Technische Universität München

Department of Informatics

Master's Thesis in Informatics

Design and Implementation of a Secure and Privacy-Preserving Notification Service for Mobile Platforms

Linus Lotz

Technische Universität München

Department of Informatics

Master's Thesis in Informatics

Design and Implementation of a Secure and Privacy-Preserving Notification Service for Mobile Platforms

Entwurf und Implementierung eines sicheren und Privatsphäre wahrenden Benachrichtigungsdienstes für mobile Platformen

Author	Linus Lotz
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Dr. Matthias Wachs
Date	November 15, 2016

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, November 15, 2016

Signature

Abstract

Push services are becoming more prevalent by expanding from the mobile platforms into the desktop operating systems and web browsers. Furthermore, the users are forced to use the push service of their platform, as these services are deeply integrated into the operating systems of modern mobile platforms. A push service provider has access to the unencrypted contents of the push messages and their metadata, as all messages need to pass through the provider. This allows a deep invasion of the user's privacy. The goal of this thesis is to provide an analysis of existing push services, show their impact on privacy and present a design and realization of a secure and privacy-preserving push service. In this thesis, three push services on mobile platforms are examined and analyzed for their security and privacy properties. To overcome the problems discovered in the analysis, a design for a secure and privacy-preserving push service is developed and implemented. We show that with this approach privacy is better protected and the impact on resource consumption and performance is negligible compared to existing push services.

Zusammenfassung

Pushdienste sind auf dem Vormarsch und verbreiten sich von den Mobilplatformen auf Desktopbetriebsysteme und Webbrowser. Nutzer sind an den Pushdienst ihrer Plattform gebunden. Dieser hat Zugriff auf die Metadaten und unverschlüsselten Inhalte ihrer Pushnachrichten. Dies ermöglicht einen tiefen Eingriff in die Privatsphäre des Nutzers. Ziel dieser Arbeit ist, existierende Pushdienste auf ihre Auswirkung auf die Privatsphäre zu untersuchen, sowie ein Design und Realisierung eines sicheren und Privatsphäre schützenden Pushdienstes vorzustellen. In dieser Arbeit werden drei existierende Pushdienste auf ihre Sicherheit und Privatsphäre schützenden Eigenschaften analysiert. Auf Basis dieser Analyse und der dabei gefundenen Probleme wurde ein Design für einen sicheren und Privatsphäre schützenden Pushdienst entwickelt. Wir zeigen, dass dieses Design die Privatsphäre besser schützt und der Einfluss auf Performance und Ressourcenverbrauch, im Vergleich zu existierenden Pushdiensten, vernachlässigbar ist.

Contents

1 Introduction 1
 1.1 Goals of the Thesis 2
 1.2 Outline 3

2 Background 5
 2.1 HTTP 2.0 5
 2.2 The Onion Router (Tor) 6

3 Mobile Push Services 9
 3.1 Differences between Push and Pull for Notifications 9
 3.2 Overview 9
  3.2.1 Use cases for Notifications 10
  3.2.2 Requirements for Mobile Push Services 10
 3.3 Architecture of Mobile Push Services 10
  3.3.1 Components 10
  3.3.2 Functionality 11
  3.3.3 Communication Life Cycle 11
 3.4 Nomenclature 12
 3.5 Google Cloud Messaging 13
  3.5.1 App Registration 13
  3.5.2 Sender Registration 13
  3.5.3 Message Submission 13
  3.5.4 Message Delivery 14
 3.6 Apple Push Notification Service 14
  3.6.1 App Registration 15
  3.6.2 Sender Registration 15
  3.6.3 Message Submission 15
  3.6.4 Message Delivery 16
  3.6.5 Privacy Concerns with Client Certificates 16
 3.7 Windows Notification Service 18
  3.7.1 App Registration 18
  3.7.2 Sender Registration 18

3.7.3 Message Submission 18
3.7.4 Life Cycle of the WNS Push Service 18
3.7.5 Notification Types 19
3.8 Comparison of the Presented Push Services 20
3.9 Summary 21

4 Security and Privacy Analysis 23
4.1 Information Model 23
4.1.1 Push Message 23
4.1.2 Push Connection 23
4.1.3 Sending a Push Message 24
4.2 Adversary and Threat Model 24
4.3 Definitions 25
4.3.1 Secure 25
4.3.2 Privacy-Preserving 25
4.4 Approaches for Reducing Information Leakage 26
4.5 Encryption in Mobile Push Services 27
4.6 Criteria for the Analysis of the Push Services 28
4.7 Comparison of the Presented Push Services 28
4.8 Summary 29

5 Related Work 31
5.1 Webpush 31
5.2 A Universal Push Service 32
5.3 Anonotify 32
5.4 Analyzing Locality of Mobile Messaging Traffic 33
5.5 Summary 33

6 Design of a Secure and Privacy-Preserving Notification Service 35
6.1 Design Goals 35
6.1.1 Achieving Security 35
6.1.2 Achieving Privacy 36
6.1.3 Achieving User Flexibility 36
6.1.4 Achieving Scalability 37
6.1.5 Providing Transport Security 37
6.1.6 Providing End-to-end Security 37
6.1.7 Providing Sender Anonymity 39
6.1.8 Summary 40
6.2 Protocol Description 41
6.2.1 Registration of the Push Client 41
6.2.2 Registration of the App 42
6.2.3 Sending a Push Message 43

6.2.4 Retrieving Push Messages 43
6.2.5 Revoking Push Access 44
6.3 Components 44
6.3.1 Push Server 44
6.3.2 Push Client 45
6.3.3 App 46
6.3.4 Push Sender 46
6.4 Protocol between App and Push Sender 47
6.5 Message Format 47
6.6 Limitations 48

7 Implementation 49
7.1 Design Decisions 49
7.1.1 Architecture 49
7.1.2 Tokens 49
7.2 Push Server 49
7.2.1 Architecture 50
7.2.2 Security 51
7.2.3 Persistent Storage 54
7.2.4 Tor Onion Service 54
7.3 Push Client 54
7.3.1 Why Android 54
7.3.2 Keeping the Connection Alive 55
7.3.3 Persistent Storage 55
7.4 Push Sender 57
7.5 End-to-end Encryption 59

8 Evaluation 61
8.1 Security Analysis 61
8.1.1 Security 61
8.1.2 Privacy 62
8.2 Performance Analysis of the Push Service 62
8.2.1 Properties to be Analyzed 62
8.2.2 Experimental Setup 63
8.2.3 Battery Consumption 67
8.2.4 Latency Measurements 67
8.2.5 Battery Consumption Measurements 67
8.3 Summary 71

9 Future Work 73
9.1 Alternative Architecture 73
9.2 Improvements to Tor 75

9.2.1 Single Onion Services 75
9.2.2 Run Own Relays 75
9.2.3 Onion Balance 75
9.3 Keep-Alive and Heartbeats 75
9.4 Certificate Pinning in the Client 76
9.5 Aggregation of Push Messages 76

10 Conclusion 77

List of Figures

2.1 A connection over the Tor network 7
2.2 Connecting to an onion service 8

3.1 Life cycle of a push service 12
3.2 Google Cloud Messaging: Instance ID acquisition 14
3.3 The generation of a token for an app in APNs 15
3.4 The verification of a token in APNs 16
3.5 Authentication between push sender (Provider) and APNs 17
3.6 Authentication between device and APNs 17
3.7 Sending push messages with WNS 19

6.1 End-to-end security in Sense 38
6.2 Architecture of Sense 41
6.3 Registration and push message sending in Sense 42

7.1 Architecture of the push server 50
7.2 The TLS state machine of the push server 52
7.3 The HTTP2 state machine of the push server 53
7.4 The database scheme used in the Sense implementation 56
7.5 Architecture of the push sender 59

8.1 The beaglebone with the GPS module used as timesource for the mobile device 64
8.2 Latency test setup 66
8.3 The results of the latency tests 68
8.4 Histogram of messages per hour 69
8.5 Impact of Tor on latency 70
8.6 Battery consumption results of sending push messages for 24 hours 71

9.1 Alternative architecture for the alternative design 74

List of Tables

3.1 Comparison between the different push services 21

4.1 Encryption in different push services 27
4.2 Comparison of the security and privacy between the different push services 29

7.1 The different supported queries for the push sender 58

8.1 Security analysis of SENSE 62

Chapter 1

Introduction

A *Notification Service* allows sending notifications to a user. On the mobile platforms it is common for the operating system to establish a connection to a notification service. These notification services are called push services since they allow the service provider of an app to "push" notifications to their app. These push services were introduced to help saving two valuable resources of mobile platforms: power and network usage.

Without push services, an app that wants to receive notifications would need to establish a permanent connection to its server. This requires these apps to run continuously and, should the device lose internet connection, require the apps to reestablish their connections.

Additionally, those platforms impose restrictions on the resource usage and will try to stop apps from running permanently in the background. For example, with Android *Marshmallow*, Google introduced a *Doze* mode, in which apps are stopped and their internet access is denied to extend the battery life of the device. This means developers are forced to use the push service of the platform if they want to send messages to the app. On iOS, only apps that classify themselves as VoIP apps are able to open persistent connections.

These push services are also tightly integrated into the mobile platform, so the user cannot simply switch the push service. Furthermore, these push service providers also maintain an app store. To use these stores a registration is required (e.g. for Microsoft Store, Apple Store, and Google Play Store), even when downloading free apps. This registration is usually done using an e-mail address and may even contain credit card information, a real name and an address when a non-free app is purchased. The push service provider can then trivially link push notifications to real identities.

Furthermore, push services have begun to come to the desktop as well. Apple's push service Apple Push Notification Service (APNs) is integrated with the newer versions of macOS, and with the establishment of Webpush — a notification service for web pages, modern browsers will also use push services to deliver notifications to the user.

Since the client always tries to open the connection to the push service, the push service knows the IP address of each client. Also, the developers of the apps often have to register and authenticate with the push service. Combined with the metadata of each push message, this allows the push service to gather information about app usage, user location, and schedules (using the IP address of the mobile device).

The collection of metadata is an explosive issue, as metadata allows far-reaching conclusions, since it was shown that even trained professionals like CIA operatives can be uncovered with metadata [1]. Metadata is also used by intelligence agencies to select people for assassination as was stated by General Michael Hayden, a former director of the NSA and CIA: "We kill people based on metadata" [2].

Even services that provide end-to-end security can reveal a lot of information with metadata. A prominent example is Apple's iMessage, which is end-to-end encrypted but gathers logs of look-ups for iMessage users [3]. These logs are given to the police, when a warrant is provided. The possibility that these logs are also shared with intelligence services cannot be ruled out, especially since Apple was one of the companies in the PRISM program [4].

It is obvious that work in reducing the metadata of push services is needed as these services become more common and thus the amount of information that can be gained with metadata increases.

This thesis presents a design and implementation of a privacy-preserving and secure push service that reduces the amount of metadata visible to the push service.

1.1 Goals of the Thesis

In this thesis, we want to build a privacy-preserving and secure push service. To do so we need to answer the following questions:

- What are push services and what does the architecture of a push service look like?
- Which push services do already exist?
- What makes a push service privacy-preserving and secure?
- Do the existing push services fulfill these properties?
- Is there related work in this field?

- What does the design of a privacy-preserving and secure push service look like?
- What does an implementation of this design look like?
- How does this implementation compare to existing push services?
- How can this design be improved further?

1.2 Outline

First, in Chapter 2, a brief introduction into HTTP2 and Tor is provided. Following, in Chapter 3, we describe what push services are and what their architecture looks like. Three existing mobile push services are presented. In Chapter 4, the properties "secure", and "privacy-preserving" are defined as well as the attacker and threat model. The push services presented in Chapter 3 are then analyzed for their security and privacy. In Chapter 5, we take a look at related work. Following, in Chapter 6, the design for the privacy-preserving and secure notification service is presented. The implementation of this design is presented in Chapter 7. This implementation is then compared to an existing push service in Chapter 8. As an outlook, future work is discussed in Chapter 9. Finally, we conclude in Chapter 10.

Chapter 2

Background

In this chapter, we will take a look at the HTTP 2.0 standard, especially with respect to its changes compared to HTTP 1.0 and 1.1. We will also take a look at The Onion Router (Tor), an anonymisation network. Both will be used later in the privacy-preserving push service.

2.1 HTTP 2.0

The *HyperText Transport Protocol* (*HTTP*) is used to serve websites. It was introduced by Tim Berners-Lee at CERN around 1989[1]. HTTP 1.0 [5] would close the connection after a successful request. This is inefficient when a client wants to send multiple requests. This was changed with HTTP 1.1[2] where the connection could be left open for subsequent requests, but a client needs to wait for the request to finish before sending the next request. To combat this problem HTTP Pipelining[3] was introduced. HTTP Pipelining makes it possible to send multiple requests without having to wait for the responses, but the requests would still be answered in the same order as requested. If a requested resource would block, all following resources were blocked as well. It could still improve performance, because a server that supported Pipelining could process the next request, while sending the data to the client. To circumvent the blocking, browsers would establish multiple connections, but this allocates more resources on the server.

HTTP 2.0 [11] is a new standard by the Internet Engineering Task Force (IETF). It was developed to circumvent the shortcomings of HTTP 1.1. In HTTP 2.0 messages between client and server are exchanged in “frames”. When a request is

[1]`http://webfoundation.org/about/vision/history-of-the-web/`

[2]Originally defined in [6], obsoleted by [7], [8], [9], [10].

[3]Pipelining was defined as part of RFC 2616 [6], Section 8.1.2.2.

started, it is assigned a stream number which is referenced by frames belonging to that stream. This allows multiplexing multiple transmissions into one connection. As header data is often redundant and very repetitive (like the `Host` header) a compression for header data was introduced.

With HTTP 2.0 it is also possible for the server to push resources to the client, without needing a request from the client. Such pushed resources can be initiated if the server knows that a client will request the resource later, after processing the original request. These push messages will be used in our push protocol (Chapter 6) to deliver messages to the client.

2.2 The Onion Router (Tor)

Tor [12] is a low-latency anonymization network. Tor uses an approach called onion routing, where the data is encrypted with several layers of encryption, so that each router on the path can only decrypt the outer layer and thus reveal the next router on the path. This has the benefit that a router on the path only knows the previous and the next router, which hides the destination from all routers except the last router and the origin from all routers except the first router. In contrast to proxies, where the operator of the proxy can see source and destination, onion routing tries to prevent one party from knowing both.

Tor works as follows: The Tor client establishes so called *circuits*. These circuits are routes over the Tor network using three *relays* (relay is the modern term for the onion routers in the paper[4]), which are nodes in the Tor network. The traffic going over these circuits is encrypted multiple times, so that each node can only decrypt the outer layer and send it to the next node.

The first node knows the IP address of the client and the second node in the circuit. The second node only knows the first node and the third node. Finally the last node, also called an *exit node*, knows only the second node and the destination IP and port, as well as all the traffic passing through. This can be seen in Fig. 2.1.

It is also possible to run Tor *hidden services* or *onion services* as they are now called[5]. Onion services allow to host web sites or other services anonymously and can only be accessed using Tor. This makes the service more resilient against censorship and hides the real IP address of the service. Since access to onion services is only possible over Tor, the client IP address is not visible to the onion service.

The procedure of connecting to an onion service can be seen in Fig. 2.2. These onion services work by building circuits to multiple relays and making them

[4] `https://www.torproject.org/about/overview.html.en`

[5] `https://media.ccc.de/v/32c3-7322-tor_onion_services_more_useful_than_you_think`

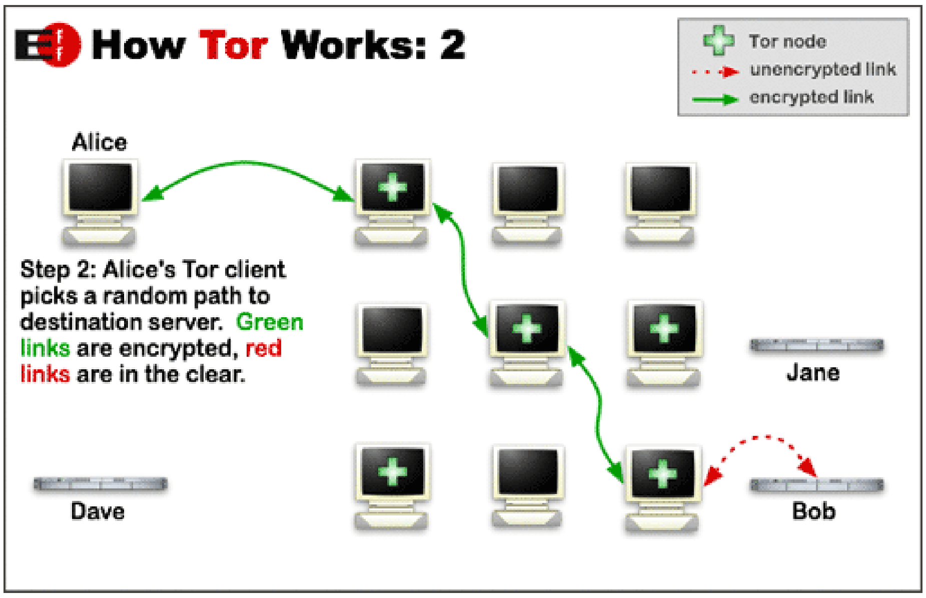
How Tor Works: 2
Tor node
unencrypted link
encrypted link
Alice
Step 2: Alice's Tor client picks a random path to destination server. Green links are encrypted, red links are in the clear.
Jane
Dave
Bob

Figure 2.1: A connection over the Tor network [13]

introduction points (step 1). These introduction points are then stored in a directory service (step 2). A client wanting to connect to the onion service asks the directory service for the introduction points (step 3) and establishes a relay as rendezvous point in the network (step 4). It then sends the information about this rendezvous point to one introduction point (steps 5,6). The client then waits for the onion service to establish a circuit to the rendezvous point (steps 7,8). The client can now communicate over the rendezvous point with the onion service (step 9). These connections are end-to-end encrypted (the endpoints establish a circuit from end-to-end). The client can verify the server using the onion host name, which is derived from the public key of the onion service.[6]

[6]https://www.torproject.org/docs/hidden-services.html.en

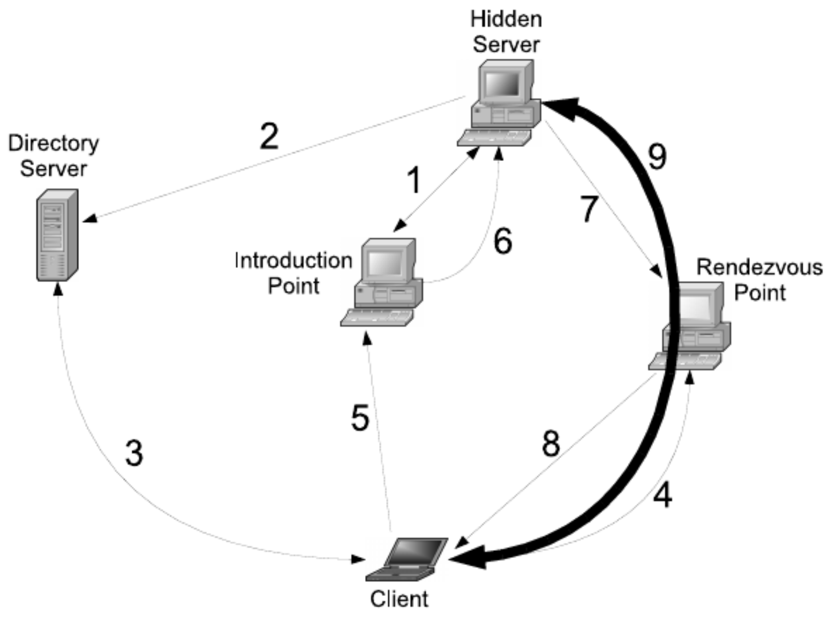

Figure 2.2: Connecting to an onion service [14].

