Informatics

Open Source and Privacy Aware Push Notifications for Mobile Phones

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Harald Jagenteufel, BSc
Matrikelnummer 0826995

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Mitwirkung: Georg Merzdovnik, PhD

Wien, 24. April 2020

Harald Jagenteufel | Edgar Weippl

Informatics

Open Source and Privacy Aware Push Notifications for Mobile Phones

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Harald Jagenteufel, BSc
Registration Number 0826995

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Assistance: Georg Merzdovnik, PhD

Vienna, 24th April, 2020

Harald Jagenteufel | Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Harald Jagenteufel, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. April 2020

Harald Jagenteufel

Danksagung

Im Schreiben dieser Arbeit haben mich viele Menschen unterstützt. Besonders hervorheben möchte ich folgende Unterstützer:

Monika, my Love, die mich gegen Ende meiner Arbeit besonders unterstützt hat. Sie hat sich viel Zeit genommen, um diese Arbeit Korrektur zu lesen. Noch viel wichtiger allerdings waren ihre unzähligen Tipps über effizientes Arbeiten und ihre Aufmunterungen, nicht die Motivation zu verlieren.

Andi, der mich einen guten Teil meines Studiums begleitet und unterstützt hat. Er war stets für angeregte Diskussionen offen und hat mir oft ausgesprochen hilfreiche Ideen vermittelt, ohne jene ich zur Lösung mancher Probleme wesentlich länger gebraucht hätte. Ich bin für seine Unterstützung als ausgesprochen guter Freund sehr dankbar.

Vera, meine Lieblingsschwester, die immer für mich da war. Ich bin ihr sehr dankbar, dass sie viele Stunden investiert hat um diesen Text zu lesen und mit ihrer sprachliche Expertise Anmerkungen zu machen. (Zum Glück wollte ich keine Applikationen *adoptieren*.)

Philipp, der durch unorthodoxe Mittel meine Motivation im Endspurt der Arbeit hoch gehalten hat. Leider hat er dennoch seine Wette verloren.

Meine Familie, insbesondere meine Eltern, die mir eh und je geholfen haben, mich unterstützt haben und nie die Hoffnung verloren haben. Ohne euch wäre ich nicht so weit gekommen.

Martin und Georg. Sie waren eine unermüdliche Unterstützung, die es selbst nach Jahren nicht leid waren, sich des Themas dieser Arbeit anzunehmen.

Last but not least SBA Research und Edgar Weippl für die Möglichkeit das Projekt in dieser Form umsetzen zu können.

Kurzfassung

Instant Messanging Dienste werden immer populärer. Die Metadaten, die diese Dienste erzeugen, sind äußerst persönlich und daher sehr schützenswert. Obwohl die Verwendung von open-source Software wie *Signal* oder *Wire* hier helfen kann, weniger Daten an Dritte weiterzugeben, löst sie das Problem dennoch nicht ganz. Selbst diese quelloffenen Programme verwenden proprietäre Push Messaging Dienste Dritter, wie etwa von Google oder Apple, um neue Nachrichten einfach und ohne große Verzögerung an die Endgeräte der Benutzer weiterzuleiten. Solche zentralen Dienste sind auch notwendig um die Batterielaufzeit hoch und den Datenverbrauch der Geräte klein zu halten.

Um auch für diese wichtigen Dienste eine open-source Alternative zur Verfügung zu haben stelle ich in dieser Arbeit eine eigene Implementierung vor: *Push Adapter*. Sie ist eine quelloffene Platform, um Pushnachrichten zentral an Android Endgeräte zu übermitteln. Dennoch kann Push Adapter von Unabhängigen betrieben und benutzt werden. Es wird sowohl ein Dienst zur Übermittelung der Nachrichten, die benötigten Android Clients als auch eine Bibliothek zum einfachen Einbinden dieses Dienstes in bestehende Apps vorgestellt.

Um die Praxistauglichkeit meines Ansatzes unter Beweis zu stellen wird er in die open-source instant messaging App *Wire* integriert, diese Integration wird im Detail beschrieben. Anhand dieser Integration wird weiters mein Dienst mit Googles *Firebase Cloud Messaging* verglichen. Es werden Messungen der Akkulaufzeit und des Datenverbrauchs von Wire unter Verwendung beider Dienste vogenommen. Diese zeigen einen klaren Vorteil zu Gunsten von Push Adapter beim Datenverbrauch. Die Ergebnisse zur Batterielaufzeit zeigen eine höhere Zuverlässigkeit von Push Adapter und gleichzeitig einen vergleichbaren Energieverbrauch. Meines Wissens nach ist Push Adapter das erste quelloffene System, das versucht, Firebase Cloud Messaging für bestehende Apps zu ersetzen.

Abstract

The popularity of instant messaging services is ever rising. The metadata exposed via the usage of these services is personal and therefore very sensitive. Open source services like *Signal* or *Wire* try to improve in this regard by going some lengths to protect their users privacy. Still, even when using these services, part of the communication metadata is shared with third parties through the use of proprietary push messaging services. These services are critical for waking up devices when new messages for the end user arrive, to download the message and display a notification. Not using these central services comes at the cost of higher network and battery usage on users' end devices.

To remedy this issue, I present *Push Adapter*. It is an open source platform for Android apps to allow pushing of messages via a service that can be hosted by any interested party. A persistent connection is shared for receiving push messages, minimizing battery and network data overhead. Furthermore, it includes an Android library to ease integration in apps that currently employ Googles' Firebase Cloud Messaging (FCM) service to receive push notifications.

To prove the usability of Push Adapter, the open source instant messaging app *Wire* was forked. It was retrofitted to push messages via Push Adapter instead of FCM. This implementation is described and evaluated. My measurements of battery usage and network bandwidth overhead using the original version of Wire and my fork show that Push Adapter incurs only half of the network overhead when compared to FCM. Additionally, battery lifetime of an Android device was evaluated, which showed that Push Adapter can deliver messages more reliably than FCM while using comparable amounts of the device's battery. To my knowledge, Push Adapter is the first open source implementation of a push notification service that tries to replace FCM in its entirety. Furthermore, Push Adapter is designed to be easily retrofitted with existing apps that currently use FCM.

Contents

CHAPTER 1

Introduction

Instant messenger services are very popular. The most popular one, *WhatsApp*, has about 1.5 billion users worldwide as of January 2018 [80]. It is a proprietary service. Such services are often accused of not properly respecting the privacy of their users [31]. This causes many users to move to open source alternatives like *Signal* or *Wire*. While these services aim to improve their users' privacy and security, they still depend on third-party services for efficient implementation of some of their features. One such feature are push notifications.

Importance of centralized Push Services There are a number of reasons why centralized push services are useful:

- They provide developers easy means to distribute messages from their backend service to their customers' mobile devices.
- Messages can arrive even when the end user's device is not being used actively, without heavily impacting the device's battery usage.

Many leading cloud service providers offer push services; *Amazon Device Messaging (ADM)*, *Apple Push Notification service (APNs)*, *Windows Push Notifications Service (WNS)* and *Firebase Cloud Messaging (FCM)* are examples of such services.

Whenever the end user's device is unused, it enters a standby mode and tries to preserve battery power. Connections over the internet are power costly and therefore reduced to a minimum in the standby mode. The mobile operating system *Android* enforces this by the so called *Doze* mode [58]. Google provides a service, *Firebase Cloud Messaging (FCM)*, which allows app developers to wake up their customers' devices remotely to receive updates from the apps' backend services. This is even possible while their end user's device is in Doze mode. Such a centralized service is very useful: it allows the device to share a single connection to Google's servers for all apps running on it. This

allows to reduce the number of open connections while still providing timely message delivery.

Although Wire does not transmit any not end-to-end encrypted data via FCM, the push service provider can still collect metadata about its users. This leakage of metadata about communication patterns can have adverse effects on users' privacy [49]. There have been cases of proprietary services and libraries not function as users might expect. This can have a severe impact on users' privacy. For example, there are reported cases where location tracking on Apples iOS and Googles Android was conducted even though users had disabled that functionality [9, 6]. Even privacy-aware apps such as Signal and Wire still prefer to use central push services. Although they provide alternatives on a per-app basis, such as websocket connections to their own backend services, they come at a high cost. Having multiple open connections for receiving push messages uses a lot more power on end users' devices. Not using such a connection, however, keeps the app from receiving messages in a timely manner and therefore provides a worsened user experience. Thus, privacy-aware users have to decide on a trade-off between usability and privacy.

Open Source Alternatives All the aforementioned push service providers host their service on proprietary platforms and are closed source. Therefore, they cannot be hosted by independent providers. In the case of Android, the proprietary libraries *Google Play services* need to be installed on end users' devices to receive FCM push notifications [59]. The Android open source developer community has started to tackle the problem by providing free and open source alternatives to most of the services provided by Google for Android devices. The *microG Project*[51] has worked on implementing open source alternatives to the proprietary Google Android libraries. While parts of the libraries have been replaced by software that is independent from Google services[1], the part that implements FCM support still uses the official Google service. Open source developers have voiced a demand for an open source alternative that does not use Google's services [36].

My work fills this need by providing an open source alternative to FCM - secure, privacy aware and independent from proprietary software and services. I implement and evaluate a prototype implementation of a push messaging service. It includes a backend for relaying messages to end users' devices as well as the necessary client apps and libraries. The service is easy to retrofit for apps that already use FCM. The client libraries resemble the API of FCM as close as possible. The REST API for sending messages via my service is built in a way compatible to FCM. To show the ease of adaption, I add support for my service to *Wire*. Specifically, I adapt the backend service and Android app of Wire to interface with Push Adapter for sending and receiving push messages. I explain and evaluate this implementation. I conduct detailed measurements of the battery power consumption and data traffic overhead of my implementation on a smartphone. These

[1] e.g. *Unified Network Location Provider*, which provides on-device geolocation support based on Cell-tower and Wi-Fi data

measurements show that my implementation outperforms FCM regarding per-message overhead. It induces less network traffic overhead than FCM while actively relaying messages. It incurs roughly the same low network data usage for maintaining an active connection when idle. Furthermore, I reveal that FCM trades reliability for better battery life. To my knowledge, this is the first open source, easy to use alternative to Google's FCM that exists to date.

The following chapter (2) gives an overview of the current state of the art. I present the work of others in the fields of privacy, security of instant messaging and push notifications for mobile applications and MQTT (Message Queuing Telemetry Transport). Chapter 3 presents background information. I show how the push notification service provided by Google, FCM, works. Another promising alternative to this prevalent service is introduced as well. An introduction to the concepts of the MQTT protocol is given, as it is the basis for my implementation. Chapter 4 gives an overview of the Push Adapter and its architecture. It states the goals for the implementation as well as an outline on how to reach these goals. Chapter 5 introduces the actual implementation. It shows the various design decisions and how they are solved within my implementation. The next chapter, 6, shows how my proof of concept was conducted. It shows how Wire, a real-world instant messaging service, works and what was needed to adapt it for Push Adapter. The evaluation method is presented in Chapter 7. It gives detailed explanations on the way my measurements are conducted. Results are presented in Chapter 8. In Chapter 9, conclusions are drawn. Future research topics can be found in Chapter 10.

CHAPTER 2

State Of The Art

Here I will present related, previous research. This will include papers in the fields of privacy, instant messenger security, push notifications for mobile applications and MQTT.

2.1 Privacy

Protecting users' privacy is a major motivation factor for my work. Various methods and countermeasures to track users have been found and analyzed in previous works.

Heurix et al. introduce a taxonomy to compare Privacy Enhancing Technologies (PETs) [35]. Zimmermann discusses several PETs and evaluates them using this taxonomy [88]. My work, although increasing users' privacy, is not classifiable as a general PET. It provides an open source alternative to a proprietary service. Merzdovnik et al. analyze popular browser extensions and android techniques for blocking third party web trackers. For analyzing browser extensions, they crawled the Alexa Top 200,000 websites in different browser configurations. Five browser extensions were evaluated in this manner. On Android, the network requests of 10,000 apps were analyzed and compared to two DNS filter lists and rule sets from *Adblock Plus for Android* [50]. [26] presents a recent literature review on web tracking papers. It categorizes the reviewed papers, distinguishing between papers focusing on technological aspects, privacy concerns and commercial aspects of web tracking.

2.2 Security and Privacy in IoT

Mobility and push messages are part the *Internet of Things (IoT)*. Security and privacy in this field is regarded as an important topic with major challenges. [63, 69, 86, 48] surveyed the state of security and privacy in the IoT world in years 2014, 2015, 2017 and 2019 respectively. These papers state that in 2019, privacy is still an open issue in

the IoT domain. [19, 38] are survey papers that focus specifically on privacy. My work picks up the topic of privacy for the use case of push messages to mobile devices. [11, 10, 12] analyze attacks on privacy in the field of smart homes. They state that, even when encrypting communication, internet service providers (ISP) and other passive adversaries with similar capabilities can retrieve sensitive, private information. My work focuses on a different adversary, the host of a mobile push notification relay.

2.3 Security in Instant Messengers

The instant messaging app *Signal* and its protocol with the same name have been analyzed by a few parties. Since *Wire*, the app that I base my work on, is very similar in functionality, some results of these analyses carry over to Wire. These research results prove that Signal and, by the mentioned extension, Wire, are worthwhile research items when it comes to privacy aware messaging. My work focuses on improving the privacy of Wire, and not on other security aspects of the app, which are discussed in the following papers.

[32] evaluates the cryptographic protocol of TextSecure and discusses the main security claims. TextSecure is the direct predecessor of Signal. It concludes that TextSecure holds most of its security properties, and, besides an unknown key-share (UKS) attack, TextSecure provides authenticated encryption. In this regard Moxie Marlinspike contradicted the severity of the issue [1]. [21] formalizes the Signal protocol by reverse engineering a formal description from its implementation. It then discusses the protocol from a security perspective. [45] introduces a way to automate cryptographic protocol verification. It implements a variant of the Signal protocol and analyzes it with regard to several security properties. [40] compares various instant messenger applications, including Signal and Wire. It provides a comprehensive overview of the security features of each application.

[64] shows several attacks on Signal. The authors present five adversaries that Signal should incorporate in their threat model. These adversaries are: a passive ISP (Internet Service Provider), an active ISP, a Signal user participating in a conversation, an adversary that can mount a MITM (man-in-the-middle) attack and finally one that can corrupt the Signal servers. The paper shows that attachments loaded via the *Giphy* service integration of Signal are not confidential in most cases. An adversary can fingerprint the victims traffic and obtain high confidence on the chosen image and the search term. Other attacks that can be carried out by a corrupted server or a MITM attacker include dropping messages unnoticed by sender and receiver as well as reordering messages. The authors further demonstrate various attacks on Signal's group messaging protocol. [22] show that the Signal Protocol does not provide post-compromise security for group chats. They provide a design which improves those properties, showing that secure post-compromise group messaging can be achieved. [62] analyzes the security of group communication in Signal, WhatsApp and Threema. They mention Wire as an interesting target for similar analysis. It shows that Future Secrecy does not hold for Signals group communication protocol. Furthermore, they show attacks on group communication in

Signal: Group conversations can be joined without being invited. A malicious server could forge the receipt status of group members. A malicious server can deliberately influence the ordering of messages.

[76] follows a systematic approach to evaluating secure messaging solutions. It proposes a framework to evaluate communication technologies in the dimensions security, usability and ease-of-adoption. It also discusses privacy properties of the evaluated solutions. It mentions *TextSecure*, the direct predecessor of Signal, as an example for various protocols and approaches that were evaluated. The analysis is comprehensive and generalizes on protocols and approaches rather than evaluating specific apps. It classifies various distributed protocols with regard to privacy, usability and adoption. As such it is a very comprehensive study on security and privacy of instant messaging technologies.

My work focuses on a different aspect of privacy. As mentioned in [76], metadata can not easily be hidden from service operators. My work tries to remedy this problem by providing users with a solution to self-host part of the infrastructure needed to provide an instant messaging service, therefore not producing metadata on untrusted services altogether.

2.4 Push Technology

Security Aspects Push services are complex and, if not used exclusively for notifications as my implementation and proof of concept is intended to do, can have critical security flaws. The following papers have conducted research in finding and mitigating attacks for such critical use cases. [47] discusses several security flaws in Google Cloud Messaging (GCM), Amazon Device Messaging (ADM) and an unnamed Chinese push service provider. They unveil that communication between the push service and the Android device as well as handling of messages on the device are error prone, can leak sensitive information or allow attackers to control critical device functionality, such as conducting a factory reset of the device. [17] describes *AUPS*, a system for authenticated publish/subscribe. It is built on top of the MQTT protocol. It improves the security of MQTT by adding a key management framework as well as a framework to enforce policies for authentication and authorization of data sources. It focuses on a general way to transmit data via MQTT while enforcing configurable access policies on MQTT topics using temporary keys. The implementation is evaluated with regard to latency of the overall system as well as memory and computational overhead and usage on the device hosting the MQTT relay. My work focuses on a specific mobile application of MQTT. While also enforcing access policies on MQTT topics, my work uses MQTT in a different way. The primary use case of my work is to notify mobile devices of a new remote state. Therefore, I evaluate my implementation with regard to amount of data transmitted to mobile clients and battery usage on those clients. Furthermore, the access policies for my use case require fixed access policies that must not be changed by administrators and are guarded by long-lived TLS client certificates. The source code to AUPS, as linked in the paper, could not be retrieved.

The authors of [18] propose an additional authentication mechanism using tokens. They argue that this method is superior to username/password combinations, as it provides better security and scalability. Their system design consists of four elements: a publisher, a subscriber, a MQTT broker and a token authentication server. They argue that their testing shows that their design fulfills the performance requirements for such a system, and can validate tokens in under one second.

Performance Aspects [46] compares different push mechanisms, including SMS/MMS, frequent polling and persistent TCP/IP connections. The comparison includes APNs and *Google Cloud Messaging (GCM)*, the predecessor of FCM. It evaluates battery and bandwidth consumption in a very basic way, not giving any measurements results other than *high* and *low* and not explaining how those results were attained. The authors of [28] built a push message system on top of XMPP. They evaluated the performance of their framework and compare it to GCM, APNs and Microsoft Push Notification Service (MPNS), showing that it is able to compete with those other services. [20] evaluates GCM, Gexin, JPush and ZYPush. The paper measures the performance regarding message delay, data compression, network traffic usage and heartbeat frequency. [53] compares the protocols nCoAP, RabbitMQ, MQTT, XMPP, SOAP and Modbus. It evaluates the memory usage, average delay of messages and whether messages are guaranteed to be received in order and reliably. [87] conducts a performance analysis of GCM in real-world usage scenarios. It shows that message delivery is not reliable and a rather slow arrival delay of upt to 10 seconds. My work focuses less on the evaluation and comparison of existing push services and more on providing a privacy friendly alternative to those centralized systems. I evaluate my implementation giving detailed results on mobile energy consumption and network bandwidth usage. [72] states MQTT as a solution for pushing messages to mobile devices. The paper includes basic measurements of energy consumption and data usage. It lacks detailed explanations of how to implement such a system or how the data and energy usages was measured. My work focuses on providing an open-source, available prototype implementation. Therefore, my work improves on this paper and goes into more detail on the actual implementation of such a system.

