrypted by the application server and decrypted by the user agent. When a request is sent the message is contained in the body of the request and because it is encrypted the content type text/plain is used. Current push services however also allow to send data with JSON without encryption. The size of these contents are in any case limited by the push service.

2.1.8 Push Subscription

A *push subscription* is a message delivery context established between the user agent and the push service. A web application requests a subscription by calling the Push API of a service worker. A push subscription has an associated endpoint in form of a uniquely identified URL exposed by the push service where the application server can send push messages to. All Push Messages are re-associated with a single subscription.

2.1.9 Permission

The term *Permission* refers to the act a user has to fulfill to allow the application to use certain features. Some native application systems ask upfront for all permissions whereas web applications ask at runtime. In case of the Push API the user has to approve the permission before a push message can be sent. This permission can be revoked at any time by the user.

2.2 Push Messaging

Push Messaging is used to send data from a server to a client. Traditionally it is only possible to have a communication between these two parties if the client sends a request to the server, because only the server has a static IP and in most cases a domain associated with it. The client changes the IP, especially on mobile devices, quite frequently and does not listen for incoming requests.

Some applications however rely on updates to be as quickly as possible. To achieve this the application could constantly send requests to the server and ask for updates. Many applications doing this at the same time would result in multiple requests running at all time and consequently draining computing time and energy. Especially for mobile devices with limited battery life this is an immense issue [2].

To tackle this problem *Push Services* have been introduced. The client establishes one performance optimized connection to a Push Service. Other servers can now send data to the Push Service where it is forwarded to the client. In this way only one constant connection has to be held and all applications can send messages from the server to the client without the need of even having the application itself running.

2.2.1 Push Notifications

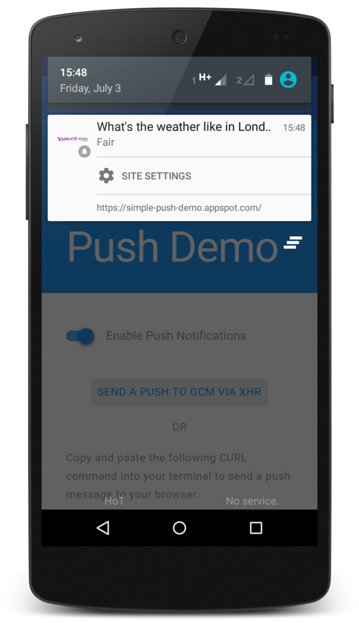
Sometimes Push Messaging gets confused with Push Notifications. However Push Notifications only display data in a notification in response to a Push Message. The look of this notification is defined by the operating system. Most commonly a headline and an informative line are presented next to an icon. In Android Lollipop notifications from web applications additionally have a button for settings to make it easier for the user to turn this feature off. See Figure 2.1 for an example notification.

Push Notifications are frequently used in messaging applications where only a small amount of data is updated. This form of displaying information is nonetheless not the only way to deal with incoming data.

2.2.2 Request to Sync

A request to sync is basically a Push Message containing only some bytes with the information for the client to sync with the server as soon as possible, because there is new data available. This can be used when bigger updates are needed, for example a news application adding a new article.

It can also make sense to request data from the server and show a notification afterward. With this approach the web application makes sure that the user gets to see the expected result after clicking on the notification. There would otherwise be cases where the notification gets delivered when the connection state is good, but the user clicks on it later on when the connection may be gone. In this case he would end up seeing a “404 - not available” page or something similar, but not the desired content.



What's the weather like in Lond.. 15:48
Fair

https://simple-push-demo.appspot.com/

Push Demo

Enable Push Notifications

Copy and paste the following CURL command into your terminal to send a push message to your browser.

HoT No service.

Figure 2.1: Notification from a web application on android. In contrast to regular notifications a button for quick access to the settings is available. Image frame generated with Android Device Art Generator [18].

Chapter 3

State of the Art

Fortunately there is a new programming interface in development[1] that should offer a solution to some of these problems by introducing a ServiceWorker [12]. Similar to a SharedWorker [7] a ServiceWorker runs in a new thread, but it can also intercept all network traffic from the website and serve local files instead. This is done in a programmatic way, so developers have better control. ServiceWorkers are not bound to a specific site and don't even need the website or the browser to be running to be called. This functionality is used to wake the ServiceWorker up as soon as a new push message arrives in order to send a notification or update the cache.

As of mid 2015 web push messaging is still under development and will need some more time to land in all major browsers with a final standard. The drawback of the web as a platform is that progress is sometimes slow, because there are many different browsers and operating systems they are running on. To not end up with different solutions there are two standardization groups that are both working on a proposal for different parts.

The World Wide Web Consortium (W3C)[2] is an international community working together to develop new Web standards. This group is mainly concerned about the front end part of the new standard. The draft they are working on is called "Push API".

The Internet Engineering Task Force (IETF) is a community working on protocol standards and informational documents [1]. They created a Working Group called "WebPush" with the goal to develop a protocol for the communication between application server and push service.

3.1 Existing Systems for Native Platforms

To compare this new API with existing technologies this section will sum up how the most popular services work. Push services for mobile devices are not

[1]as of early 2015 in W3C Editors Draft

[2]http://www.w3.org/Consortium/

a new invention. BlackBerry started off in 2003 with instant push emails [19]. Since then it has gone a long way and is by far not email exclusive anymore. Every major mobile operating system nowadays has their own push service and app developers are required to use them. To compare all existing systems a closer look at the most common systems is necessary.

3.1.1 Google Cloud Messaging (GCM)

Google Cloud Messaging is used for the Android operating system and was first featured in Android 2.2 under the name “Android Cloud to Device Messaging” (C2DM) [20]. At Google’s annually developer conference Google I/O 2015 they announced support for iOS as well.

The Procedure of getting GCM to work on Android devices is rather simple. First a registration on the Google Developer Console is required where a project is set up and an API key is generated. This API key is used in the authorization header for every message sent. A HTTP 1.1 POST request for sending a message looks like the following example from the Google developers website [21].

```
Content-Type:application/json
Authorization:key=AIzaSyZ-1u...0GBYzPu7Udno5aA

{
  "to" : "APA91bHun4MxP5egoKMwt2KZFBaFUH-1RYqx...",
  "data" : {
    ...
  },
}
```

The Device ID used in the “to” property is a 183 byte string with letters A to Z, both lower and uppercase, as well as numbers 0 to 9 and lower dash, dash and backslash. The data property holds custom information for the device to show a notification or update the data model. In total the payload of the whole message is limited to 4kb for data messages.

For only displaying notifications on the user agent the “notification” property with values for “body” and “title” can be used instead of the data property. This will show a notification instantly and is limited to 2kb payload.

```
{
  "to" : "APA91bHun4MxP5egoKMwt2KZFBaFUH-1RYqx...",
  "notification" : {
    "body" : "great match!",
    "title" : "Portugal vs. Denmark"
  }
}
```

There are several other predefined properties that can be set. The “registration_ids” property can be used to send push messages to multiple recipients at once to improve performance of sending a lot of messages. The “collapse_key” identifies notifications and groups all with the same key into one collapsed notification. Although GCM usually delivers messages immediately it could be that the device is turned off, offline or otherwise unavailable. In this case the “delay_while_idle” property specifies whether to send the message later or not. In some cases with a short time of importance it can be better to not show a notification afterward. For example a traffic jam alert is not of importance after a couple of hours while the default waiting time would be 4 weeks. With the “time_to_live” property the expiration date for a message can be set even more precisely.

If the HTTP status of the response is 200 the message is processed successfully. A status 200 does however not tell if the message is delivered to the device or not, but additional information about the status of the message can be found in the body of the response.

```
{ "multicast_id": 108,
  "success": 1,
  "failure": 0,
  "canonical_ids": 0,
  "results": [
    { "message_id": "1:08" }
  ]
}
```

If the message is sent or stored successfully the counter of the success item increases and an object with “message_id” is added to the result array. If there is also a property with “registration_id” the registration id in the database should be updated with this new value. On an error the “message_id” property is not present but instead an error property is. All errors are listed in Table 3.1. A detailed description is shown on the Google developers website [22].

3.1.2 Apple Push Notification Service (APNs)

Apple’s push service was launched with iOS 3.0 and added to Mac OS X later. Getting started is more complex than with GCM, because a payed developer account is required to even test push messaging. Also setting up push messaging is harder because Apple has a different security concept that verifies the sender of each message with a certificate. This certificate is gained after the registration when a new one is generated for your account and must be used in every request. Apple calls this layer “connection trust”. Another layer they call “token trust” assures the message is delivered to the right device. This happens in a similar way as in GCM with a 64 byte

Table 3.1: Google Cloud Messaging error codes

HTTP Code + Response	Error
200 + error:MissingRegistration	Missing Registration Token
200 + error:InvalidRegistration	Invalid Registration Token
200 + error:NotRegistered	Unregistered Device
200 + error:InvalidPackageName	Invalid Package Name
401	Authentication Error
200 + error:MismatchSenderId	Mismatched Sender
400	Invalid JSON
200 + error:MessageTooBig	Message Too Big
200 + error:InvalidDataKey	Invalid Data Key
200 + error:InvalidTtl	Invalid Time to Live
5xx or 200 + error:Unavailable	Timeout
500 or 200 + error:InternalServerError	Internal Server Error
200 + error:DeviceMessageRateExceeded	Device Message Rate Exceeded
200 + error:TopicsMessageRateExceeded	Topics Message Rate Exceeded

hexadecimal encoded string that is obtained by the device after registration with APNs. The payload of a notification can contain any kind of information in JSON format, but must contain the “aps” property with at least one of the following items:

alert A message to display to the user in form of a notification.

badge A number next to the app icon.

sound A sound of the app bundle to play.

content-available If set to 1 the message is silent, meaning no notification is shown. This is the equivalent to data messages of GCM.