Chapter 3

Mobile Push Services

In this chapter, we have a look at what push services are and how their architecture looks like. With this knowledge, we will examine three push services and their architecture.

3.1 Differences between Push and Pull for Notifications

Push and pull are different approaches to deliver notifications. When using pull notifications, the party interested in the information will regularly check for new information. This is usually done in predefined intervals, but can also be initiated when the user explicitly checks for new information or when the device is woken up from standby.

With push notifications, a different approach is used. The server can reach the client at any time, e.g. using a persistent connection or sending UDP packets to the client. As soon as an event occurs, the information will be sent from the server to the client. Push notifications can be faster than pull notifications, since the client does not need to ask the server for updates to get the new information.

Boonkrong and Dinh [15] compared the battery consumption of push and pull notifications and found that push was more efficient.

3.2 Overview

In this section we examine the use cases and the challenges that arise for push services on mobile devices.

3.2.1 Use cases for Notifications

There are several reasons for wanting notifications in mobile platforms: They could be used by the mobile operating system to get notifications about software updates. Third party apps could use them to notify the user about incoming messages, events in social media, parcel delivery status updates, or news.

3.2.2 Requirements for Mobile Push Services

There are several challenges that face notification systems for mobile devices. First, mobile devices change their IP address relatively often compared to desktop computers with a fixed Internet connection. Such changes occur, for example, when the device switches from mobile data to wireless LAN.

Secondly, the device will be mostly behind a router with Network Address Translation (NAT), either the carrier-grade NAT which is commonplace in mobile networks [16–18] or a wireless LAN with NAT. These restrictions make it difficult for the push service provider to connect to the mobile device directly (e.g. using UDP for notifications). To ensure a timely push message delivery the device should try to have an open connection to the push service provider when an Internet connection is available.

Finally, the push service should be resource-efficient. Mobile devices have limited battery power and data usage in mobile networks is often limited as well. A push service for mobile platforms needs to be battery efficient and keep data usage minimal.

3.3 Architecture of Mobile Push Services

Before analyzing existing mobile push services, it is important to understand the general architecture of these push services, their components, and how these components interact.

3.3.1 Components

There are four parties involved in a mobile push service. First, there is the push client, running on the mobile device of the end user. This push client receives the notifications from the second party, the push server. The third party is the app that wants to get notifications for the user. The app developer runs a server, to which the app connects and from which the developer wants to send notifications to the app. This server is the push sender, which is the fourth party.

3.3.2 Functionality

The different components in the push service have different functionalities, which together enable the delivery of push messages.

3.3.2.1 App registration

An app that wants to receive push messages needs to register with the push client on the device. Depending on the push service, the push client can either directly give the app a registration or needs to retrieve it from the push server. This registration needs to be passed to the push sender.

3.3.2.2 Sender Registration

Some push services require a registration of the push sender. The push service provider gives the push sender some means of authentication. A part of this authentication might need to be included with the app so it can pass it to the push client when registering.

3.3.2.3 Message Submission

When the push sender wants to send a push message to the app, it uses the registration information received to authenticate itself with the push server. Some push services allow to set a priority or a time to live for the push message or allow to replace another push message which has not been delivered yet.

3.3.2.4 Message Delivery

If the client has a connection open to the push server, the push server can use this connection to deliver the push message. If no connection is established, the push server waits until the push client connects or the time to live of the push message expires and the push message is deleted. The push server might also aggregate push messages to reduce the time the radio of the mobile device needs to be active and the device awake.

3.3.3 Communication Life Cycle

The life cycle of a mobile push service can be seen in Fig. 3.1. First, the app registers itself with the push client on the device (step 1) and gets a registration (step 2) to allow the push sender of the app to send notifications to the app. This registration is passed by the app to the push sender (step 3). When the push

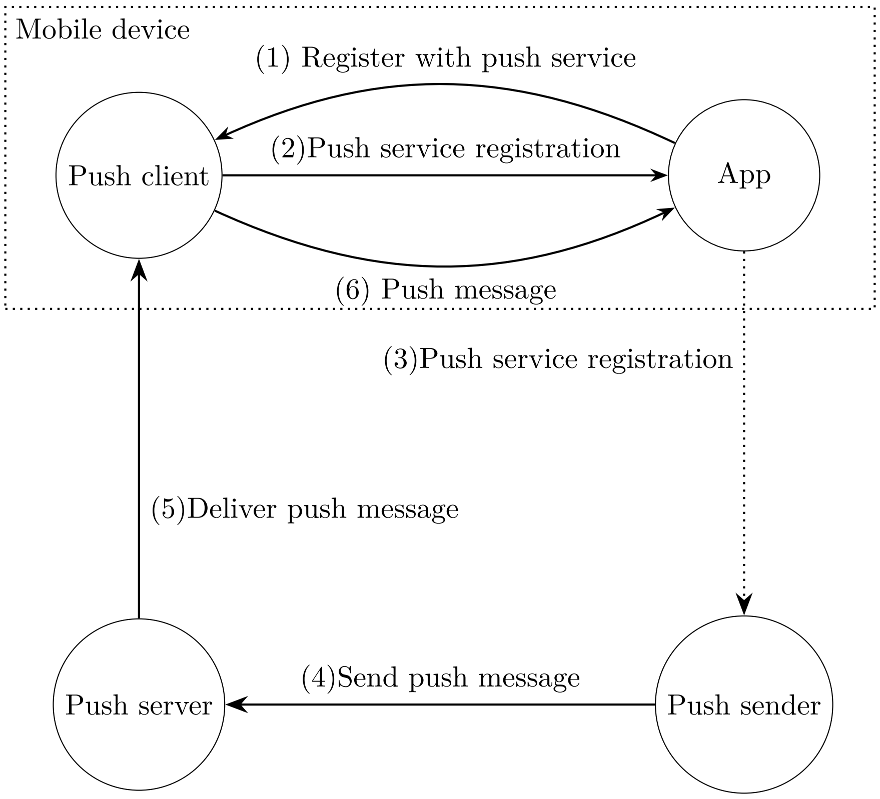

Figure 3.1: Life cycle of a push service. The connection between app and push sender is not part of the push service.

sender wants to send a push message to the device it uses the push registration data given to it to connect to the push server (step 4). As soon as the push server receives the push message, it will try to deliver it. If there is an open connection to the push server from the push client, the push server will send the message directly (step 5). Otherwise the push server will wait until the push client opens a connection to the push server and then deliver it. On the mobile device the message is then given to the app (step 6).

3.4 Nomenclature

As there is no predefined nomenclature, different push services have defined their own names for these four parties. When describing other approaches the corresponding name of this nomenclature will be provided. This naming schema was chosen to describe push notification systems independently of their use case (mobile, desktop, etc.). For example, current websites use websockets to provide notifications, in which case the push server and push sender is the server providing the websocket and the push client is the JavaScript application running in the web browser.

3.5 Google Cloud Messaging

Google Cloud Messaging (GCM) is a push service developed by Google for Android devices. It replaced Cloud to Device Messaging (C2DM) as the primary push service for Android and is now in the process of being replaced by Firebase Cloud Messaging (FCM)[1].

3.5.1 App Registration

The registration process is described in its documentation by Google (see literature [19–21]). An app that wants to receive push messages needs to acquire an *Instance ID*. The process can be seen in Fig. 3.2. This Instance ID can be used to generate security tokens that grant access to the push service. The Instance ID can also be used to verify the name of the app and whether its signature is valid (provided the app is distributed using Google Play). Additionally, the Instance ID Server can be queried to tell "when the device on which your app is installed was last used" [21].

3.5.2 Sender Registration

To send messages, a "project" needs to be registered in the Google Developer Console. This enables access to two identifiers: The Sender ID and the API key. The Sender ID is included in the app and sent with the request for an Instance ID. The API key is used to authenticate the push sender to the GCM servers.

3.5.3 Message Submission

To send a push message, the push sender needs to transmit the API Key and the Instance ID of the targeted device along with the push message. Then the servers verify that the Sender ID associated with the Instance ID matches the API Key sent with the push message.

GCM distinguishes between two types of payload for the push messages: notification and data [22]. With the notification payload the GCM client on the device displays the notification on behalf of the targeted app. On Android the data payload is delivered to the app over an *Intent* — an IPC call on Android. It is

[1] `https://developers.googleblog.com/2016/05/google-cloud-messaging-and-firebase.html`

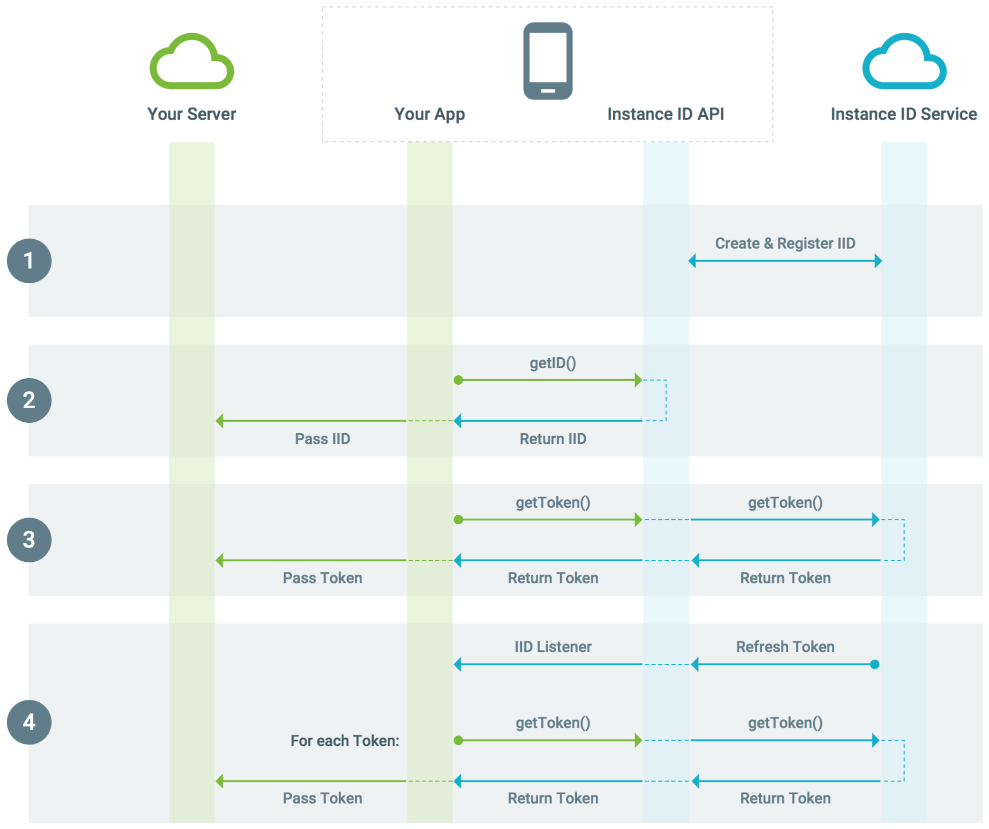

Figure 3.2: The process of acquiring an *Instance ID* for Google Cloud Messaging [21]

possible to send both payloads in one push message. The notification payload has a size limit of 2KB and the data payload a limit of 4KB.

3.5.4 Message Delivery

Notifications in GCM can have a priority and a time to live, which are specified when sending the push message. High priority push messages can wake the device, while normal priority messages are delivered, when the device exits idle. GCM can send push messages to Android, Chrome and iOS.

3.6 Apple Push Notification Service

Apple describes their push service in [23]. It was first announced in 2008 [24] and released in 2009 [25] with iOS 3.0. It is now part of “iOS (and, indirectly, watchOS), tvOS, and macOS” [23].

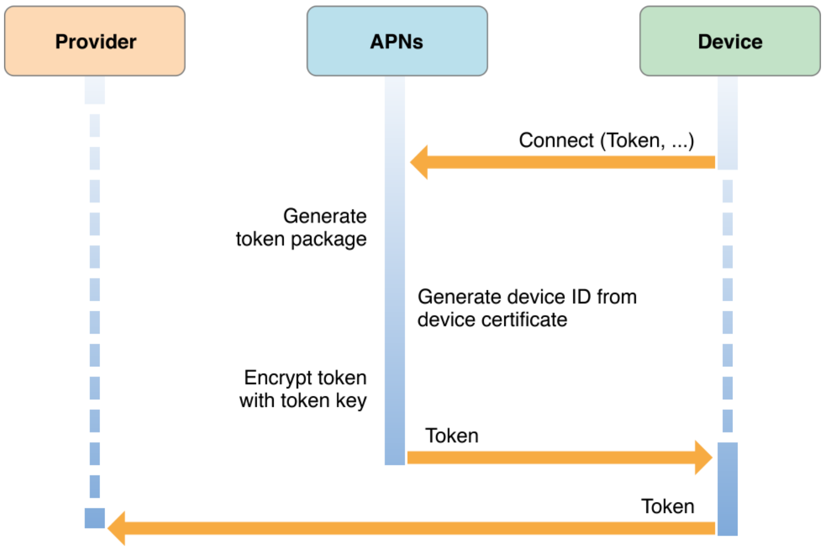
Provider
APNs
Device
Connect (Token, ...)
Generate token package
Generate device ID from device certificate
Encrypt token with token key
Token
Token

Figure 3.3: The generation of a token for an app in APNs [23]

3.6.1 App Registration

When an app wants to register to the push service, the request is forwarded to the APNs server. The server then generates a device token for this app using information from the device certificate (Fig. 3.3). This token is encrypted and sent back to the device. The app can then send this token to the push sender.

3.6.2 Sender Registration

The developer needs to request a certificate from Apple for push messages. A developer license is required to request the certificate. This certificate is used to authenticate the provider to the APNs using TLS client certificate authentication (Fig. 3.4).

3.6.3 Message Submission

Using the certificate together with the device token allows the APNs server to authenticate the push sender and verify that the push sender is allowed to send messages to the device (Fig. 3.4). The decrypted device token contains the *Device ID* which is used to deliver the message.

Until iOS 9, the payload was limited to 256 bytes. The limit has been raised to 4KB when using the HTTP2 API and 2KB when using the deprecated binary interface. VoIP applications can receive notifications with 5KB payload over the

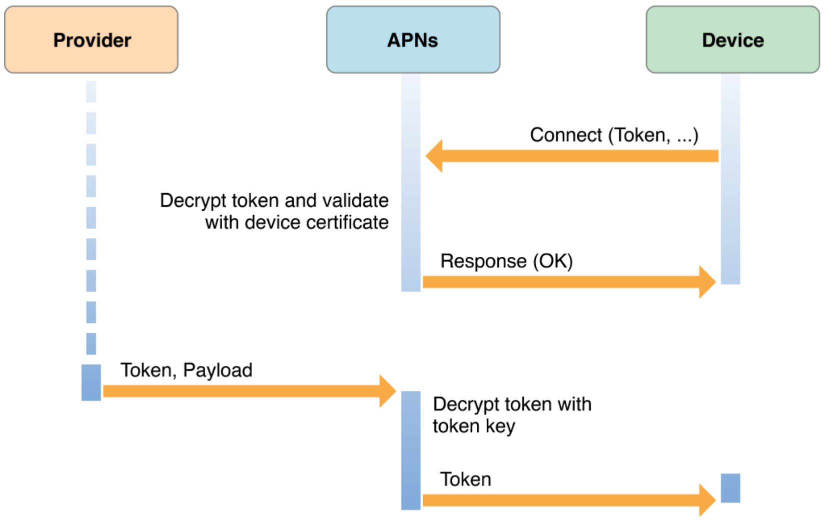
Provider
APNs
Device
Connect (Token, ...)
Decrypt token and validate with device certificate
Response (OK)
Token, Payload
Decrypt token with token key
Token

Figure 3.4: The verification of a token in APNs [23]

HTTP2 interface[2]

3.6.4 Message Delivery

Apple Push Notification Service (APNs) also uses TLS peer certificate authentication to authenticate the device (Fig. 3.6) to APNs. Connections can be established over TCP port 443 or 5223.

3.6.5 Privacy Concerns with Client Certificates

Client certificates are transmitted without encryption and are thus visible to an eavesdropper. The certificate for the device is generated when the device is activated ("The device obtains its certificate and key at device activation time and stores them in the keychain." [23]). As every device has an unique certificate this could enable an attacker that monitors all connections to APNs to track the public IP of every iPhone. The problems with TLS client certificates is also discussed in [26].