CHAPTER 3

Background

Before I describe *Push Adapter*, my implementation to provide push messages for Android devices, I will give an overview of existing technologies, which relate to my work. This includes the existing push message service *FCM (Firebase Cloud Messaging)* and *OwnPush*, a push notification service for Android. I also present the workings of MQTT (MQ Telemetry Transport), a protocol to relay messages, which I use to implement Push Adapter on top of.

3.1 FCM - Firebase Cloud Messaging

Firebase Cloud Messaging (FCM) is a service provided by Google. It allows sending of push messages for various platforms including browsers and mobile phones.

3.1.1 Architecture

Here I present an overview of the architecture of FCM. This section is based on the available developer documentation [59] as well as papers [47] and [78]. How Google operates the service and the internal workings are not part of my work.

FCM supports clients on iOS, Android, the Unity framework and bindings for C++ and JavaScript. Sending messages to clients is supported via a REST API [3]. The following provides some details on the API provided for Android clients.

Dependencies Apps using FCM need the *Android SDK for Firebase*. Parts of this SDK are available under the Apache 2 license [27]. The library `com.google.firebase:-firebase-messaging` needed for FCM is, however, only available under the `Android Software Development Kit License Agreement` [5]. Furthermore, a Google Account is needed to use FCM.

FirebaseMessagingService The class `FirebaseMessagingService` plays a core role when handling FCM push messages. It needs to be subclassed by apps that want to receive push messages. The entry point for receiving push messages is the method `onMessageReceived`. It receives the data sent by the apps' server through the only parameter `RemoteMessage`. FCM wakes up apps when a high priority message is pending for them. It allows those apps a few seconds to process the message. Furthermore, it allows network communication for this short period. This is true even if the device is in Doze mode, where apps are restricted from using background data or CPU time.

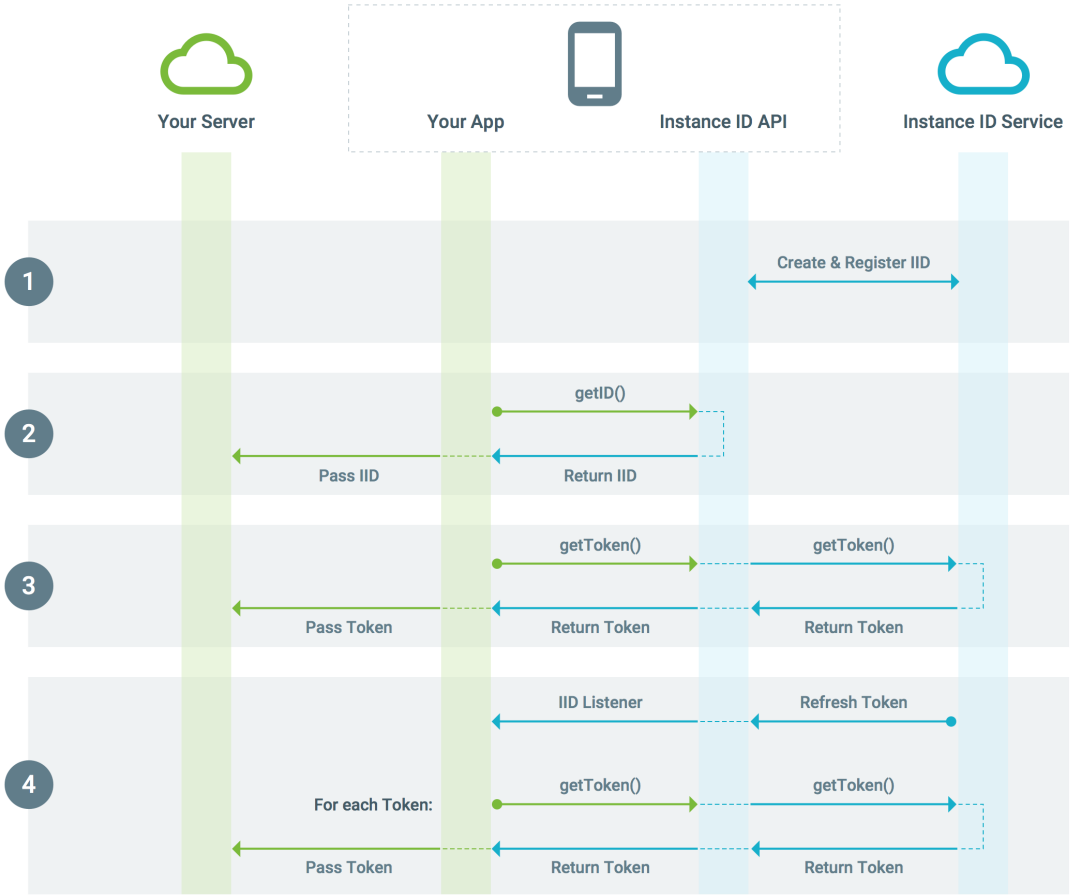

Figure 3.1: Instance ID Architecture overview as found in the official documentation [79].

InstanceID The class `InstanceID` provides a method to generate a push token that uniquely identifies this app and device combination. This token is needed by content providers to push to this device. Figure 3.1 shows the flow of messages between a developer's app, their server and Google's Instance ID Service and Instance ID API on Android. First, the device registers with the Instance ID Service. This initializes

the Instance ID with a private key stored on the Android device. The matching public key is registered with the Instance ID Service. Next, the developer's app requests an `InstanceID` instance via the Instance ID API method `getID()`. The app can then request a push token using the method `getToken()`. This token is generated on the Instance ID Service and returned to the calling Android app. It is unique and secure for this app. This token can be transmitted and stored on the developers' server to use it for pushing messages to this Android device. The Instance ID API also allows to refresh tokens. For this, the app developer needs to implement an *Instance ID Listener* callback to be called when a fresh token is generated. App developers can also request a fresh token, discarding the previous one, by calling `getToken()` again.

3.2 OwnPush

OwnPush [60] is a project to provide an open source alternative to the push services provided by Google. It focuses mainly on low resource usage and easy usability, but also mentions improved user privacy as a use case. OwnPush is based on a central service hosted by the library developers. Similar to FCM, app developers use this service to send push messages to Android devices. This central service then relays the messages to an OwnPush component on the Android device, which in turn delivers the message to the specific app. This flow of messages is shown in Figure 3.2.

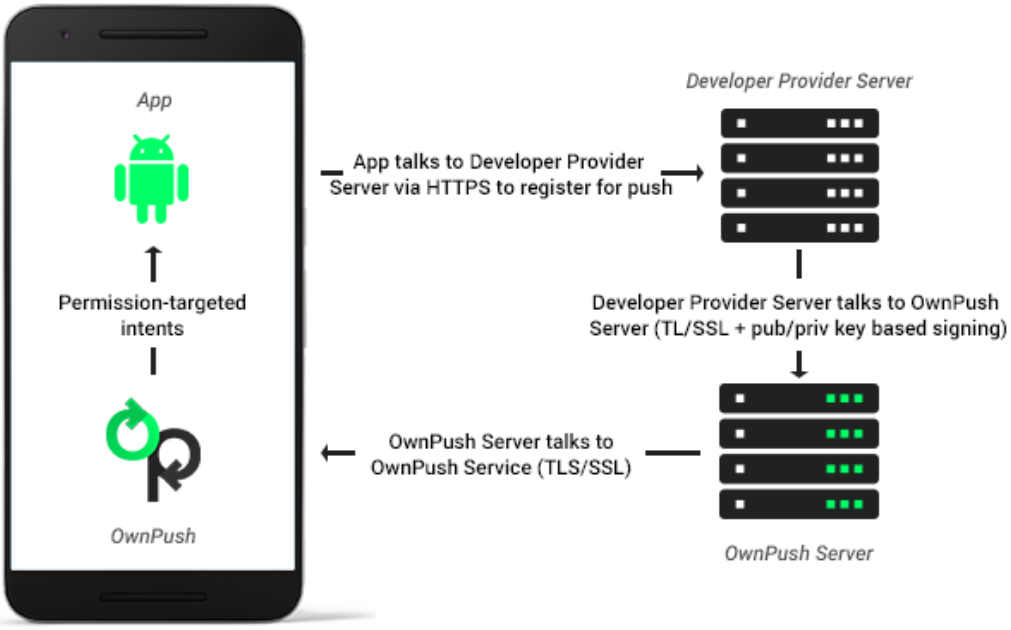

Figure 3.2: OwnPush Architektur as found on the OwnPush website [60].

Sadly, development of this project seems to have seized. The Homepage of the project *ownpush.com* is not available anymore and therefore was visited via `archive.org`. Source code repositories hosted on GitHub are still available. The last change in those repositories is dated to 2018-10-30. The completeness of the source code published on GitHub was not evaluated.

3.3 MQTT

MQTT (MQ Telemetry Transport) is a lightweight binary messaging protocol, built on top of TCP connections. It is built on a publish/subscribe message model. Senders publish messages to certain topics, receivers subscribe to these topics to receive said messages. MQTT is an open standard under OASIS[1]. The most recent version is 5.0, published in 2019. *Eclipse Paho*, the MQTT library used in my implementation, adheres to version 3.1.1 from 2014 [52, 41, 56, 57].

Client and Broker *Clients* are identified by their client id. This is a UTF-8 string with the length of one to 23 bytes that is allowed to only contain alphanumeric characters. The central entity of the system is a so called *broker*, a server, which relays messages between clients. Clients do not connect to each other directly but only through such a broker. There, the communication is structured in arbitrary topics [52, 56].

Topics and Subscriptions The entity for organizing messages and subscriptions is a topic. Topics are organized hierarchically, separated by a forward slash `/`. Subscribers can choose to receive messages of an explicit topic or can use wildcards to listen to messages on a whole branch of the topic hierarchy. There are two wildcards available:

plus + specifies a wildcard for a single level of the hierarchy.
hash # is a wildcard to select the whole sub tree. Therefore it must be the last character when specifying a topic selection [41, 52, 56, 57].

Quality of Service MQTT specifies three QoS (Quality of Service) levels:

zero *fire and forget* messages will be delivered once, without confirmation.
one *delivered at least once* messages will be delivered at least once, confirmation by the other party is required. Duplicated messages are possible.
two *delivered exactly once* messages will be delivered exactly once. This is ensured by additional confirmation messages.

Messages are sent and received at the QoS that the client specifies. The receiving client specifies the highest QoS it wants to receive with. If a message is published at a lower QoS than the subscriber has specified, then the lower QoS will be provided. Higher QoS settings incur a higher latency and bandwidth usage [41, 52, 56, 57].

[1] Organization for the Advancement of Structured Information Standards

Persistent Messages Messages can be set to be retained by the broker. This orders the broker to keep the messages even after sending them to all current subscribers of the specified topic. This allows new subscribers to receive these messages, even if they have been sent before the subscription was active. This mechanism is limited to the most recent retained message per topic, though [41, 52, 56, 57].

Last Will Clients can register a *last will and testament* message with their broker. This message will be sent by the broker if the client disconnects unexpectedly. Other than that, the message will be treated the same as any other message, i.e. it has a QoS, topic, content and persistence flag [41, 52, 56, 57].

Security MQTT supports authentication of clients using a username and password combination. Connections can be encrypted using TLS. Apache ActiveMQ also supports authentication through TLS Client Certificates, in addition to username and password authentication mechanisms [41, 52, 56, 57].

Dynamic Keep Alive To ensure that dead TCP connections are detected and closed, a keep alive mechanism is needed. Heartbeat messages at regular, fixed intervals, however, incur unnecessary overhead on clients' network usage, power usage and CPU time. MQTT solves this problem by sending ping messages only if no other messages are transmitted during the negotiated time interval. The interval length is set by the client. It can be dynamically changed without the need to reconnect to the broker. It is also possible to disable the heartbeat mechanism. The maximum allowed keep alive interval is 18 hours, 12 minutes and 15 seconds [44, 56, 57].

CHAPTER 4

Architecture of Push Adapter

As discussed in previous chapters, FCM and other alternatives all have their problems, especially when it comes to effectively protecting the privacy of the users. Therefore, I designed and implemented a new system, named *Push Adapter*.

This chapter introduces the overall goals of Push Adapter, its architecture and the high-level interaction patterns within the system. The following Chapter 5 discusses the implementation of each component.

4.1 Goals

The goals of Push Adapter are as follows:

Privacy The users of our implementation should not have to compromise their privacy. They should not have to trust any arbitrary third party with such privacy critical data. Furthermore, the system's topmost priority must be to preserve the user's privacy. To guarantee this, individuals must have the possibility to run the service themselves, independent of any third party services. Furthermore, to properly support the claim that the service is written with the user's best interest in mind, this claim has to be verifiable. For this reason alone the implementation has to be Open Source.

Open Source There are other reasons to create open source software, and many of them are goals for Push Adapter as well. The software stays modifiable and chances of continued support are higher. Interested parties can modify the software, improve it and add new features. The software can rely on existing open source software that would otherwise be incompatible due to licensing issues.

Security Push Adapter should be built in a secure way. The software should be designed in a way that no obvious security flaw is inherent in the design and architecture of the software. Providing the source code of the system should make it easy for anyone with the right expertise to verify and point out security flaws and problems, and even improve the software in these regards.

FCM Compatibility To ease the switch from FCM to this implementation, interfaces similar to those provided by FCM should be provided. This includes client side interfaces on apps running on Android as well as the HTTP API to push messages to clients. Of course, some changes will be required, but these will be documented and justified in the following sections.

Power Saving The implementation should be energy efficient on Android clients. It should allow apps using its service to save overall system battery consumption. This should be achieved by using one single connection to the push service, instead of every app relying on an individual connection to their own backend. This should be similar to how FCM holds only a single connection for all apps using it. Energy efficiency of the server implementation will not be evaluated.

4.2 Design Decisions

Obviously to reach such goals, a lot of work would be needed. To realistically finish the implementation and allow it to be verified in a timely manner, some design decisions were made. The resulting limitations are given as follows.

Prototype The implementation has to be regarded a *prototype*. This means that some flaws and bugs are expected. Only a limited set of features is implemented to test the design in principle. The implementation is not ready for a production environment. It is not thoroughly tested. No performance testing is conducted and therefore no optimizations are implemented.

One-To-One Downstream Push Messages Only FCM provides many features like *upstream messages* (sending messages from the device to the app server) and broadcast messages to groups of devices with a single request. While those features might be useful for smaller apps that do not want to maintain their own message facilities, it's not the use case we are interested in this work. Upstream messages are not the kind of push messages that are as hard to implement as downstream pushing messages to devices at any time. Broadcast messages are not that useful in the context of instant messaging.

Limited Number Of Users Since this is a prototype implementation, it is not optimized to serve large number of users. Furthermore, federation of multiple instances of this service and the like are not going to be implemented.

Ease Of Setup And Documentation Setting up the service does not have to be easy. Although some basic documentation is provided alongside the source code, the source code itself is the ultimate truth describing how the service works. No guarantees about the up-to-dateness of other documentation can be made.

User Management User management is very limited. Only a registration process for devices and apps is established. No modification or deletion of registered devices or apps is possible.

4.3 Architecture

A system of this size needs a proper architecture to function. Furthermore, the previously discussed goals had to be reached. These depend on a well thought architecture that can accommodate the required properties. From the goals mentioned previously, several properties for the design of the system can be deducted:

- Basic security principles have to be followed, e.g. communication has to be encrypted and certificates need to be verified by all communication partners.
- Power consumption by android clients should be low.
- A single persistent backend connection must be usable for all apps on one client.
- Multiple users should be able to use one backend instance. So functionality implemented in the backend has to follow basic multi-user separation, e.g. confidentiality must be guaranteed by the server implementation.

In this section, an overview and discussion of the basic system design as a whole will be presented. Communication patterns and protocols used by this service will be discussed. Implemented use cases will be presented at a high level to give an overview of what actors, processes, devices and external resources are involved. Details of the implementation and a close look at the individual components are presented in Chapter 5.

4.3.1 Overview

Push Adapter consists of several parts: the server *Push Relay*, the Android apps *MQTT-Client* and *Push Adapter*, the Android library *push-integration-lib* and the universal helper library *pinning-tls-lib* for verifying TLS certificates in regard to a central, self-signed Certificate Authority (CA). Third parties wanting to push messages to their Android client apps using Push Adapter are named *Push Provider*. The server run by *Wire* will be adapted to be such a Push Provider. For more details on the proof of concept using Wire, see Chapter 6. A basic overview of these components can be seen in Figure 4.1.

There are three main operations that are necessary to implement our push model: 1. register a new Android device, 2. register a new app on a registered device and 3. send

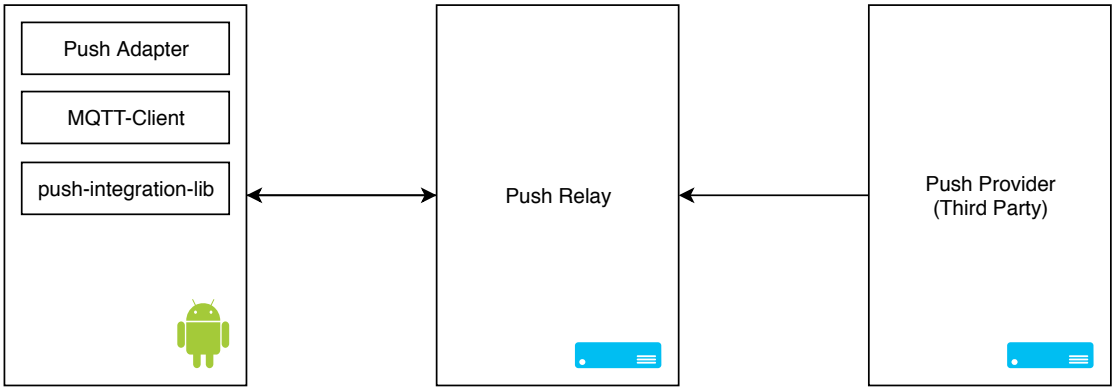

Figure 4.1: Overview of Push Adapter architecture.

push messages to a registered app. The following sections describe those operations from an architectural point of view.

4.3.2 Register Device

Push Relay needs to know about each device that messages should be pushed to. A simple, automated registration process was established to ease the registration of new devices. This registration process consists of the following steps:

1. Push Adapter sends a registration request to Push Relay.
2. Push Relay registers the new device and assigns it a unique identifier.
3. Push Relay returns settings and credentials for setting up a persistent connection to Push Adapter.
4. Push Adapter requests a persistent connection to be created by MQTT-Client.
5. MQTT-Client connects to Push Relay.
6. Push Relay accepts these connections if they use the previously provided credentials.

The registration process is depicted in Figure 4.2. As those steps are crucial to set up a secure, persistent connection between Push Relay and Android clients, we will discuss each step in detail.

Registration Request This request is a simple HTTP POST request. The hostname and port of Push Relay to use must be previously configured. It uses TLS transport encryption. To verify the possibly self-signed certificate provided by Push Relay, a copy of the certificate must be stored by Push Adapter. In contrast to all other requests made by Push Adapter, this one is the only request that does not provide a client certificate.