All other properties of the JSON payload can be used to send up to 2kb data in total.

```
{
    "aps" : {
        "alert" : {
            "title" : "Game Request",
            "body" : "Bob wants to play poker"
        },
        "badge" : 5,
    },
    "foo" : "bar",
    "foo2" : [ "bang",  "whiz" ]
}
```

Table 3.2: APNs response status codes

Status code	Description
0	No errors encountered
1	Processing error
2	Missing device token
3	Missing topic
4	Missing payload
5	Invalid token size
6	Invalid topic size
7	Invalid payload size
8	Invalid token
10	Shutdown - server closed connection (e.g. maintenance)
255	None (unknown)

Another big difference to GCM is the way of sending data from the application server to the push service. Instead of a POST request a streaming TCP socket design is used. Each push message is sent in one frame and aside from the payload also the *device token*, *notification identifier*, *expiration date* and *priority* have to be set.

In comparison to http response codes APNs uses only one byte status codes. Table 3.2 lists all possible status codes as also shown on the Apple developer website [23].

3.1.3 Windows Notification Services (WNS)

Up to 2014 Microsoft used two different services for push messaging. Microsoft Push Notification Service (MPNS) was used for mobile devices while Windows Notification Service (WNS) was used for desktop applications. Technically MPNS can still be used but just works as a shim for WNS, meaning it forwards all messages to WNS [38]. Therefore this chapter just covers the new universal solution (WNS).

To get started with WNS the app must be registered on the Windows Store Dashboard where a Package Secure Identifier (SID) and a secret key is generated. Each app has an own set of credentials. On first start of the app a channel URL can be requested. This channel URL can be compared to the GCM device ID or the APNs device token with the identifying token set as parameter of the URL. The URL is used as the target for sending push messages. Microsoft states on their developer website [24] that URLs expire after 30 days and should be updated in the database on every start of the app.

For authentication WNS uses OAuth 2.0 [3]. First a HTTP 1.1 request to the microsoft live login is made with the credentials from setting up the app on the Windows Store Dashboard as seen in the example below.

```
POST /accesstoken.srf HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: https://login.live.com
Content-Length: 211

grant_type=client_credentials&client_id=ms-app
   -1-15-...-650196962&client_secret=Vex8L9WOFZuj7XyoDhLJc7
   &scope=notify.windows.com
```

The response to this request includes an access_token that has to be used when sending a push message until it expires. For sending payload WNS uses the XML format with a size limit of 5000 bytes.

```
POST https://cloud.notify.windows.com/?token=
    AQEbU2fSjZOCvRjjpILow HTTP/1.1
Content-Type: text/xml
X-WNS-Type: wns/tile
Authorization: Bearer Fv4Ck1OUrKNmtxRO6Njk2MgA=
Host: cloud.notify.windows.com
Content-Length: 24

<body>
 ....
```

The header field *X-WNS-Type* can be set to one of four predefined values.

wns/badge creates an overlay over the tile on the homescreen.

wns/tile updates the content text of a tile.

wns/toast displays a notification.

wns/raw sends custom payload without a visual action.

Other available header fields are *X-WNS-Cache-Policy* for cache settings, *X-WNS-RequestForStatus* for requesting the connection status of the device, *X-WNS-TTL* to specify when a message expires, and some more that can be found on the Microsoft developer website [25].

For response codes WNS uses only HTTP status codes as described in Table 3.3.

3.1.4 Mozilla Simple Push

Mozilla also got an own push service for their Firefox OS platform. Firefox OS is based on web technologies so this approach is quite similar to the now

[3]http://oauth.net/2/

Table 3.3: WNS response status codes

Status code	Description
200 OK	The push message was accepted
400 Bad request	One or more headers specified incorrectly
401 Unauthorized	OAuth token invalid
403 Forbidden	Not allowed to send to this recipient
404 Not Found	Chanel URI not valid or found
405 Method Not Allowed	Invalid method (GET, CREATE)
406 Not Acceptable	Throttle limit reached
410 Gone	The channel expired
413 Request Entity Too Large	Payload exceeds 5000 byte size limit
500 Internal Server Error	Internal failure caused to fail delivery
503 Service Unavailable	The service is currently not available

developed solution and Mozilla wants to adopt the webpush protocol and PushAPI as soon as they are recommended standards[4]. For now Mozilla implemented only a limited service in terms of functionality.

To enabled push messaging for a Firefox app the property "push" has to be set in the permissions of the manifest file as well as a message property that indicates the page that will receive the push events.

In the app the device has to register to get an endpoint in form of an URL with a token inside. This is, as in the other services, sent to the server and stored until a message should be sent.

```
var req = navigator.push.register();

req.onsuccess = function(e) {
  var endpoint = req.result;
  // send endpoint to server
}
```

To receive events an event listener for push is attached.

```
window.navigator.mozSetMessageHandler('push', function(e) {
  // display notification or do something else
});
```

If a push subscription expires a "push-register" event is fired and a new registration should be fulfilled.

To send a push message from the application server a PUT request to the endpoint has to be made. A custom payload is not accepted, only a version

[4]See Mozilla Developer Network website for more information: https://developer.mozilla.org/en-US/docs/Web/API/Simple_Push_API.

number can be passed in the body of a message. The server responds with HTTP status code 200 or 202 if the message was accepted and a HTTP error message with detailed information in JSON if the sending failed.

3.2 W3C PushAPI

The PushAPI is a JavaScript programming interface that is usable by developers on the front end of the applications, meaning in the context of a browser.

3.2.1 Prerequisites

To use push messaging for web applications developers have to use ServiceWorkers and Web App Manifests. On an explanation page on Github[5] Alex Russell, one of the W3C editors of the ServiceWorker draft, explains ServiceWorkers like this:

> The ServiceWorker is like a SharedWorker in that it:
>
> - Runs in its own global script context (usually own thread)
> - Isn't tied to a particular page
> - Has no DOM access

Furthermore Russell writes about the necessity of HTTPS and its event-driven nature, meaning it can terminate when it is not in use and be woken up when needed. Also a SericeWorker can run without any page as well and has a defined upgrade model.

The Web App Manifest[6] is usually just a JSON[7] file with some meta information about the web application. This is mainly used for pinning a website to the homescreen on mobile devices. The manifest states what the name should be and how the icons should look like.

3.2.2 How it Works

To receive push messages the user has to give permissions. In Chrome for Android this is done by confirming a dialog appearing on the bottom of the screen while the app is active. The user has the options to either block or allow the app to send notifications (as shown in Figure 3.1). After receiving the permissions a call to the new Push API [6] can be made to get a registration id and an endpoint URL indicating the location of the push service.

[5]https://github.com/slightlyoff/ServiceWorker/blob/master/explainer.md

[6]http://www.w3.org/TR/appmanifest/

[7]JavaScript Object Notation http://json.org

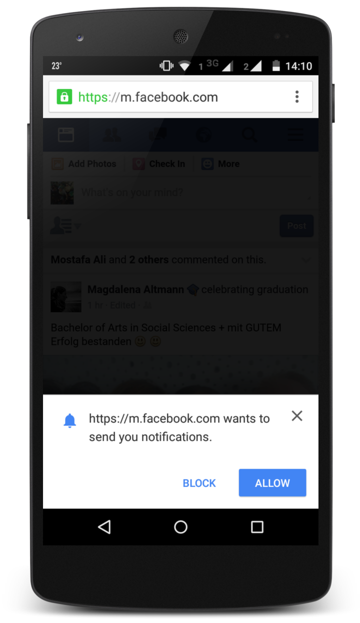


Figure 3.1: Facebook asking for permissions to send notifications. Image frame generated with Android Device Art Generator [18].

This information is now sent to the application server where the web application is hosted to store it. This server can now send push messages to the push service when need, where it is forwarded to the device.

3.2.3 Browser Support

Google Chrome, Mozilla Firefox and Opera are showing enthusiasm for ServiceWorkers and therefore push messaging. However only Chrome has a working first implementation for the PushAPI with limited functionality as of version 40. Details and current status for each part of the API can be found at a website called "is ServiceWorker ready?" [26].

A polyfill[8] is unfortunately not possible, because a ServiceWorker is completely new concept that no other existing web technology can recreate.

Despite the lack of current support it can be expected that other vendors will implement this functionality if it becomes a success.

[8]A polyfill is a piece of code that imitates a missing functionality that the browser would usually provide.

3.3 IETF WebPush

To deliver messages from the application server to the Push service a uniform protocol is necessary. Otherwise developers would have to implement a different solution for every available Push service and keep all of them up to date. This protocol is developed by a working group of the Internet Engineering Task Force with members from Mozilla, Google, Microsoft and many more big companies. The goal is to find a simple solution for sending messages in realtime and at the same time minimizing the amount of additional information that is revealed to the push service as described in the charter of the working group [27].

3.3.1 Planned Protocol

The main players of this new protocol [14] are the web application running on the client device, the user agent in which context the application runs, the push service, and the application server (definitions for these terms can be found in section 2.1). To initialize the process the web application requests to subscribe for push messages on the user agent, which sends a HTTP/1.1 POST request over the network to the push service. The responded subscription is the base for the following interaction. It consists of an endpoint URL which indicates the location where future messages should be sent to, a link header [9] for receiving push receipts and a location for retrieving push messages.