[2]`https://developer.apple.com/library/prerelease/content/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/CreatingtheNotificationPayload.html`

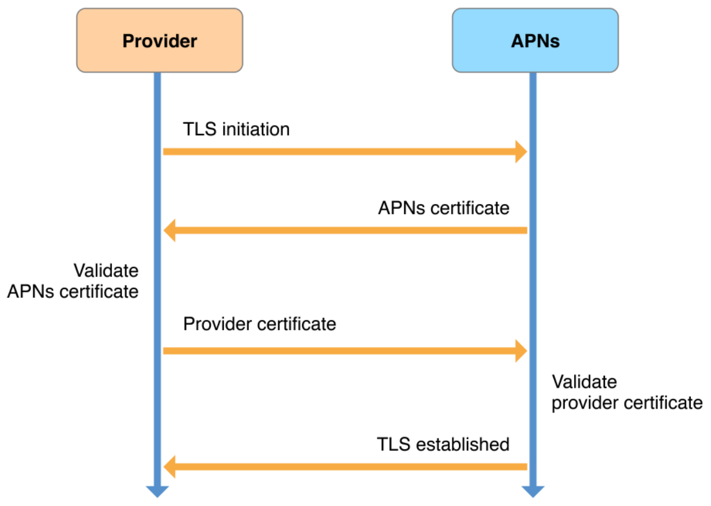

Figure 3.5: Authentication between push sender (Provider) and APNs [23]

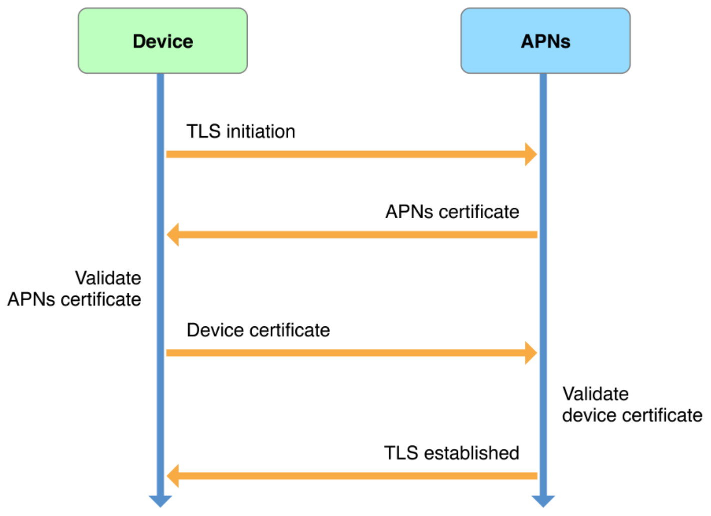

Figure 3.6: Authentication between device and APNs [23]

3.7 Windows Notification Service

The Windows Notification Service (WNS) is the push service for the Universal Windows Platform (UWP). It can be used to send push messages to apps running on Windows 8, 8.1, 10, 10 Mobile and Windows Phone 8.1.[3]

WNS is the successor to the Microsoft Push Notification Service (MPNS). The architecture of WNS is described by Jacobs [27] and Microsoft [28]. It follows the design described in Section 3.3.

3.7.1 App Registration

An app requests a *Notification Channel* from the *Notification Client Platform*. The Notification Client Platform forwards this request to the WNS. WNS then returns a *channel URI* to the Notification Client Platform. The Notification Client Platform then forwards the channel URI to the app.

3.7.2 Sender Registration

To send push messages, the developer needs to register the app with the *Windows Store Dashboard*. The developer is then given credentials to authenticate with WNS. These credentials consist of a *Package security identifier (SID)* and a secret key. The authentication with WNS uses these credentials and utilizes the client credentials profile of OAUTH 2.0 to retrieve an authentication token. The retrieved token only has a limited lifetime after which a new token has to be requested for new messages.

3.7.3 Message Submission

The push sender sends the push message over the channel URI and authenticates using the authentication token. If the token is no longer valid, he has to acquire a new token.

3.7.4 Life Cycle of the WNS Push Service

The process of an app requesting push messages, and retrieving messages can be seen in Fig. 3.7. An app that wants to receive notifications requests a Notification Channel (Step 1) from the Notification Client Platform. The Notification Client Platform requests this channel from WNS and is given the channel URI (Step 2). This channel URI is forwarded to the app (Step 3). The app informs its *Cloud*

[3] `https://en.wikipedia.org/wiki/Windows_Push_Notification_Service`

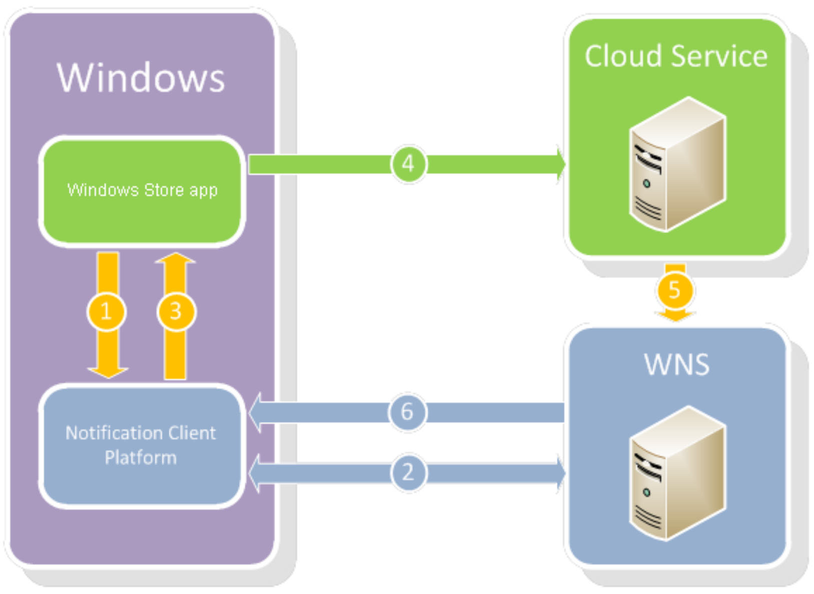

Figure 3.7: Sending push messages with WNS Microsoft [28]

Service (the push sender) of this channel URI (Step 4). The push sender uses the security credentials to authenticate with WNS. When the authentication is successful, the push sender is given an authentication token. push messages are sent by sending POST requests to the channel URI including the authentication token retrieved during the authentication (Step 5). The WNS then delivers the message when the mobile device is available (Step 6).

3.7.5 Notification Types

Windows Notification Service offers four kinds of notifications:

1. toast notification
2. tile notification
3. badge notification
4. raw notification

The first three kinds do not require any interaction with the notification by the app itself. Their respective format must follow the schema for its notification type. With these notifications the push sender can update tiles (tile notification) or display notifications (badges and toast notifications). Raw notifications do not have any restrictions on their format and are handled by the app itself.

3.8 Comparison of the Presented Push Services

Table 3.1 shows a comparison of the presented push services. We used the following criteria to show differences in the services. These criteria are very general and do not have a focus on security and privacy, as we will investigate those aspects in the next chapter.

Payload size—The sender can send data for the app with the notification, this is called the payload of the notification. The presented services have a limitation on the size of this payload. This size restriction has to be considered when sending notifications over the respective service.

Operating system support—The push services we analyzed so far support different platforms, a developer targeting specific platforms will want to minimize the amount of push service APIs to implement. It is also easier to adopt new platforms, if the platform is already supported by the push service.

Own push server support—Although the majority might be content with the push server of their platform, individuals who want to have more control over their personal data might want to setup their own push server.

Costs—The push service provider might want to charge the developer for using their push service. This might be important for small developers who have a tighter budget.

	Push service		
	GCM	APNs	WNS
Payload size	6KB[a]	4KB[b]	5KB[c]
OS support	Android, iOS, Chrome	iOS, macOS	Windows (8, 8.1, 10, 10 Mobile, Phone 8.1)
Own push server	no	no	no
Costs	free	developer license (99 US$/year)	registration fee (~19-99 US$)[d]

[a]2 KB predefined set of user-visible keys, 4KB data payload[29]

[b]When using the HTTP2 API, otherwise 2KB and 256b before iOS 8. VoIP apps can receive 5KB

[c]For raw payloads: `https://msdn.microsoft.com/en-us/library/windows/apps/xaml/jj676791`

[d]Individual ~19 US$, company ~99 US$, see: `https://msdn.microsoft.com/en-us/windows/uwp/publish/account-types-locations-and-fees`

Table 3.1: Comparison between the different push services

3.9 Summary

In this chapter we introduced push services, analyzed their architecture and presented three different push services in use today. We found that their architecture is very similar, but differ in authentication and the supported platforms. Unfortunately, no push service supported setting up an own server as the push server. In the next chapter, we will analyze the security and privacy of these push services.

Chapter 4

Security and Privacy Analysis

In this chapter, we will define the adversary and threat model which we will use. Then we will define what the properties *secure* and *privacy-preserving* imply for a push service in this threat model. Finally we will look at how secure and privacy-preserving the push services presented in Chapter 3 are.

4.1 Information Model

Before analyzing the security and privacy it is important to know what information is exposed during the communication and what information is included within a push message.

4.1.1 Push Message

A push message contains at least the following information:

- target app and target mobile device
- content
- size

The sender might only have a single identifier for a specific combination of mobile device and target app.

4.1.2 Push Connection

The push connection is the connection between push client and push server. The push server knows at least the following information:

- public IP address of the device

- unique identifier of the mobile device
- point in time of delivery for each push message

The unique identifier is needed to deliver the push messages to the correct device.

4.1.3 Sending a Push Message

When sending a push message, the push sender has to give the push server at least the following information:

- push message (target, content, size)
- authorization for sending a push message to the targeted app
- point in time of the connection
- public IP address of the sender

Optionally the sender might include the following information for the push server:

- priority
- time to live

This information can be used by the push server to decide if and how push messages are aggregated.

4.2 Adversary and Threat Model

There are two adversary models that will be considered. First we have the honest but curious observer O. This observer controls the push server and has access to all information accessible to the push server. This includes all the information we presented in Section 4.1.

This attacker is a passive observer and does not modify the push messages or drop them (unless this is specified by the protocol e.g. by a time to live). In this case the push server cannot be considered trusted. This adversary wants to gain personal information about the user.

The second adversary model is a state-level actor S. This actor can modify and eavesdrop on the communication to and from the mobile device. Furthermore, he has access to all of the information of the observer O and can coerce the push service provider to modify, drop, or deliver arbitrary push messages. The goal of this attacker is to spy on the user, gain information about their communication and manipulate or block unwanted communication. An example would be a repressive state wanting to censor and persecute political opponents.

We consider the app, the push sender, and the push client trusted in for both attacker models.

4.3 Definitions

It is important to define what is meant by a “secure” and “privacy-preserving” push service. We define these properties in the environment of a mobile push service. In this model we will consider only the app, the push sender, and the push client trusted. We do not cosider the push service provider to be trusted.

4.3.1 Secure

To ensure a secure operation of a push service, the push service needs to fulfill certain requirements.

First, the push service needs to ensure the *Authenticity* of the push message. Authenticity is defined by ISO 27000 as follows: “Property that an entity is what it claims to be” [30]. In the case of the push service this means that the push message is verified to be from a valid push sender. The push server needs to verify the authenticity to prevent a malicious push sender from spamming the push client. For the push client, authenticity implies that the push message is from the push sender it claims to be from.

The second property a secure push service needs to fulfill is the property of *integrity.* Integrity is defined by ISO 27000 as follows: “Property of accuracy and completeness” [30]. In the case of the push service this means that the push client can verify that the message was not modified between the sender and the client. For the attacker model with observer O it is sufficient to ensure that the message was not modified between push sender and push server and from push server to push client.

Finally, the push service needs to ensure the *Confidentiality* of the push message. The definition of Confidentiality in ISO 27000 is: “Property that information is not made available or disclosed to unauthorized individuals, entities, or processes” [30]. For a push service that protects Confidentiality, this means that the information within the push messages is only accessible to the push sender, push client, and the app.

4.3.2 Privacy-Preserving

The IETF published an RFC with “privacy considerations for inclusion in protocol specifications” [31]. In this document privacy is defined as: “privacy is the sum of what is contained in this document" [31]. A short definition is given in RFC 4949: "The right of individuals to control or influence what information related to them may be collected and stored and by whom and to whom that information may be disclosed" [32].

Therefore a privacy-preserving push service needs to minimize the amount of information that is leaked during the operation of the service. This gives the user more control, which information he wants to disclose and thus improves his privacy.

Furthermore, a privacy-preserving push service needs to provide *Anonymity*. ISO 29100 defines Anonymity as follows: "characteristic of information that does not permit a personally identifiable information principal to be identified directly or indirectly" [33]. For the push service this means that the push server does not know who sent the push message. Since the push message should be delivered to a specific push client, the push service needs to know where the push message should be delivered to. It is not important for the delivery of the push message to know who sent it.

4.4 Approaches for Reducing Information Leakage

In Section 4.1, we listed the information that is included in the push messages or leaked during the operation of the push service. From this we can deduce which information can be minimized.

For the push message, we can reduce the information leak to the push server by encrypting the content. Also it is possible to hide the target app from the push server.

Considering the push connection (between push client and push server), it would be possible to hide the public IP address, by using an anonymization network like Tor. However, this would increase the battery and data usage and thus will not be considered here.

The identifier could also be protected from the push server by using an approach like described in [34]. Here the notifications are anonymous but the client has to resort to pull notifications. But this is also impractical for mobile environments, as it also increases both battery and data usage.

Protecting the point in time of the delivery would mean increasing the latency significantly which is not desired by the user, as this would make the push service impractical for instant messaging and VoIP applications.

When sending a push message, the push message contents can be protected by using end-to-end encryption. The size of the message could be hidden by having a fixed length for the push message, however this increases the data usage of the mobile device and will not be considered for this reason. The push server only needs to know the target mobile device, but not the targeted app, meaning that the information about the target app should only be visible to the push client.

To protect the public IP address of the push sender, using an anonymization network is needed. Furthermore, the client needs to send only one message per connection to prevent the push server from correlating the push messages sent over the same connection. The authorization for sending push messages must not allow the identification of the push sender as this would make hiding the IP address useless.

Analogously to the push connection, it is undesirable to hide the point in time of the connection, as this would increase the latency.

Sender anonymity is achieved when the following conditions apply:

- hidden public IP address of the push sender
- non-identifying authorization of the push sender
- end-to-end encrypted content

4.5 Encryption in Mobile Push Services

To achieve security, it is important to encrypt the communication paths using a protocol that ensures confidentiality, authenticity and integrity. We distinguish between transport encryption — the encryption between push sender and push server, and between push server and push client — and end-to-end encryption — the encryption between push sender and push client.

The encryption usage in the push services can be seen in Table 4.1. While all push services use transport encryption, no push service enforces end-to-end encryption.

	Push service		
Encryption	GCM	APNs	WNS
Transport			
server ↔ client	yes	yes	yes
server ↔ sender	yes	yes	yes
End-to-end	no	no	no

Table 4.1: Encryption in different push services

4.6 Criteria for the Analysis of the Push Services

We will compare the push services using the security and privacy properties described in Section 4.3. Furthermore we will take a look at the following properties:

- perfect forward secrecy (end-to-end and on transport)
- certificate pinning used on the client
- open source

Perfect forward secrecy—We use the following definition for perfect forward secrecy (PFS): "A protocol is said to have perfect forward secrecy if compromise of long-term keys does not compromise past session keys" [35, p. 496].

If the push service provides end-to-end encryption this encryption should also have perfect forward secrecy.

Certificate pinning on the client—Certificate pinning is the process of verifying that specific certificates are in the certificate chain of the connected endpoint. Since the push services do not let the user choose the push server, they should ensure the authenticity of the push server using certificate pinning. Using certificate pinning prevents man-in-the-middle attacks if a certificate authority is compromised or a malicious third party certificate authority is added to the trust store of the operating system.

Open source—Having an open source implementation of all components would allow security researchers to better analyze security problems and make them public. While having the source code available is not a requirement, it makes it easier to analyze the push service.

4.7 Comparison of the Presented Push Services

Table 4.2 shows a comparison of the presented push services. In Section 4.5, we found that no tested push service provides end-to-end encryption, which would provide the security necessary to prevent the observer O from eavesdropping. Furthermore, all push services have a mandatory registration for the sender, which prevents sender anonymity.

The attacker S is a more powerful version of the observer O, as he has access to the same information and can manipulate the communication. Since all push services already fail with the observer O, they also fail with this attacker.

	Push service		
	GCM	APNs	WNS
Observer O			
Security			
Authenticity	no	no	no
Integrity	no	no	no
Confidentiality	no	no	no
Privacy			
Sender anonymity	no	no	no
Attacker S			
Authenticity	no	no	no
Integrity	no	no	no
Confidentiality	no	no	no
Privacy			
Sender anonymity	no	no	no
General			
Forward secrecy	yes[a]	yes[b]	yes[c]
Certificate pinning	yes[d]	yes[e]	unknown[f]
Open source	no	no	no

[a]tested: mtalk.google.com:5228, forward secret cipher suites were preferred, but non-forward secret ciphers were allowed

[b]tested, server: `pop-deu-central-courier.push-dapple.com.akadns.net`

[c]tested, server: `client.wns.windows.com`, forward secret cipher suites were preferred, but non-forward secret ciphers were allowed

[d]According to [36], and [37], p. 153.

[e]Since iOS 7: `https://github.com/meeee/pushproxy/blob/master/README.md`

[f]Windows 10 seems to pin the certificate: `https://hexatomium.github.io/2016/09/24/hidden-w10-pins/`

Table 4.2: Comparison of the security and privacy between the different push services

Despite these results, we found that all push services use forward secrecy and, as far as we could tell, used certificate pinning in the client to prevent man in the middle attacks.

4.8 Summary

In this chapter, we discussed the information exposed at each point in a mobile push service. Two attackers were introduced — an honest, but curious observer O, and a state-level attacker S, a more powerful attacker. Hence, we defined the terms “secure” and “privacy-preserving” and how they apply to mobile push services under these attacker models.

Additionally, approaches to reduce the information leakage were presented. We looked at the encryption employed in the mobile push services and found that they do not use end-to-end encryption to protect the push messages. The authentication of the push sender in all analyzed push services could also be used to identify the push sender, which prevented sender anonymity. These problems prevented the push services from achieving security and privacy under both attacker models.

For the user this means that the push service has access to the content of the push messages, if they are not encrypted by the push sender. The observer O can analyze the app usage patterns of the user and a state-level attacker could coerce the push server to manipulate push messages, which could result in information not reaching the user, presenting wrong information to the user or possibly exploiting third party apps.

Chapter 5

Related Work

In this chapter we will take a look at related work concerning push services and privacy in mobile platforms.

5.1 Webpush

Webpush [38] is a new standard for sending push messages to web applications. It allows the operator of a website to send push messages to the web browser. To do this the JavaScript code of the website requests permission to receive push messages. If the request is granted by the user, the browser will supply the JavaScript code with a push URL. This push URL can then be sent to the web server. The code running on the web server can use the push URL to send push messages to the browser over the push server given by the URL. The push server then delivers the push message to the web browser. The web browser can then execute the JavaScript code associated with the push message. In Firefox and Chrome, Webpush is only available when the site is served over HTTPS [39–41].

This standard comes with a JavaScript API [42], which defines the interface in the web browser.

Two other standards are developed together with Webpush: Thomson [43] describes an optional encryption that should be used by the website to encrypt push messages end-to-end. This is important, since push messages are sent over a third party push provider like GCM. Compared to the push service presented in Chapter 6, the encryption does not change keys for each message, so there is no forward secrecy and a compromise of the shared secret allows the decryption of all sent messages.

The other standard is *The Voluntary Application Server Identification for Web Push* (VAPID)[44]. This can be used by the push server to notify the sender when any problems occur. The authentication is voluntary as the name already states, so the push sender can choose to omit the authentication. Not using VAPID will not guarantee anonymity, as a registration of the push sender is required for FCM, which is used for Chrome. Further information can also be found in the Mozilla Blog[1].

These new standards, together with the Push Web API, allow the website operators to send push messages to the browser without having to establish a persistent connection themselves. These push messages even work if the web page is closed. A drawback is that the end-to-end encryption is optional, leaving the privacy of the user to the developer of the web page.