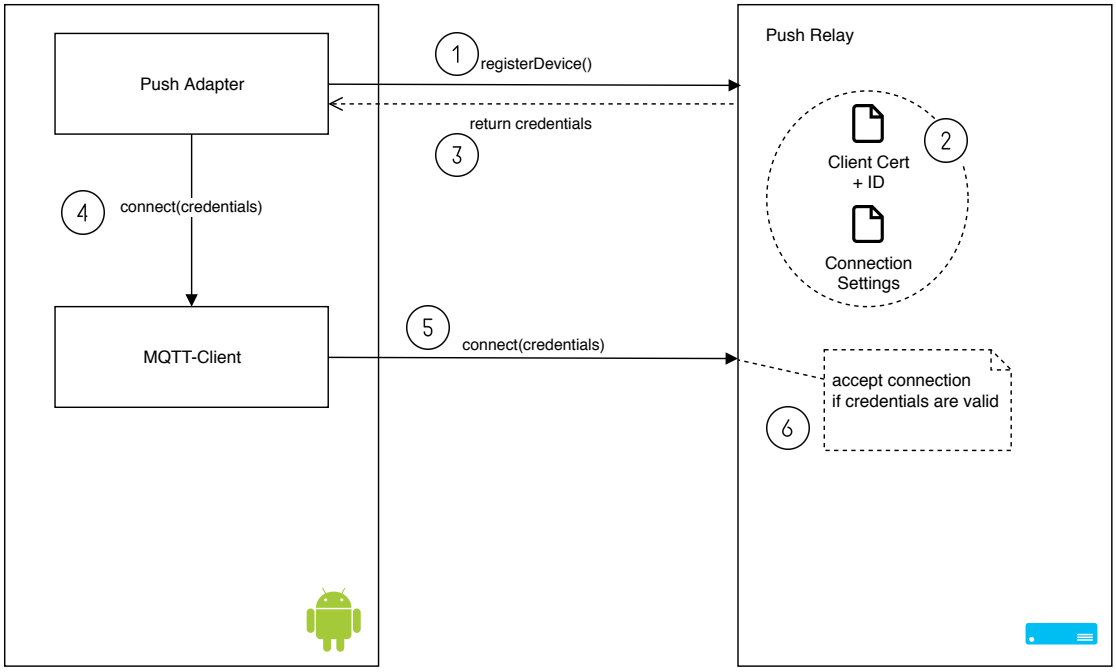

Figure 4.2: The device registration workflow.

Push Relay Registers New Device Upon receiving a new registration request, Push Relay generates a new, unique TLS client certificate and a new, unique `registration ID` for this registration request. The registration ID is set as the *Common Name (CN)* property on the client certificate. The certificate is signed by the Certificate Authority (CA) hosted by Push Relay.

Returning Persistent Connection Credentials To ease configuration but leave some space for hosting Push Relay, only the REST API endpoint of Push Relay needs to be configured in Push Adapter. Persistent connections are using MQTT, not HTTPS, as protocol for communication. This requires a different port and possibly a different hostname for MQTT connections. Alongside the signed certificate, the MQTT connection settings for setting up a persistent connection with this Push Relay are transferred.

Request Persistent Connection Push Adapter stores the connection settings and client certificate permanently on the device. This is needed to be able to reconnect on reboot and use the client certificate for further requests. Push Adapter doesn't create and manage a persistent MQTT connection itself. These responsibilities have been encapsulated by MQTT-Client. For detailed reasons on why this separation was deemed useful see Section 5.2. To create such a connection, Push Adapter is signaling to MQTT-Client that such a connection should be created. Alongside this signal it transfers the connection settings and the client certificate to MQTT-Client.

Create Persistent Connection MQTT-Client uses the provided connection settings and client certificate to set up and manage the persistent connection. The persistent connection uses TLS over MQTT, which ensures that no eavesdropping on the connection is possible. Since MQTT is built around a publish-subscribe communication model using topics, we need to break this down to a one-to-one model. Simply put, this is achieved by assigning each client a unique topic. See Section 5.1.4 for all technical details on the usage of MQTT for this purpose.

Push Relay Accepts Connection Push Relay accepts new MQTT connections only if the connecting client provides a client certificate signed by this Push Relay's own CA. The encapsulated registration ID is used to identify the device. To not allow any eavesdropping by other clients, Push Relay restricts each client connection to the topic it got assigned while registering. Details on how this is achieved can be found in Section 5.1.4 as part of the discussion of Push Relay's implementation.

4.3.3 Register App

App registration is the process in which an app can request a unique *push token* to be able to push messages to the requesting app instance. To obtain this push token, the following steps have to be taken:

1. The app requests a new token from Push Adapter.
2. Push Adapter extracts details about the requesting app and forwards these to Push Relay.
3. Push Relay stores these details, generates a push token and sends this token back to Push Adapter.
4. Push Adapter stores the token alongside the app details.
5. Push Adapter returns the token and the connection settings to the calling app.

Figure 4.3 depicts this app registration process.

From the point of view of an app developer this process is very similar to FCM. There is only one major difference: in addition to receiving a push token upon successful registration, the app also receives connection settings (hostname, port, TLS certificate) of the Push Relay instance this device registered with. The registration process must not leak push tokens to any third party. Neither to another app installed on the same device, nor while in transit from Push Relay to the app. After the requesting app has received the push token, Push Adapter can no longer guarantee its proper usage. This is similar to FCM. In the following paragraphs I will discuss the app registration steps in more detail.

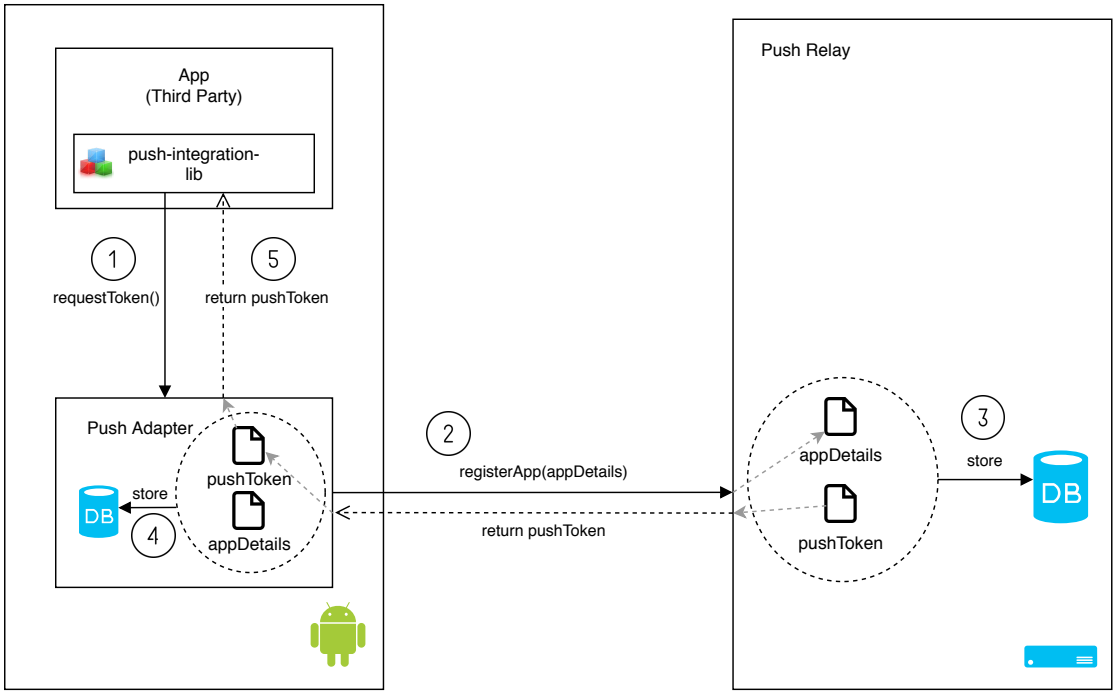

Figure 4.3: The app registration workflow.

Token Request By App Any app running on an already registered device can request a push token. The process is similar to the one provided by FCM and facilitated by integrating push-integration-lib into the app.

Push Adapter Handles Request Upon receiving a registration request by an app, Push Adapter has to gather some basic data about the requesting app. By using services provided by Android it extracts details to uniquely identify the app. Which details and how they are extracted is described in Section 5.2.2 when discussing the implementation details of Push Adapter. The app's details are then sent via HTTPS to Push Relay. For this connection, the client certificate generated during device registration is used to identify the device uniquely.

Push Relay Generates Token Push Relay accepts app registration requests only by valid devices (i.e. devices providing a certificate signed by the CA of Push Relay). It saves and associates the details of the requesting app alongside the requesting device's registration ID. It generates a new push token and associates it with the saved details. Later on, this allows to forward any push message requests made with this specific push token to the correct device and app. After successfully storing these data, the newly generated push token is returned to Push Adapter.

Push Adapter Stores Token Upon receiving the push token from Push Relay, Push Adapter stores the token in a persistent, private database on the device. Although not implemented, this would allow the return of the push token to the same app again, should it request it again.

Return Token To App Push Adapter returns the push token to the requesting app. It also returns the connection settings and TLS certificate for the Push Relay this device is registered with. This is necessary since the app has to make requests against this Push Relay instance. From an app developers point of view, this is the only difference to FCM, where no connection settings are provided. This change is inevitable and inherent to being able to self host a Push Relay without needing any central coordinator.

Now the push token can be transferred to the backend services of the requesting app. From now on the token can be used to push messages to this device. For details on how the architecture of this process is built see the following section.

4.3.4 Sending Push Messages

Similar to FCM Push Providers can send push messages to their client apps by sending a request to Push Relay. Push Relay provides a REST API for this purpose. It is similar to the API provided by the FCM service. The process, as depicted in Figure 4.4, is outlined as follows:

1. Push Provider sends the push message with the push token to Push Relay
2. Push Relay validates the message
3. Push Relay forwards the push message to MQTT-Client
4. MQTT-Client forwards the message to Push Adapter
5. Push Adapter forwards the message to the app

Push Provider Sends Message Push Provider has to provide the push token it previously received from Push Adapter. Optionally it can include arbitrary data in the message as well.

Message Validation Push Relay validates the message. It has to conform to the set format and must contain at least a valid push token. More details on the actual message format can be found in Section 5.1 where the implementation of Push Relay is discussed.

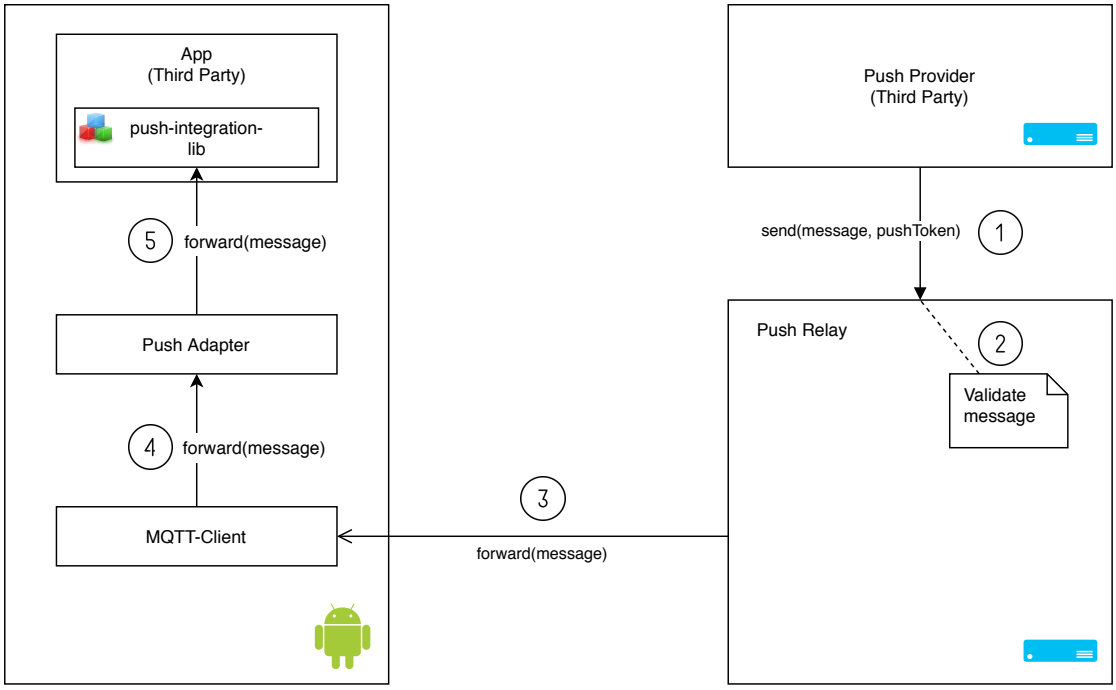

Figure 4.4: Sending a message using Push Adapter.

Forwarding to MQTT-Client Push Relay uses the push token received from the Push Provider to look up which device is assigned to this token. It enqueues the message and the app package name assigned to this token on the actual MQTT-Relay running inside Push Relay. It makes sure that the message is only queued on the right MQTT-Topic, the one the device got assigned to while registering. This is crucial to ensure that no eavesdropping can happen. Messages sent via MQTT should be received by the corresponding MQTT-Client instance in a timely manner, as long as the client device is connected.

Forwarding to Push Adapter Upon receiving any MQTT push message, MQTT-Client processes the content. It parses the binary content embedded in the MQTT message and extracts message data and the app package name the message is intended for. It then forwards the parsed content to Push Adapter.

Forwarding to App Push Adapter receives the message from MQTT-Client. It uses the provided app package name to lookup up the correct app the message is intended for. It does so by using the registration data it created when the app registered previously. It then forwards the message payload the Push Provider sent when pushing the message to Push Relay. If no message data was provided it forwards an empty message.

CHAPTER 5

Implementation of Push Adapter

Push Adapter[1] is a software system for relaying messages from arbitrary services to Android clients. It is open source software, published under various open source licenses. For details on the license of each part and the reasons why no uniform license was chosen, see the respective section below.

The previous chapter, Chapter 4, introduces the goals of the system. Furthermore, it describes the architecture and high-level interaction patterns of the system in Section 4.3. This chapter introduces the components that make up the whole system. Interesting implementation details are emphasized. Issues and implementation choices are discussed as well.

5.1 Backend - Push Relay

Push Relay[2] is the backend of our push implementation. It is open source software, published under the terms of the GPLv3 [30]. It replaces the server components of FCM. It handles registration of new devices and apps on those devices. It provides an interface for Push Providers to push messages to their registered apps and relays those messages to the appropriate devices. Proper identity and access management needs to be implemented, so the confidentiality of push messages is not violated.

Push Relay is a Java application, running inside an Apache Tomcat web application server. It is implemented with the use of *Spring Framework* and *Spring Boot* [71, 70]. These frameworks provide dependency injection and ease test setup. Integrating libraries such as *Apache ActiveMQ* [7] for hosting a MQTT Relay inside the application server

[1] The source-code of Push Adapter is available on github `https://github.com/haja/push-adapter`

[2] The source-code of Push Relay is available as part of Push Adapter on github `https://github.com/haja/push-adapter`

is facilitated by Spring Boot. Push Relay needs to store persistent data for device and app registrations, the database *H2* [34] is used for this purpose. Schema setup and migration is done with *Liquibase*. SQL queries are managed with the help of *JDBI*. Push Relay needs to generate, sign and verify TLS certificates. For these cryptographic tasks, *Bouncy Castle* [73] is used. To reduce boilerplate and allow a more functional oriented style of writing Java code *Lombok* is used. Immutable and functional-oriented Collections provided by *vavr* are used where appropriate. These libraries and tools allow us to write expressive and readable Java code.

Push Relay offers a few REST endpoints for client devices as well as Push Providers. Authentication and authorization on those endpoints are implemented with *Spring Security*. Only HTTPS is offered for using these endpoints. The REST endpoints offered are the following:

- `/registration/device` for registering new Android devices
- `/registration/new` for registering new apps
- `/push` for receiving push messages from Push Providers

Furthermore, Push Relay offers an MQTT relay for receiving push notifications. This service is provided via MQTT over TLS only and is only usable with a valid TLS client certificate, signed by the CA of Push Relay. Security on this MQTT endpoint is handled through Apache ActiveMQ and JAAS. For more details on this, see Section 5.1.4.

5.1.1 Registering a Device

The first step in setting up a new device is to register it with its Push Relay. This sets up client credentials for the device. Those are needed to register apps and eventually receive push messages.

Push Relay accepts device registration requests on the REST endpoint `/registration/device`. This endpoint is provided over HTTPS only. In contrast to other endpoints described below, this endpoint does not verify clients by means of TLS client certificates. Upon receiving a new request on this endpoint, Push Relay generates a new client certificate for this new device. It signs the certificate with the CA hosted by this Push Relay. Requests for app registration (see section 5.1.2) and MQTT connections (see section 5.2.1) are verified by this certificate. Only clients that present a valid signature from this CA are allowed to connect.

Push Relay needs to identify each client uniquely. For this, every new client certificate holds a unique, 32 byte long, randomly generated identifier in it's CN property. This property cannot be modified without invalidating the signature. Therefore, this field is appropriate for usage as an identifier. Each client is assigned a separate MQTT topic for receiving push messages. At device registration, such a new MQTT topic is created. The topic's permissions are set up in such a way that only Push Relay can write, and only the client with this specific certificate can read from this topic. For details on how

MQTT topics are set up to facilitate one-to-one messaging, see section 5.1.4. Since no other data than this certificate is exposed over this endpoint, it is safe to do so without verifying the requesting client.

5.1.2 Registering an App

Every app needs to register with Push Relay before it can receive push messages. Like with FCM, this generates a new *push token* for this app, that can be used to push messages to this app instance.

Push Relay offers the REST endpoint `/registration/new` for registering new apps. The request has to include `app package`, `app signature` and `sender ID`. Those values are obtained and transmitted by Push Adapter. Push Relay only processes app registration requests if they are made with a previously generated TLS client certificate. This is configured in the class `WebSecurityConfig` and enforced by the Spring Security framework. Upon receiving a new registration request, Push Relay extracts the device ID from the provided client certificate. It then generates a new, 32 byte long push token. The registration request details including the device ID are stored together with the push token in a persistent database. Finally, the token is returned to Push Adapter.

5.1.3 Pushing a Message

Push Relay offers the REST endpoint `/push` for sending push messages. This interface was made as compatible with the respective FCM interface as possible. Messages can be transmitted by using the HTTP method `POST` and providing JSON in the request body. This endpoint is secured by HTTPS. Push Providers can verify the authenticity of the endpoint by using the TLS certificate that is provided to the app while registering for a push token. The JSON body has to conform to the format as seen in listing 5.1.

```
{
    "validate_only": false,
    "message": {
        "data": {
            "key": "value"
        },
        "token": "TOKEN"
    }
}
```

Listing 5.1: JSON structure of push messages

The JSON fields are interpreted as following:

validate_only boolean, value ignored; required for FCM compatibility.

message JSON object; contains details about the message.

message.data JSON object; this is the actual payload transmitted to the client app.

message.token string; push token as generated during app registration.