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:52 GMT
Link: </p/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
  rel="urn:ietf:params:push"
Link: </receipts/xjTG79I3VuptNWS0DsFu4ihT97aE6UQJ>;
  rel="urn:ietf:params:push:receipt"
Location: https://push.example.net/s/LBhhw0OohO-
    Wl4Oi971UGsB7sdQGUibx
Cache-Control: max-age:864000, private
```

The user agent stores the location header and does not reveal it to any other party. The endpoint for sending push messages and requesting receipts are sent to the application server where they are usually stored in a database for future usage as seen in Figure 3.2.

For requesting a push message delivery the application server needs to send a HTTP/1.1 POST request to the stored endpoint (see Figure 3.3) with the message in the body of the request. The push service then forwards it to the device and further to the user agent with HTTP/2 server push [3]. In comparison to existing push services this method relies on standard HTTP requests rather than XMPP or websockets to make the implementation easier.

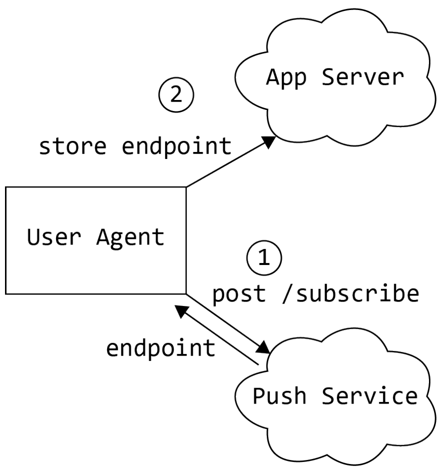


Figure 3.2: Simplified subscription process of the webpush protocol proposal.

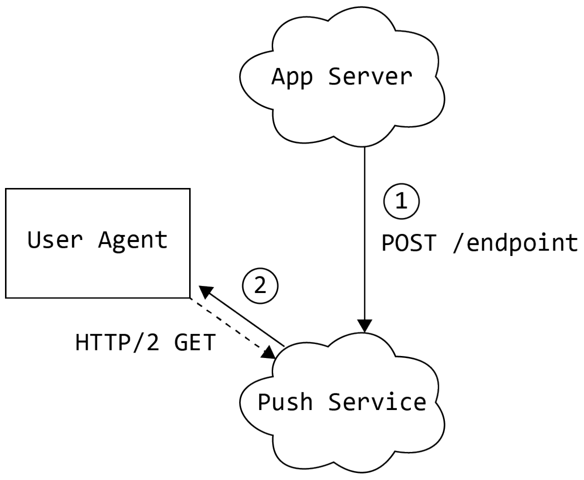


Figure 3.3: Simplified sending process of the webpush protocol proposal.

Security

Security concerns are one major topic in the process of designing this protocol. Not at last because of recent scandals. Therefore all network communication must use HTTP over TLS [11]. This however does not protect the messages from being read by the push service. The solution to this problem is End-to-end encryption. This additional security is accomplished by an asynchronous key pair that is generated on subscription by the user agent. The public key is then sent to the application server whereas the private key remains secret and is not even exposed to the web application. When the application server wants to send a message it encrypts the body of the message with the public key and sends it to the push service. The message is decrypted by the user agent and handed over to the web application.

The authorization to send a message to a certain subscriber is given by knowledge of the capability URL [13]. With a high enough complexity of at least 120 bits of random entropy the URL is almost impossible to guess. For further confidentiality the push service can implement an authorization based on a HTTP-compatible method.

Aggregation

Although delivery of the same message to a large number of recipients is a common feature in push services the current proposal of the protocol does not offer a solution for this problem. This is because end-to-end encryption is used individually for each recipient with a different key. A different aggregated channel would have to be created and is in discussion already, but the first version of the draft will not include it.

Receipts

To allow seeing the state of a sent push message, the protocol includes receipts. When a message is sent to the push service a receipt URL is received in response. This URL can be used by the application server to subscribe for receipts by sending a HTTP/2 GET request. Once the status changes the push service can respond with a status code. The code 410 is used if the message got delivered to the device and is now “gone” from the push service. To ensure an unmitigated delivery to the app, the user agent acknowledges a message after successful decryption.

3.3.2 Push Service Support

Currently no push service fully supports this new protocol because it is still under development, but Google is working on a prototype implementation [28]. They also mentioned to implement it requiring some sort of sender authentication [29]. Employees of Mozilla and Microsoft are also contributing a lot to the current draft, so it is likely[9] to see an implementation soon.

3.3.3 The Problem with Push Services

This protocol is the critical part of the whole API, because there are already a lot of different Push services and all of them have different proprietary protocols. In theory the new protocol sounds great, but until the protocol

[9]Mozilla Bugzilla report about Push API can be found here https://bugzilla.mozilla.org/show_bug.cgi?id=1038811.

is finished and all push servers offer this protocol a lot of time will pass and there is no guarantee all push servers will even implement it at all. Until then browers that offer the PushAPI will probably rely on their existing proprietary protocol as Google Chrome currently does. To solve this problem a new approach is necessary.

Chapter 4

New Approach

Existing solutions matured over time and are used by thousands of apps. Alone GCM delivers messages to 1.5 billion devices [36]. A radical change to the new webpush protocol is therefore highly unlikely. Push Services need to implement the new protocol as a second channel, but this could take some time. An intermediate or even long term solution for this problem could be to introduce another server in between that handles the authentication to the proprietary PushService. This relay server offers an API with the webpush protocol. With the help of it the sender does not need to know which push service it is talking to and just send a POST request to the URL it got as endpoint. All traffic for web applications would run over this relay service, therefore performance and scalability plays an important role. Key contributors for bad performance and added latency are access to databases and waiting for other network requests. Also a big amount of kept alive connections can slow down servers. To guarantee good performance this relay service should not persist any data and use the push services in an efficient way with as few kept alive connections as possible.

4.1 Overview

Instead of only three for push messaging a fourth player is introduced as shown in Figure 4.1 with this new approach. This relay server is strategically best located close to the push service, because it will be the main communication partner. To offer such a relay service some requirements have to be fulfilled. This solution can for example not be used as a third party shim, but has to be implemented by the user agent itself, because it is the one registering and providing the endpoint URL, so it needs to know the location of the relay service. It would however also be possible to let the push service implement the relay and return the relay endpoint on device registration. This paper has a focus on the first of these options. Technically this approach can be used with any push service, but for sake of simplicity this

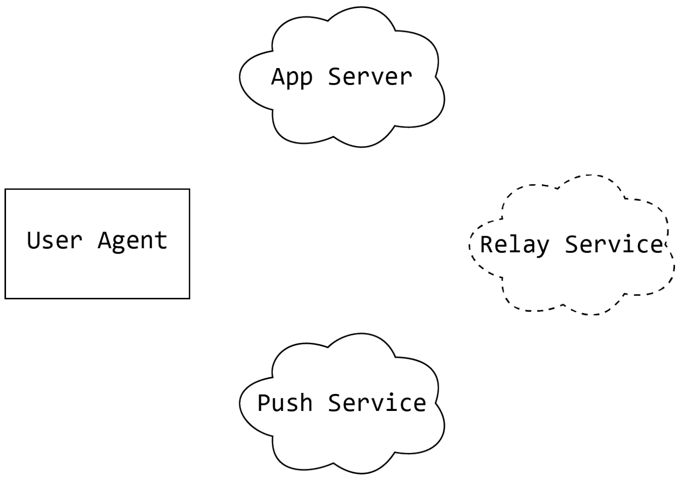


Figure 4.1: A new server is added to the system. This server can also be integrated in the push service.

paper deals only with GCM, APNs and WNS. An example implementation for GCM is shown in Chapter 5.

4.2 Mapping Device Tokens

The user agent still uses the proprietary push service to subscribe for messages (see Figure 4.2). In return to this initial request usually some kind of token is received. The webpush protocol states in Section 8.3:

> 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.

All three tested push services fulfill this requirement, but in the case of WNS the token consists of a full URL with the origin “sin.notify.windows.com”. The actual token is set as the “token” parameter of the request. There are several ways to tackle this problem. If the origin never changes the token can be extrapolated from the URL and used instead. The origin is stored in the relay service. However sometimes the origin changes for different subscriptions because of different push service locations or load balancing reasons. If the origins are static and all are known it is possible to use different relay service instances for each of them with different URLs or ports. For dynamically generated origins it makes sense to encode the whole URL and use it as token. The relay service has to decode it later and send the push messages to the according URL instead of using a static push service location.

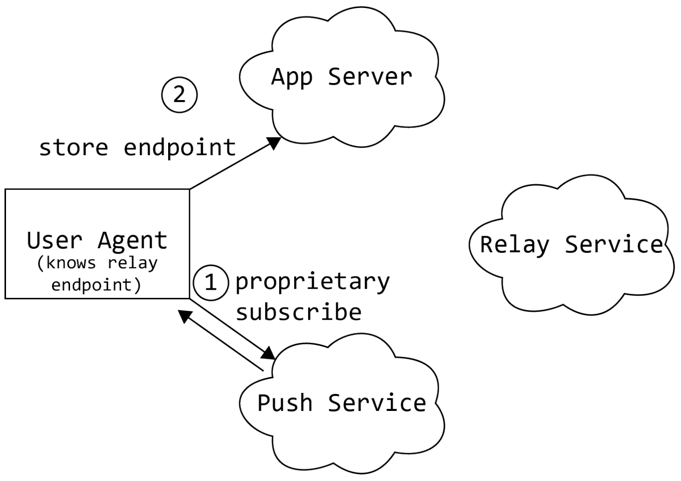


Figure 4.2: Subscribing with a relay service does not change much on first sight, but the device tokens have to be changed.

Once the token generation is done it is added to the base relay server origin path, set as new endpoint URL, and handed over to the application. The application then sends this data to the application server where it is stored in a database.

4.3 Security Concerns

According to the webpush protocol each push subscription is bound to a user agent generated encryption key. Although none of the three tested push services however offer a mechanism like this, encryption can still be applied because it is used independently of the push service. Therefore the user agent just needs to make sure to generate a key pair. The public key is then sent to the server along with the endpoint URL and stored in the database. All push messages for this user agent are encrypted with this key. This ensures that push services cannot read the data sent in the messages. Because of this End-to-End Encryption there is not a big risk of introducing another server in the middle, because no sensitive data is exposed. However the relay service still needs to be trusted, because it has knowledge of the capability URLs and is therefore technically authorized to send messages. It cannot send different data or alternate its content because the decryption of the message would fail and the message would not be handed over to the web application. Still it could send the same message over and over again and start a DDOS attack on the push service that has to deliver the messages. This could either result in availability problems of the push service or lead to a ban of the original sender or web app in general.

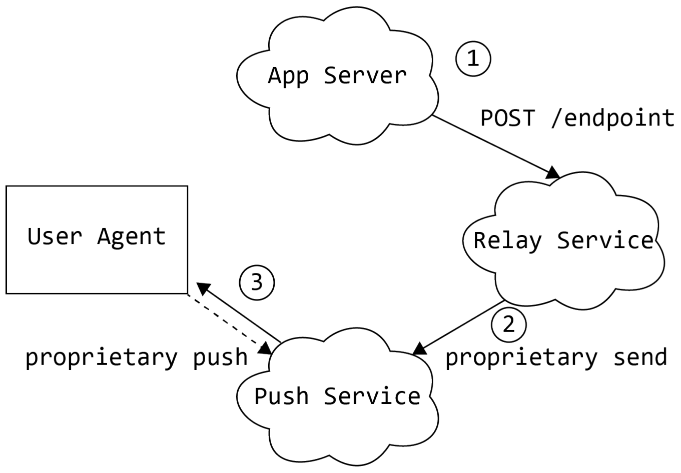


Figure 4.3: Sending a push message with a relay service.

4.4 Sending Messages

To send messages the application server uses the stored public key to encrypt the message and transmits it to the also stored endpoint URL with the standardized request as shown in Figure 4.3.