5.2 A Universal Push Service

Brustel and Preuss [45] analyze three push services in their paper: APNs, C2DM, and MPNS. In the analysis they check whether encryption is enforced from the push sender and the mobile device to the push service. They also compare payload sizes of the three push services. They propose a universal *Push Interface* which can be used to send messages over any of these push services.

While the paper also analyzes different push services for their security, the analysis does not focus on privacy, but rather simplifies the usage of multiple push services. Furthermore, their new push service does not provide solutions to the privacy and security shortcomings discussed in this thesis.

5.3 Anonotify

Anonotify by Piotrowska, Hayes, Gelernter, *et al.* [34] is an anonymous notification system, that provides anonymity for the sender and the recipient of a notification. The sender and the recipient use mix networks to hide their IP address. The notifications are stored on the server in “shards” that have an attached bloom filter. The bloom filter allows the client to check whether the notification might be in the shard or if it is not.

This design makes both the sender and the recipient of the notification anonymous, but this comes at the cost of having to regularly check for new notifications, as the client needs to check for notifications even when there are no new ones. This is undesirable in a mobile environment, where every wake-up drains battery power. Furthermore, it is possible that the bloom filter returns false positives, causing

[1]`https://blog.mozilla.org/services/2016/04/04/using-vapid-with-webpush/`

the client to download the shard even when there is no notification for it. In the mobile environment this uses data unnecessarily and also drains battery.

5.4 Analyzing Locality of Mobile Messaging Traffic

In their paper Scheitle, Wachs, Zirngibl, *et al.* [46], analyzed the traffic of different mobile messengers. They focus on the locality of the traffic. They looked at how likely it was for the traffic to leave the originating region (Europe, Oceania, Asia, South America, North America) and how likely the traffic would be visible to any of the Five Eyes partners. They found that the traffic of TextSecure and WhatsApp was always passing through Five Eyes countries. They argue that this has privacy implications for the user as the information passes through countries with strong network surveillance.

5.5 Summary

In this chapter, we took a look at different related work. The related work covers new push services, anonymous notification sending, and the privacy implications of mobile messengers. Even though there is work on anonymous notification sending, this approach is not practical on mobile devices due to its resource consumption. We see that the related work does not solve the problems discovered in the analysis of the push services.

Chapter 6

Design of a Secure and Privacy-Preserving Notification Service

In this chapter, the design for a privacy-preserving and secure push service, called SENSE (SEcure Notification SErvice), is presented.

6.1 Design Goals

In Chapter 4, the properties secure and privacy-preserving were defined in the context of two attackers. The design of the push service needs to provide these properties, while being resource-friendly. We can achieve this by having comparable battery usage to existing push services and by only transmitting data when necessary.

Furthermore, we want to enable the users to have more control over the push service by allowing them to choose the provider for the push service. Since a push service should be able to provide a service to many mobile devices, it is important for the push service to be scalable and handle many connections efficiently. In the following sections we elaborate the steps needed to achieve these goals.

6.1.1 Achieving Security

We want to achieve integrity, confidentiality and authenticity assuming a state-level attacker who can eavesdrop on the communication to and from the push server and the mobile device. We also want protection against an honest-but-curious push service provider who is providing the push service without modifying the push messages, but is interested in the content of these push messages.

To keep the honest-but-curious push service provider from reading the push messages, end-to-end security between push sender and push client is needed.

Furthermore, adding end-to-end security to the connection ensures that the contents of the push messages were not modified during the transport and that the push message is from the correct push sender.

To protect against the state-level attacker performing man-in-the-middle attacks, we additionally need transport security between the push sender and the push server and between the push server and the mobile device. This provides authenticity, integrity, and confidentiality on these connections.

Together with the end-to-end encryption this prevents the state-level attacker from modifying or eavesdropping on the push messages.

6.1.2 Achieving Privacy

To provide privacy in the push service setting, reducing the metadata available to the push server is important. The only metadata the push server needs to know to operate is the destination of the push message. The push server does not need to know who the push sender is, but needs to know if the sender is allowed to send push messages to the client. This is desired to prevent spamming the push client, as this would consume battery and cause unnecessary data usage.

Thus, an authentication method is needed, which is able to prove the legitimacy of the push sender, while hiding its actual identity from the push service.

If all requests come from the same IP address, the anonymous authentication is useless, since the push server can simply correlate messages to IP addresses. Therefore it is also important to protect the IP from leaking to the push service.

Another privacy concern is the content of the push messages. This will be protected by the end-to-end security we introduced in Section 6.1.1.

6.1.3 Achieving User Flexibility

In the push service designs analyzed in Chapter 3, users cannot easily switch push service providers, as the push client is deeply integrated into the user's mobile platform and only connects to its "own" push service. This forces the user to use the push server of their mobile platform.

In this design, we want to prevent this kind of vendor lock-in. In SENSE we realize this, by allowing the user to select the push server in the push client. The push client then passes the push server to the app during the registration, which can then pass it to the push sender. Making the push server exchangeable can prevent centralizing all push messages over one company.

This also helps to improve privacy as the user can also choose to operate their own push service or choose a push server they trust. It also allows the user to switch the push server if they are not content with the provided service.

6.1.4 Achieving Scalability

To achieve a scalable push service it is important for the implementation to use a networking API that can efficiently handle many parallel connections that are mostly idle. These idle connections should use as little memory as possible such that the server is able to handle many connections while keeping memory pressure minimal.

6.1.5 Providing Transport Security

In SENSE we will use TLS v1.2 [47] (or newer) to provide transport security. TLS protects integrity, confidentiality and authenticity between the endpoints and thus prevents the state-level attacker from manipulating the connections. To increase security only ciphers that use an ephemeral key exchange will be used. This provides Perfect Forward Secrecy (PFS). PFS protects the communications with the push server in the event of a compromise of the push server's private keys in the future (see Section 4.6).

6.1.6 Providing End-to-end Security

The end-to-end security is provided through the use of the Double Ratchet Algorithm [48]. The Double Ratchet Algorithm was developed by Trevor Perrin and Moxie Marlinspike and was originally called Axolotl Ratchet but was renamed in March 2016 [49]. This algorithm is designed with asynchronous communication in mind which makes it ideal for a mobile environment.

The Double Ratchet Algorithm was developed as part of the Signal Protocol [49], which includes the key exchange and specifies the message formats, elliptic curves for the Diffie-Hellman, symmetric encryption and MAC. A formal security analysis of the Signal protocol was done by Cohn-Gordon, Cremers, Dowling, *et al.* [50].

In the description of the Double Ratchet Algorithm by Marlinspike [51] two properties of the Double Ratchet are presented: "Future Secrecy" and "Forward Secrecy". These properties describe how the compromise of the encryption key of one message affects previous and future messages.

With "Future Secrecy" future messages are protected. This is possible, because the Double Ratchet Algorithm uses a Diffie-Hellman key exchange for every communication round trip. This prevents an attacker from deducing the new message key from an older message key.

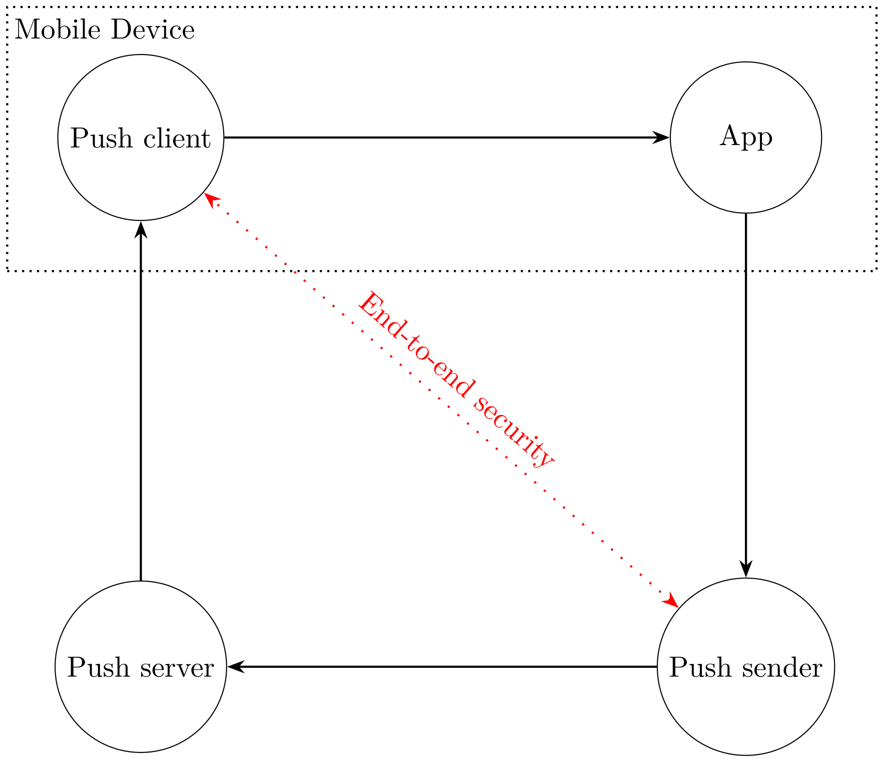

Figure 6.1: End-to-end security in SENSE. Black solid arrows denote message flow of messages.

"Forward Secrecy" protects older messages from compromise if the message key of a message is compromised. This is achieved by deriving a new message key from the Diffie-Hellman exchange or hashing the last message key. This makes it difficult to derive the previous message key from a given message key.

The Double Ratchet Algorithm also defines a header encryption scheme for the exchanged messages. With header encryption enabled, the Diffie-Hellman (DH) parameters sent with each message and message counters are encrypted as well. This is needed for SENSE, as the DH parameters only change when a message is received. This is intended, as only then the receiver knows that the other party has received the previous DH parameters and only then new DH parameters are generated. Multiple messages sent by one sender, without receiving a message in between, can thus easily be tied to this sender.

In SENSE, we will use the Signal Protocol to provide end-to-end security for the push messages between push sender and push client. The message flow and how the end-to-end security is enclosed in SENSE can be seen in Fig. 6.1.

6.1.7 Providing Sender Anonymity

The push service provider needs to authenticate the push senders, since only authorized push senders should send push messages to the device. This is desired, as push messages draw battery power and use mobile data. These are both limited resources in a mobile environment. This authentication needs to be anonymous, otherwise the push service can use the authentication to distinguish between the push senders.

To ensure anonymity of the push sender to the push service, the connections to the push server must be established through an anonymization network. In our design, we will use Tor [12].

To ensure that the sender uses Tor to send the push messages to the push server, the push server is only reachable using a Tor onion service. An onion service provides anonymity to both sides of the connection. The push client will tell the app the onion URL used to connect to the Tor onion service of the push server. This ensures that the push sender uses Tor and does not leak its IP address to the push server.

To prevent the push server from correlating push messages only one push message can be sent over a connection. If more than one push message would be sent over one connection, the push server could easily infer that all these push messages come from the same push sender. This is enforced by both the push server and the push sender, who both close the connection after a successful delivery.

The anonymous authentication is provided through the use of HMAC [52] tokens. Being a cryptographically proven Message Authentication Code, HMAC allows to prove authenticity of a message. In this case the authenticity of the token is proven. The push client produces random messages, appends the HMAC, generated with a key that is shared with the push server, and gives them to the push sender. When given to the push server by the push sender, the push server can verify the authenticity of these messages, without knowing the push sender. These tokens are generated as follows: The push client generates a random key k that they share with the push server. When an app wants to receive push messages, the push client generates n tokens t_i using the following algorithm:

```
for i ←1 to n do
    $r_i \leftarrow random(\{0,1\}^{len(k)})$
    $h_i \leftarrow \textsc{hmac}(k, r_i)$
    $t_i \leftarrow (r_i, h_i)$
end for
```

The tokens are then encrypted with the Signal Protocol, which is initialized with the identity key and pre-keys (the first DH parameters) of the push sender, which the app supplies to the push client during registration.

$\mathit{encryptedTokens} \leftarrow \textsc{signalEncrypt}_{\text{push sender}}(, (t_1, t_2, \ldots, t_n))$

The push sender can then decrypt the encrypted tokens and use one of the tokens to send a push message. The push server receives the push message and can verify the token t for a key k as follows:

```
function isValidToken($k, t$)
    $(r, h) \leftarrow t$
    if inBlacklist($k, r$) then
        return False
    end if
    if hmac($k, r$) $==$ $h$ then
        blacklistToken(k, r)
        return True
    else
        return False
    end if
end function
```

The idea behind this authentication is that the push server does not know who received the token, but can verify that it is a valid token. The tokens are blacklisted as soon as they are found valid, to prevent token reuse. This blacklist could grow very large over time, as more and more tokens get used. To prevent this problem, more than one key can be stored on the server and the push client signals the push server to remove the keys when all tokens for this key were used. Since the push server does not know which key was used to create the token, it has to try every key belonging to this user. This method of anonymous authentication was inspired by the design of Pond.[1]

6.1.8 Summary

From the description so far, we know that the architecture of Sense looks like Fig. 6.2. The connections to and from the push server are protected from the state-level attacker and the push messages going through the push server are protected with end-to-end security from the honest-but-curious push service operator.

[1] `https://github.com/agl/pond`

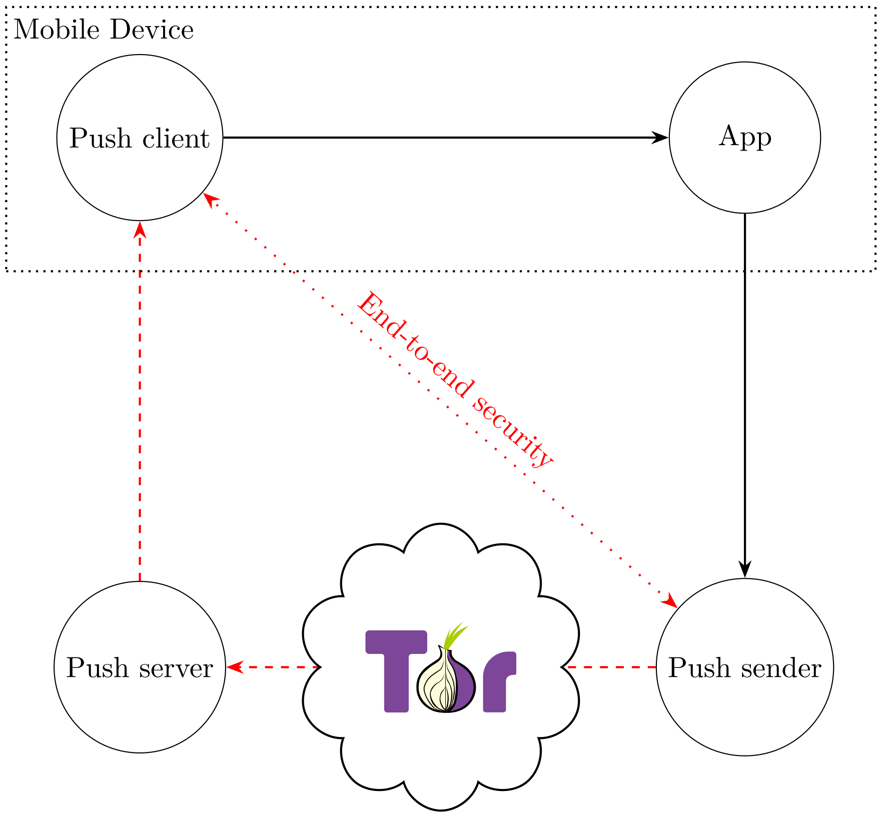

Figure 6.2: Architecture of Sense. Red (dashed or dotted) lines are encrypted.

6.2 Protocol Description

Now that we know how the security is ensured and the privacy preserved we will take a look at how the four parties interact. We will go through the steps needed to register with the push server for the delivery of a push message. These steps can be seen in Fig. 6.3.

6.2.1 Registration of the Push Client

To register the push client establishes a TLS-secured HTTP2 connection to the push server. The push client then sends a GET request to the path `/register` and using HTTP Basic Authentication with an empty username and a 16 byte random secret as password. This makes guessing the secret difficult as there are $2^1$28 possible combinations. This secret is used to authenticate the client to the server. The server creates a random Universally Unique Identifier (UUID) as defined in the UUID specification [53] as “Version 4.” and returns it to the client. The random UUID is chosen as it does not use any external information for its generation. This is important, since the UUID is passed to third parties, which

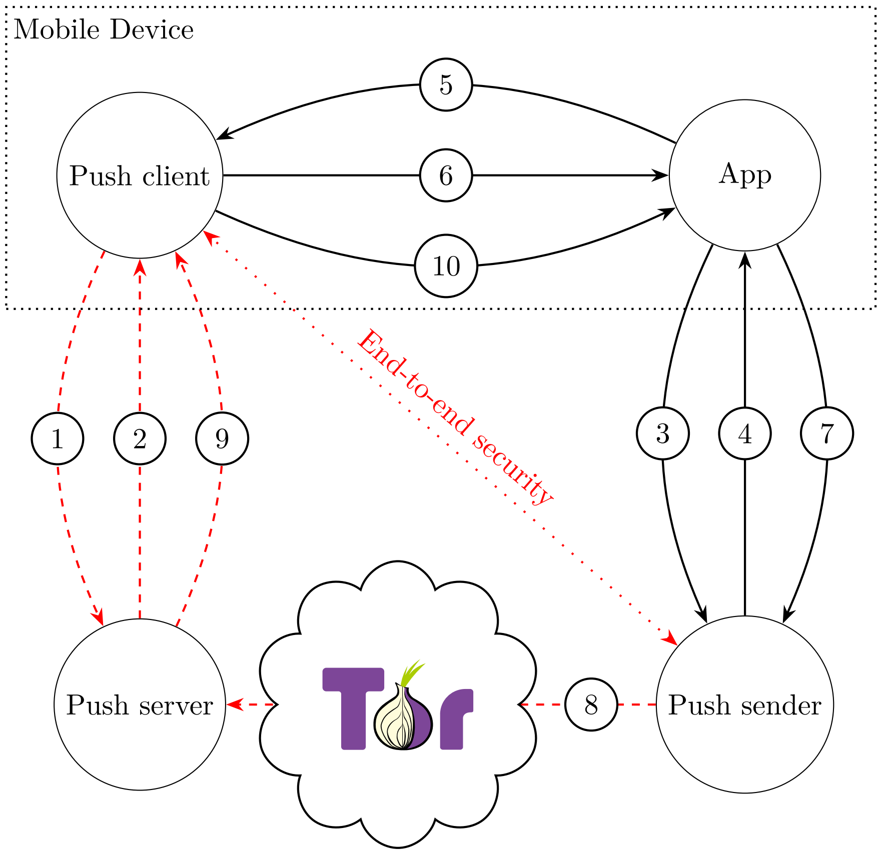

Figure 6.3: Registration to SENSE (1-7) and sending (8-10) of push messages. Red (dashed or dotted) lines are encrypted.

might extract this information. This UUID is used to identify the client to which the push sender wants to send the push message.