Is the message valid and successfully processed by Push Relay, the `MessageID`, a random UUID, will be returned to the calling Push Provider. Valid messages are then transmitted to client devices through the MQTT protocol. For this purpose Push Relay hosts an *Apache ActiveMQ* MQTT relay internally. Registered devices connect to this MQTT relay. Push Relay interacts with this relay through the standardized *Java Message Service (JMS)* API [43]. Spring Boot provides integration with JMS through the class `JmsTemplate`. The JmsTemplate is set up as a Spring Bean in the configuration class `MqttBrokerConfig`. It uses a special user that has writing access to all topics. This user is secured by a password, which is randomly generated on each server startup. The password has a length of 48 bytes. In addition to using a randomly generated password, Push Relay never accepts authentication with username and password on its MQTT over TLS socket. Connections with this authentication method are only accepted from within the same *Java Virtual Machine (JVM)*. This means, an attacker would not only need to extract the password, but also be able to run inside the same JVM instance. This is a very unlikely attack surface, which would include total compromise of the server instance anyways. For this reason, no further steps were taken to secure this user with write privileges.

MQTT only supports transmission of binary message content. Since a variable length encoding was needed and messages size is not of utmost importance, I decided to encode messages as JSON. This allows easy en- and decoding since it is a widely used encoding. Android has JSON decoding libraries built in. Messages transmitted via MQTT are then handled on the client devices. This is done via the app's MQTT-Client (described in Section 5.2.1) and Push Adapter (see Section 5.2.2). For more details on how MQTT is used for pushing to individual clients, see the next Section 5.1.4.

5.1.4 MQTT for one-to-one Messaging

MQTT has a communication model that is based on publish-subscribe. Clients can *subscribe* and *publish* to *topics*. So inherently, MQTT does not have one-to-one messaging out of the box. Still, I built a system with one-to-one semantics by assigning each client to an individual topic.

When a device first registers, it gets assigned to a new, unique topic. The topic identifier (or name) is the same as the device registration ID created when the device registers. This registration ID is a random string with a length of 32 byte. It is never reused and only assigned to exactly one registration request. Therefore, only one client can be assigned the same MQTT-Topic. The registration ID is used in the TLS client certificate that is created at the registration time as well. It is saved as the `CN` field as specified in IETF

RFC4519 [65]. This approach has another benefit: only the registration ID and later generated push tokens needs to be stored. Push tokens can simply be assigned to those registration IDs. There is no need to store the MQTT-Topic separately. This simplifies the handling of topics and permissions, making the system easier to understand, and reduces the memory footprint per client.

Message Security Having a MQTT-Topic for each client is required, but not sufficient, for secure one-to-one messaging. Clients must not be able to read other clients messages. Therefore, clients must be restricted to their assigned MQTT-Topic. This is achieved by leveraging Apache ActiveMQ's customizable architecture and its JAAS [39] capabilities. ActiveMQ allows to plug in several `Broker` and `Plugin` classes to customize almost every aspect of the MQTT-Broker implementation.

Out of the box, ActiveMQ allows complex authentication and authorization setups. However, user authentication using a TLS client certificate is not provided. Since the system should authenticate and authorize clients solely through their TLS certificates, some customization is needed.

Authentication We want to authenticate all client devices through their TLS client certificates. Furthermore, Push Relay itself needs a special MQTT client to be able to push messages to each client. Therefore, we need to authenticate this special client as well. This requires some special setup. The client running inside Push Relay will use a random password, generated at startup, for authentication. This access will be authenticated by `SimpleAuthenticationPlugin` which is provided by ActiveMQ out of the box. To allow authentication with TLS client certificates, `JaasCertificateAuthenticationPlugin`, provided by ActiveMQ as well, will be used. The custom-made class `JaasCertOnlyOrSimpleAuthenticationPlugin` merges these two authentication strategies. `JaasCertificateAndSimpleAuthenticationBroker` leverages this plugin then to decide which authentication method is appropriate. It examines the type of connection that should be authenticated, differentiating two cases:

1. TLS connection, therefore coming from a client device and
2. non-TLS connection.

The rest of the service is configured in such a way, that no external connections can be made without using MQTT over TLS. Therefore, every non-TLS connection must be from withing Push Relay. So in case (1) `JaasCertificateAuthenticationPlugin` and only this plugin will be used for authentication. No simple password authentication will be tried. This guarantees that every client device connecting must provide a valid certificate. Case (2) covers connections from within Push Relay. As an additional safety measure, these connections will be authenticated against the password generated at startup.

For every authenticated client, a unique username must be extracted from the credentials provided. In case of TLS certificate authentication some customization was needed. This

is done by the class `SimpleCertificateLoginModule`. It extracts the `CN` property of the certificate provided by the client and uses it as a username.

Authorization For authorization the usernames provided by the authentication infrastructure are used. While client devices are only allowed to read from their specific topic, the internal client must have unrestricted access rights. It has to create new topics for newly registered clients and has to write to all existing topics to be able to transmit push messages to client devices. Therefore, the authorization components distinguish these two cases. The authorization infrastructure consists of two main components: `CustomAuthorizationBroker` and `CustomAuthorizationDestinationFilter`. The former extends the `AuthorizationBroker` class provided by ActiveMQ. It uses the username provided by the connection to decide if the client is allowed to proceed. If the username matches the username set as the internally used username, then unrestricted access is granted. If the username matches the topic name, then read access is granted. Otherwise, a security exception is thrown and no access is granted. The latter extends `DestinationFilter`, which is responsible for subscribing to topics. It processes authorization requests in the same manner as `CustomAuthorizationBroker`. To be able to plug these customizations into the actual MQTT-Broker instance, the plugin `CustomAuthorizationPlugin` wraps this setup and provides it as a plugin for ActiveMQ. The wiring and setting up of the internal client's password takes place in the class `MqttBrokerConfig`. It leverages dependency injection and other features of Spring to set up the ActiveMQ broker in a convenient way.

Resource Usage Every registered client device only holds one MQTT connection to its Push Relay, no matter how many apps have registered for push notifications. This allows very efficient resource usage. It is expected to be more efficient when two or more apps are using the Push Adapter service compared to maintaining their own backend connection for receiving push messages. MQTT as a protocol is designed to be very efficient with control data. It supports long timeout intervals, which means longer periods where no interaction between client and server is needed. It also supports dynamic keep-alive frequencies. This allows delaying keep-alive messages if there was data transmitted anyways, making keep-alive messages unnecessary. This makes a single MQTT connection for multiple apps even more useful, since it can further reduce the number of extra keep-alive messages. These features allow for potentially less data being transmitted as well as longer sleep periods on mobile devices. The MQTT library used by Push Adapter on Android devices is *Eclipse Paho*. This library supports this dynamic keep-alive messages.

5.1.5 Test Setup

Some basic, automated tests for asserting some basic properties of the implementation were implemented. These cover the following base cases:

- A client providing a valid client certificate is allowed to read the topic assigned to it.
- A client providing a valid client certificate is not allowed to read from a topic not assigned to it.
- Registering a device generates a new client certificate every time.
- Registering an app generates a push token.
- Using a valid token for pushing messages succeeds.
- Using an invalid token for pushing messages fails.
- Registering a device, then an app on this device and using the provided push token to push a messages results in a message being received by the device.

These tests, however, are not very thorough. They are in no way sufficient for developing a production ready service. However, tools and helper classes were implemented to make these tests maintainable and readable. Extending the test suite based on this test setup should be quite straightforward.

5.2 Components On Android

The client implementation consists of three components, each having their distinct purpose: *MQTT-Client*, *Push-Adapter* and *Push-Integration-Library*. The first two are Android apps, the latter is a library that can function as a drop-in replacement for push notifications from FCM, as far as this is possible.

MQTT-Client is solely responsible for keeping a persistent connection to Push Relay. Push Adapter on the other hand has higher level responsibilities, like registering the device with the relay and processing registration requests from apps, which wish to use the push service. Push Adapter is loosely based on MicroG [51], an open source implementation of various Google service clients on Android. MicroG is licensed under APLv2 [8], wherefore Push Adapter uses the same license. There are several reasons for splitting MQTT-Client from Push Adapter. On one hand, it facilitates keeping a somewhat reasonable and decoupled interface between the high-level and not so performance critical tasks of Push-Adapter and the actual receiving of push messages, which is a completely different concern. This would allow to somewhat easily exchange the current MQTT implementation with different technology, like e.g. SSE (Server-sent Events) or websockets. On the other hand, separating those two concerns into separate components solves a license issue. The MQTT library in use by MQTT-Client is Eclipse Paho [24], which is licensed under EPLv1 [25]. Since the remaining source code is published under APLv2 [8] or GPLv3 [30], this imposes some restrictions e.g. Eclipse Paho cannot be used with GPLv3 or APLv2 licensed code. Therefore, separating those library usages into a different, loosely coupled app solved these problems.

All apps and the library are compatible with Android 9 *Pie*. Android 7 *Nougat* introduced Doze Mode, which restricts apps from conducting certain tasks to preserve energy. Android 8 *Oreo* imposed further restrictions on background processing and network IO for apps not currently being engaged with user input. For specifics on how these features were dealt with see the following sections.

5.2.1 MQTT-Client

MQTT-Client is responsible for holding a persistent connection to Push Relay. It leverages the MQTT library *Eclipse Paho* [24] to set up and maintain a MQTT connection. Since Eclipse Paho is made available under the terms of the EPLv1 (Eclipse Public License version 1) [25] and Eclipse Distribution License [23], MQTT-Client has to follow a similar license. It is therefore also licensed under the term of the EPLv1[3].

The app offers a basic user interface. It consists of only three buttons. One button to request exemption from *battery optimizations*. Another button allows to manually disconnect the current MQTT connection to Push Relay, stopping any background activity. The last button allows to request the needed user permissions. After the initial setup, users are not required to interact with the app anymore.

Permissions There are two permissions MQTT-Client needs for its operation:

1. `at.sbaresearch.android.gcm.mqtt.intent.SEND` and
2. `at.sbaresearch.android.c2dm.permission.CONNECT`.

(1) is declared and used by MQTT-Client itself. It makes sure that if other apps want to send push notifications as well, the user is informed about this. Push Adapter accepts push intents only if the app sending those intents holds this permission. This protects users from fabricated push messages. For more details, see Section 5.2.2 about the implementation of Push Adapter.
(2) is declared by Push Adapter and used by MQTT-Client. Only apps that hold this permission are allowed to receive connect intents from Push Adapter. By itself, MQTT-Client does not establish a connection to a Push Relay. It does so only when it receives an intent from Push Adapter, requesting a connection. This intent contains the connection details of the Push Relay and is guarded by permission (2). This allows the user to restrict which app should be able to control MQTT-Client. Most probably the only apps with permissions to send or receive these connect intents are MQTT-Client and Push Adapter.

Messages received through this MQTT connection are parsed and broadcast as an `Intent`. Messages sent by Push Relay use a simple JSON format. On receiving such a message, metadata information is extracted and transformed into `Intentextras`.

[3] The source-code of MQTT-Client is available as part of Push Adapter on github `https://github.com/haja/push-adapter`

The actual JSON payload provided by the Push Provider is transferred as a string object. To ensure that no other app on the Android device can eavesdrop and intercept the intent, the intent is broadcast to Push Adapter only. This is achieved by using `Intent.setClassName()` with the package of Push Adapter and the corresponding `BroadcastReceiver` within Push Adapter. This should mitigate attacks on intent interception as described by [47].

Battery Optimization Since version 8, Android implemented new energy saving methods. These restrict background processing and network IO. This affected the functionality of MQTT-Client. Therefore, MQTT-Client needs to be exempt from most of those restrictions. This is done by configuring Android such that MQTT-Client is whitelisted from *battery optimizations*. MQTT-Client requests the corresponding permission `REQUEST_IGNORE_BATTERY_OPTIMIZATIONS`. This allows the user to easily configure the whitelist for MQTT-Client by triggering the action `ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS` via a button in the MQTT-Client UI. MQTT-Client is almost always running in the background. To still be able to do any network IO, a *foreground notification* needs to be displayed [4].

5.2.2 Push-Adapter

Push Adapter implements most of the client logic. It closely cooperates with *push-integration-lib* to provide apps with an interface similar to FCM. Push-integration-lib is discussed in detail in Section 5.2.3. It is responsible for:

- registering the device,
- registering new apps and providing them with push tokens,
- maintaining Push Relay connection data and requesting MQTT connections and
- processing and distributing new push messages to registered apps.

The implementation[4] is licensed under APLv2 [8]. It reuses some parts from the MicroG project [51], which is licensed with the same license. MicroG implements Google Play Service clients as open source apps and libraries. The parts of MicroG that implement push messages based on the outdated *Google Cloud Messaging (GCM)* framework were examined and partially reused. Through those parts some insight in the workings of GCM and its follow-up service, FCM, were gained.

Push Adapter uses `OkHttpClient` and `Jackson` for making HTTP calls. To pin the possibly self-signed Push Relay certificate, the custom-made `pinning-tls-lib` is used. More details on this library, common to all components that have to interact with Push Relay via HTTPS, can be found in Section 5.3.

Push Adapter's UI offers a few functions for setting the device up:

[4] The source-code of MQTT-Client is available as part of Push Adapter on github `https://github.com/haja/push-adapter`

- request required user permissions,
- register the device with the pre-configured Push Relay,
- ensure that an MQTT connection is established and
- request exemption from *battery optimizations*.

Permissions The permissions required by Push Adatper are:

- `at.sbaresearch.android.c2dm.permission.RECEIVE`,
- `at.sbaresearch.android.c2dm.permission.SEND` and
- `at.sbaresearch.android.c2dm.permission.CONNECT`.

The RECEIVE permission is used to authorize apps for receiving push messages. It is created and held by Push Adapter to assert that only Push Adapter can send the appropriate push message intents. Apps wanting to receive push messages must therefore request this permission as well.
The SEND permission is held to assert that only Push Adapter can send push messages to apps. Apps listening for push messages need to declare that they only accept intents on their `IntentReceiver` if the issuing app holds this SEND permission.
The CONNECT permission is held to be able to send intents requesting new Push Relay connections to MQTT-Client. See Section 5.2.1 for a detailed explanation.

Device Registration This action triggers the device registration process. It issues an HTTPS request to the pre-configured Push Relay. Upon its completion, the device is registered with the Push Relay. A CONNECT intent with the host, port, MQTT topic and TLS client certificate will be sent to MQTT-Client. From now on, apps can register for push notifications.

Ensure MQTT Connection This action triggers the same connection setup that will be triggered after successfully registering the device. Its main purpose is for debugging or after manual disconnection from the Push Relay via MQTT-Client. The MQTT connection is also set up by Push Adapter after the device has been booted.

Battery optimization Push Adapter needs to be on the whitelist for *battery optimizations*. This is needed on recent versions of Android to guarantee proper and timely notification processing. See the previous Section 5.2.1 for in depth information about Android's *battery optimization* features.

App Registration Other clients of Push Adapter besides its UI are Android apps that want to register for and receive push messages. Therefore, Push Adapter listens to app registration intents issued by push-integration-lib. Those registration requests are then received in a `JobIntentService`. The usage of JobIntentService is necessary since Android 8. This version of Android introduced various new restrictions on background

processing. Specifically, `WakefulBroadcastReceiver` can no longer hold a WakeLock. This makes it essentially useless[5]. As a replacement, JobIntentService was introduced. This allows Android to wait for several background tasks to bunch up and process them in one go, reducing the wake up count of the device and therefore allowing greater energy savings. The implementation by MicroG relied on WakefulBroadcastReceiver. This part of the implementation from MicroG was rewritten to use a JobIntentService. Since registration requests normally only arrive when the device is awake and in use anyways, this should not delay the registration process unnecessarily.

Once processing of the registration process has started, Push Adapter verifies the `packageName` of the registering app. Furthermore, it extracts the signature with which the app was signed. This prevents forging of the requesting app. Finally, Push Adapter forwards this data to Push Relay via HTTPS. It uses the method POST on REST endpoint `registration/new`. Successful registration returns a push token to Push Adapter. This push token is then used alongside with the Push Relay endpoint and certificate to generate the response to the app. This data is then returned to push-integration-lib via an intent.

5.2.3 Push-Integration-Library

Any app that wants to use Push Adapter for downstream push messaging can leverage the provided *push-integration-lib*[6]. It is written in Java and usable on Android. This library provides an easy to use interface for requesting push tokens and receiving push messages once app registration is complete. Its interface is based on the one provided by the FCM client libraries. This allows easy adaption for apps that are already using FCM to receive push messages. Since parts of this library are based on *MicroG* project [51], it is licensed under APLv2 [8].

App Registration For registration, the class `FirebaseInstanceId` is provided. This class is directly based on code provided by the *MicroG* project [51]. It implements the two methods that are necessary to successfully register an app. The static method `getInstance(context)` returns a singleton instance of FirebaseInstanceId. Furthermore, it provides the class `CloudMessagingRpc` with the ApplicationContext. This is needed to access details of the registering app and communicate with Push Adapter for the registration process itself. The method `getToken(authorizedEntity, scope)` returns the actual registration details. This method sends an app registration request to Push Adapter and returns a `RelayConnection` object. This object consists of the push token and the connection details of the Push Relay used to relay messages. This includes the URL for the relay as well as the TLS certificate this relay uses.

[5] `https://developer.android.com/reference/android/support/v4/content/WakefulBroadcastReceiver`

[6] The source-code of push-integration-lib is available as part of Push Adapter on github `https://github.com/haja/push-adapter`

Receiving Push Messages The library provides push messages received through Push Adapter in a similar way to how FCM provides them. For this, the abstract class `FirebaseMessagingService` is provided. This class is loosely based on a similar class provided by the *MicroG* project. It extends the Android class `Service`. Apps should extend this class, similar to FCM, to be able to receive push messages. `FirebaseMessagingService` receives an intent from Push Adapter whenever a new message arrives. It parses the intent and provides the the app with the contained message via the method `onMessageReceived(message)`. During parsing and delivering the message to the actual app, the service is run as a foreground service. This is necessary on newer versions of Android. Otherwise, Android might defer this background activity to a later point in time, which in this case is not desired.

Differences to FCM The data returned by the call to `getToken()` is different from FCM. While at this point FCM only provides the push token, my implementation also returns data for the connection to Push Relay. This is necessary since relays are self-hosted and not managed by a central entity, as this was a primary design goal of the implementation.

Apps that integrate this library have to be exempt from Doze Mode. Details on how to do this with simple intents are described in the discussion of 5.2.1. Doze Mode restricts network IO, although intents can be received and processed. So if apps receiving push messages want to do network IO upon receiving such a notification, they have to be excluded from Doze Mode. To counter this problem, the idea was to hold a wake lock on behalf of the respective client app But this does not seem to be possible. Further investigation is needed. How FCM client libraries achieve the goal of allowing network IO for clients receiving push messages was not investigated, but might prove fruitful in tackling this problem.

5.3 Pinning TLS Library

`pinning-tls-lib` is a library for verifying TLS certificates and using custom TLS private keys. It is written in java using the package `java.security`. To reduce boilerplate code, `lombok` was used.

It provides the class `PinningSslFactory`. This class can be used to initialize a `SSLSocketFactory` to create secure and verified TLS connections. Furthermore, it exposes the `X509TrustManager` created during initialization. These allow creating and verifying connections using a provided Certificate Authority and optionally provided client keys.

Setup PinningSslFactory is initialized with a Certificate Authority and optionally a `ClientKeyCert` object. The ClientKeyCert object contains a TLS private key and client certificate. The Certificate Authority is provided as a java `InputStream`. All certificates must be provided in binary *DER* format. Private keys must be provided in

binary *PKCS#8* format. The exposed SSLSocketFactory is generated by a SSLContext. This context is initialized with a X509TrustManager generated from the provided CA. If a ClientKeyCert object is provided as well, a `KeyManager` is initialized with the provided key and used for the SSLContext.