```
request({
  'method': 'POST',
  'url': endpoint,
  'body': encryptedMessage
})
```

The relay service reacts to the incoming request by sending a request to the push service with a proprietary protocol and authentication and responds to the initial request according to the response of the push service. The push service handles the message as usual and delivers it to the device as soon as possible.

4.5 Authentication

As described in Section 3.3 the sole knowledge of the endpoint allows the application server to send messages, because the entropy of the URL is too high to guess. Some push services however will need further authentication by requiring developers to sign up and use some sort of key to authenticate the sending application server. How this authentication looks like in detail is not directly specified in the current version of the webpush protocol draft but it is stated in Section 8.3:

> A push service MAY choose to authorize requests based on any HTTP compatible authorization method available, of which there are numerous options.

GCM already uses a HTTP-compatible authorization method for their first version of the implementation by setting the "Authorization" header. This approach is also used for the prototype where the header is read and just passed along. For WNS the sender has to retrieve a bearer token via OAuth 2.0 first and also set it as Authorization header. APN on the other hand uses a completely different method by uploading certificates. This cannot be accomplished with a HTTP-compatible authorization method, so instead the relay service itself has to authorize using a certificate and let the application servers send messages without additional authorization. This of course can result in a ban of the relay service quickly because of too many messages if there are too many websites sending messages at the same time or a denial of service attack happens. This problem can be worked around by running multiple relay server instances with different certificates, but this also increases cost and complexity of the system.

4.6 Mapping requests and responses

To use existing push services their API has to be mapped to work with the webpush protocol. A specific call for a proprietary operating system functionality cannot be considered, instead only the data channel of existing push services is used, because via the PushAPI it is possible to specify in the ServiceWorker which action (e.g. show a notification) should take place. Other OS specific actions (e.g. badges, tiles, toasts, ...) should be shown via JavaScript APIs.

The response codes can be matched quite easily to webpush as shown in Tables 4.1 and 4.2. Only APN lacks the direct response of expired token which is quite important to create no redundant requests when trying to send messages to non existent devices. For this functionality APN got a "Feedback Service" that is used to get information about expired registrations. The feedback service should be queried once a day and returns a list of expired tokens. Implementing this in the relay service would not be a big problem, but violate the performance rules set. To know expired tokens is however necessary and therefore an exception to the rule should be made. In the end only a list of tokens needs to be stored. Once the application server tries to send a message to one of these tokens a 410 response is sent and the token is deleted. It is then the application server's responsibility to delete the registration. A problem exists if the application server does not delete the token and use it again.

Table 4.1: Mapping GCM responses to webpush

Description	GCM	webpush
Success	200 +message_id	201
Invalid token	200 +error:NotRegistered	404
Expired token	200 +registration_id	410
Payload too large	200 +error:MessageTooBig	413
Rate Exceeded	200 +error:DeviceMessageRateExceeded	406
Invalid TTL	200 +error:InvalidTtl	400
Internal Server Error	500 or 200 +error:InternalServerError	500
Method not allowed	-	405

Table 4.2: Mapping APNs and WNS responses to webpush

Description	APNs	WNS	webpush
Success	0	200	201
Invalid token	2, 5 or 8	404 or 403	404
Expired token	-	410	410
Payload too large	7	413	413
Rate Exceeded	-	406	406
Invalid TTL	-	400	400
Internal Server Error	1, 10 or 255	500	500
Method not allowed	-	405	405

4.6.1 Delivery Receipt

Webpush however also includes the option of push message receipts. This is used to ensure the message is properly delivered. GCM does not support this feature in basic HTTP downstream, but another method of GCM does. The GCM “Cloud Connection Server” (CCS) uses the XMPP protocol and can send delivery receipts over a persistent XMPP connection. However CCS is limited to 100 connections per sender and the results would need to be stored if the application server requests the receipts later. This plays against the performance goals of the relay service and cannot be implemented in this way.

APN and WNS do not have any direct receipt functionality implemented either. This issue does not allow a full mapping of webpush to existing push services, but the core functionality is still given. Furthermore an acknowledgment can also be made by the app itself with a request to the application server.

Chapter 5

Implementation

A relay service can only be used in conjunction with a corresponding user agent. This constraint is given because the user agent needs knowledge about the relay service to adopt the endpoint URL. In this example Google's push service GCM is used in conjunction with a custom browser running on the Android operating system. In similar ways this approach could easily be used for other operating systems.

5.1 User Agent

To represent the standardized format how push messaging registrations should be handled the user agent has to register to a push service. For native applications this represents a fairly common procedure and detailed instructions and examples for this can be found online. On Android a library has to be included in the project and the API Key needs to be specified in a configuration file. The connection with the push service is handled by the operating system itself and not by the app. This means the app does not register for push messages at the push service directly, but has to ask the operating system to subscribe for push messages. In contrast to native apps web applications do not talk to the operating system directly, but to the browser first. The browser then acts as a regular native app and registers the web application at the push service. This means a website cannot "polyfill" a push service by adding a library if there is no existing underlying implementation of the user agent.

5.1.1 Custom Browser for Android

The user agent has to know the location of the relay service to hand the correct URL to the web application. To test the approach with a relay service a new browser is needed (see Figure 5.1). The easiest way to create an app that works like a browser is to use Apache Cordova™[1], which is a native shell for web application that exposes a set of device APIs to JavaScript and runs the application in a sandboxed webview. Other frameworks use similar techniques [10] and would also be possible to use for this project but the big community around Apache Cordova simplifies error detection and usage of third party plugins. It also allows to develop applications with web technologies and publish them to native app stores and look like a native application. This is possible by rendering the website in a webview without any URL bar or controls. The numerous plugins available expose otherwise not accessible device functions to the web application. Two of these which are used in this prototype are called "InAppBrowser" and "gcm".

Apache Cordova Plugins

The plugin "InAppBrowser" is used to work like a basic web browser with an editable URL status bar. The original version of the plugin is designed to offer a preview of a website within an app without giving the possibility to change to URL. It is used for sign-in forms and quick preview of newspaper articles. For this prototype however the status bar needs to be editable and therefore the plugin had to be adopted by creating fork of the main repository and changing parts of the code. This fork can be found on a Github repository [30].

Usually Cordova applications use local HTML files to show content and whitelist all external calls, because otherwise all websites would have access to the Cordova API and therefore all permissions the app got granted on installation. The InAppBrowser provides a save context where websites can run without access to these developer interfaces. For the prototype websites will however need access to one of these interfaces. The gcm plugin provides access to the Google Cloud Messaging service over a proprietary API. To register for push messages the register function has to be called inside the app.