The client then generates a key for the HMAC tokens and sends it with a POST request to the server using the `/addKey` path. To authenticate this query, the push client uses HTTP Basic Authentication with the UUID as username and the 16 byte secret as password.

To get the address of the Tor onion service of the push server, used to deliver the push messages to the it, the client sends a GET request to the `/onionHost` path.

The push client is now registered with the push server. This corresponds to the steps 1 and 2 in Fig. 6.3.

6.2.2 Registration of the App

The app needs to get the necessary information for starting a signal ratchet from the push sender (steps 3,4), before registering with the push client. This includes the identity key of the push sender as well as pre-keys. With these keys, the app can request the registration from the push client(step 5). The push client then generates the tokens for this app and encrypts them using the Signal Protocol, which is initialized using the pre-keys and the identitiy key of the push sender. The encrypted tokens are then given to the app together with the onion address of the onion service, and the UUID for the push client (step 6). The app gives this information to the push sender (step 7) who is able to send push messages with this information.

6.2.3 Sending a Push Message

By decrypting the encrypted tokens the push sender can initialize its side of the Signal Protocol. This allows him to encrypt push messages. When the push sender wants to send a push message he encrypts the data for the app in a Signal message.

To ensure the anonymity of the push sender, the header encryption of the Double Ratchet Algorithm needs to be used, otherwise information about the push sender is leaked.

The push sender connects to the onion address of the push server — which was given to him by the app — over the Tor network using a Tor client connected to the Tor network.

Using this connection the push sender establishes a TLS-protected HTTP2 session and delivers the push message by sending a POST request with the push message as body to the UUID path of the user with removed dashes. E.g. for the user with the UUID `427ac153-3e72-4aa4-a5a3-790943743d7f` the sender would send a POST request to the path `/427ac1533e724aa4a5a3790943743d7f` with the Signal message as request body. One of the decrypted tokens is Base64 [54] encoded and included as the "token" header of the POST request. This is step 8 in Fig. 6.3.

6.2.4 Retrieving Push Messages

The client establishes a HTTP2 connection to the server (or reuses the connection used to register with the push server). This connection is used to deliver the push messages to the client. To request push messages, the client starts a GET request to the path `/retrieve`. The HTTP2 stream opened by the client is never closed by the server and stays open until the client closes the stream with a HTTP2 `RST_STREAM` frame. The client authenticates to the server by using HTTP Basic Authentication with his UUID as user and the 16 byte secret as password. While this stream is open, new push messages are delivered by the push server using

HTTP2 `PUSH_PROMISE` frames. The `PUSH_PROMISE` frame opens a new stream and includes the token that was used to send the push message in the header. The payload of the push message is then delivered over this new stream (step 9 in Fig. 6.3).

The client uses the token to look up for which app the push message is intended. Using this information, the push client can decrypt the payload of the push message and deliver it to the app (step 10).

6.2.5 Revoking Push Access

When the user does not want to receive push messages from a specific push sender any longer, all the tokens for this push sender can be revoked on the push server. This will prevent the push sender from sending any messages in the future, as the push server will reject any attempts to send push messages with an invalid token. After the push access has been revoked no new messages should be sent by the push sender, as the push server could store which tokens were invalidated together. and thus could correlate these messages.

In some cases, it is not possible to inform the push sender of the revocation, e.g. if the app was uninstalled. In this case the push access should not be revoked. Since the push sender only has a limited amount of tokens (at most 100 tokens), only a limited amount of notifications can be sent after the app has been uninstalled. These push messages are simply ignored by the push client running on the mobile device.

6.3 Components

Each component of Sense has specific requirements it needs to fulfill to provide the service.

6.3.1 Push Server

The push server has the following duties:

- allow registration of new push clients
- store client UUIDs and secrets
- store HMAC keys for each client
- provide onion service
- accept push messages

- verify the authenticity of the push sender
- disconnect after accepting a push message
- manage token blacklist for each HMAC key
- store push messages
- deliver push messages

The push server is responsible for the delivery of the push messages. He needs to accept registrations, store the new UUIDs and secrets and keep track of each clients HMAC keys.

To accept push messages, a onion service needs to be created. This can be done by running a Tor client on the same machine that establishes the onion service and connects to the push service, when a new connection to the onion service is established.

The push messages delivered over the onion service needs to be authenticated and stored until the successful delivery to the push client. After having accepted a push message, the push server needs to terminate the connection, to prevent the push sender from sending any more data. Furthermore, the push server needs to store any used tokens in a blacklist until the key is deleted.

When a client is connected, the push messages need to be delivered and removed from the store, when the delivery was successful.

6.3.2 Push Client

The push client has the following duties:

- register with the push server
- manage HMAC keys
- manage tokens
- accept registration from apps
- store Signal Protocol state for each app
- encrypt tokens for apps
- establish connection to push server and keep it alive
- verify tokens of incoming push messages
- assign push messages to apps
- decrypt push messages
- deliver push message payloads to apps

Before any app can register with the push client the push client needs to register with the push server and store the first HMAC key on the server. The client needs to keep track of the HMAC keys on the server, how many tokens were already generated for each key, and if any app has no more tokens left.

Apps that want to register need to provide the push client with the identity key of the push sender and the pre-keys to establish the Signal Protocol. The Signal Protocol State has to be stored permanently by the push client. The tokens for the app need to be encrypted using the Signal Protocol and passed to the app for the push sender.

To receive push messages the client has to establish the connection to the push server as soon as Internet connectivity is available and keep this connection alive. The token of each arriving push message needs to be verified and assigned to an app. If the token is valid, but cannot be assigned to an app, the push message must be discarded.

When the app for the push message is known, the client decrypts the push message using the Signal Protocol. The decrypted message is then delivered to the app.

6.3.3 App

The app that wants to use SENSE as push service needs to:

- know the identity key of the push sender for this app
- retrieve new pre-keys from the push sender
- register with the push client
- accept new encrypted tokens
- accept notifications from the push client

The app needs to have the identity key for the push sender and new pre-keys to register with the push client. When the push client sends new encrypted tokens to the app, they need to be passed to the push sender, so that the push sender can start or continue sending push messages. Furthermore, the app needs to accept new push messages from the push client and process the data from the push message.

The app uses a library to communicate with the push client, which handles the Inter Process Communication (IPC) between push client and app.

6.3.4 Push Sender

To be able to send push messages over SENSE the push sender needs to provide the following features:

- allow the app to retrieve new pre-keys
- store Signal Protocol state for each mobile device
- accept new encrypted tokens
- store tokens per app
- store UUID per app
- store push server per app
- connect to push servers over Tor using a Tor client
- encrypt push messages
- deliver push message to push server
- disconnect after each push message

The push sender needs to supply the apps with pre-keys, so that they can register with their push clients. Every pre-key is stored in the Signal Protocol state, so that it can be used to decrypt the encrypted tokens. When the app delivers new tokens, they have to be decrypted using the Signal Protocol and stored in a permanent storage along with the onion address of the push server and UUID for this push client.

To send a push message a connection to the corresponding push server needs to be established over the Tor network. This is done by using the SOCKS server provided by a Tor client accessible to the push sender. Before sending the push message, it needs to be encrypted for the push client of the targeted app using the Signal Protocol.

With each push message the push sender needs to send a valid, unused token in the “token” header. After a successful delivery, the push sender needs to disconnect from the push server.

6.4 Protocol between App and Push Sender

The protocol between app and push sender is not specified, as it is specific for each app. In Chapter 7 we will present the protocol we used in our test implementation.

6.5 Message Format

For the push messages we use the Signal Message format, as specified by the Signal Protocol. The decrypted contents of the push message is application-specific.

For the encrypted tokens the Signal PreKeyMessage format is used, if it is the first encrypted token message, otherwise a Signal Message is used. The content of this message are concatenated tokens in binary representation. Each token is 64 bytes long, 32 bytes for the random message and 32 bytes for the HMAC.

6.6 Limitations

Even though privacy is improved with the proposed push service, the recipient of the message is not hidden from the push server. If the push protocol wanted to hide this information, the message would have to be transmitted to more than one push client. This is not desirable in a mobile environment since it increases traffic and keeps the device awake, which would decrease the battery lifetime.

Chapter 7

Implementation

To test the design presented in Chapter 6, the four parties involved in a push service (see Section 3.3) were implemented to test them against an existing push service.

7.1 Design Decisions

The design describes the high level protocol, but does not specify all variables. In this section, we will take a look at how we implemented the design.

7.1.1 Architecture

For simplicity and to ease testing the implementation, the push client and app are one application, but separating them later would not be a problem.

7.1.2 Tokens

The design specifies the token generation using HMAC [52]. In this implementation we will use SHA256 as the secure hash function and both the key and the random part of the token will be 32 bytes long. This results in a 64 byte token, which will be encapsulated with a Base64 encoding, when transmitted as cleartext.

7.2 Push Server

The push server was implemented in Rust[1]. Rust was chosen for several reasons. First, Rust is a memory safe language without using garbage collection. This has

[1]`https://www.rust-lang.org/`

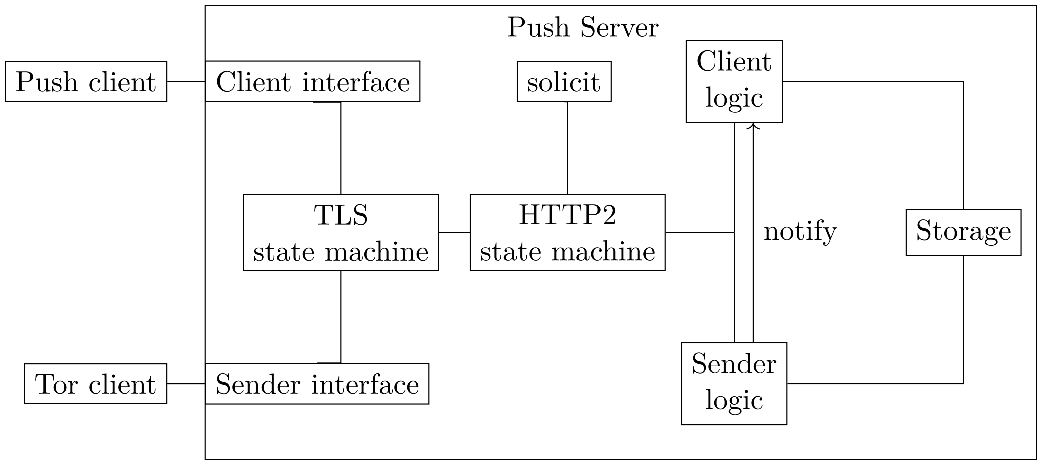

Figure 7.1: Architecture of the push server

the benefit of more predictable performance. The memory safety prevents several security problems and the thread safety guaranteed by the language prevents data races. Although the server is currently single threaded, adding multiple threads to handle high workloads would not constitute a problem. A more extensive overview of the benefits of Rust can be found in [55].

The networking is done asynchronously to handle many concurrent connections efficiently. Asynchronous networking scales better than multi-threaded networking with many connections [56]. The server uses a Rust library that implements a high-level interface around *epoll*, which is very efficient for handling many connections [57].

7.2.1 Architecture

An overview of the push server architecture can be found in Fig. 7.1. The server uses two state machines to handle all incoming connections. The first state machine is used for the TLS layer (see Fig. 7.2). It handles the TLS handshake and wraps the second state machine. The second state machine is used to handle HTTP2 connections (see Fig. 7.3). It checks for the preamble required by HTTP2 and exchanges the `SETTINGS` frames. When the HTTP2 connection is established and all buffers are empty, the HTTP2 state machine is in the state *idle*. This ensures that the allocated buffers are freed and the server uses as little memory as possible. When new frames arrive or messages need to be sent, the state is changed to *active* and the incoming frames are handled and outgoing frames are sent.

The frames are handled by the Rust HTTP2 library solicit.[2] This library tracks

[2]`https://github.com/mlalic/solicit`

the state of the HTTP2 connection and the streams. It can also automatically answer `PING` frames and handle `SETTINGS` frames. This library features a server implementation that would handle the complete connection, but this feature was not used as it did not use asynchronous I/O. The push server features an own asynchronous HTTP2 server implementation in Rust.

The streams are handled depending on the type of the connection — either from a push client or from a push sender.

The push client can perform four different actions. First, he can register using the URL `/register` and using HTTP Basic Authentication with a 16 byte password and an empty username. The server then generates a random UUID as defined in the UUID specification [53] as "Version 4".

After having registered, the client can add new secret keys for tokens using POST requests to `/addKey`. These keys are stored and used to verify the tokens sent by the push sender.

To remove a key, the client can use a POST request to `/delKey`. This ensures that all tokens with this key are considered invalid. This helps the server as the key's blacklist can then be discarded.

Finally, the client can use a GET request to `/retrieve`. This will cause the server to send `PUSH_PROMISE` frames and the accompanying data whenever a push message is received. This stream is never closed by the server, unless the client uses a `RST_STREAM`.

The push sender can only send a POST request to the path corresponding to the UUID, as specified in Section 6.2.3.

7.2.2 Security

The connections are secured using TLS v1.2 [47] and using the cipher suites recommended by Mozilla for Server Side TLS [58]. These settings enforce Forward Secrecy by only using ciphers with a forward secret key exchange. All ciphers use elliptic curves for the key exchange, which uses less data, as the keys are shorter and the calculation is faster. These properties are good for mobile devices, as CPUs are slower and data is often limited.

The certificate for the privacy-preserving push service was created using *Let's Encrypt* [59], to have a certificate from a certificate authority accepted by the Android OS. This ensures that the client can verify the authenticity of the push server. As the user should be able to change the push server via the user interface of the app, no certificate pinning was used. To increase security the push server could tell the client to pin the public key.

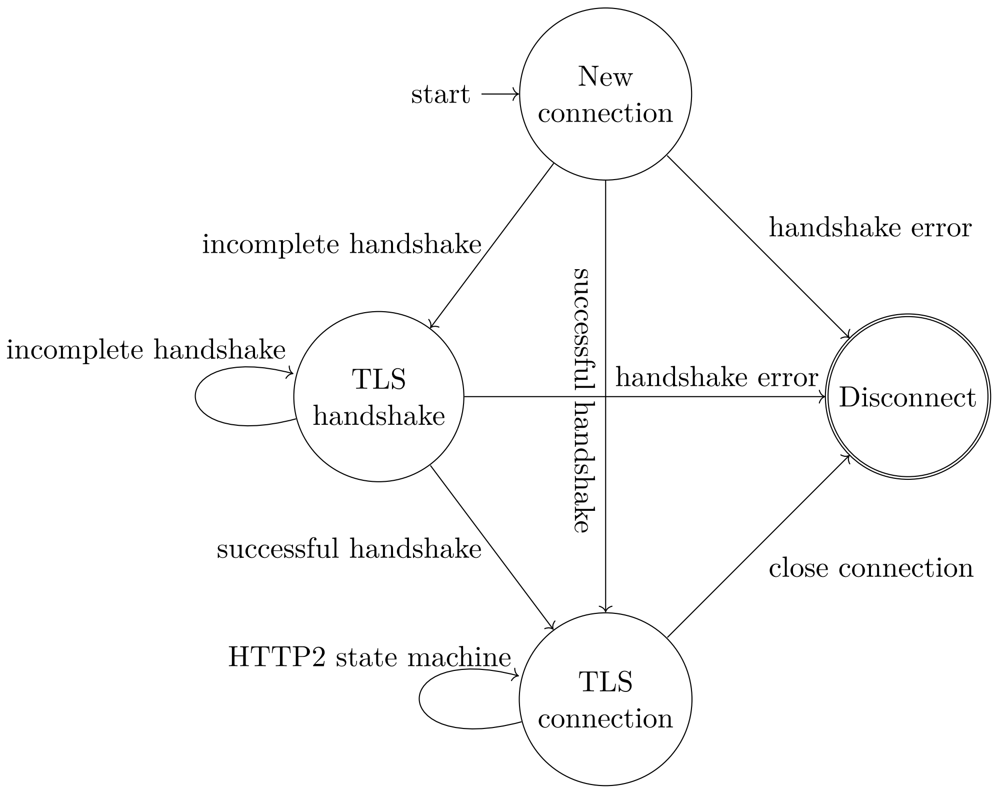

Figure 7.2: The TLS state machine of the push server

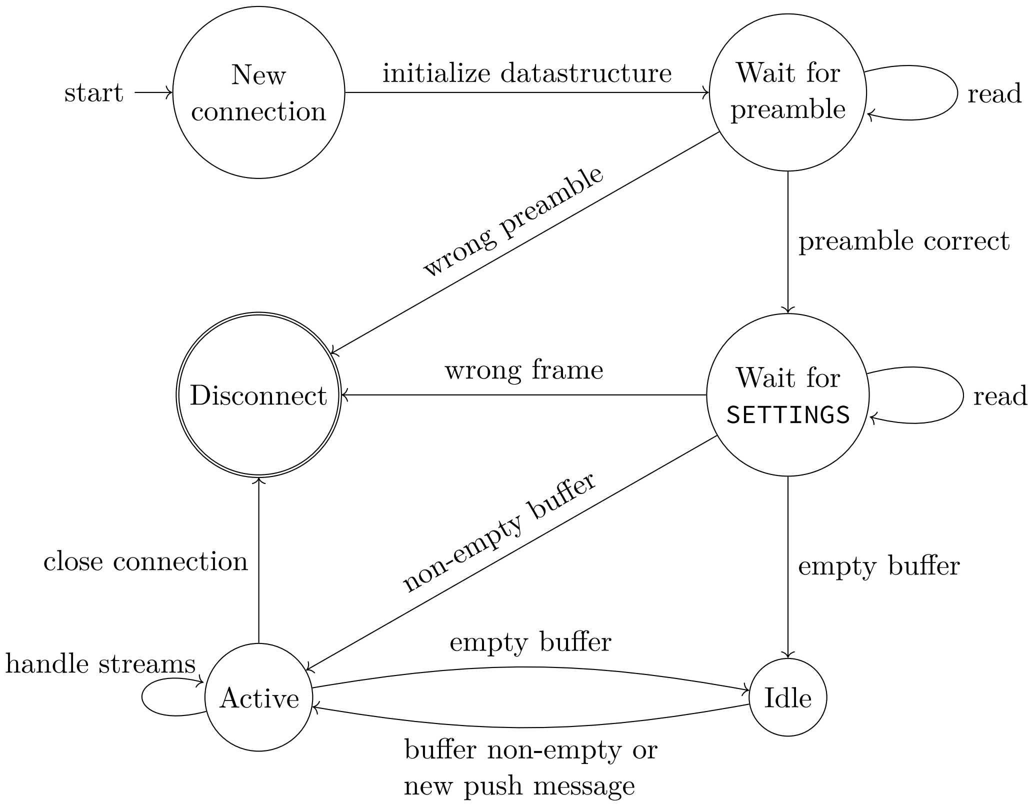

Figure 7.3: The HTTP2 state machine of the push server

7.2.3 Persistent Storage

The push server needs to store the undelivered push messages, the HMAC keys, UUIDs, and the secrets for the authentication persistently. Currently all storage is in memory, but behind a generic interface, which would allow putting it into e.g. a relational database.