Pinning-tls-lib is used in Push Relay (see Section 5.1), Push Adapter (see Section 5.2.2) and for integrating Push Adapter in Wire. See Chapter 6, Section 6.4.1.2 for details on the Push Adapter integration of Wire.

CHAPTER 6

Proof Of Concept: Wire

The Push Adapter implementation on its own is not very useful. To evaluate Push Adapter, integrating it in an instant messaging app is imperative. For this purpose, the instant messaging app *Wire*[1] was chosen. There are several reasons why Wire is a good choice:

Wire offers everything expected form a modern instant messaging app:

- Text messages
- Different kinds of multimedia messages, including picture, audio- and video messages and arbitrary files.
- Audiocalls
- Videocalls
- End-to-end encryption for all the media types above, by default.

Wire provides a security white paper [33] where the core functionality and security of the system is outlined. Furthermore, it hosts an overview of its security and privacy features [66]. They substantiate their security claims by providing a series of recent security reviews [16, 14, 15, 13].

Wire is an open source application. Not only the Wire Android client is open source, but the backend *Wire-Server* as well [82, 84]. This is crucial for successfully adapting Push Adapter. It is in the nature of Push Adapter, that not only client-side, but also server-side changes are necessary. The development culture of Wire is, although mainly developed by a single company, rather open. Although *Wire Swiss GmbH*, the main company behind Wire, has stated that they are currently not accepting github merge requests beyond bug fixes, they are open to forks and have extensive documentation on self-hosting a Wire Server instance [81]. Therefore, successfully hosting or even integrating a further developed version of Push Adapter into a productive Wire instance seems feasible.

[1] `https://wire.com`

Wire is a company based in Switzerland, where privacy regulations are way higher than in other countries, e.g. the US. [2]. They continue to improve support to run Wire without any Google dependencies, in code as well as on a service level. This is discernible from the ever so maturing websocket support for push notifications. Wire does not include any third-party dependencies in their web-client, so no third party can track users of Wire-Webapp [85]. Wire-Android explicitly excludes *firebase-analytics* during the build process of the Android client [83]. Wire-Android can be used even without access to local contacts. For registration and usage, no phone number is needed. An account can be created just by using an email address [33, 83].

6.1 Overview of Push Messages in Wire

Wire consists of a server and multiple clients. Official source code repositories are hosted on GitHub [83, 84]. Supported operating systems are Android, iOS, Windows, macOS, Linux and web browsers [74]. Since the Android client is the only one relevant for our work, other clients will not be discussed in detail.

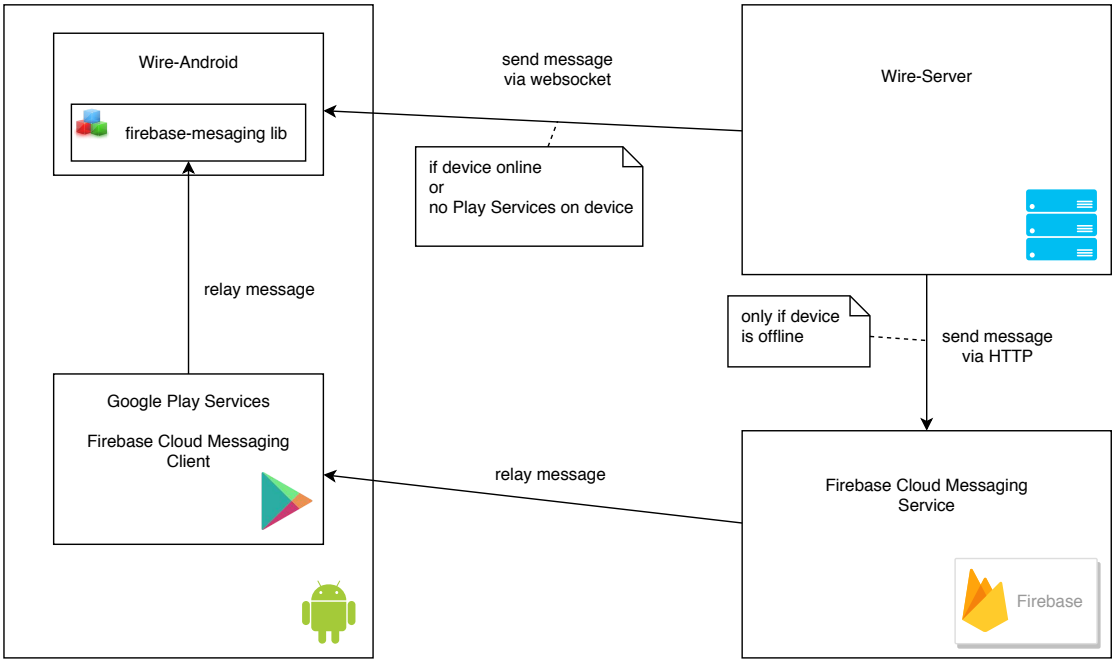

Figure 6.1: Overview of Wire-Server and Wire-Android communication using FCM.

Pushing Messages Wire has two means of pushing messages to Android clients:

- If the client is currently running in the background and has Google Play Services installed, FCM will be used to notify the client about new messages. No message content is sent through FCM, only a generic client ID.

- Is the client app currently in use by the user (e.g. running in the foreground) or are no Google Play Services installed on the device, a websocket connection to Wire-Server is being held.

In case Wire-Server detects an active websocket connection, it will push notifications over that connection, ignoring FCM for the respective device. This communication flow is shown in Figure 6.1.

6.2 Wire-Server

Wire-Server is the backend for Wire. It is an open source project hosted on github [84]. Except for some libraries, the entire service is implemented in Haskell. As depicted in Figure 6.2, the service is split up into distinct modules. Each module is named after distinct parts of sailing vessels:

Brig User service. Holds Gundeck instances to communicate with users.

Gundeck Push notification Hub. Handles sending push notifications via *Amazon Web Services* AWS SNS or Cannon. SNS delivers the notifications via FCM or APNs.

Cannon Manages client websocket connections.

Galley Conversation Service.

Cargohold Asset Storage.

Restund Used for audio and video calls. It is a modified version of restund [61], a STUN[2] and TURN[3] server.

Spar Handles integration with SSO (Single Sign On) identity providers.

All modules are running behind a Nginx reverse proxy with custom auth module, referred to as *nginz* [54]. Modules are communicating via HTTP and can be scaled up individually. For example *Cannon* can only hold a limited amount of open websocket connections. If more connections are needed, new Cannon instances can be started on the fly.

Wire-Server relies on various external services. AWS is used for storage, delivery of push messages and queue management. Wire offers integration with YouTube, Spotify, SoundCloud and Google Maps. For ease of development and deployment, wire-server can be run via docker. Specifically for development purposes, a so called *demo-mode* of wire-server is available, which does not depend on AWS or any other external services.

[2]Session Traversal Utilities for NAT
[3]Traversal Using Relays around NAT

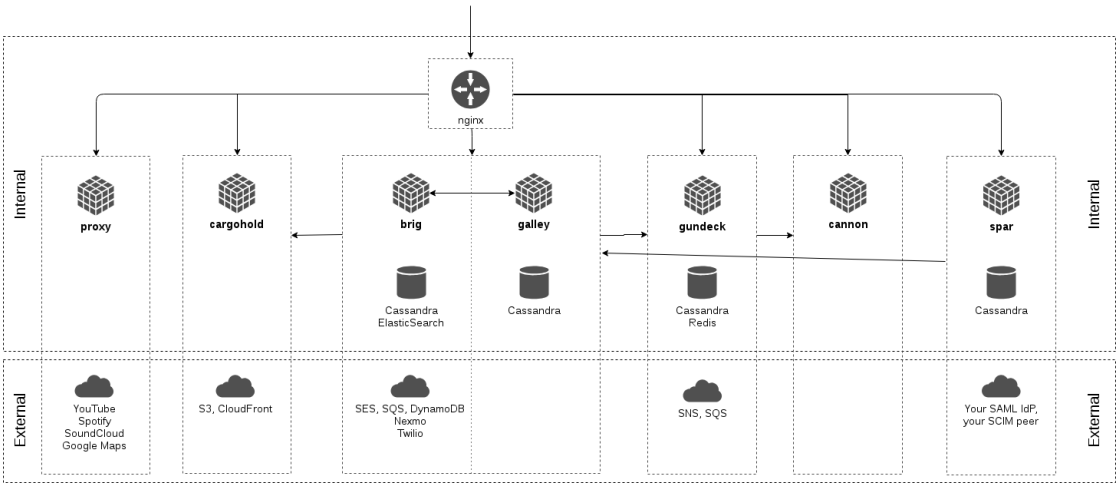

Figure 6.2: Wire-Server module overview as found in the Wire github repositories [84].

Push Notifications Wire-Server has a dedicated component responsible for notifying devices about new messages: *Gundeck*. Gundeck can send push messages using a custom websocket implementation or via *AWS SNS* which in turn uses FCM. The websocket connection is preferred if it is available. Android clients establish a websocket connection as soon as the app is running in the foreground. If no FCM is available on the Android device, Wire-Android can keep a websocket connection alive in the background.

6.3 Wire For Android - Wire-Android

Wire for Android (Wire-Android hereafter) is the official Wire client for Android. It is an open source app, the source code can be found on github[4]. Major parts of the app are written using Scala, while Java is also used. Scala, being a functional oriented language, seems to be a good fit considering most parts including all the business logic of Wire-Server are written in Haskell. No further evaluation on the reasons of the mix of languages was conducted.

[4] `https://github.com/wireapp/wire-android`

Wire-Android relies on a few libraries developed specifically for Wire:

- Wire-Android-Sync-Engine[5]
- Audio Video Signaling[6]
- generic-message-proto[7]
- wire-andoid-translations[8]

6.3.1 Wire-Android-Sync-Engine

Wire-Android-Sync-Engine is an open source library written specifically for Wire-Android. It is written almost entirely in Scala. SBT is used for build and dependency management. It is responsible for handling most communication with the backend service. Push messages are processed in the library, just the actual receiving class extending `FirebaseMessagingService` is implemented in Wire-Android. It handles seamless switching between websocket and FCM for push messages.

6.4 Adapting Wire For Push Adapter

Integrating Push Adapter in Wire-Server and Wire-Android was a relatively straightforward task. My prototype implementation is functional and comprehensible. Integration was achieved with only minor, local changes to Wire-Server as well as Wire-Android and Wire-Android-Sync-Engine.

The overall changes to the flow of data when pushing messages can be seen in Figure 6.3. If the device is not online, i.e. not currently connected to Wire-Server via websocket, messages are relayed via Push Adapter. In order to do that, Wire-Server sends an HTTP request to Push Relay, which relays the message to MQTT-Client on the user's Android device. Afterwards, the message is forwarded to Push Adapter running on Android, and through that further to push-adapter-integration-lib. This library then transmits the push message to Wire-Android via the same interface as FCM did previously. Changes to Figure 6.1 are highlighted with a blue background. The case for an online device has not been altered.

In the following sections, a detailed explanation of the required changes with reasons for why each change was necessary will be given.

[5] https://github.com/wireapp/wire-android-sync-engine
[6] https://github.com/wireapp/wire-audio-video-signaling
[7] https://github.com/wireapp/generic-message-proto
[8] https://github.com/wireapp/wire-android-translations

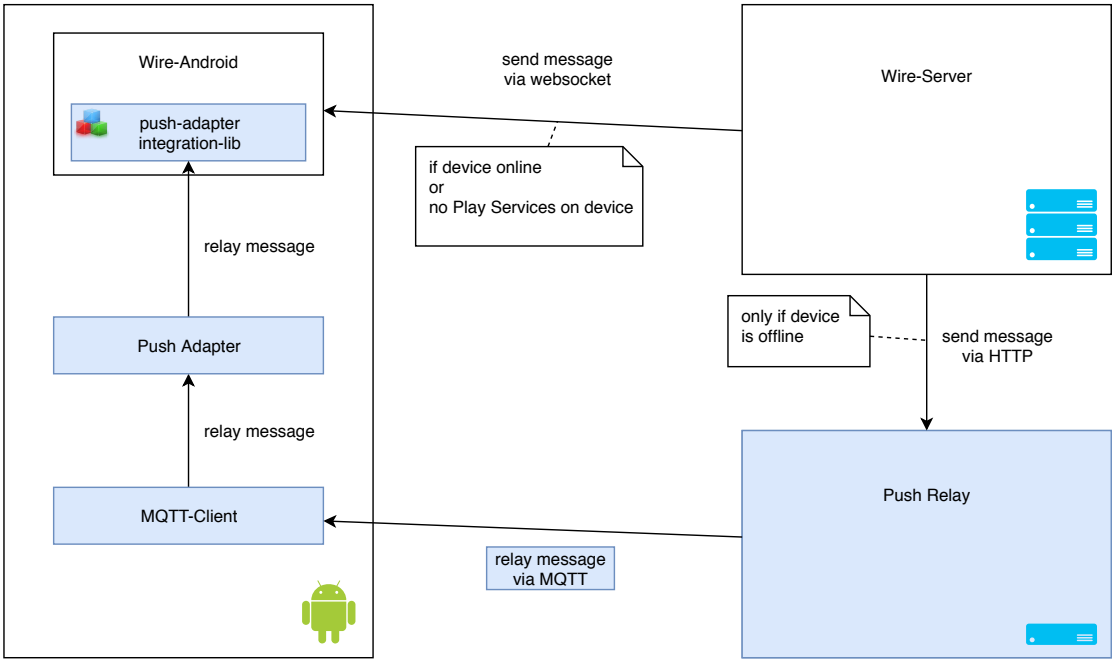

Figure 6.3: Overview Wire-Server and Wire-Android communication using Push Adapter.

6.4.1 Wire-Server

Naturally, Wire-Server had to be changed to support Push Adapter. Push messages had to be transmitted to Push Adapter, so Push Adapter can forward them to the desired device. Wire-Server's clean separation of concerns in dedicated modules and components was very beneficial when adapting it for Push Adapter. All logic and interfacing with AWS SNS and therefore with FCM was handled by Gundeck. Basically only Gundeck and nginx had to be adapted to support Push Adapter as a replacement for FCM.

6.4.1.1 Gundeck

Gundeck is the central component responsible for actually pushing messages to devices. It is implemented using *Haskell*. Wire leverages AWS SNS for pushing messages to FCM. Obviously, AWS had to be removed and be substituted with calls to our Push-Relay service.

A new Haskell module `Gundeck.PushRelay` was added to Gundeck. Previous calls to AWS were migrated and implemented in this new module. This migration was rather straightforward, once the required calls were found in the source code of Wire-Server. The module `Gundeck.PushRelay` now implements two functions:

create Function `create :: AppName -> Token -> Gundeck()` has a noop implementation. It is called once a new registration token is received. Since this token is

already intercepted by the component *Push-Integration* before it reaches Gundeck, this function has currently no purpose. See section 6.4.1.2 for details on this component and registration handling.

push The function `push :: NativePush -> Address -> Gundeck Result` is more interesting. It is responsible for forwarding the notification request to Push-Integration. It receives `NativePush` and `Address` as parameters. The Haskell definitions of those can be seen in listings 6.1 and 6.2. `Gundeck.PushRelay` only uses a subset of the provided data. From NativePush, only `notificationId` is used. It is exclusively used for logging the push request.

```
data NativePush = NativePush
    { npNotificationid :: NotificationId
    , npPriority       :: Priority
    , npApsData        :: Maybe ApsData
    }
```

Listing 6.1: Haskell definition of NativePush

```
data Address = Address
    { _addrUser      :: !UserId
    , _addrConn      :: !ConnId
    , _addrPushToken :: !PushToken
    }
  deriving (Eq, Ord)

data PushToken = PushToken
    { _tokenTransport :: !Transport
    , _tokenApp       :: !AppName
    , _token          :: !Token
    , _tokenClient    :: !ClientId
    } deriving (Eq, Ord, Show)
```

Listing 6.2: Haskell definition of Address and PushToken

The actual push token is stored in `Address`. Furthermore, Android-Wire requires the field `UserId` to correctly process incoming push messages. Therefore, those two fields are extracted and transmitted via HTTP with JSON body to Push-Integration. For this transmission the custom data type `FcmPushRequest` was created. It stores all information relevant for pushing to wire clients and implements the JSON serialization. Transmission is done by utilizing the libraries *Aeson*[9] for JSON serialization and *http-*

[9] https://github.com/bos/aeson

conduit[10], as in the rest of Wire-Server. `FcmPushRequest` implements the `ToJSON` class provided by Aeson to facilitate JSON serialization. The data type definition including the `ToJSON` implementation can be seen in listing 6.3.

```
data FcmPushRequest = FcmPushRequest !NativePush !Address

instance ToJSON FcmPushRequest where
    toJSON (FcmPushRequest _ addr) = object
        [ "validate_only" .= False
        , "message" .= object
            [
            "token" .= (addr^.addrToken)
            , "data" .= object
                [
                "user" .= (addr^.addrUser)
                ]
            ]
        ]
```

Listing 6.3: Haskell definition and JSON serialization of FcmPushRequest

6.4.1.2 Push-Integration

For simplicity and re-usability most logic for performing the actual HTTP request to Push Relay was implemented in a separately deployable service, `Push-Integration`[11]. It can be reused by other applications' servers wanting to integrate Push Relay into their environment. Decoupling the needed logic on the application provider's side in this way should ease adding support for Push Relay. It is licensed under the terms of the GNU Affero General Public License version 3 [29].

This service is written in Java, using tomcat as a web container. Spring Boot was used to ease development and save on boilerplate code. In general, this service follows Push Relay implementation in various parts and reuses common libraries, like *tls-pinning-lib*. It leverages a SQLite database to persistently store data.

It has two main endpoints:

POST /push/tokens for handling new registration requests from Wire-Android clients.

POST /send for pushing messages to Push Relay from within Gundeck.