Whenever a new site loads in the InAppBrowser a piece of JavaScript is injected that sets the endpoint and therefore exposes parts of the original API. This could all be done in a more elegant way by modifying the open source chromium browser to let it provide the new endpoint itself, but for a prototype this solution is sufficient. The only drawback of this approach is the delay it takes until the script is successfully injected into the website.

Push API Mock

Webviews lack the support of ServiceWorkers and can therefore not offer the API needed for subscribing for push messages, therefore it has to be imitated by mocking ServiceWorker and PushManager registration. Also receiving

[1] https://cordova.apache.org/

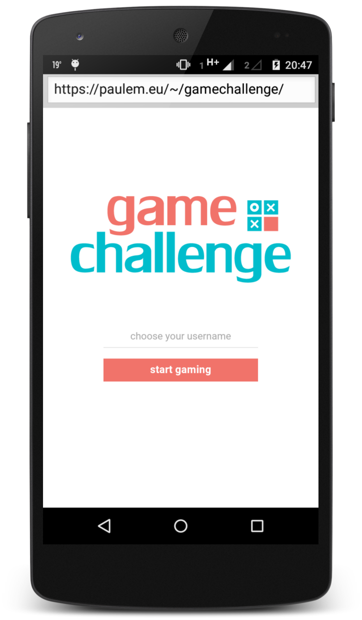


Figure 5.1: Custom browser based on Apache Cordova. Image frame generated with Android Device Art Generator [18].

push messages inside the application would not be possible because this can only happen inside a ServiceWorker according to the Push API. To still see a result the contents of the push message are sent to the application where a notification can be shown.

5.1.2 Plugin Solution for Existing Systems

Chrome announced partial support for push messages in version 42. However, the messages are still sent over the regular GCM API and not with the new webpush protocol. On registration the browser returns the proprietary endpoint for GCM. This endpoint consists of a location from which the push server is reachable. To use the relay service the token has to be extracted by the plugin and added to the relay server location as shown in Figure 5.2. This plugin solution won't work in user agents that do not support push messaging, because the registration and connection is still handled by the user agent itself.

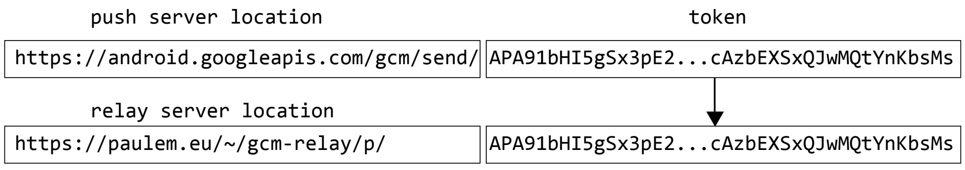

push server location
token
https://android.googleapis.com/gcm/send/
APA91bHI5gSx3pE2...cAzbEXSxQJwMQtYnKbsMs
relay server location
https://paulem.eu/~/gcm-relay/p/
APA91bHI5gSx3pE2...cAzbEXSxQJwMQtYnKbsMs

Figure 5.2: Plugin based solution changes server URL.

5.2 Push Relay Service

The push relay service is the main focus of the implementation, because the performance relevant parts happen here. With an enormous high number of incoming requests this service needs to perform perfectly without sacrificing too many features of the proposed webpush protocol. As described in chapter 4 a way of achieving this performance is to avoid using databases and CPU intensive tasks. Luckily CPU critical operations like encryption and decryption take place on the application server and user agent and databases were be avoided in this approach.

5.2.1 Technology Stack

Choosing the right tools for a project is the key for getting good performance. A well working technology stack does not only utilize the best programming language but also its best libraries and features.

Language Comparison

While some programming languages are designed to do CPU intensive calculation others perform well in networking operations. Some are known for a good community while others offer long term support for enterprise customers. To find the best fit for a relay service a small benchmark setup with different langsuages is revealing the strengths and weaknesses clearly.

In this test the popular web server languages *GO*, *Java*, *JavaScript* (node.js), *PHP*, *Python* and *Ruby* are compared in a setup that simulates the relay server by forwarding an incoming POST request to another service. This other service also runs locally and waits 100ms the respond. In combination with a high number of concurrent requests this creates congestion on the tested systems. It is important to note that all test servers are deliberately kept small (maximum 60 lines of code) and do not use any framework or libraries. Performance of some of these tests could dramatically increase with the right tools in use or further code optimizations. Therefore the captured data should provide only an approximate value about strengths and weaknesses of the according languages. Detailed information on how to rerun

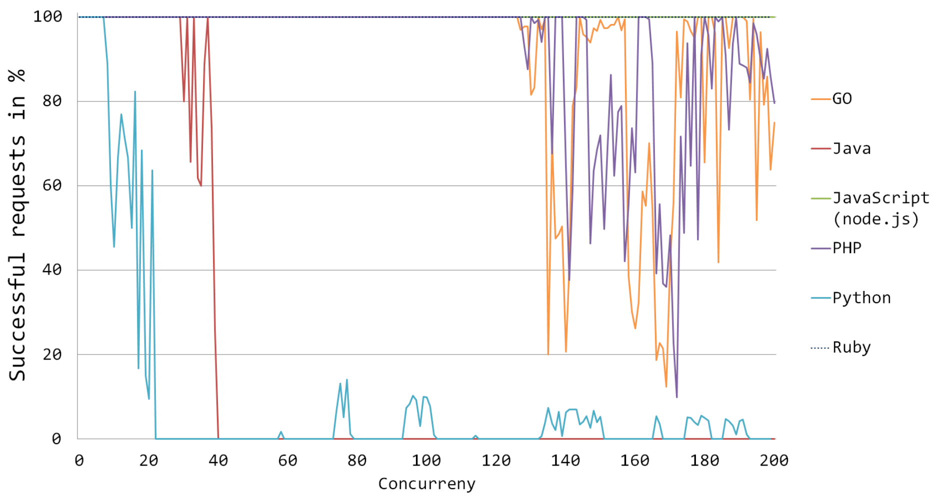


Figure 5.3: Success rate of requests on different levels of concurrency.

tests on a different system and where to find the code for this benchmark can be found in Section 6.3.4.

The tests start of with a concurrency of one and rise up to 200. While some programming languages queue the incoming amount of requests others drop requests because of an overfull buffer as shown in Figure 5.3. Python drops the first request at concurrency of 9 and almost stops serving requests afterward. The same effect applies for Java where requests start to drop at 30 concurrent requests. The other languages have considerably less problems with a first decrease of the success rate at around 120 parallel requests. JavaScript and Ruby did not even drop a single request during the test and are therefore clear winners.

However, another view at the collected data reveals some even more interesting results. While comparing the success rate is one major factor for good performance another one is the time the requests need to be processed. In Figure 5.4 the request per seconds in conjunction with concurrency are compared. Although Ruby was able to process all incoming requests it could only handle five at a time. This means some requests at a concurrency of 100 took as long as 20 seconds. Java and Python behave similar by steadily increasing the requests per second but lack far behind GO, JavaScript or PHP. While GO starts to perform best the dropped requests are noticeable at 130 parallel requests where also PHP starts to have problems. JavaScript shows no drops of requests per second with a mean average duration of only 414ms per request at a concurrency of 200. This means the JavaScript server did not even reach its limits in this benchmark and therefore suits perfectly for a prototype implementation of a relay server.

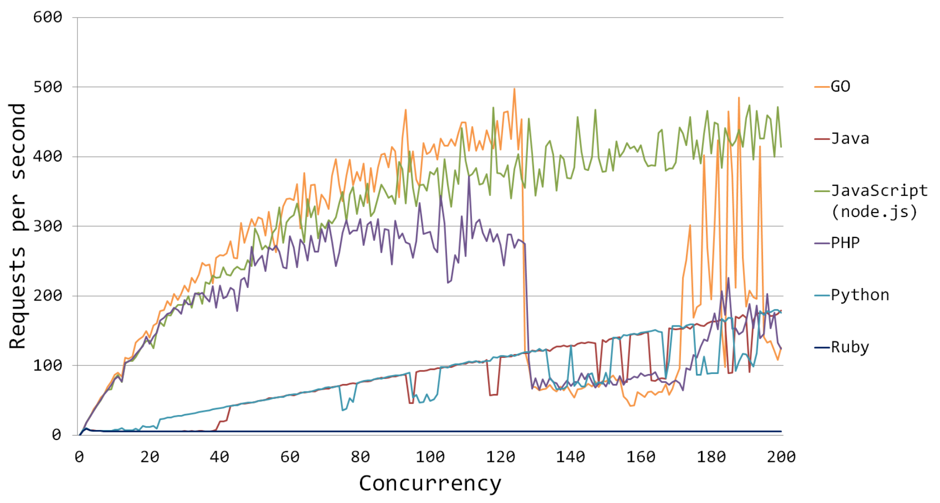


Figure 5.4: Request per second language comparison. Go, PHP, and JavaScript start off with similar characteristics but only JavaScript keeps a constant performance.

Node.js

To run JavaScript on a server node.js[2] is used. It has good scaling characteristics for real-time networking applications as also other tests show [15]. Node.js uses Google's V8 engine for JavaScript interpretation and just-in-time compilation (JIT). To extend the basic JavaScript functionality node offers core modules for accessing the file system and operating system functions as well as cryptography libraries and much more. Compared to JavaScript in the browser in node.js there is no DOM available and therefore the DOM API is neither. Also node.js uses a simple module loading system for better separation of concerns instead of including all files. This can contribute to a better code overview as dependencies are included with a *require* statement at the beginning of the file.

Although it is a relatively new runtime with its first release in 2009 big companies like ebay [31] and Walmart [37] are using it already in parts of their websites. Because of the asynchronous event driven nature of JavaScript an incoming request does not block other requests. This allows node.js to run in a single thread and use less memory than other multi-threaded frameworks. JSON manipulation is no problem with JavaScript and the problem of bad CPU intensive performance of node.js is made less relevant by reducing the complexity of the program.

[2]https://nodejs.org/

Modules

Furthermore, node.js offers a rich ecosystem of modules available via the node package manager (npm). This project uses “restify” for setting up a “Representational State Transfer” (REST) http server which is later used as endpoint for push messages. Also used are “request” for simple http requests, “minimist” for reading command line inputs and “pem” to generate self signed certificates if non is set via the command line interface. The certificates are necessary because the server runs on SSL to provide better security. Although the actual content of the message is encrypted by the application server with a separated key the meta information about the message sent in the header as well as the registration token in the URL should be encrypted as well.

5.2.2 Application Structure

To send the request to GCM the registration key needs to be set in the “to” property and the content of the request body in a nested data property, because there are only certain properties allowed. Web push messages just use the data channel which is used by the data property inside of the general data property. The TTL header can be mapped directly to the “time_to_live” property and “dryrun” is used only for testing.