7.2.4 Tor Onion Service

The onion service is created by adding these lines to the `torrc` configuration file:

```
HiddenServiceDir /var/lib/tor/pushservice/
HiddenServicePort 3001 127.0.0.1:3001
```

This causes Tor to create a private key for the onion service and the associated hostname is stored in `/var/lib/tor/pushservice/hostname`. This hostname has to be stored in the configuration of the push server so that the client can fetch this information from the push server. When the Tor client is started, it will announce the onion service and clients can connect to the push server. To enforce the usage of the onion service, the server binds to 127.0.0.1:3001, so it can only be accessed from the server running the push server. Since the Tor client runs on the same machine it can connect to this port, when a new connection to the onion service is established over the Tor network.

7.3 Push Client

The push client was developed for Android in Java. It registers with the push server and sends tokens and UUID to the push sender. It can also receive push notifications over FCM. This will be used to compare the efficiency of Sense with FCM.

For the HTTP2 protocol the okhttp library[3] was used. Although the okhttp library theoretically supports HTTP2, the `PUSH_PROMISE` support was not exposed via the public API, so the library was modified to expose an API for these frames.

7.3.1 Why Android

During the design phase, a client for iOS was considered. It became obvious that the restrictions of iOS would make a practical implementation infeasible. APNs is very tightly integrated into iOS and the IPC is very simple, only allowing file

[3] `https://square.github.io/okhttp/`

transfers and calling other apps with an URI scheme[4]. In Android, it is possible to create a push client that provides a similar interface to FCM for receiving push messages as the IPC used by FCM is available to every app.

7.3.2 Keeping the Connection Alive

To prevent the Android system from disconnecting the connection to the push server, the battery optimization was disabled for the push client. To keep the app in memory a persistent notification is displayed, as long as the push client is running. Furthermore, the app registers an alarm that fires approximately every hour to send a HTTP2 `PING` frame. The Android *Doze* mode, which was introduced in Android 6.0 *Marshmallow*, disables this alarm. To keep the idle time of the connection as short as possible during Doze, a `PING` frame is also sent when entering and exiting idle.

Although it is possible to enable TCP keep-alive in Android, the kernel seems to stop sending keep-alives during sleep[5].

A more sophisticated strategy for keep-alives is presented in the future work Section 9.3.

7.3.3 Persistent Storage

The client has to store certain information persistently. The client needs to store the current push server and the onion address of the push server. Furthermore, it needs to keep track of the issued tokens, so that they can be linked to the app that requested them. The client also needs to store the keys for the tokens and the state of the Signal Protocol.

This information is stored in a SQLite[6] database. SQLite support is integrated into Android as a method for storing persistent data. The database scheme used in the implementation can be seen in Fig. 7.4.

[4]`https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Inter-AppCommunication/Inter-AppCommunication.html`
[5]`http://stackoverflow.com/a/30904117`
[6]`https://www.sqlite.org/`

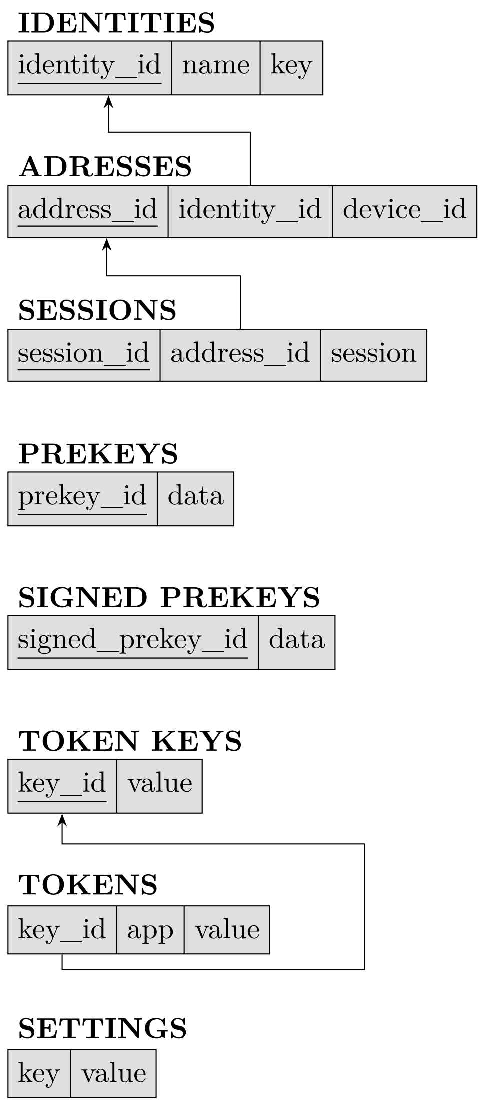

Figure 7.4: The database scheme used in the SENSE implementation

7.4 Push Sender

The push sender is also written in Java and provides a simple HTTP interface for registration of the client and for sending messages to a client. This interface was implemented using spark.[7]

The push sender supports sending over FCM or over Sense. The HTTP interface allows to specify which push service should be used. An overview of the possible commands can be seen in Table 7.1.

During the registration the onion address of the push server can be passed to the push sender. This provides the possibility of setting the push server in the push client, which was introduced as user flexibility in the design goals. The Sense interface then uses the SOCKS [60] server provided by the Tor client to connect to the push server.

The sender stores the Signal encryption data, and the registration information (the UUIDs and corresponding tokens) permanently using Java Object (de-)serialization.

The push sender was controlled during the tests using a simple python script that used this interface to send push messages. This design also makes it possible to use this push sender implementation as a back-end for any application that wants to send messages over Sense. An overview of the push sender architecture can be seen in Fig. 7.5.

[7] `http://sparkjava.com/`

Path:	/register
Method:	POST
Description:	Used by the client to register with the push sender
Request body:	Encrypted Tokens
Parameters:	
uuid	The UUID of the push client
endpoint	The *.onion* hostname of the push server

Path:	/instanceid
Method:	POST
Description:	Used by the client to give the push sender the instanceID token used to send messages over FCM
Parameters	
uuid	the UUID of the push client
instanceid	The InstanceId authentication token for FCM

Path:	/prekey
Method:	GET
Description:	Used by the client to retrieve a prekey for the Signal Protocol
Response body:	Prekeys, Identity keys as JSON data

Path:	/message
Method:	POST
Description:	Used to send messages to a client over SENSE or FCM
Request body:	The unencrypted message for the client
Parameters	
uuid	The UUID of the push client
fcm	Either 0 or 1, use FCM as transport(1=true, 0=false, default=0)

Table 7.1: The different supported queries for the push sender

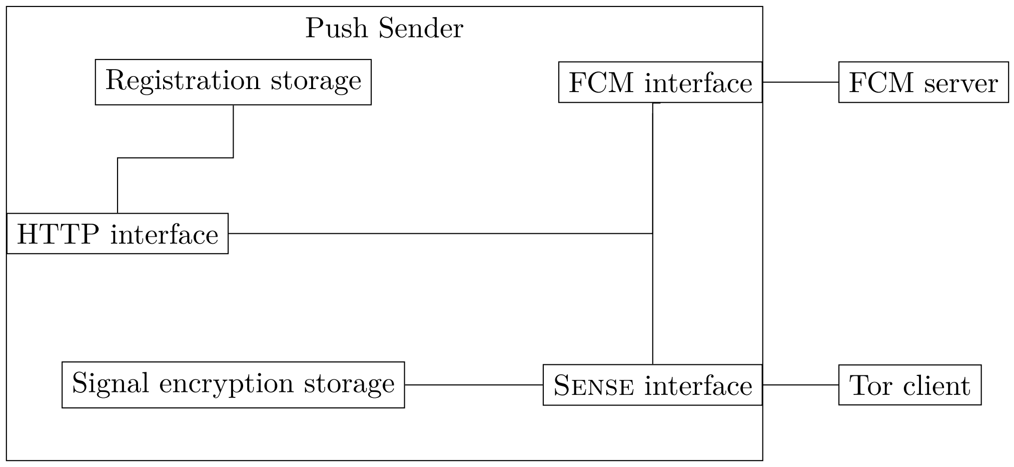

Figure 7.5: Architecture of the push sender

7.5 End-to-end Encryption

The end-to-end encryption is implemented using Java implementation of the Signal Protocol from Open Whisper Systems[8]. This implementation unfortunately does not support header encryption. Implementing the header encryption was not possible during the thesis due to time constraints. An implementation that is used productively should replace this implementation with an implementation that uses header encryption. The author is aware of three implementations of the Double Ratchet Algorithm with header encryption: one in GO for Pond,[9] another implementation in JavaScript,[10] and one in Objective-C.[11] The author of the later two implementations strictly warns against the usage of them and Pond is no longer actively developed[12] and suggests using Signal.

[8]`https://github.com/WhisperSystems/libsignal-protocol-java`
[9]`https://github.com/agl/pond/blob/master/client/ratchet/ratchet.go`
[10]`https://github.com/emundo/MobileEdge-Server/blob/master/node/libs/axolotl.js`
[11]`https://github.com/emundo/MobileEdge-iOS/tree/master/MobileEdge-iOS/Security/Axolotl`
[12]`https://github.com/agl/pond`

Chapter 8

Evaluation

In the last chapter, an implementation of the SENSE push service was presented. To see how this design and implementation compares to existing push services we will compare the security properties and performance of this implementation to an existing push service.

8.1 Security Analysis

The motivation for the new push service was improved privacy and better security than existing solutions. We will take a look at the security properties we defined in Chapter 4 and see if the implementation does provide these properties. The result can be seen in Table 8.1.

8.1.1 Security

SENSE uses TLS v1.2 with enforced PFS for the transport security and uses the Signal Double Ratchet Protocol for securing the communication between push sender and push client. This provides integrity, authenticity, and confidentiality against an eavesdropping push server (observer O), as well as any eavesdropper monitoring the traffic from the mobile device or the push server. Since the end-to-end security prevents forging or modifying push messages, SENSE protects against the state-level attacker we introduced in Section 4.2. Furthermore, it prevents the push server from dropping single messages, as the Signal Protocol can detect missing messages, through message counters included in the header of the Signal Messages.

	Sense
Observer O	
Security	
Authenticity	yes
Integrity	yes
Confidentiality	yes
Privacy	
Sender anonymity	yes
Attacker S	
Authenticity	yes
Integrity	yes
Confidentiality	yes
Privacy	
Sender anonymity	yes[a]
General	
Forward secrecy	yes
Certificate pinning	no[b]
Open source	yes

[a]when using the header encryption of the Double Ratchet Algorithm
[b]Future work

Table 8.1: Security analysis of Sense

8.1.2 Privacy

With the anonymous authentication using tokens, the push server cannot infer the push sender using authentication information and the Tor layer protects the IP of the sender. By using one connection per message, the push server cannot infer the origin of the push message.

8.2 Performance Analysis of the Push Service

Sense should offer similar performance as existing push services to ensure that it can be used for the same use cases.

8.2.1 Properties to be Analyzed

To analyze the performance of the push service we need another service to compare it with and the properties that should be compared. As the push client runs on

Android the obvious candidate for comparison is the biggest push service on that platform — FCM which is provided by Google.

One performance metric that was chosen is the latency of the notification. Latency is important, as the user might be waiting for a message and wants to be notified instantly. A latency that is too high would be undesirable for the user.

Another case where the latency would be important is an VoIP application. The VoIP application could send a notification to the mobile device of the callee so that the mobile device starts ringing. The caller has to wait for the notification to arrive and the callee to pick up the call. In the normal mobile network it takes around 8 seconds to establish a call to another mobile device: "Current trends seem to indicate a call set-up time of 4 seconds for a mobile to PSTN call and about 8 seconds for a mobile-to-mobile call." [61, p. 260] SENSE should provide a similar or better performance.

The second metric that was chosen is power usage. In the mobile space it is important not to waste power, as battery capacity is limited. The power consumption of the SENSE push service should be comparable to FCM or better.

8.2.2 Experimental Setup

The experiments were done on a rooted *Motorola Moto E 4G* running Android 6.0.1 (Marshmallow). The push client app is running on the device. The push client registers with the push server and FCM after being started and waits for messages. The push client registers with the push sender by forwarding the registration information about FCM and SENSE. When a push message is received, it is logged to the Android log buffer.

The setup for latency tests can be seen in Fig. 8.2. During the test, the sender sends the current timestamp as message payload to the client. The client then logs the payload to the log buffer. When reading the log buffer with "adb logcat", every log entry is shown with the time it was logged. By comparing the time it was logged and the time the message was sent, it is possible to calculate the latency.

Both sender and mobile device were synchronized with stratum 1 NTP time servers, to reduce the error of differently running clocks. For the mobile device a stratum 1 time server was connected to the WiFi access point used by the mobile device. This time server uses a GPS signal to get the accurate time. This timeserver was built using a BeagleBone Black[1] with a GPS module that has a Pulse Per Second (PPS). This setup can be seen in Fig. 8.1.

[1] `https://beagleboard.org/black`

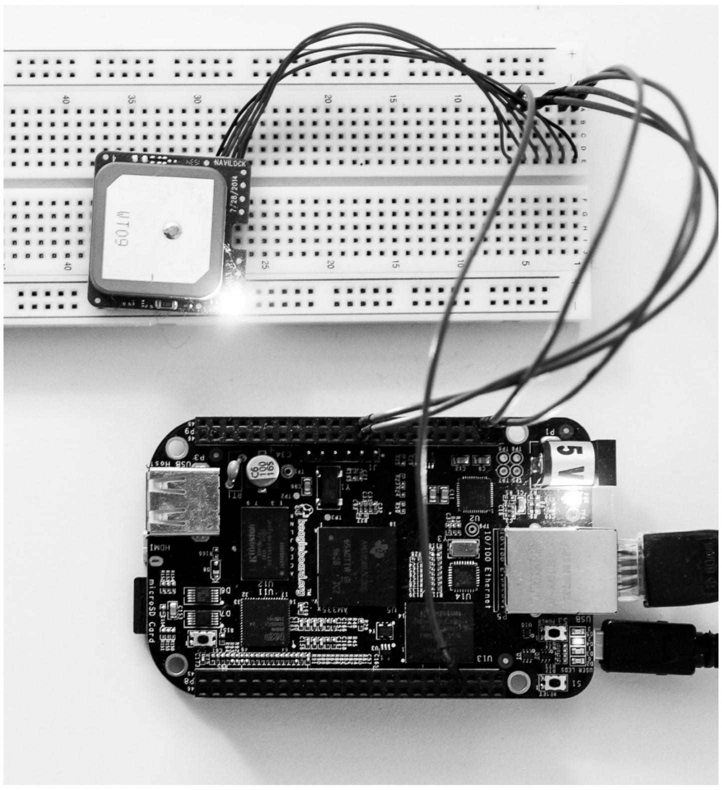

Figure 8.1: The beaglebone with the GPS module used as timesource for the mobile device

To synchronize the clock of the android device the ntpd from busybox[2] was started before the start of the test.

The sender was synchronized with the NTP servers of the Leibniz Rechenzentrum[3], where the sender was located. The LRZ uses a DCF77 (radio) and a GPS reference clock for their time servers. The NTP server was configured to prefer the GPS time server.

Battery consumption tests are run by disconnecting the device from the power and reading the battery consumption data as soon as the test is finished using "adb bugreport". The resulting file can be analyzed using *Battery Historian*[4]. During the battery consumption test no NTP synchronization was used, as this would consume battery power and distort the results.

In the test, notifications are sent randomly. The probability of receiving a notification is uniformly distributed over the timespan of the test. The test duration was set to 24h with 240 notifications arriving during this timespan. This corresponds to an average of 10 notifications per hour. To make the test reproducible, the notification times are generated and stored before the test. Each test uses the same pre-generated notification times.

Because the Tor network adds latency, it is interesting to know the impact on the notification latency. As the notifications are sent to an onion service, in total 7 Tor nodes are between sender and push server. Additionally, each notification has to be sent over a new connection, adding more time to the notifications. The decryption of each message can also have an impact on the latency.

Starting with Android 6 Marshmallow, a power saving mode called *Doze* [62] was introduced. When the device is disconnected and lying on a flat surface the device enters a deep sleep mode. During this sleep mode the device will wake up for short periods called *maintenance windows*. When the device is in *Doze*, only messages with high priority are delivered by FCM. During the *maintenance windows* the push messages with normal priority are delivered to the device. Since the device sleep between the *maintenance windows* can last for several hours, it can take several hours until the message reaches the device.

Besides the priority, the TTL setting can affect the notification time with FCM. From the FCM documentation:

> "Another advantage of specifying the lifespan of a message is that FCM never throttles messages with a `time_to_live` (TTL) value of 0 seconds. In other words, FCM guarantees best effort for messages that must be delivered 'now or never.' Keep in mind that a `time_to_live`

[2] `https://busybox.net/`
[3] `https://www.lrz.de/services/netzdienste/ntp/`
[4] `https://github.com/google/battery-historian`

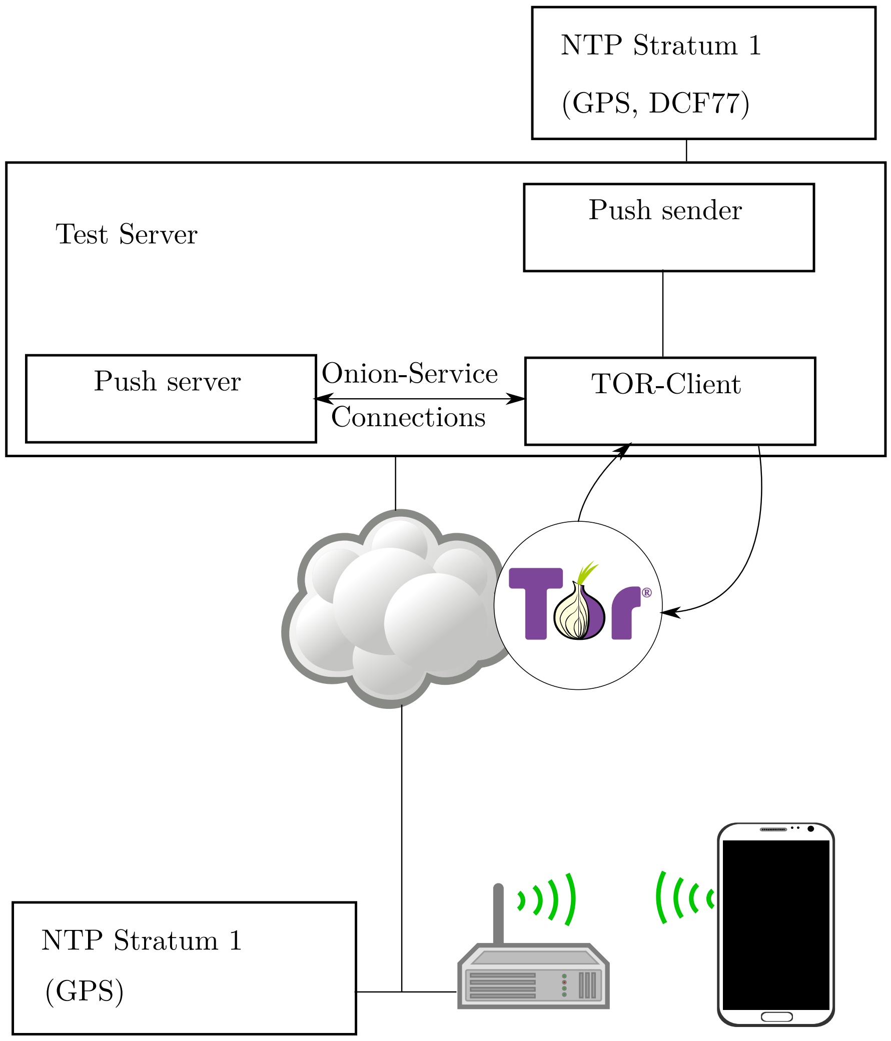

Figure 8.2: The test setup for the latency tests. The test device is connected to the power.