[10] `https://www.stackage.org/lts/package/http-conduit`

[11] The source-code is available on github as part of wire-server `https://github.com/haja/wire-server`

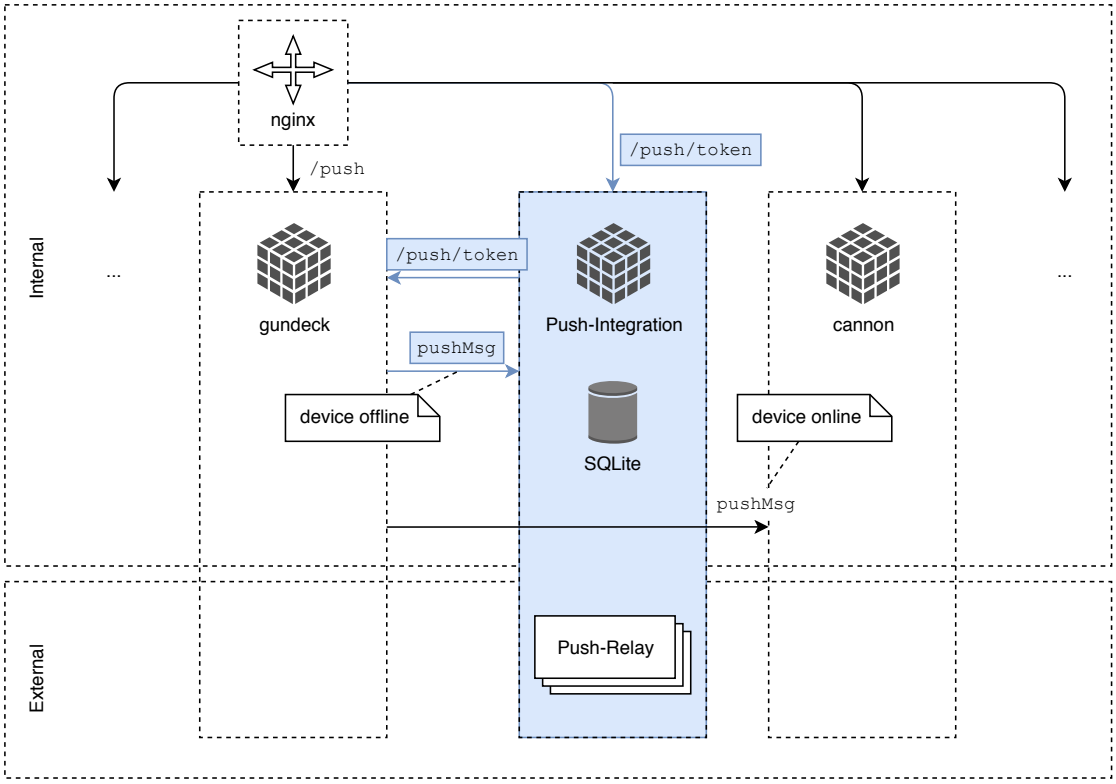

Figure 6.4: Overview Wire-Server with Push-Integration and Push-Relay.

Other implemented HTTP verbs on `/push/tokens` are `GET` and `DELETE`, which are simply forwarded to Gundeck unchanged.

Push-Integration holds all data needed for making a successful HTTP push request to the Push Relay assigned to each registered device. This data consists of the following: 1. host 2. port 3. TLS certificate 4. FCM-compatible registration token. The TLS certificate is used to verify the Push Relay in a secure manner without having to rely on any PKI. Potentially each device has its own Push Relay instance. Therefore, an individual host and port mapping per device is needed.

It proxies incoming HTTP requests for device registration. Each time a client receives a new registration token, Push-Integration will intercept this API call. This allows it to store or update the registration details mentioned above to its local database. The FCM-compatible registration token is subsequently forwarded to the corresponding *Gundeck*-endpoint. This has the added benefit that no change to the internal handling of tokens in Gundeck was necessary. The architecture overview of wire with changes incorporated to use Push-Integration are depicted in Figure 6.4. Push-Integration and corresponding changes in data flow are highlighted in blue.

Wire-Server utilizes Nginx for handling TLS termination and as a reverse proxy to redirect

incoming traffic to the corresponding component instance. To redirect the registration calls to Push-Integration, the predefined Nginx configuration had to be slightly modified. Requests to `/push/token` are now being redirected to Push-Integration. Since the websocket push implementation of Wire-Server was not changed, Wire continues to use all existing features without being intercepted by *Push-Integration*.

The Push-Integration component can be deployed using Docker. This is similar to how the other components of Wire-Server are deployed. The new component is integrated with the existing *docker-compose* deployment process, so no special setup for running Push-Integration is needed.

6.4.2 Wire-Android and Wire-Android-Sync-Engine

The Android client had to be adapted to use Push Adapter. It had to be linked against the `push-integration-lib` library, to use the FCM compatibility layer provided by Push Adapter. FCM can provide automatic registration of new apps, creating an `InstanceID` and push token automatically [67]. Using the Push Adapter library, the request for a new token had to be made manually. Alongside the push token, connection settings of the corresponding Push Relay are returned. These need to be transmitted to Wire-Server, therefore handling of registration details and the backend call had to be adapted.

App Registration For registration, a few changes in Wire-Android and Wire-Android-Sync-Engine were implemented. The class `GoogleApiImpl` of Wire-Android had to be adapted. The changes made to this class are summarized as follows:

1. The function `checkGooglePlayServicesAvailable` was modified to always return true, to allow the usage of Push Adapter regardless of whether Google Play Services are installed on the Android device.

2. `getPushToken` returns the push token generated by InstanceID. This was changed to save the connection details provided by Push Adapter as well as the actual push token. For this, the data structure `PushToken` implemented in Wire-Android-Sync-Engine was extended to hold these fields.

3. Code to initialize Googles `FirebaseApp` was removed, as the dependency on this class was removed.

Since the permission to receive messages via Push Adapter is marked as *dangerous* and therefore requires user approval, this permission had to be requested during launch of the app. The class `FirstLaunchAfterLoginFragment` was changed to request this permission during the first creation of its view. Finally, the gradle build file of Wire-Android was adapted to include push-integration-lib and the modified Wire-Android-Sync-Engine library.

Some more changes only affecting Wire-Android-Sync-Engine had to be done. Since this library is actually transmitting the push token to Wire-Server, a few more changes were needed:

1. The file `Uid.scala` contains the case class `PushToken`. This definition was extended to hold `relayUrl` and `relayCert` in addition to the actual push token. The de- and encoder definitions were adapted here as well.
2. The class `SyncRequest` is responsible for transmitting the push token to Wire-Server. It was adapted to handle the new PushToken structure properly.
3. Minor changes to handle the adapted PushToken data structure were needed in the classes `Preferences`, `AccountDataOld`, `Event`, `PushTokenService`, `PushTokenClient`, `PushTokenSyncHandler` and `JsonDecoder`.

Overall, those changes were only minor and pretty straightforward once the design and inner workings of Wire-Android were discerned.

Receiving Messages The changes required to receiving messages via Push Adapter instead of FCM were even less. Only the class `FCMHandlerService` had to be touched. Here, no structural changes were made. Only the imports had to be adapted to import the classes `FirebaseMessagingService` and `RemoteMessage` from push-integration-lib instead of FCM's client libraries.

Android 8 and Doze Whitelist Android 8 introduced new power saving features. Among other changes, these introduced restrictions on background processing and network IO. This made certain changes to Wire-Android necessary: While processing push messages, a foreground service needs to be started. This includes displaying a low priority notification. A service started in that way can successfully receive and process intents from other apps, such as from Push Adapter when receiving a new push message. Still, certain restrictions apply. Wire-Android is only waking up via these push messages to load the actual message content via a direct request to Wire-Server. This requires network access, which is still restricted on these new Android versions, even for foreground services. Therefore, Wire-Android, just like Mqtt-Client, needs to be exempt from *battery optimization*. Otherwise, the push message will be received and processed by Wire-Android, but any network access will receive a timeout after a few seconds.

CHAPTER 7

Evaluation

The following section (7.1) describes how Push Adapter was evaluated and compared to FCM. Measurements of network data usage and battery consumption were conducted. Detailed explanations of how the measurements were conducted are presented, including problems encountered during the evaluation process. Section 7.2 presents the adversary model for my implementation.

7.1 Comparing Push Adapter to FCM

Technical aspects of the implementation that have been measured include:

- Energy Usage on an Android device
- Network Data Usage

All measurements were conducted on the same smartphone. A *Samsung Galaxy S4 GT-I9506* with Android version *9* as provided by *LineageOS 16.0* was used. Automatic app battery optimizations offered by Android were disabled so they would not interfere with the receiving of messages. Further details about the Android version used can be found in Figure 7.1.

For all measurements using *Push Adapter*, my customized Wire Server instance as described in Chapter 6 was used. For comparing Push Adapter to FCM, the unmodified, but otherwise identical version of the Wire Android app was used. It was connected to the official Wire Server instances. This should yield comparable results, as energy consumption and data usage should only depend on the messages sent and the Wire Android version used, not on the way the backend service is hosted. FCM was evaluated using Google apps provided and packaged by *The Open GApps Project*[75]. This project provides Google apps in various builds, architectures and sizes. For the phone used, the

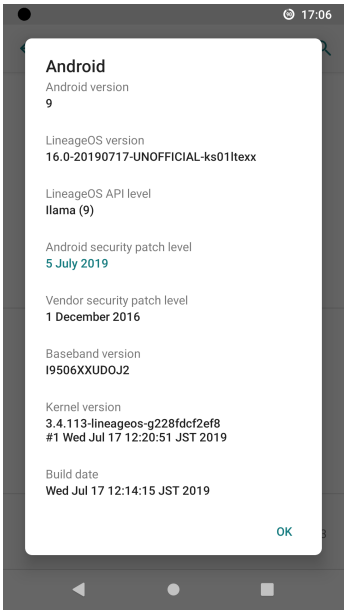

Figure 7.1: Android version used for analysis.

build for ARM (32 bit) for Android 9.0 was required. Since only basic Google services were needed to provide FCM, the variant *nano* was chosen. The Open GApps version from January the 3rd, 2020[1] was used.

My measurements focus on energy consumption and data usage on Android. Other factors for evaluation, e.g. timeliness of messages, were evaluated as well, but might not be as representative. See [78] for a thorough evaluation of MQTT, FCM and APNs performance characteristics.

7.1.1 Wire-Analysis

Wire lacks an easy to script client. For the measurements *wire-analysis*[2], a wrapper around wire-webapp, was created. It leverages Selenium[3] to interact with the wire web client. It provides the following features through a Java API:

- Login with a previously created user
- Sending of messages to a predefined user
- Detection of message received notifications[4]
- Logout detection and reestablishing of user sessions

[1] The full changelog can be found in the changelog[55].
[2] `https://github.com/haja/wire-analysis`
[3] `https://selenium.dev`
[4] Messages are being considered received when the string `delivered` is displayed next to a message in the Wire Webapp conversation. This check is carried out once per second. This adds some extra delay to the actual instant a message is received on the device.

- Gathering of statistics of message sent and received timestamps
- Reliable export of these statistics to file

To be able to compare the results, the intervals at which messages are sent were generated using a pseudo random number generator with a predefined seed. This seed was used in every measurement run for both technologies, FCM and Push Adapter. This guarantees the same interval times for each run and technology. Interval limits and seed can be configured using a simple properties file.

7.1.2 Energy Usage

Energy usage on Android devices is a major design goal of Push Adapter. To evaluate this property of my implementation, the following process was conducted repeatedly:

1. Fully charge the phone and disconnect it from the charger.
2. Connect the device to mobile networks.
3. Place the phone in a spot with acceptable reception.
4. Send messages to the device at random intervals between 1 and 180 seconds.
5. Take note of the duration until the phone powers itself off.

Ideally, a stable Wi-Fi connection would have been preferable. However, at least with this phone running Android 9, this was not possible. Wi-Fi turned itself off consistently after a few minutes after switching off the screen. Also, setting the Wi-Fi policy to never turn off Wi-Fi did not help to remedy this. The intervals noted in item (4) should represent valid usage patterns. It should allow for some sleep intervals, where the phone can save power but also some high intensity usage patterns. The phone is assumed to be out of battery as soon as new messages are not received anymore. Verification of this fact are conducted manually (e.g. checking that the phone is actually dead). Furthermore, to pinpoint the time of the shutdown further, manual checks of the Android logs were conducted. They showed that the phone shutdown correlates closely with the last message received timestamp as observed by Wire-Analysis.

The first two runs with FCM were inconclusive, seemingly FCM stopped delivering messages to the device after some time. Even after waiting several hours, the device was not receiving any new messages. For the third and fourth run, Wire was configured to be exempted from battery optimizations, suspecting some power saving algorithm might be interfering with the measurements. Yet this did not eliminate the problem completely. Latency spikes still remained. Why FCM did not work for the whole test duration remains unclear. Perhaps, if the app receiving messages via FCM is not interacted with for some time or has received a certain amount of messages without interaction, FCM stops delivering messages for that app. Further investigation is needed to understand this problem better. The goal was to conduct two measurement runs for each technology. Since FCM was quite problematic and inconclusive, more than two runs had to be conducted.

7.1.3 Data Usage

Efficient network data usage is another important design goal of Push Adapter. The network data usage measurements were conducted similarly to the energy consumption measurements. Two measurement types were conducted: one to establish how much data is used when actively using Wire, and another one to measure the data usage of maintaining an idle connection. For the latter measurement step 4 was simply skipped.

1. Fully charge the phone and disconnect it from the charger.
2. Connect to mobile networks.
3. Place phone in a spot with acceptable reception.
4. Send messages to the device at random intervals between 1 and 180 seconds.
5. After 2 hours, take note of the data usage for the relevant apps.

The same reasons as in the previous Section 7.1.2 regarding the use of mobile networks apply here. For data usage readings as described by item (5), the data usage as recorded and provided by Android was used.

7.2 Security Evaluation

My implementation is a prototype. As such, it does not handle all attack surfaces equally well. Nevertheless, my implementation follows basic security concepts and tries to avoid major security issues. To evaluate Push Adapter regarding its security concepts I describe an adversary model. My results on addressing this adversary are presented in Section 8.2. Some security improvements that broaden the threat model are described in Section 10.2.

7.2.1 Adversary Model

I assume an adversary that can intercept traffic and mount MITM (man-in-the-middle) attacks. Furthermore, the adversary has capabilities to generate valid certificates for any domain. The attacker is assumed to be able to register apps and devices. It can use malicious clients, generating malformed and invalid requests. Furthermore, it is assumed to be able to operate a general purpose MQTT-Client.

I assume the provider of Push Adapter, hosting Push Relay, is trustworthy and competent. As such, the provider is expected to keep client certificates, CA certificates, sensitive configuration options (such as passwords to certificate stores) and generated access tokens confidential and secure. Furthermore, I assume that the provider is not spying on its users. I exclude denial of service attacks from the threat model.

CHAPTER 8

Results

Here I discuss the results for my measurements. First, the battery runtime evaluation of Android devices using Push Adapter and FCM are presented. Some interesting findings regarding latency when using FCM were found and are discussed here as well. Secondly, the results of my network data usage measurements for Android are presented. These show a clear advantage for Push Adapter.

Finally, Section 8.2 presents features and countermeasures regarding security. It describes the security architecture of my implementation and details which countermeasures help against which capability of the adversary. Further considerations about the system's security as well as excluded attack surfaces are presented in Chapter 10, Section 10.2.

8.1 Comparing Push Adapter to FCM

This section presents the results for measuring the battery lifetime and network data usage. I compare the performance of my implementation *Push Adapter* to Google's *FCM* service.

8.1.1 Battery Usage

Table 8.1 shows the aggregated measurement results for battery runtime for each run respectively.

For Push Adapter, battery lifetimes of *16 hours, 30 minutes* and *15 hours, 44 minutes* were measured for runs one and two respectively. The latency for messages was *4.8* and *6.5* seconds on average. The highest latency spike was *42 seconds* for run one and *1 minute 18 seconds* for run two. In total, Push Adapter managed to deliver *666* messages for run one and *631* for run two until the phone battery died.

	run	msgs recv	battery lifetime	mean/min/max latency
Push Adapter	1	666	16h 30m 15s	4.88s/2.10s/41.69s
	2	631	15h 44m 45s	6.51s/2.44s/1m 18.44s
FCM	1		inconclusive	
	2		inconclusive	
	3	860	21h 16m 32s	3m 42.82s/2.29s/38m 26.01s
	4	821	20h 19m 51s	3m 59.55s/2.07s/32m 13.73s

Table 8.1: Results for measuring battery lifetime for Push Adapter and FCM.

FCM managed to keep the phone alive longer. Run three, the first run to deliver usable results, showed a battery lifetime of *21 hours and 16 minutes*. Run four had a similarly long battery lifetime with *20 hours and 19 minutes*. The longer lifetime comes at a cost of service quality, though. Message latency was way higher on average than when using Push Adapter. While minimum latency for both implementations is slightly above two seconds, the average latency for FCM is *three minutes and 43 seconds* for run three and *4 minutes* for run four. The highest latency spikes observed are *38 minutes* and *32 minutes* for runs three and four, respectively. FCM managed to deliver *860* messages during run three and *821* messages during run four in total.

8.1.1.1 Latency of Messages

Figure 8.1 shows latency of messages for each run. The messages are plotted with their time sent relatively to starting the measurements and their latency until marked as received. This clearly shows how large the latency spikes for FCM are compared to Push Adapter.

Push Adapter Figure 8.2 shows the latency of messages sent to my Wire proof of concept implementation, using Push Adapter to relay push notifications. There are some latency spikes, especially at the end of each run. Overall, these spikes are rarely higher than one minute. The graph shows the time of day when each message was sent. The spikes do not seem related to the time of day. Only during run one, a slight overall increase in latency can be seen.

FCM Figure 8.3 shows the latency of messages when using FCM to relay push notifications to Wire. It can be seen that there are *eleven* and *13* spikes higher than 15 minutes where no messages were delivered. This time without delivery to the phone adds up. If all latency spikes higher than 15 minutes are summed up, the phone is not receiving messages for *4 hours and 58 minutes* during run three and for *5 hours and 8 minutes* during run four. Subtracting this offline time from the measured runtime, FCM provides roughly the same battery lifetime as Push Adapter, while being less reliable.

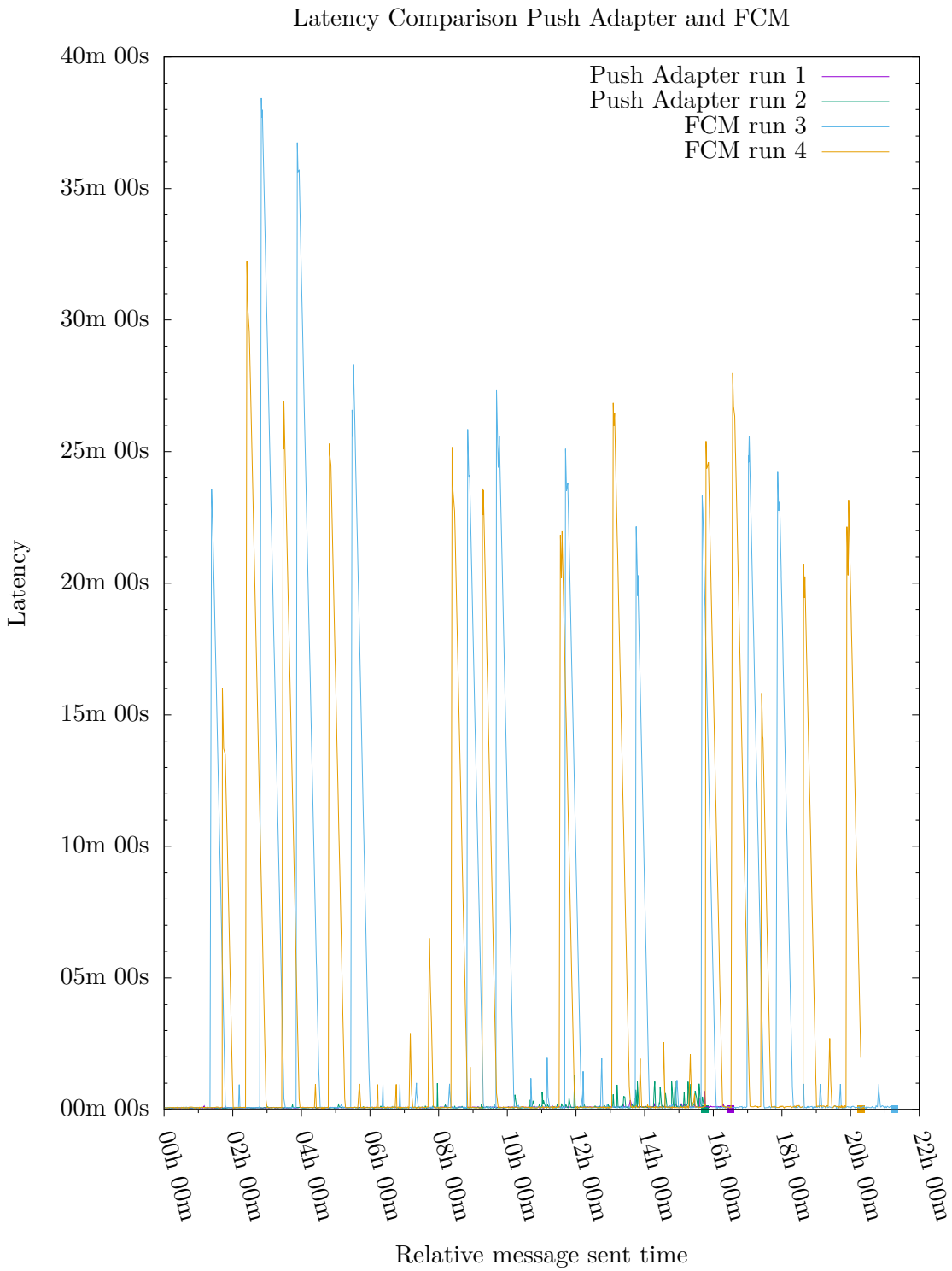

Figure 8.1: Comparison of latency of Push Adapter and FCM.