```
request({
  'method': 'POST',
  'headers': {
    'Content-Type': 'application/json',
    'Authorization': req.headers.authorization
  },
  'uri': 'https://android.googleapis.com:443/gcm/send',
  'json': true,
  'body': {
    'to': req.params.id,
    'data': {
      'data': req.body
    },
    'time_to_live': req.headers.ttl,
    'dry_run': dryRun
  }
}, function (err, resService, resBody) {
```

The server uses if and switch statements to map the response from GCM to the webpush protocol. First a check is made for any internal errors like network errors. In the webpush protocol it is not defined which status to send in this case but status 500 seems appropriate. The detailed error description is not set as body of the message for security reasons. The error should however be logged and frequently observed, because the push service could for example ban the IP of the relay server.

```
if (err) {
  return res.send(500);
}
```

Next the status code of the response is tested. If the code is not 200 there could be a chance of an invalid registration. In this case a 404 is returned, indicating the subscription could not be found and should be deleted on the application server. Otherwise the same status code is sent. This should be rarely the case, because GCM handles errors in the response body rather than in the status. Sending a status that is not specified in the webpush protocol could lead to some discrepancy but is still more informative than sending an *Internal Server Error* status.

```
if (resService.statusCode !== 200) {
  if (resService.statusCode === 400 && resBody.indexOf('
      INVALID_REGISTRATION') !== -1) {
    return res.send(404);
  }
  return res.send(resService.statusCode);
}
```

If there were no erros so far the JSON response body can be examined. Is the success property set to 1 the message was processed by the push server and a 201 response can be sent. Only the "registration_id" field has to be checked. If it is set, there is a new registration id available and the old one should be discarded. There is no such process in the webpush protocol, but setting the expiration header to 0 should indicate that this registration is expired and a new registration is needed. The application server is in charge of deleting the subscription from its database and requesting a new subscription from the application.

```
if (resBody.success === 1) {
  if (resBody.results[0].registration_id) {
    res.set('Expires', '0');
  }
  return res.send(201);
}
```

Finally if the success property was not set to 1 the error property of the result is mapped to the status code of the webpush protocol. By doing so some errors have to be merged because the protocol does not support them and therefore information is lost. By adding the detailed information to the body of the response it can be preserved and developers can at least see the reason in the server logs.

```
switch (resBody.results[0].error) {
  case 'NotRegistered':
  case 'MissingRegistration':
  case 'InvalidRegistration':
    res.send(404);
    break;
  case 'MessageTooBig':
    res.send(413);
    break;
  case 'InvalidTtl':
    res.send(400);
    break;
  case 'DeviceMessageRateExceeded':
  case 'TopicsMessageRateExceeded':
    res.send(406);
    break;
  case 'InternalServerError':
  case 'InvalidDataKey':
  case 'Unavailable':
  default:
    res.send(500, resBody.results[0].error);
}
```

Chapter 6

Evaluation

To evaluate how good this prototype performs and if it actually makes sense it is important to keep the goals in mind. Web developers should be able to use the new standard faster to make development easier, without having to change the code later on if the push service fully supports the protocol. The problems about who has to implement this relay service however remains. Only browser manufacturers themselves are able to do this but it can be a huge benefit for them as well because they can have a low cost alternative to make the new API available without having to implement their existing solution for native applications again for the web. With the approach described in this thesis they can test the response of developers and see if it is worth the effort to develop a complete solution.

6.1 Coverage of Webpush Protocol

An important role of the relay approach is to behave as closely as possible like an actual implementation of the push protocol from the view of the application server and web application. The biggest problem are receipts, because current push services do not offer similar methods. This could result in problems for developers if they rely on this feature to be available, because in the webpush protocol it is stated as required and not optional. The browser vendor has to make this clear to the developers. If they know about the absence of this feature a simple if statement would be enough to detect a not exiting push-receipt header. To make this even more obvious to developers a new header could be set indicating that the application server is talking to a relay service and therefore functionality is limited.

6.2 Ease of Use

The new protocol will reduce the amount of code and complexity a lot and will bring more developers to use this energy saving method of sending messages to devices. In the best case developers would have to implement the protocol once without bothering about differences between different push services. In real life this will however not be completely true, because companies like Google already announced to require authentication to use the service, so developers still need to differentiate between them. A common way to do so is by scanning the endpoint URL for certain words. In a sample endpoint that looks like this

```
https://android.googleapis.com/gcm/send/APA91bHDsiWcJT50...
```

it would make sense to search for the appearance of the first part of the string without the token to set the the authorization header.

```
var GCM_URL = 'https://android.googleapis.com/gcm/send';
var GCM_AUTH = 'key=AIzaSyCjwXopyMFOpL0C5SOzvKdC9U3hVe2LZ';

var authorization;
if(endpoint.indexOf(GCM_URL) === 0){
  authorization = GCM_AUTH;
}
request({
  'method': 'POST',
  'url': endpoint,
  'content-type': 'text/plain;charset=utf8',
  'headers': {
    'Authorization': authorization
  },
  'body': message
}, function (error, response, body) {
      // ...
});
```

This URL is hence hardcoded into application and difficult to change. This has to be considered when offering a temporary solution with a relay service. If the authentication method or key changes a different URL has to be chosen as well so the if statement does not match any more. The same is valid for the opposite: If the authentication method and key stay the same, the URL should stay the same as well to not require an additional statement. This can be accomplished by adding a different sub path.