> value of 0 means messages that can't be delivered immediately are discarded. However, because such messages are never stored, this provides the best latency for sending notification messages." [29]

To compare the latency of the Sense push service with FCM, we test the latency of FCM with a `time_to_live` value of 0, and with the default `time_to_live` each for high and normal priority. A `time_to_live` value of 0 and high priority should result in the lowest latency.

The device is connected to a power supply during latency testing to prevent *Doze* from batching the notifications.

8.2.3 Battery Consumption

As described in Section 8.2.2, the *Doze* mode of Android stops applications during device sleep and disconnects their connections. This would prevent the push client from receiving messages during *Doze*. In order to prevent Android from disconnecting the push connection, the Android battery optimization setting for the push client was changed from “Optimize” to “Don’t optimize”.

Testing the battery consumption of the privacy-preserving push service was done by sending push messages at predefined times over either FCM or the privacy-preserving push service. The mobile device was fully charged before each test and the battery level and other information during the test were gathered via “adb bugreport”.

8.2.4 Latency Measurements

The results of comparing different settings for FCM with the privacy-preserving push service can be seen in Fig. 8.3. Note that the setting `time_to_live = 7w` and `priority = normal` is not included in this graph. This was done since it was discovered that with these settings FCM would aggregate push messages and send them in bulk. This causes latencies of several hours and does not show the actual performance of the push service. We can see that the median for Sense is below 2 seconds and that at least 75% of the messages arrive within 3 seconds.

The aggregation of messages can be seen in Fig. 8.4. For the non-aggregating settings, the histograms are the same, but for the setting `time_to_live = 7w` and `priority = normal`, messages arrive up to three hours later (top left).

To study the impact of using a Tor onion service for anonymisation, measurements were taken on the push server to see the latency from the push sender to the push server and the latency from the push server to the mobile device instead of just observing the latency between sender and mobile device. The results can be seen in Fig. 8.5. This graph shows that all high latency messages can be attributed to the Tor network introducing the latency.

8.2.5 Battery Consumption Measurements

The results of the battery consumption test can be seen in Fig. 8.6. We see that Sense uses more battery than FCM. Although no messages were sent over FCM, the FCM connection was active during the Sense test, since the Google framework is part of the firmware of the device. Furthermore, Sense is relatively aggressive with sending HTTP2 PING frames to keep the connection alive. A more intelligent approach could save more battery power (see Section 9.3).

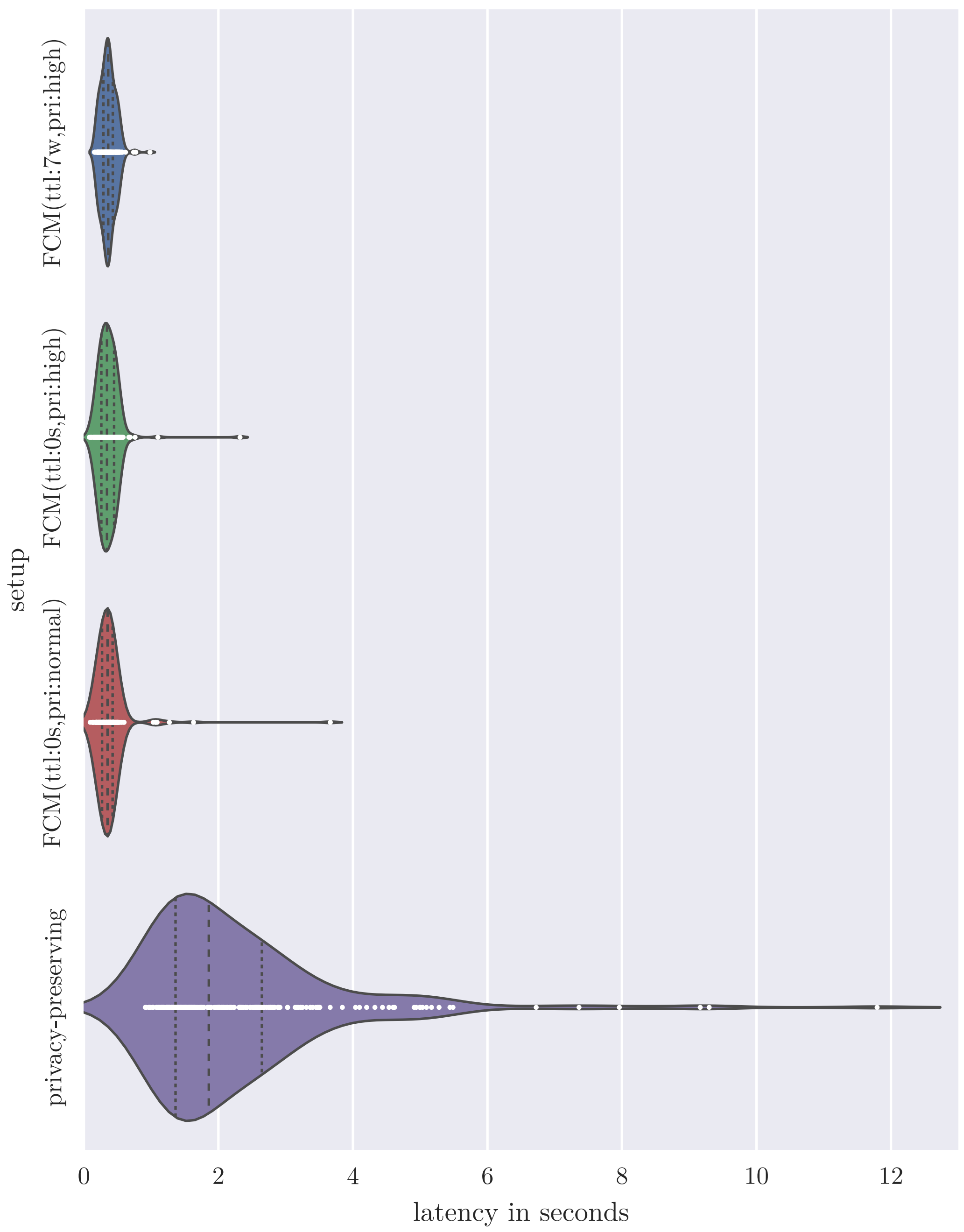

Figure 8.3: The results of the latency tests. White dots represent single measurements. The black dashed lines represent the quartiles, violins represent density.

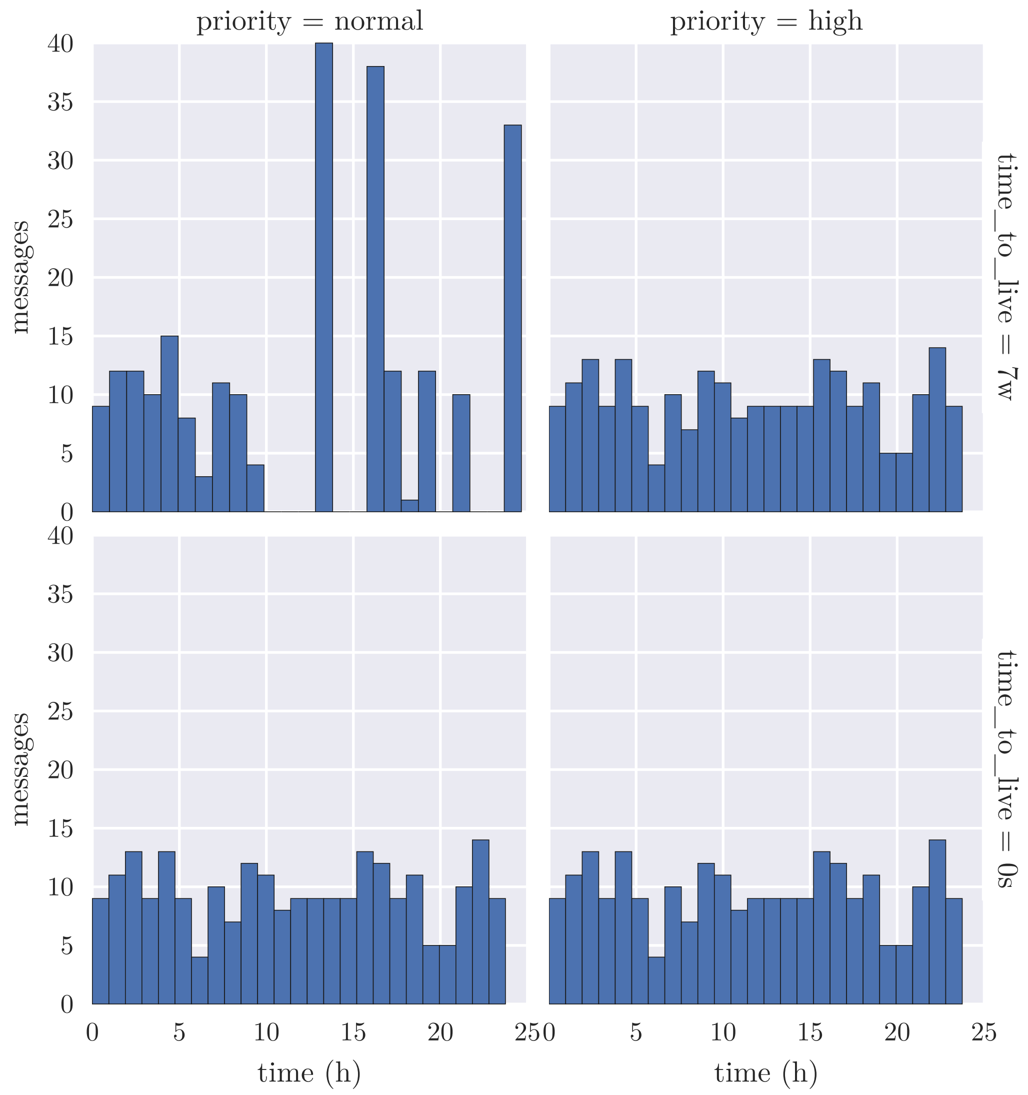

Figure 8.4: This is a histogram of the messages over time arriving for each of the tested settings for FCM. When using a TTL of 7 weeks and priority normal, messages might get aggregated (top left).

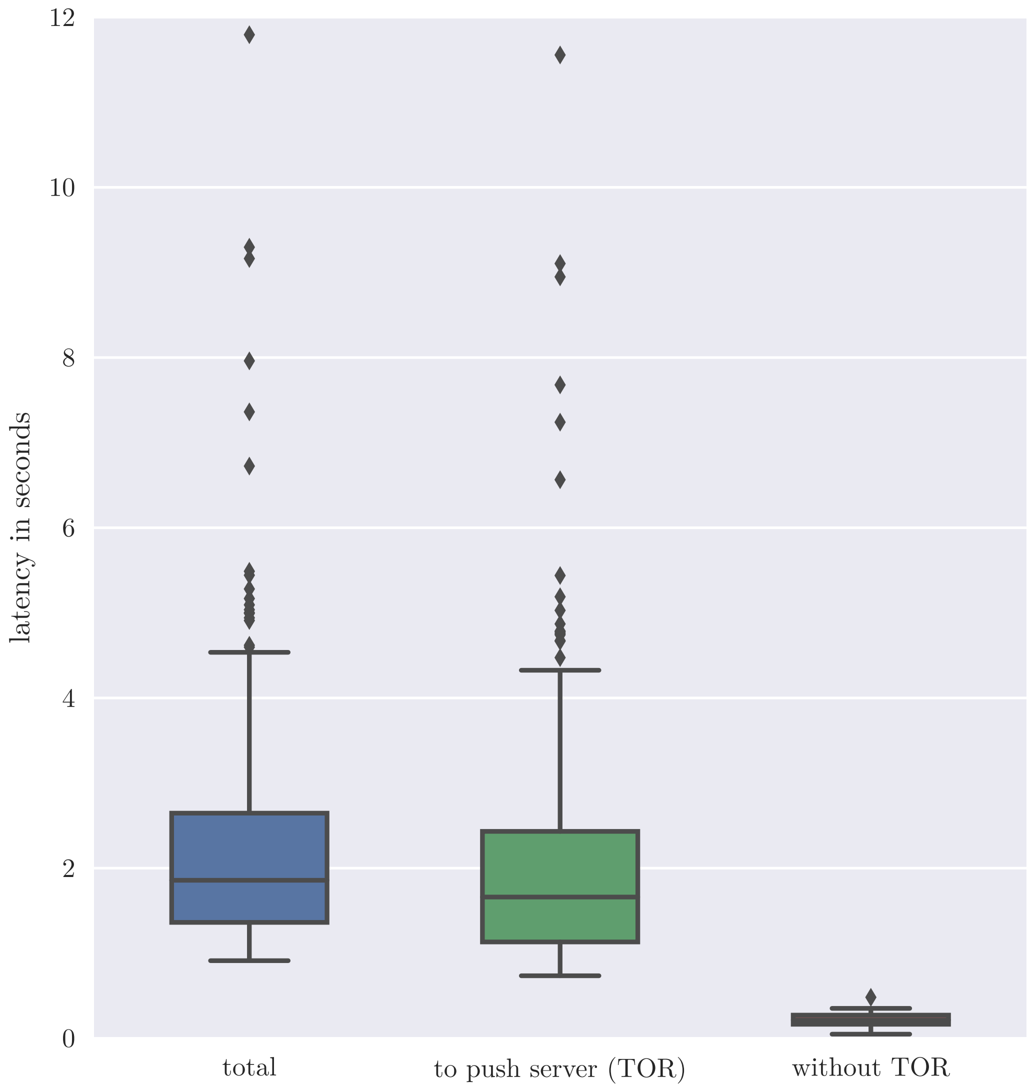

Figure 8.5: This boxplot shows the impact of Tor on the latency. The whiskers are set to maximum 1.5x inter quartile range, routes show outliers. The left box is the total latency, in the middle the latency to the push server via Tor and on the right the latency from the push server to the mobile device.

This data only covers a device connected to one network and not changing networks. If the push client has to reconnect, the device has to wake up and wait for the connection to be established, which needs battery power.

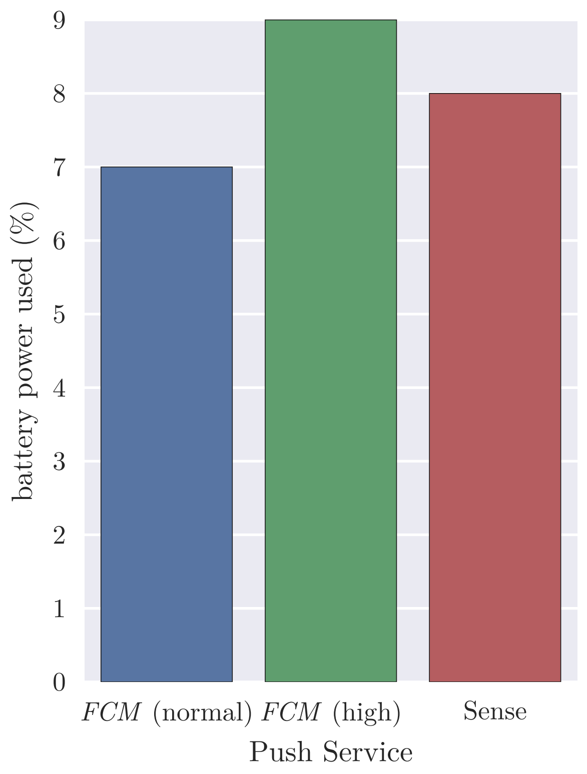

Figure 8.6: Battery consumption results of sending push messages for 24 hours. Graphs show difference of the reported battery power of the OS between before and after the test. FCM was tested with different priorities using the default `time_to_live` value.

8.3 Summary

We tested the latency of Sense against FCM and found that the battery consumption is comparable to FCM and the latency was, in 50% of the cases, below 2 seconds and, in 75% of the cases, below 3 seconds. Since an increase in latency was expected and the latency is most of the time lower than the call establishment of a mobile to PSTN call, we think that the latency is acceptable for notifications on mobile platforms.

Chapter 9

Future Work

In the previous chapter we had a look at how the implementation of the SENSE design compares to FCM. We saw that the latency increased and the battery consumption could be better. In this chapter, we want to present possible improvements that tackle these problems and an alternative architecture which improves the privacy further at the cost of complexity for the sender.

9.1 Alternative Architecture

A different architecture (see Fig. 9.1) would be to directly run the Tor hidden service on the mobile device. The push sender then connects directly to the hidden service running on the mobile device. This combines the push server and push client on the mobile device. This approach improves the privacy further, as the third party running the push server is removed. By creating an hidden service per app it is possible to remove access by not announcing the hidden service for this app.

Furthermore, this removes the necessity of a separate end-to-end security with the Signal Protocol as connections to Tor hidden services provide end-to-end security. The tokens can be replaced by a static secret as authentication.

This increased privacy comes at the price of additional burden for the push sender. The push sender needs to store messages and try to redeliver them if the service is not available.

This design also makes aggregation of messages to save battery more difficult, as the device would need to signal every push sender to send aggregated messages.