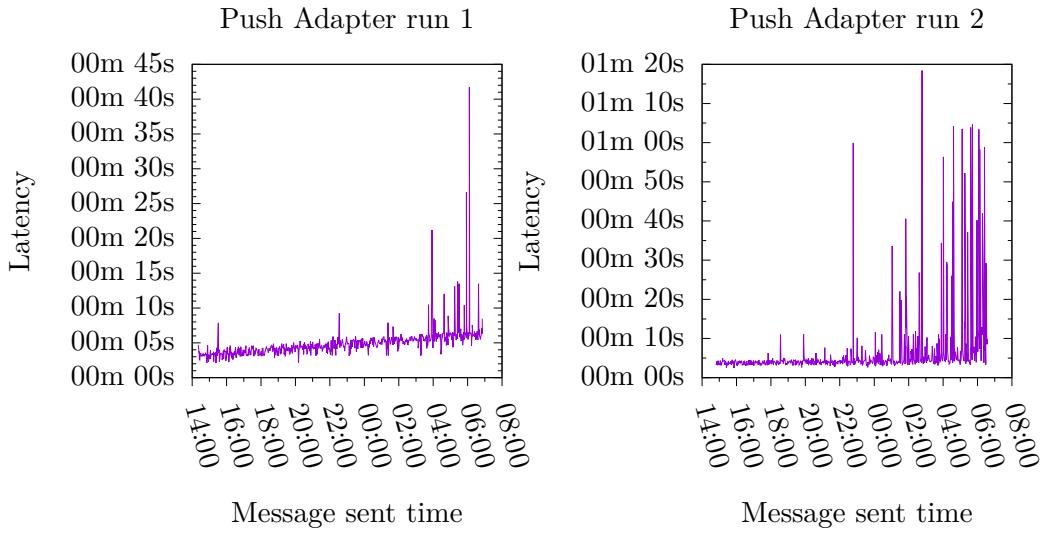

Figure 8.2: No high latency spikes for Push Adapter on run 1 or 2.

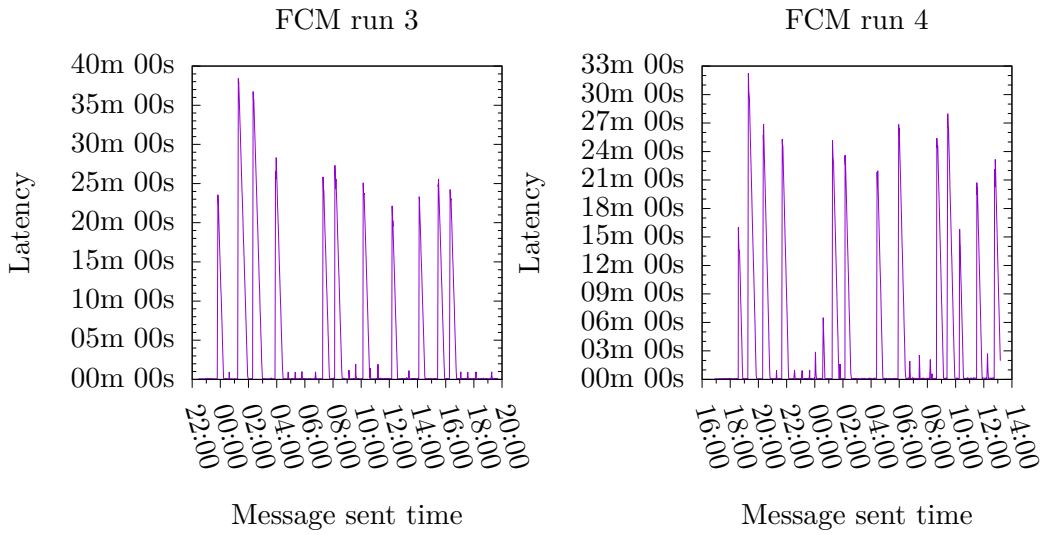

Figure 8.3: Latency spikes of FCM run 3 and 4.

	messages	data usage			
		Wire	push	Android OS	push + OS
Push Adapter	84	329 kB	76 kB	100 kB	176 kB
FCM		581 kB	360 kB	43 kB	403 kB
Push Adapter	0	< 1 kB	27 kB	2 kB	29 kB
FCM		< 10 kB	40 kB	8 kB	48 kB

Table 8.2: Data usage after two hours for Push Adapter and FCM.

8.1.2 Data Usage

Table 8.2 shows the measurement results regarding network data usage. Two runs for each technology were conducted, each two hours long. One run included sending messages at the same interval as for measuring battery lifetime. This means that during the measurement time, 84 messages were sent and received by the device. During the second run, no messages were sent. This establishes a baseline for how much data is used while the system is idle. Figure 8.4 shows that no major latency spikes were measured during the test period, neither for FCM nor for Push Adapter.

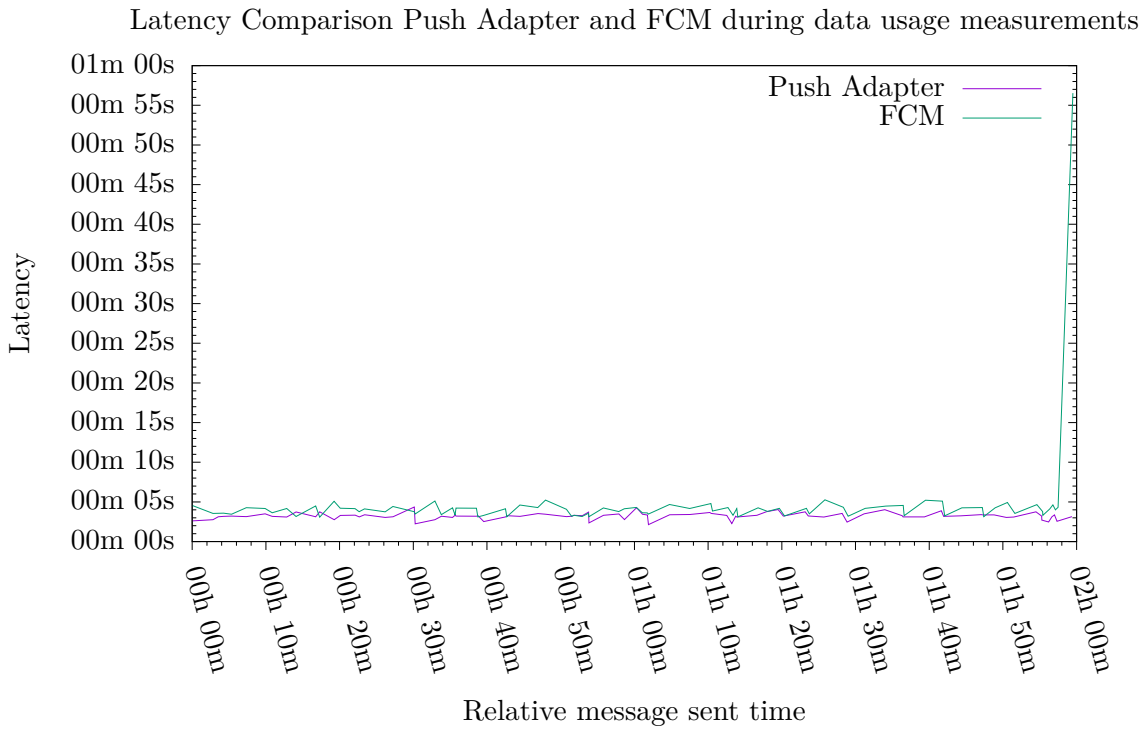

Figure 8.4: Comparison of latency of Push Adapter and FCM during data usage measurements.

Push Adapter uses 27 kB of data transfer while being idle, while FCM uses 40 kB. During usage, these statistics differ more. Push Adapter needs to transfer 76 kB and induces 100 kB of data usage on the Android OS. FCM has more overhead. For notifying about the same 84 messages, FCM uses 360 kB itself and incurs additional 42 kB on the Android OS. In total, FCM uses more than twice as much network data as Push Adapter. Whether all of Android OS data usage should be counted towards the push message service is hard to decide. Sadly, other apps can still make network requests and therefore be responsible for this usage. Although no apps except the ones needed for the setup were installed, certain pre installed apps still remain on the phone, e.g. the Google Play store. Wire data usage itself also differs for both technologies. It is hard to tell why Wire apparently uses almost twice as much data when running with FCM instead of Push Adapter.

8.2 Security Features of Push Adapter

Push Adapter implements several security features that help to protect against the adversary model described in Section 7.2.1. The following paragraphs give a detailed explanation of each feature. In Section 8.2.1 I will explain how this helps to protect against each capability of the expected adversary.

TLS for all connections All traffic in the system is encrypted using TLS 1.2. This includes all REST endpoints and the connections to the MQTT relay. No fallback to not encrypted endpoints is configured.

TLS Certificate Pinning The Android apps MQTT Client and Push Adapter are built with the relays' certificate embedded. They only use this certificate to verify connections to the relay. Therefore, they do not rely on third party public key infrastructure. This applies to HTTPS connections as well as to MQTT connections. These measures implement certificate pinning for Android devices. Push Adapter allows third party app developers to verify the certificate of Push Relay as well. Push Adapter on Android hands over the certificate of Push Relay to client apps at registration time. This allows app developers to verify their connection with Push Relay when using its API. Therefore, third party app developers can also implement certificate pinning for their connections to Push Relay.

TLS Client Certificates for Authentication Push Relay has a unique Certificate Authority (CA) built in. Each Android device receives a unique TLS Client Certificate upon registration with the Push Adapter service. This certificate is signed by the CA of Push Relay. Each connection to the MQTT relay as well as each HTTP request to register a new Android app is verified against this CA. If no client certificate is provided or the provided certificate is invalid, the connection is rejected.

Access to MQTT Topics MQTT uses a publish/subscribe messaging pattern. Publishers can write to a *topic* and interested observers can subscribe to these topics. This

means that potentially multiple clients can read on the same topic or write to arbitrary topics. As described in Section 5.1.4, much effort has gone into developing a system that allows confidential one-to-one messaging on top of this publish/subscribe model.

The MQTT broker used to implement Push Relay is *Apache ActiveMQ*[7]. This broker has built in plugins to control topic access based on Access Control Lists (ACLs). ACLs can be configured at runtime. If this is not sufficient, custom authorization plugins can be implemented to further limit access. Acceptance or rejection of each event (e.g. topic creation, topic subscription etc.) and each message on a topic can be decided by arbitrary logic.

My implementation limits write access to one special user that is created at the startup of Push Relay. It is initialized with a new, random password on each startup of the service. To further limit this user, it can only connect via the internal `jvm` protocol. This keeps external users, connected in any other way (e.g. via the TLS endpoint used for pushing messages to Android clients), from writing to *any* topic at all. Read access is handled based on TLS client certificates. Since each Android device gets assigned a unique certificate, this is sufficient to identify each device uniquely. Each certificate holds a unique value in its `CN` field. This value is used to give the client read access to the topic of the same value. Subscription requests on other topics than the one matching the `CN` field will be rejected.

Keeping Push Token confidential The push token plays a central role in securing communication between Push Providers and their apps using Push Adapter. Therefore, it is paramount to keep this token secure, avoiding interception of this token. Some measures were put in place to ensure the confidentiality of the token:

- The token is only transmitted via TLS encrypted connection.
- Only at app registration time is the token being transmitted between Push Relay and Push Adapter. It can not be requested after this point.
- On Android, the push token is passed only by intents with defined target package, tailored to the requesting app. Other apps can not intercept these requests.
- Furthermore, the intent is secured by a custom permission of the security class `dangerous`. This ensures that apps need to be approved by the phone's user to use Push Adapter.

However, no guarantees about the handling of the token outside of my system (e.g. by Push Providers) can be made, naturally. This remains an issue, since the token needs to be permanently stored by the Push Provider to use Push Adapter. Nevertheless, this is the same behavior as with FCM, with the same problems attached to it.

8.2.1 Security Features in Context to the Adversary Model

The adversary model employed by the implementation is described in Section 7.2.1. This threat model has several capabilities that can be discussed separately.

The first capability of the adversary model includes intercepting and modifying traffic between Push Adapter and its clients. Against this class of attacks, TLS is employed to encrypt and verify packets transmitted between Push Relay, Push Adapter, MQTT-Client and third party push providers. Since the latest version of TLS, 1.2, is used, it should protect against this adversary adequately at the time of this writing.

The adversary model also includes a hardened version of the classic MITM attacker. It is extended to also include capabilities to infiltrate PKI (Public Key Infrastructure). It is expected that an attacker can acquire certificates that are valid for any domain when queried against PKI authorities. To prevent such attacks from succeeding, certificate pinning is employed. Push Adapter and MQTT-Client are pre-configured at compile time with the certificate of their configured Push Relay instance. They refuse to establish a connection to a service that does not present exactly this certificate and proof of having the according private key, as specified by TLS. No PKI infrastructure is queried.

Attackers that acquire a copy of Push Adapters client apps can successfully connect to the matching Push Relay. They can use the service like any other user. But the measures to separate each client on their own MQTT topic still apply, as described in Section 5.1.4. Therefore, attackers cannot eavesdrop on other clients' push messages. They can also not send messages to other clients' end devices as they would need to guess their push token. Since the push token consists of a random string with a length of 32 bytes, this is currently believed to be impossible.

CHAPTER 9

Conclusion

I implemented and evaluated *Push Adapter*, an open source framework and service for pushing messages to Android devices. In that regard, it is an alternative to Firebase Cloud Messaging (FCM) downstream messages. Although not all functionality of FCM was implemented, a core mechanism essential for instant messaging was created. This shows that it is feasible to replace FCM with an open source implementation for pushing messages to Android devices. The provided service is a distributed system which can be hosted by independent people and organizations for their own uses.

The prototype implementation is built around the protocol *Message Queuing Telemetry Transport (MQTT)*. It is a binary protocol that is optimized for small overhead and offers interesting quality of service capabilities. The central component is a MQTT Broker that relays messages between clients. Since MQTT uses a publish/subscribe model based on topics, much effort has gone into setting up the broker in such a way that it can be used for pushing messages to one device and one device only.

Security was another concern for my implementation. TLS is used throughout the whole application. For securing the permanent background connection between Android devices and Push Relay, TLS Client Certificates are used.

Another cornerstone of the implementation is its compatibility to FCM from an app developers perspective. Existing apps that are currently dependent on FCM to deliver push messages can easily be retrofitted to use my implementation. To prove this point, a working proof of concept was implemented. The instant messaging app *Wire* was adapted to use my implementation for delivery of push messages instead of FCM. My work documents what changes to Wire were needed to adapt Push Adapter.

I have conducted extensive measurements to compare Push Adapter with FCM using the proof of concept implemented in Wire. The measurements include comparison of battery lifetime and network data usage of Android devices employing either implementation. The average battery lifetime when using FCM is 20 hours and 45 minutes, compared to

16 hours and 7 minutes when using Push Adapter. However, my measurements show that FCM, although providing longer battery lifetime, also incurs high latency spikes that decrease the quality of the service. These spikes occur frequently at irregular intervals. On average, a spike occurs every two hours and last an average 26 minutes. During these latency spikes, the Android device is not woken up although new messages are waiting to be delivered. These spikes are substantial. Taking into account the time that the device appears as offline, the online time of both services evens out. Regarding network data usage, Push Adapter can clearly outperform FCM. It uses less than half the data when being actively used to push messages to Android devices every few minutes. Even while being idle, Push Adapter transfers less data for maintaining a connection than FCM. The reduction in data usage is about one quarter.

CHAPTER 10

Future Work

As Push Adapter is a novel yet complex system, there is still room for improvement. Here I want to present some tasks for future research regarding the features of Push Adapter, its threat model and security features as well as other technical tasks.

10.1 Limitations of Push Adapter

Push Adapter is a prototype implementation. As such, it has certain limitations and leaves areas for improvement. Here I want to give an overview of future research topics regarding Push Adapter and its implementation.

Improve Adversary Model The current implementation does not implement measures to handle some harder to counter attacks. As such, (D)DOS[1] attacks and network analysis attacks are beyond the current project's scope.

More than one Push Relay per Device Currently, only one Push Relay, the server component of Push Adapter, can be used per device. Users might want to further separate their push service provider on a per-app basis. With the current implementation, such a scenario is not supported.

Rate Limiting Push Adapter is not limiting the rate at which push messages might be sent to end user's devices. This could be potentially abused for all kinds of attacks. Rate limiting the requests via the REST API for sending push messages as well as registering new devices and apps on those devices could remedy some of those problems.

[1] (Distributed) Denial of Service

Device Registration Currently, new devices can be registered without limit. No valid email address or similar means of identification are required by Push Adapter. Therefore, entities which provide Push Adapter as a service cannot limit their users based on such means of identification.

User Management Devices requesting registration are always granted a new identity and can register. Once registered, there are currently no means to remove devices or registered apps on those devices. Push Adapter service providers might want to remove users from their service for various reasons.

Reliability and Operations The service is only a prototype implementation. Operating the service in a production environment was no design or implementation goal. How to make Push Adapter ready for production was not evaluated.

Battery Runtime Optimizations If the offline periods of FCM shown during my measurements in Chapter 8 are regarded as a power saving feature, implementing a similar feature for Push Adapter can be seen as a future improvement. For example, gathering usage statistics of client apps and implementing an algorithm which optimizes battery lifetime using these statistics could be useful.

10.2 Broaden Adversary Model of Push Adapter

The adversary model described in Section 7.2.1 is chosen in such a way that it is useful for evaluation and addresses the most pressing issue with using FCM currently. Yet a broader threat model could prove useful should this prototype be expanded on. The following paragraphs provide a non exhaustive list of issues and ideas the current implementation can be improved on in the future.