```
https://android.googleapis.com/gcm/send/relay/APA91bHDsi...
```

6.3 Performance

Node.js is known as a high performance framework for applications with high numbers of requests and little needed CPU intense calculations. To test the performance of the relay service actual requests are sent to GCM with the flag “dryrun” set to true so the push messages are not processed by Google, but still a real status message is returned. This way actual latency and load can be measured without risking to get a “Rate Exceeded Error” while testing.

As a test setup for the relay server a virtual server from hosteurope is used [32]. It runs on two virtual cores with 4GB RAM and 100Mbit/s guaranteed peak bandwidth. As operating system Ubuntu 12.04.5 LTS is used. This system is deliberately chosen to be relatively low powered compared to commonly used hardware in production systems to easier discover bottlenecks and weaknesses of the application.

6.3.1 Latency

Some applications have tight time constraints and require the delivery of push messages to be as fast as possible or otherwise the push message is not of importance any more. One case would be a call app in which a message about an incoming call should be sent as quickly as possible. An additional relay server in the middle adds additional latency to the whole process, because one more request has to be made.

To measure this added time a computer on another location sends requests to the relay server as well as to GCM directly and measures the difference in time it takes to get a response. On a production system a high amount of requests can be expected. To simulate this load concurrent requests are sent. The test starts with one and measures until 50 simultaneous requests. For every step of this test 100 requests per concurrency are sent and stored. Out of these results a mean average is calculated and plotted in combination with the number of concurrency as shown in Figure 6.1.

The results show an expected small increase of latency with the relay service but it is not a significant increase and even with higher concurrency there is no additional latency added. In total 127.500 requests were sent with an error quote of 0.13% (502 Gateway errors) and on average $47ms$ added delay. For most applications this is not easily distinguishable from the regular sending.

Another test measures the time added from the application server to the device. This test is what is going to be most relevant for real time applications but at the same time it is harder to measure because the application server and the device clock need to be in sync. With the Network Time Protocol (NTP) this can be accomplished but even rooted Android devices do not get synced to a server highly accurate. To solve this problem all requests got normalized so the slowest request takes 100 milliseconds and all others are relative to this one. The important part in this test is not the time it takes in general but the delta time between the two approaches. 1000 requests are sent to both the relay service and GCM directly and the time

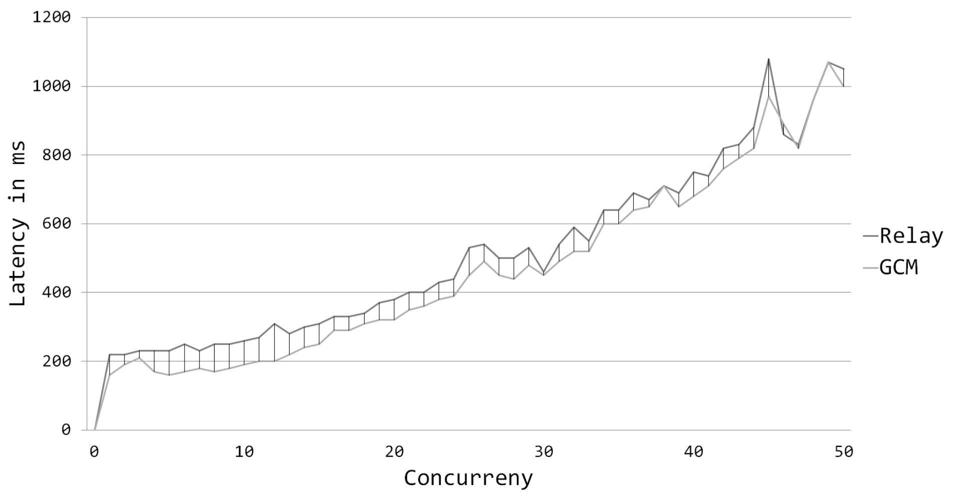


Figure 6.1: Latency comparison of relay service and plain GCM requests with different concurrency and mean average of 100 requests per concurrency.

is measured until the message arrives in a sample Corodova Phongap app. The time of the requests are grouped into segments of 10ms and plotted on chart as shown in Figure 6.2. While plain GCM performs overall better with a mean average of 184ms compared to 257ms of the relay service not a huge difference is visible. The better Google servers perform more reliably than the cheap relay service but this problem could be solved because push service providers have the possibilities to put the relay service on the same location as the push service to further reduce the latency to a minimum.

6.3.2 CPU load and memory consumption

Push services have to handle a big load of requests and servers can therefore be very expensive. The next test is aimed to monitor CPU load on a high number of concurrent requests as well as memory consumption. One request is sent to the relay service at the beginning and gradually more are added until a concurrency of 80 is reached. The CPU load, shown in Figure 6.3 quickly climbs up to around 70% beginning at 20 concurrent requests, but as the latency test shows it has no significant impact on time usage. The whole test averages a CPU load at 63 percent with a peak at 97. Node.js is single threaded and uses only one CPU core. It is easy to run more instances to use the other cores because no session states have to be stored.

Memory consumption is always low because no state has to be stored. In Figure 6.4 a constant value between 140 and 160MB can be seen. With an average of 147MB the process uses little memory and therefore a high number of instances can run on the same server.

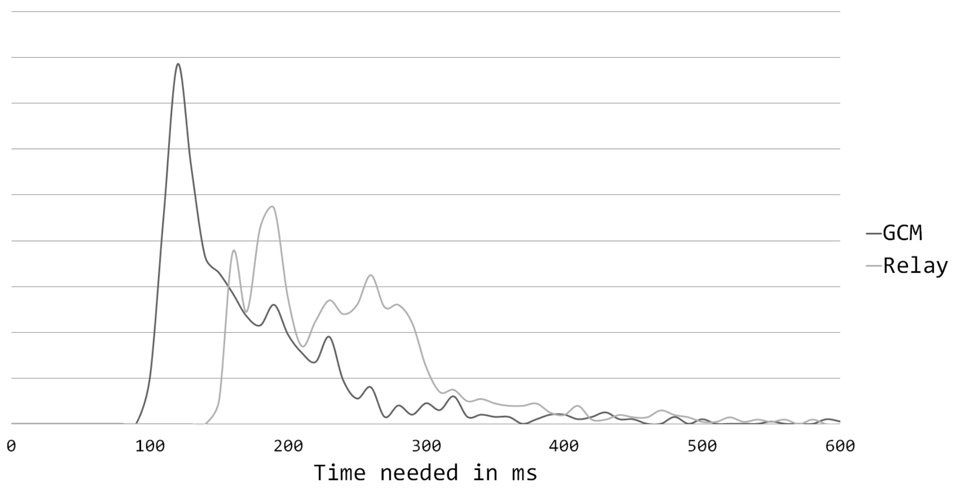


Figure 6.2: End to end latency comparison of relay service and plain GCM requests normalized to 100ms for the lowest request.

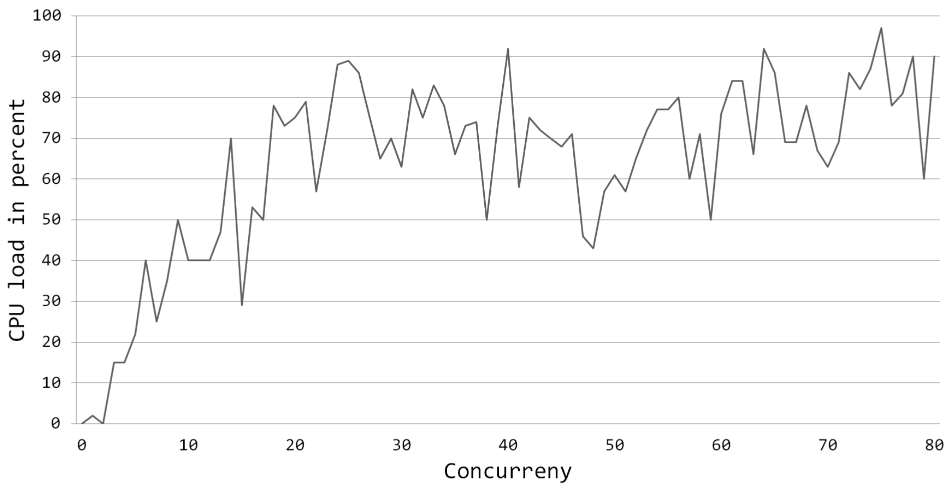


Figure 6.3: CPU load of the tested system.

6.3.3 Bandwidth

As CPU load and memory consumption don't seem to be of a big problem the bottleneck is more likely to be the network itself. A single request can have a maximum payload of 4.096 bytes, but not a high amount of request will use all of it. It is rather likely to send a request to sync which only uses few bytes. A theoretical average of 100 bytes payload per request should

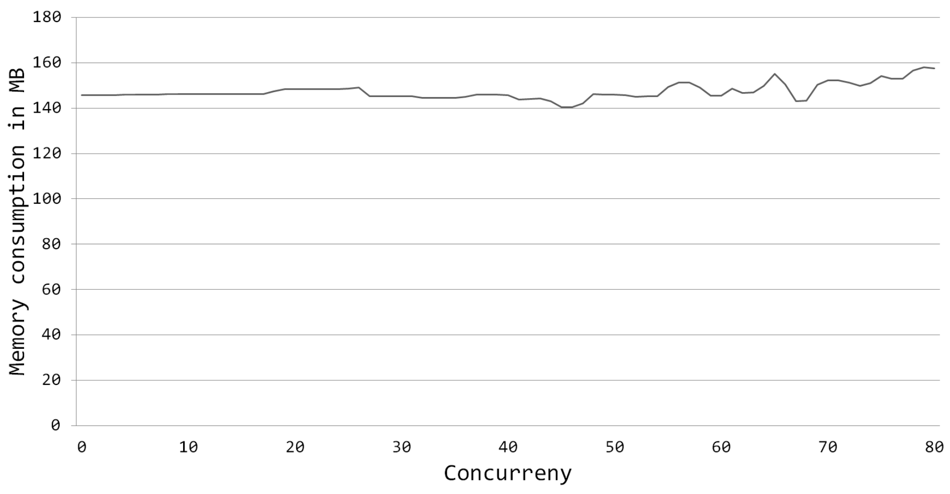


Figure 6.4: Memory consumption of the tested system.

be applicable. A full request including an authorization header with a 100 byte payload would be around 200 bytes depending on other set headers and length of the endpoint URL. This may not seem a lot but with a high amount of requests the limit is quickly reached if the servers are not very powerful. A solution for this bottleneck is to integrate the relay service into the existing push service to use the bandwidth of it and forward messages over the internal network.

6.3.4 Rerun Tests

The measured results only reflect one single tested server and can vary highly with different hardware. All tests can be found in the project added to this thesis or on Github [33]. Requirements for testing are a free Google Developer account, a server for running the relay prototype and computer to send the requests. For testing latency to the device an Android smartphone is needed. All tests are written in node.js and use the module “loadtest” [34] to create requests and “usage” [35] to monitor CPU load and memory consumption. To run these tests the relay prototype must be available on a remote server. This can be confirmed by opening the root location in a browser. If the message “Relay server running” appears in the browser it started successfully. Furthermore, all tests need some adoption of parameters as described in the according README files in the test directories.

Return Latency

For this test the relay service has to run with the flag “dryrun” to not actually send the messages to the tested device. The test can be started by entering the following commands:

```
cd tests/returnLatency
node index.js
```

. This will send requests to GCM and the relay service and save the results in the results.json file. Not only the mean latency of all requests but also detailed information about statistical percentiles can be found as well as the error rate of the requests.

Device Latency

This test is a more complex version of the simple return latency test and measures the time between sending the message from the application server and receiving the message on the device. In this case the “dryrun” flag on the relay service needs to be disabled so messages are sent.

Additionally to sending requests to the relay service and GCM directly the time has to be measured on the device as well as on the sending computer. Therefore they should be in sync as described in section 6.3.1. On Windows, Linux or MacOS these tasks can be accomplished rather easily, but on Android this takes root access. It is however also sufficient to sync them to roughly have the same time and normalize the data afterwards.

On the Android device a Cordova Phonegap application has to be installed which registers on GCM and prints the registration token. The sending computer can then be connected to the phone to inspect the application with Chrome device inspector to copy the registration token as well as the gathered data afterward. The copied token has to be inserted into the sending script to finish the setup process.

Once these steps are done the sending script can be started by entering the commands

```
cd tests/deviceLatency
node ./applicationServerTest/sendPush.js
```

. This will send 1000 request to the relay service and GCM directly and save the results in a local variable on the phone. This variable can then be copied with the device inspector to a file called results.json. To better sort the data a script called mapResults.js can be run which will store the mapped data in mappedResutls.json for further evaluation.

CPU and Memory

To measure CPU load and memory consumption the test is split into two different locations again. One part is the measurement of the CPU and memory on the relay server and the other one is sending the requests and storing how many concurrent requests were sent at a certain time. The “dryrun” flag should be turned off for this test because a high amount of requests (up to half a million) are sent to GCM.

First the index file must be started on the relay server to monitor the system and store the results in a file every minute. Then the local computer can start sending the requests be initiating the sendLoad script with

```
cd tests/cpuRam
node sendLoad.js
```

which results in logs appearing in the console informing about the status of the test. This test can take up to an hour to finish and once it is done the monitor script on the relay should not be stopped immediately but after the next automatic save of the file. This saved file with name results.json should then be copied to the local computer where it can be mapped with the timetable by running the map.js script to combine to CPU and memory data with the request intensity.

Bandwidth

Testing the bandwidth cannot be easily done in node.js and requires to use other tools to monitor the system. In Ubuntu for example the command “iftop” can be used to measure incoming data. On the local computer load is sent by calling

```
cd tests\bandwidth
node index.js
```

to the relay server. The test can be aborted when sufficient data was collected.

Language Comparison

As discussed in chapter 5.2 the programming language comparison shows a clear outcome. To rerun the benchmark all the tested languages (GO, Java, Node.js, PHP, Python and Ruby) need to be installed in the newest version. In the folder “tests languages” all implementations as well as the tools needed to run the test can be found. First the endpoint server needs to be started by running the command

```
cd tests\lanuages
node _endpoint\index
```

on the project root. Next the language specific server needs to be started by running the proper command in the language directory. For GO the command *go run index.go* is used while for all other languages the *run* keyword is removed. PHP has to be treated a bit differently because it requires an Apache server to run and to adopt the URL to send POST requests to in the file `tests\languages\load\_index.js`. Note that only one language can be tested at a time. To finally run the test enter the command