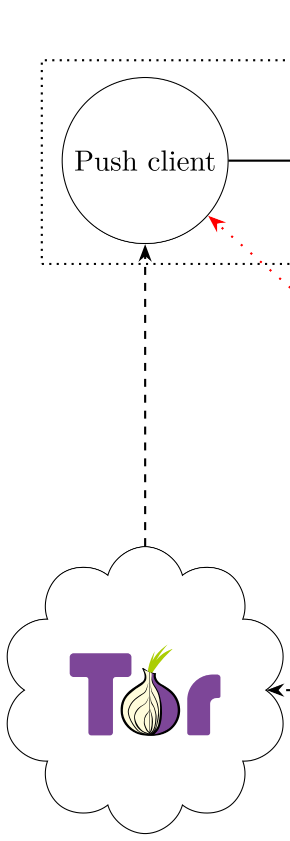

Figure 9.1: Architecture of the alternative design. Red (dashed or dotted) lines are encrypted.

9.2 Improvements to Tor

In Chapter 8 we saw that the Tor network had the biggest influence on the latency. This is not very surprising, considering that the message needs to pass 6 nodes to reach its destination. When sacrificing the anonymity of the push server this latency could be reduced with the approaches we present here.

9.2.1 Single Onion Services

If the anonymity of the push server could be sacrificed, the number of hops could be reduced. There were proposals to add “Single Onion Services” [63], [64] to Tor. Single Onion Services reduce the anonymity of the server, by reducing the hops to the server. The two approaches differ in the announcement of the server. While the “Rendezvous Single Onion Services” [63] use the introduction point and rendezvous nodes, the “Single Onion Services” [64] announce the relay responsible for the single onion service and allow the client to “directly” connect to the service. The drawback is that Single Onion Services need an externally reachable port, whereas Rendezvous Single Onion Services work behind NAT, but have higher latency. Using either approach would improve the latency of the push service, as the amount of Tor nodes involved is reduced.

9.2.2 Run Own Relays

Even when Single Onion Services are not an option, performance could be improved by running own relays and forcing the Tor client running the hidden service to use them. If all nodes run in the same network, the latency would most likely be smaller. This would come with the cost of reduced anonymity for the server.

9.2.3 Onion Balance

Onion Balance [65] distributes the load of a onion service over different endpoints. It does not have the goal to reduce the latency. Still, this approach could help reduce the load of the push service, as the traffic is distributed.

9.3 Keep-Alive and Heartbeats

Currently the push client sends hourly HTTP2 `PING` frames and on device idle changes. This is very aggressive, and might wake the device to often in some networks or might not suffice to keep the connection alive in other networks.

Additionally, using TCP keep-alive would be more efficient, as less data needs to be transferred.

A better approach would be to use short intervals in the beginning and increase the interval until the connection is dropped. Then use the last “good” interval to send the keep-alive or HTTP2 `PING` frame. This interval can then be remembered per network to skip the learning of the correct interval.

Google Cloud Messaging apparently uses a dynamic heartbeat interval[1] and it also seems to differentiate between mobile data and wireless LAN.

A good keep-alive strategy could improve the notification speed and ensure that notifications can be received, while minimizing battery and data usage.

9.4 Certificate Pinning in the Client

To increase the security against rogue certificate authorities the client and push server could use a similar system as is described in RFC 7469 [66]. The push server would tell the client which certificates in the certificate chain of the push server the client should pin and for what period. The idea is to provide certificate pinning, even when exchanging the push server is possible.

9.5 Aggregation of Push Messages

We saw in Chapter 8 that FCM aggregates push messages. Aggregating push messages helps saving battery, as it reduces the time the radio and the device need to stay awake. The current implementation of Sense does not implement aggregation. To ensure the best resource usage, the push client could inform the push server when the device exits idle and request all aggregated push messages.

[1] `http://stackoverflow.com/a/18428357`

Chapter 10

Conclusion

In the introduction, the problem of metadata collection was introduced. This problem cannot simply be solved by introducing end-to-end encryption or transport encryption. It requires protocol designs that reduce the metadata and ensure that only the information that each party needs is exposed to this party.

The existing push services analyzed in this thesis were not designed to minimize metadata on the push server. The related work did not provide a solution for mobile devices.

The privacy-preserving and secure notification service called SENSE does provide a solution to this problem. This service protects the privacy of the user by enforcing end-to-end security between the push sender and the push client, reducing the amount of metadata accrued at the push server, and using an anonymous authentication system that puts the user in control.

Furthermore, the new push service allows the user to choose which push service to use and trust with his personal data. Additionally, the user has control over the authorization of push messages, which lets the user — and not the push service provider — decide which push messages to receive.

The implementation of this design showed comparable battery performance to the push service of the Google Android platform, FCM, while keeping an acceptable latency of less than 2 seconds for 50% of the messages and less than 3 seconds for 75% of the messages.

With the possible future work presented in Chapter 9, we showed that future research could reduce the latency impact at the cost of anonymity of the push server or further increase the anonymity of the user, by directly delivering push messages to the mobile device over Tor hidden services.

The research in this field is just beginning and it is important to find solutions for private push notifications as these push services become more and more prevalent.

The amount of metadata accrued at Google, Apple and Microsoft will increase unless the architecture of these push services is changed. And with the increasing amount of metadata, these networks will become the focus of intelligence agencies.

With this work we hope we increased awareness to the privacy problems of push services and provided a usable approach for more privacy in mobile push services.

Bibliography

[1] O. Reißmann. (Aug. 2, 2013). Telefonüberwachung: Handy-daten verraten illegale cia-operation. S. Online, Ed., [Online]. Available: `http://www.spiegel.de/netzwelt/netzpolitik/telefon-ueberwachung-metadaten-verraten-illegale-cia-operation-a-914470.html` (visited on 11/05/2016).

[2] D. Cole. (May 10, 2014). 'we kill people based on metadata'. T. N. Y. R. of Books, Ed., [Online]. Available: `http://www.nybooks.com/daily/2014/05/10/we-kill-people-based-metadata/` (visited on 11/05/2016).

[3] S. Biddle. (Sep. 28, 2016). Apple logs your imessage contacts — and may share them with police. T. Intercept, Ed., [Online]. Available: `https://theintercept.com/2016/09/28/apple-logs-your-imessage-contacts-and-may-share-them-with-police/` (visited on 11/01/2016).

[4] B. Gellman and L. Poitras. (Jun. 6, 2013). NSA slides explain the PRISM data-collection program. T. W. Post, Ed., [Online]. Available: `http://www.washingtonpost.com/wp-srv/special/politics/prism-collection-documents/` (visited on 11/14/2016).

[5] H. F. Nielsen, R. T. Fielding, and T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.0*, RFC 1945, May 1996. DOI: `10.17487/rfc1945`.

[6] J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, RFC 2616, Jun. 1999. DOI: `10.17487/rfc2616`.

[7] R. T. Fielding and J. F. Reschke, *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, RFC 7231, Jun. 2014. DOI: `10.17487/rfc7231`.

[8] R. T. Fielding and J. F. Reschke, *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests*, RFC 7232, Jun. 2014. DOI: `10.17487/rfc7232`.

[9] R. T. Fielding, M. Nottingham, and J. F. Reschke, *Hypertext Transfer Protocol (HTTP/1.1): Caching*, RFC 7234, Jun. 2014. DOI: `10.17487/rfc7234`.

[10] R. T. Fielding and J. F. Reschke, *Hypertext Transfer Protocol (HTTP/1.1): Authentication*, RFC 7235, Jun. 2014. DOI: `10.17487/rfc7235`.

[11] M. Belshe, R. Peon, and M. Thomson, *Hypertext Transfer Protocol Version 2 (HTTP/2)*, RFC 7540, Nov. 2015. DOI: `10.17487/rfc7540`.

[12] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," DTIC Document, Tech. Rep., 2004.

[13] Tor Project. (). Tor: Overview, [Online]. Available: `https://www.torproject.org/about/overview.html.en` (visited on 11/05/2016).

[14] L. Overlier and P. Syverson, "Locating hidden servers," in *2006 ieee symposium on security and privacy (s p'06)*, May 2006, p. 15. DOI: `10.1109/SP.2006.24`.

[15] S. Boonkrong and P. C. Dinh, "The comparison of impacts to android phone battery between polling data and pushing data," in *International conference on computer networks and information technology (iccnit 2013)*, Unpublished, 2013. DOI: `10.13140/2.1.4731.7447`.

[16] A. Müller, F. Wohlfart, and G. Carle, "Analysis and topology-based traversal of cascaded large scale nats," in *Proceedings of the 2013 workshop on hot topics in middleboxes and network function virtualization*, ser. HotMiddlebox '13, Santa Barbara, California, USA: ACM, 2013, pp. 43–48, ISBN: 978-1-4503-2574-5. DOI: `10.1145/2535828.2535833`.

[17] S. Triukose, S. Ardon, A. Mahanti, and A. Seth, "Geolocating ip addresses in cellular data networks," in *Passive and active measurement: 13th international conference, pam 2012, vienna, austria, march 12-14th, 2012. proceedings*, N. Taft and F. Ricciato, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 158–167, ISBN: 978-3-642-28537-0. DOI: `10.1007/978-3-642-28537-0_16`.

[18] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An untold story of middleboxes in cellular networks," in *Proceedings of the acm sigcomm 2011 conference*, ser. SIGCOMM '11, Toronto, Ontario, Canada: ACM, 2011, pp. 374–385, ISBN: 978-1-4503-0797-0. DOI: `10.1145/2018436.2018479`.

[19] Google. (May 18, 2016). Google cloud messaging: Overview | cloud messaging | google developers, [Online]. Available: `https://developers.google.com/cloud-messaging/gcm` (visited on 07/10/2016).

[20] Google. (Jan. 14, 2016). Registering client apps | cloud messaging | google developers, [Online]. Available: `https://developers.google.com/cloud-messaging/registration` (visited on 07/10/2016).

[21] Google. (Sep. 21, 2015). What is instance id? | instance id | google developers, [Online]. Available: `https://developers.google.com/instance-id/` (visited on 07/10/2016).

[22] Google. (Mar. 11, 2016). Messaging concepts and options | cloud messaging | google developers, [Online]. Available: `https://developers.google.com/cloud-messaging/concept-options` (visited on 07/10/2016).

[23] Apple. (Oct. 24, 2016). Apns overview, [Online]. Available: `https://developer.apple.com/library/content/documentation/NetworkingInternet/`

Conceptual/RemoteNotificationsPG/APNSOverview.html (visited on 11/07/2016).

[24] D. Melanson. (Jun. 9, 2008). Iphone push notification service for devs announced. Engadget, Ed., [Online]. Available: https://www.engadget.com/2008/06/09/iphone-push-notification-service-for-devs-announced/ (visited on 11/06/2016).

[25] N. Patel. (Mar. 31, 2009). Iphone 3.0 beta 2 released, push notifications are a go. Engadget, Ed., [Online]. Available: https://www.engadget.com/2009/03/31/iphone-3-0-beta-2-released-push-notifications-are-a-go/ (visited on 11/06/2016).

[26] A. Parsovs, "Practical issues with TLS client certificate authentication," in *Proceedings 2014 network and distributed system security symposium*, Internet Society, 2014. DOI: 10.14722/ndss.2014.23036.

[27] M. Jacobs. (Feb. 8, 2016). Windows push notification services (wns) overview. Microsoft, Ed., [Online]. Available: https://msdn.microsoft.com/en-us/windows/uwp/controls-and-patterns/tiles-and-notifications-windows-push-notification-services--wns--overview (visited on 11/02/2016).

[28] Microsoft. (2016). Windows push notification services (wns) overview (windows runtime apps), [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh913756 (visited on 11/02/2016).

[29] Google. (2016). About fcm messages | firebase, [Online]. Available: https://firebase.google.com/docs/cloud-messaging/concept-options (visited on 09/06/2016).

[30] "Information technology — Security techniques — Information security management systems — Overview and vocabulary," International Organization for Standardization, Geneva, CH, Standard, Feb. 15, 2016, p. 34.

[31] A. Cooper, H. Tschofenig, D. B. D. A. Ph.D., J. Peterson, J. B. Morris, M. Hansen, and R. Smith, *Privacy Considerations for Internet Protocols*, RFC 6973, Jul. 2013. DOI: 10.17487/rfc6973.

[32] R. W. Shirey, *Internet Security Glossary, Version 2*, RFC 4949, Aug. 2007. DOI: 10.17487/rfc4949.

[33] "Information technology — Security techniques — Privacy framework," International Organization for Standardization, Geneva, CH, Standard, Dec. 15, 2011, p. 28.

[34] A. Piotrowska, J. Hayes, N. Gelernter, G. Danezis, and A. Herzberg, *Anonotify: A private notification service*, Cryptology ePrint Archive, Report 2016/466, http://eprint.iacr.org/2016/466, 2016.

[35] S. A. V. A. J. Menezes Paul C. van Oorschot, *Handbook of applied cryptography.* Taylor & Francis Inc, Oct. 16, 1996, 810 pp., ISBN: 0849385237.

[36] D. Iland and C. Munger, “Secure resilient messaging for android devices,” 2012.

[37] J. D. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski, *Android hacker’s handbook*. Wiley, 2014, 576 pp., ISBN: 978-1-4920-0583-4.

[38] M. Thomson, E. Damaggio, and B. Raymor, “Generic Event Delivery Using HTTP Push,” Internet Engineering Task Force, Internet-Draft draft-ietf-webpush-protocol-11, Oct. 2016, Work in Progress, 33 pp.

[39] J. Medley. (Oct. 13, 2016). Web push notifications: Timely, relevant, and precise. Google, Ed., [Online]. Available: https://developers.google.com/web/fundamentals/engage-and-retain/push-notifications/ (visited on 10/13/2016).

[40] Mozilla. (Sep. 26, 2016). Push api, [Online]. Available: https://developer.mozilla.org/de/docs/Web/API/Push_API (visited on 10/14/2016).

[41] Mozilla. (Sep. 13, 2016). Using service workers, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers (visited on 10/14/2016).

[42] M. v. Ouwerkerk, M. Thomson, B. Sullivan, and E. Fullea, “Push api,” World Wide Web Consortium, Tech. Rep., Dec. 15, 2015.

[43] M. Thomson, “Message Encryption for Web Push,” Internet Engineering Task Force, Internet-Draft draft-ietf-webpush-encryption-04, Oct. 2016, Work in Progress, 12 pp.

[44] M. Thomson and P. Beverloo, “Voluntary Application Server Identification for Web Push,” Internet Engineering Task Force, Internet-Draft draft-ietf-webpush-vapid-01, Jun. 2016, Work in Progress, 11 pp.

[45] J. Brustel and T. Preuss, “A universal push service for mobile devices,” in *2012 sixth international conference on complex, intelligent, and software intensive systems*, Institute of Electrical & Electronics Engineers (IEEE), Jul. 2012. DOI: 10.1109/cisis.2012.105.

[46] Q. Scheitle, M. Wachs, J. Zirngibl, and G. Carle, “Analyzing locality of mobile messaging traffic using the matador framework,” in *Passive and active measurements conference (pam) 2016*, Heraklion, Greece, Mar. 2016. DOI: 10.1007/978-3-319-30505-9_15.

[47] T. Dierks, *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC 5246, Oct. 2015. DOI: 10.17487/rfc5246.

[48] T. Perrin and M. Marlinspike. (Jun. 2014). Axolotl ratchet, [Online]. Available: https://github.com/trevp/axolotl/wiki (visited on 11/06/2016).

[49] M. Marlinspike. (Mar. 30, 2016). Signal on the outside, signal on the inside. O. WhipserSystems, Ed., [Online]. Available: https://whispersystems.org/blog/signal-inside-and-out/ (visited on 11/04/2016).

[50] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, *A formal security analysis of the signal messaging protocol*, Cryptology ePrint Archive, Report 2016/1013, `http://eprint.iacr.org/2016/1013`, 2016.

[51] M. Marlinspike. (Nov. 26, 2013). Advanced cryptographic ratcheting. O. WhipserSystems, Ed., [Online]. Available: `https://whispersystems.org/blog/advanced-ratcheting/` (visited on 11/04/2016).

[52] D. H. Krawczyk, M. Bellare, and R. Canetti, *HMAC: Keyed-Hashing for Message Authentication*, RFC 2104, Feb. 1997. DOI: `10.17487/rfc2104`.

[53] P. J. Leach, R. Salz, and M. H. Mealling, *A Universally Unique IDentifier (UUID) URN Namespace*, RFC 4122, Mar. 2013. DOI: `10.17487/rfc4122`.

[54] S. Josefsson, *The Base16, Base32, and Base64 Data Encodings*, RFC 4648, Oct. 2006. DOI: `10.17487/rfc4648`.

[55] J. Blandy, "Why rust? Trustworthy, concurrent systems programming," O'Reilly Media, Tech. Rep., Sep. 4, 2015.

[56] D. Kegel. (2014). The c10k problem, [Online]. Available: `http://www.kegel.com/c10k.html` (visited on 11/02/2016).

[57] L. Gammo, T. Brecht, A. Shukla, and D. Pariag, "Comparing and evaluating epoll, select, and poll event mechanisms," in *Linux symposium*, vol. 1, 2004.

[58] Mozilla. (Feb. 23, 2016). Security/server side tls, [Online]. Available: `https://wiki.mozilla.org/Security/Server_Side_TLS` (visited on 10/13/2016).

[59] Linux Foundation. (2016). Let's encrypt - free ssl/tls certificates, [Online]. Available: `https://letsencrypt.org/` (visited on 10/14/2016).

[60] M. D. Leech, *SOCKS Protocol Version 5*, RFC 1928, Mar. 1996. DOI: `10.17487/rfc1928`.

[61] A. Mishra, *Fundamentals of cellular network planning and optimisation: 2g/2.5g/3g... evolution to 4g*, piegel, Ed. Wiley, 2004, ISBN: 978-0-4708-6268-1.

[62] Google. (2015). Optimizing for doze and app standby - android developers, [Online]. Available: `https://developer.android.com/training/monitoring-device-state/doze-standby.html` (visited on 11/02/2016).

[63] T. Wilson-Brown, J. Brooks, A. Johnson, R. Jansen, G. Kadianakis, P. Syverson, and R. Dingledine. (Oct. 17, 2015). Rendezvous single onion services, [Online]. Available: `https://gitweb.torproject.org/torspec.git/tree/proposals/260-rend-single-onion.txt` (visited on 11/02/2016).

[64] J. Brooks, P. Syverson, and R. Dingledine. (Jul. 13, 2015). Single onion services, [Online]. Available: `https://gitweb.torproject.org/torspec.git/tree/proposals/252-single-onion.txt` (visited on 11/02/2016).

[65] D. O. Cearbhaill. (2016). Onionbalance, [Online]. Available: `https://github.com/DonnchaC/onionbalance` (visited on 11/02/2016).

[66] C. Evans, C. Palmer, and R. Sleevi, *Public Key Pinning Extension for HTTP*, RFC 7469, Oct. 2015. DOI: `10.17487/rfc7469`.