```
node tests\languages\_load\index
```

and the script will start to send requests. When finished the data is written in a file called “results.json”.

Chapter 7

Conclusion

As I started with the research what topic to write my thesis about, it was clear I wanted it to have something to do with new web technologies and new ways to create better mobile web experiences, because compared to native the web still lags behind on mobile devices. Not only statistics but also my personal experience showed this. As I was creating slides for a talk about good mobile websites and their native counterparts I realized that even I, a web developer, had dozens of apps installed that had really good web applications as well. I was asking myself about why this is a case and quickly I realized that notifications are a big part of the problems web applications have. So I tried to create my own push systems with websockets and native apps that would act as a push service, but got to the limits of the operating system or the browser context quite fast.

After further research I found out about the w3c push API and read everything I could find about it. At this time (August 2014) there was not a lot to find though, but I found out about the IETF webpush protocol and joined their working group. The problem I had with all this work was the slow progress. Not only the progress of the protocol itself, but also the insights that for example Apple has no people working on it. This showed me that it would take several years before this technology could be used for a broad audience and I wanted to do something about it. As part of my research and development of the prototype I participated in the 93rd IETF meeting in Prague and got to know the other people working on the standard and got some good feedback about my work.

The webpush protocol is a great idea and developers should have the chance to use it as quickly as possible. A relay service helps to speed up that process but browser producers still need to show commitment to even want to offer this API. The relay service makes most sense for browsers that do not have their own push service running in the OS and want to make the webpush API available although the used push service does not allow it. In this case there is no way to change the existing push service but a solution would be to offer a relay service that looks like it supports the protocol. The cost of the relay service would not be high as the performance evaluation shows. Probably the biggest problem with the relay service however is the lack of push receipts. In the actual implementation it should be evaluated again if it is possible to add this feature for given operating system and used push service and if the performance would decrease too much because of it. Also the webpush encryption mechanism should be implemented in a production environment which is missing in this prototype.

Appendix A

Contents of the CD-ROM/DVD

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 PDF Files

Pfad: /

Emathinger_Paul_2015.pdf	Master thesis with instructions (entire document)

Pfad: /OnlineSources

ECMA-262.pdf	[17]
DeviceArtGenerator.pdf	[18]
BlackBerryHistor.pdf	[19]
Android2.pdf	[20]
GCM.pdf	[21]
GCM-errors.pdf	[22]
apns.pdf	[23]
wns.pdf	[24]
wns2.pdf	[25]
sw.pdf	[26]
charter.pdf	[27]
webpush-demo.pdf	[28]
webpush-auth.pdf	[29]
cordova-inapp.pdf	[30]
ebay.pdf	[31]

hosteurope.pdf	[32]
gcm-relay.pdf	[33]
loadtest.pdf	[34]
usage.pdf	[35]
gcm-stats.mp4	[36]
walmart.mp4	[37]
channel9.mp4	[38]

A.2 Images

Pfad: /images

*.ai	Original Adobe Illustrator files
*.png	Rendered images

A.3 Source Code

Pfad: /source

relay.zip	Source code and tests of the developed prototype

References

Literature

[1] H. Alvestrand. *A Mission Statement for the IETF*. RFC 3935. Internet Engineering Task Force, 2004. URL: http://www.ietf.org/rfc/rfc3935.txt (cit. on p. 9).

[2] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. “Energy consumption in mobile phones”. In: *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference - IMC ’09* (2009), p. 14. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-84877730481%5C&partnerID=tZOtx3y1 (cit. on p. 7).

[3] M. Belshe, R. Peon, and M. Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. Internet Engineering Task Force, 2015. URL: http://www.ietf.org/rfc/rfc7540.txt (cit. on p. 18).

[4] S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. RFC 2119. Internet Engineering Task Force, 1997. URL: http://www.ietf.org/rfc/rfc2119.txt (cit. on p. 4).

[5] Andre Charland and Brian LeRoux. “Mobile Application Development: Web vs. Native”. In: *Communications of the ACM* 54.5 (2011), pp. 49–53 (cit. on p. 1).

[6] Eduardo Fullea and Bryan Sullivan. *Push API*. W3C Working Draft. W3C, Apr. 2015. URL: http://www.w3.org/TR/2015/WD-push-api-20150427/ (cit. on p. 16).

[7] Ian Hickson. “Web Workers - W3C Candidate Recommendation 01 May 2012”. 2012. URL: http://www.w3.org/TR/workers/ (cit. on p. 9).

[8] Mehdi Jazayeri and Navid Ahmadi. “End-user programming of web-native interactive applications”. In: *Proceedings of the 12th International Conference on Computer Systems and Technologies - CompSysTech ’11* (2011), pp. 11–16 (cit. on p. 1).

[9] M. Nottingham. *Web Linking*. RFC 5988. Internet Engineering Task Force, 2010. URL: http://www.ietf.org/rfc/rfc5988.txt (cit. on p. 18).

[10] Arno Puder, Nikolai Tillmann, and Michał Moskal. “Exposing native device APIs to web apps”. In: *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems - MOBILESoft 2014* (2014), pp. 18–26 (cit. on p. 29).

[11] E Rescorla. *HTTP Over TLS*. RFC 2818. Internet Engineering Task Force, 2000. URL: http://www.ietf.org/rfc/rfc2818.txt (cit. on p. 19).

[12] Alex Russel and Jungkee Song. “Service Workers - W3C Working Draft 18 November 2014”. 2014. URL: http://www.w3.org/TR/service-workers/ (cit. on p. 9).

[13] Jeni Tennison. *Good Practices for Capability URLs*. W3C Working Draft. W3C, Feb. 2014. URL: http://www.w3.org/TR/2014/WD-capability-urls-20140218/ (cit. on p. 20).

[14] M. Thomson, E. Damaggio, and B. Raymor. “Generic Event Delivery Using HTTP Push”. 2015. URL: https://tools.ietf.org/html/draft-ietf-webpush-protocol-00 (cit. on pp. 4, 18).

[15] Stefan Tilkov and Steve Vinoski. “Node.js: Using JavaScript to build high-performance network programs”. In: *IEEE Internet Computing* 14.6 (2010), pp. 80–83 (cit. on p. 33).

[16] Chad Tossell et al. “Characterizing web use on smartphones”. In: *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems - CHI ’12* (2012), pp. 2769–2778 (cit. on p. 1).

Online sources

[17] URL: http://www.ecma-international.org/ecma-262/6.0/ (visited on 09/09/2015) (cit. on pp. 4, 48).

[18] URL: https://developer.android.com/distribute/tools/promote/device-art.html (visited on 09/09/2015) (cit. on pp. 8, 17, 30, 48).

[19] URL: http://www.bbscnw.com/a-short-history-of-the-blackberry.php (visited on 08/07/2015) (cit. on pp. 10, 48).

[20] URL: http://developer.android.com/about/versions/android-2.2-highlights.html (visited on 09/09/2015) (cit. on pp. 10, 48).

[21] URL: https://developers.google.com/cloud-messaging/server (visited on 09/09/2015) (cit. on pp. 10, 48).

[22] URL: https://developers.google.com/cloud-messaging/server-ref#error-codes (visited on 09/09/2015) (cit. on pp. 11, 48).

[23] URL: https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/CommunicatingWIthAPS.html (visited on 09/09/2015) (cit. on pp. 13, 48).

[24] URL: https://msdn.microsoft.com/en-us/library/windows/apps/hh913756.aspx (visited on 09/09/2015) (cit. on pp. 13, 48).

[25] URL: https://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh868245.aspx (visited on 09/09/2015) (cit. on pp. 14, 48).

[26] URL: https://jakearchibald.github.io/isserviceworkerready/ (visited on 09/09/2015) (cit. on pp. 17, 48).

[27] URL: https://datatracker.ietf.org/doc/charter-ietf-webpush/ (visited on 09/09/2015) (cit. on pp. 18, 48).

[28] URL: https://docs.google.com/document/d/1kDVLMddPJL6iHLJ6QuqNFc1D5X9rASx0PfDd1IlxUE4 (visited on 09/09/2015) (cit. on pp. 20, 48).

[29] URL: https://github.com/webpush-wg/webpush-protocol/issues/44 (visited on 09/09/2015) (cit. on pp. 20, 48).

[30] URL: https://github.com/paul-em/cordova-plugin-inappbrowser (visited on 09/09/2015) (cit. on pp. 29, 48).

[31] URL: http://www.ebaytechblog.com/2013/05/17/how-we-built-ebays-first-node-js-application/ (visited on 09/09/2015) (cit. on pp. 33, 48).

[32] URL: https://www.hosteurope.de/en/Server/Virtual-Server/Advanced/ (visited on 09/09/2015) (cit. on pp. 39, 49).

[33] URL: https://github.com/paul-em/push-relay-gcm (visited on 09/09/2015) (cit. on pp. 42, 49).

[34] URL: https://github.com/alexfernandez/loadtest (visited on 09/09/2015) (cit. on pp. 42, 49).

[35] URL: https://github.com/arunoda/node-usage (visited on 09/09/2015) (cit. on pp. 42, 49).

[36] *Google I/O 2015 - Google Cloud Messaging 3.0.* 2015. URL: https://www.youtube.com/watch?v=gJatfdattno (cit. on pp. 5, 22, 49).

[37] *Node.js at Walmart: Introduction.* 2013. URL: https://www.joyent.com/developers/videos/node-js-at-walmart-introduction (cit. on pp. 33, 49).

[38] *Notification Platform Development on Windows.* 2014. URL: https://channel9.msdn.com/Events/Build/2014/2-521 (cit. on pp. 13, 49).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —

Breite = 100 mm
Höhe = 50 mm

— Diese Seite nach dem Druck entfernen! —