Resource Exhaustion All kinds of attacks that lead to resource exhaustion are excluded from the adversary model. This includes (Distributed) Denial of service (DDoS) attacks among others. Registering new devices can lead to unexpected high loads on Push Relay since a new certificate needs to be generated and signed and a new MQTT topic has to be created for each new device. Registering new apps can have a similar effect since a Push Token is generated for each request and stored in an internal database. The latter issue is lessened by the requirement of having a valid TLS Client Certificate to register new apps.

Certificate renewal No certificate renewal mechanism is currently implemented. Certificates once issued by Push Relay and signed by its CA are never exchanged. Also, no fallback certificate for Android clients to use in case the certificate of Push Relay needs to be renewed is defined. These issues should be addressed for a production system. However, to just provide a proof of concept, these details were excluded from the current implementation.

Push Token renewal A similar issue persists with Push Tokens. These are never exchanged and therefore can be used indefinitely. However, apps registering with Push Adapter can request a new Push Token by simply registering again.

End-to-end Encryption of Messages Since the adversary model assumes the host of Push Adapter to be trustworthy, no end to end encryption mechanism was implemented regarding the payload of messages. I assume that Push Adapter is only used for waking up end user's devices. Therefore, messages should not include any payload that contains plain text information. As this is sufficient for my proof of concept using *Wire* (see Chapter 6), I regard this as a minor issue.

Use of TLS 1.3 This work is published at a time where TLS 1.3 is not readily available. Therefore, it uses TLS 1.2 Client Certificates, which are transmitted in plain text. This creates a possibility to track users based on their TLS Client Certificates for passive observers. This attack surface is mitigated by TLS 1.3 [77].

10.3 MQTT Alternatives

Push Adapter employs MQTT to transport push messages from Push Relay to its Android clients. MQTT, however, is not the only viable protocol for this task. There are several competitors that could be useful or better alternatives. Thorough evaluation of these alternatives should be conducted in the future.

Server-sent Events Server-sent Events (SSE) are part of the HTML Standard [37]. These are specified as a simple way to push messages from a server to clients. The specification includes a section on energy saving support via *connectionless push*.

CoAP CoAP (Constrained Application Protocol) is a protocol designed by the CoRE (Constrained Resource Environments) IETF group [68]. It is a document transfer protocol, tailored to requirements of typical IoT devices. Unlike MQTT, CoAP is built on top of UDP. It might allow for even lower data transfer overhead while still offering two QoS levels. [42]

List of Figures

List of Tables

Listings

Bibliography

[1] *[Messaging] How Secure Is TextSecure?* URL: https://moderncrypto.org/mail-archive/messaging/2014/001030.html (visited on 12/19/2019).

[2] *About · Wire*. URL: https://wire.com/en/about/ (visited on 01/29/2020).

[3] *About FCM Messages*. URL: https://firebase.google.com/docs/cloud-messaging/concept-options (visited on 07/20/2019).

[4] *Android Developer Guides: Services Overview*. URL: https://developer.android.com/guide/components/services (visited on 01/24/2020).

[5] *Android Software Development Kit License Agreement*. URL: https://developer.android.com/studio/terms (visited on 07/26/2019).

[6] *AP Exclusive: Google Tracks Your Movements, like It or Not*. Aug. 13, 2018. URL: https://apnews.com/828aefab64d4411bac257a07c1af0ecb (visited on 12/21/2019).

[7] *Apache ActiveMQ Homepage*. URL: http://activemq.apache.org/ (visited on 07/28/2019).

[8] *Apache License, Version 2.0*. Jan. 2004. URL: https://www.apache.org/licenses/LICENSE-2.0.html (visited on 01/24/2020).

[9] "Apple Challenged over iPhone Location Settings". In: *BBC News. Technology* (Dec. 5, 2019). URL: https://www.bbc.com/news/technology-50673031 (visited on 12/21/2019).

[10] Noah Apthorpe, Dillon Reisman, and Nick Feamster. "A Smart Home Is No Castle: Privacy Vulnerabilities of Encrypted IoT Traffic". In: (May 18, 2017). arXiv: 1705.06805 [cs]. URL: http://arxiv.org/abs/1705.06805 (visited on 12/18/2019).

[11] Noah Apthorpe, Dillon Reisman, and Nick Feamster. "Closing the Blinds: Four Strategies for Protecting Smart Home Privacy from Network Observers". In: (May 18, 2017). arXiv: 1705.06809 [cs]. URL: http://arxiv.org/abs/1705.06809 (visited on 12/18/2019).

[12] Noah Apthorpe et al. "Spying on the Smart Home: Privacy Attacks and Defenses on Encrypted IoT Traffic". In: (Aug. 16, 2017). arXiv: 1708.05044 [cs]. URL: http://arxiv.org/abs/1708.05044 (visited on 12/18/2019).

[13] JP Aumasson and Markus Vervier. *Wire Security Review – Phase 1*. Feb. 8, 2017. URL: https://wire-docs.wire.com/download/Wire+Audit+Report.pdf (visited on 12/22/2019).

[14] JP Aumasson and Markus Vervier. *Wire Security Review – Phase 2 – Android Client*. Mar. 7, 2018. URL: https://www.x41-dsec.de/reports/X41-Kudelski-Wire-Security-Review-Android.pdf (visited on 12/22/2019).

[15] JP Aumasson and Markus Vervier. *Wire Security Review – Phase 2 – iOS Client*. Mar. 7, 2018. URL: https://www.x41-dsec.de/reports/X41-Kudelski-Wire-Security-Review-iOS.pdf (visited on 12/22/2019).

[16] JP Aumasson and Markus Vervier. *Wire Security Review – Phase 2 – Web, Calling*. Mar. 7, 2018. URL: https://www.x41-dsec.de/reports/X41-Kudelski-Wire-Security-Review-Web-Calling.pdf (visited on 12/22/2019).

[17] "AUPS: An Open Source AUthenticated Publish/Subscribe System for the Internet of Things". In: *Information Systems* 62 (Dec. 1, 2016), pp. 29–41. ISSN: 0306-4379. DOI: 10.1016/j.is.2016.05.004. URL: https://www.sciencedirect.com/science/article/abs/pii/S030643791630237X (visited on 12/18/2019).

[18] A. Bhawiyuga, M. Data, and A. Warda. "Architectural Design of Token Based Authentication of MQTT Protocol in Constrained IoT Device". In: *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)*. 2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA). Oct. 2017, pp. 1–4. DOI: 10.1109/TSSA.2017.8272933.

[19] Shi-Cho Cha et al. "Privacy Enhancing Technologies in the Internet of Things: Perspectives and Challenges". In: *IEEE Internet of Things Journal* 6.2 (Apr. 2019), pp. 2159–2187. ISSN: 2372-2541. DOI: 10.1109/JIOT.2018.2878658.

[20] Wei Chen et al. "On Measuring Cloud-Based Push Services:" in: *International Journal of Web Services Research* 13.1 (Jan. 2016), pp. 53–68. ISSN: 1545-7362, 1546-5004. DOI: 10.4018/IJWSR.2016010104. URL: http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/IJWSR.2016010104 (visited on 12/18/2019).

[21] K. Cohn-Gordon et al. "A Formal Security Analysis of the Signal Messaging Protocol". In: *2017 IEEE European Symposium on Security and Privacy (EuroS P)*. 2017 IEEE European Symposium on Security and Privacy (EuroS P). Apr. 2017, pp. 451–466. DOI: 10.1109/EuroSP.2017.27.

[22] Katriel Cohn-Gordon et al. "On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees". In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, Aug. 10, 2018, pp. 1802–1819. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243747. URL: http://dl.acm.org/citation.cfm?id=3243734.3243747 (visited on 12/19/2019).

[23] *Eclipse Distribution License - Version 1.0*. URL: https://www.eclipse.org/org/documents/edl-v10.html (visited on 01/27/2020).

[24] *Eclipse Paho - MQTT and MQTT-SN Software - Android*. URL: https://www.eclipse.org/paho/clients/android/ (visited on 07/20/2019).

[25] *Eclipse Public License - Version 1.0*. URL: https://www.eclipse.org/legal/epl-v10.html (visited on 01/24/2020).

[26] Tatiana Ermakova et al. "Web Tracking - A Literature Review on the State of Research". In: *Hawaii International Conference on System Sciences 2018 (HICSS-51)* (Jan. 3, 2018). URL: https://aisel.aisnet.org/hicss-51/os/information_security/5.

[27] *Firebase Android SDK. GitHub*. Firebase, July 25, 2019. URL: https://github.com/firebase/firebase-android-sdk (visited on 07/26/2019).

[28] Huber Raul Flores Macario and Satish Srirama. "Mobile Cloud Messaging Supported by XMPP Primitives". In: *Proceeding of the Fourth ACM Workshop on Mobile Cloud Computing and Services - MCS '13*. Proceeding of the Fourth ACM Workshop. Taipei, Taiwan: ACM Press, 2013, p. 17. ISBN: 978-1-4503-2072-6. DOI: 10.1145/2497306.2482983. URL: http://dl.acm.org/citation.cfm?doid=2497306.2482983 (visited on 01/25/2019).

[29] Free Software Foundation. *GNU Affero General Public License Version 3*. Nov. 19, 2007. URL: https://www.gnu.org/licenses/agpl-3.0.en.html (visited on 01/27/2020).

[30] Free Software Foundation. *GNU General Public License Version 3*. June 29, 2007. URL: https://www.gnu.org/licenses/gpl-3.0.html (visited on 01/24/2020).

[31] Chris Foxx. "Google and Facebook Face GDPR Complaints". In: *BBC News. Technology* (May 25, 2018). URL: https://www.bbc.com/news/technology-44252327 (visited on 12/21/2019).

[32] Tilman Frosch et al. "How Secure Is TextSecure?" In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2016 IEEE European Symposium on Security and Privacy (EuroS&P). Saarbrucken: IEEE, Mar. 2016, pp. 457–472. ISBN: 978-1-5090-1751-5. DOI: 10.1109/EuroSP.2016.41. URL: http://ieeexplore.ieee.org/document/7467371/ (visited on 01/25/2019).

[33] Wire Swiss GmbH. *Wire Security Whitepaper*. Aug. 17, 2018. URL: https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf (visited on 12/22/2019).

[34] *H2 Database Engine*. URL: https://www.h2database.com/html/main.html (visited on 07/28/2019).

[35] Johannes Heurix et al. "A Taxonomy for Privacy Enhancing Technologies". In: *Computers & Security* 53 (Sept. 2015), pp. 1–17. ISSN: 01674048. DOI: 10.1016/j.cose.2015.05.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167404815000668 (visited on 11/15/2019).

[36] *How Tutanota Replaced Google's FCM with Their Own Notification System | F-Droid - Free and Open Source Android App Repository*. URL: https://www.f-droid.org/en/2018/09/03/replacing-gcm-in-tutanota.html (visited on 12/21/2019).

[37] *HTML Standard - Server-Sent Events*. URL: https://html.spec.whatwg.org/multipage/server-sent-events.html (visited on 12/03/2019).

[38] Sana Imtiaz, Ramin Sadre, and Vladimir Vlassov. "On the Case of Privacy in the IoT Ecosystem: A Survey". In: *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). July 2019, pp. 1015–1024. DOI: 10.1109/iThings/GreenCom/CPSCom/SmartData.2019.00177.

[39] *JAAS Reference Guide*. URL: https://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASRefGuide.html (visited on 03/08/2019).

[40] Christian Johansen et al. "Comparing Implementations of Secure Messaging Protocols (Long Version)". In: *978-82-7368-440-0* (2017). URL: https://www.duo.uio.no/handle/10852/60949 (visited on 03/05/2019).

[41] Roxanne Joncas. *MQTT and CoAP, IoT Protocols | The Eclipse Foundation*. URL: https://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php (visited on 01/24/2020).

[42] Roxanne Joncas. *MQTT and CoAP, IoT Protocols | The Eclipse Foundation*. URL: https://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php (visited on 01/25/2020).

[43] *JSR# 343: Java Message Service 2.0*. URL: https://jcp.org/en/jsr/detail?id=343 (visited on 07/28/2019).

[44] *Keep Alive and Client Take-Over - MQTT Essentials Part 10*. URL: https://www.hivemq.com/blog/mqtt-essentials-part-10-alive-client-take-over/ (visited on 01/24/2020).

[45] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. "Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach". In: 2nd IEEE European Symposium on Security and Privacy. Apr. 26, 2017, pp. 435–450. DOI: 10.1109/EuroSP.2017.38. URL: https://hal.inria.fr/hal-01575923/document (visited on 01/25/2019).

[46] Na Li, Yanhui Du, and Guangxuan Chen. "Survey of Cloud Messaging Push Notification Service". In: *2013 International Conference on Information Science and Cloud Computing Companion*. 2013 International Conference on Information Science and Cloud Computing Companion (ISCC-C). Guangzhou, China: IEEE, Dec. 2013, pp. 273–279. ISBN: 978-1-4799-5245-8. DOI: 10.1109/ISCC-C.2013.132. URL: http://ieeexplore.ieee.org/document/6973604/ (visited on 01/25/2019).

[47] Tongxin Li et al. "Mayhem in the Push Clouds: Understanding and Mitigating Security Hazards in Mobile Push-Messaging Services". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*. The 2014 ACM SIGSAC Conference. Scottsdale, Arizona, USA: ACM Press, 2014, pp. 978–989. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660302. URL: http://dl.acm.org/citation.cfm?doid=2660267.2660302 (visited on 01/25/2019).

[48] Yang Lu and Li Da Xu. "Internet of Things (IoT) Cybersecurity Research: A Review of Current Research Topics". In: *IEEE Internet of Things Journal* 6.2 (Apr. 2019), pp. 2103–2115. ISSN: 2372-2541. DOI: 10.1109/JIOT.2018.2869847.

[49] Jonathan Mayer, Patrick Mutchler, and John C. Mitchell. "Evaluating the Privacy Properties of Telephone Metadata". In: *Proceedings of the National Academy of Sciences* 113.20 (May 17, 2016), pp. 5536–5541. ISSN: 0027-8424, 1091-6490. DOI: 10.1073/pnas.1508081113. URL: http://www.pnas.org/lookup/doi/10.1073/pnas.1508081113 (visited on 12/18/2019).

[50] Georg Merzdovnik et al. "Block Me If You Can: A Large-Scale Study of Tracker-Blocking Tools". In: *2017 IEEE European Symposium on Security and Privacy (EuroS P)*. 2017 IEEE European Symposium on Security and Privacy (EuroS P). Apr. 2017, pp. 319–333. DOI: 10.1109/EuroSP.2017.26.

[51] *microG Project*. URL: https://microg.org/ (visited on 12/22/2019).

[52] *MQTT Man Page*. Sept. 20, 2018. URL: https://mosquitto.org/man/mqtt-7.html (visited on 01/24/2020).

[53] Piotr Nawrocki, Mikolaj Jakubowski, and Tomasz Godzik. "NOTIFICATION METHODS IN WIRELESS SYSTEMS". In: *Computer Science* 17.4 (2016), p. 519. ISSN: 1508-2806. DOI: 10.7494/csci.2016.17.4.519. URL: https://journals.agh.edu.pl/csci/article/view/1898 (visited on 01/25/2019).

[54] *NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy*. URL: https://www.nginx.com/ (visited on 12/22/2019).

[55] *Open GApps: Version 2020-01-03 Changelog*. URL: https://liquidtelecom.dl.sourceforge.net/project/opengapps/arm/20200103/open_gapps-arm-9.0-stock-20200103.versionlog.txt (visited on 01/06/2020).

[56] OASIS Open. *MQTT Version 3.1.1 Specification*. 2014. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html (visited on 02/06/2019).

[57] OASIS Open. *MQTT Version 5.0 Specification*. 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.pdf (visited on 01/24/2020).

[58] *Optimize for Doze and App Standby*. URL: https://developer.android.com/training/monitoring-device-state/doze-standby (visited on 12/22/2019).

[59] *Overview of Google Play Services | Google APIs for Android*. URL: https://developers.google.com/android/guides/overview (visited on 12/22/2019).

[60] *OwnPush Website*. Apr. 13, 2018. URL: https://web.archive.org/web/20180413014244/https://ownpush.com/ (visited on 07/20/2019).

[61] *Restund - Open Source STUN/TURN Server*. URL: http://www.creytiv.com/restund.html (visited on 12/22/2019).

[62] Paul Rösler, Christian Mainka, and Jörg Schwenk. "More Is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema". In: *2018 IEEE European Symposium on Security and Privacy (EuroS P)*. 2018 IEEE European Symposium on Security and Privacy (EuroS P). Apr. 2018, pp. 415–429. DOI: 10.1109/EuroSP.2018.00036.

[63] J. SathishKumar and Dhiren R. Patel. "A Survey on Internet of Things: Security and Privacy Issues". In: *International Journal of Computer Applications* 90.11 (Mar. 26, 2014), pp. 20–26. ISSN: 09758887. DOI: 10.5120/15764-4454. URL: http://research.ijcaonline.org/volume90/number11/pxc3894454.pdf (visited on 12/18/2019).

[64] Michael Schliep, Ian Kariniemi, and Nicholas Hopper. "Is Bob Sending Mixed Signals?" In: *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society - WPES '17*. The 2017. Dallas, Texas, USA: ACM Press, 2017, pp. 31–40. ISBN: 978-1-4503-5175-1. DOI: 10.1145/3139550.3139568. URL: http://dl.acm.org/citation.cfm?doid=3139550.3139568 (visited on 12/19/2019).

[65] A. Sciberras. *Lightweight Directory Access Protocol (LDAP): Schema for User Applications*. RFC4519. RFC Editor, June 2006. DOI: 10.17487/rfc4519. URL: https://www.rfc-editor.org/info/rfc4519 (visited on 01/29/2020).

[66] *Security & Privacy · Wire*. URL: https://wire.com/en/security/ (visited on 12/22/2019).

[67] *Set up a Firebase Cloud Messaging Client App on Android*. URL: https://firebase.google.com/docs/cloud-messaging/android/client (visited on 01/25/2020).

[68] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. URL: https://tools.ietf.org/html/rfc7252 (visited on 01/25/2020).

[69] S. Sicari et al. "Security, Privacy and Trust in Internet of Things: The Road Ahead". In: *Computer Networks* 76 (Jan. 2015), pp. 146–164. ISSN: 13891286. DOI: 10.1016/j.comnet.2014.11.008. URL: https://linkinghub.elsevier.com/retrieve/pii/S1389128614003971 (visited on 12/18/2019).

[70] *Spring Boot*. URL: https://spring.io/projects/spring-boot (visited on 07/28/2019).

[71] *Spring Framework*. URL: https://spring.io/projects/spring-framework (visited on 07/28/2019).

[72] Konglong Tang et al. "Design and Implementation of Push Notification System Based on the MQTT Protocol". In: *Proceedings of the 2013 International Conference on Information Science and Computer Applications (ISCA 2013)*. 2013 International Conference on Information Science and Computer Applications (ISCA 2013). Changsha, Hu Nan, China.: Atlantis Press, 2013. ISBN: 978-90-78677-85-7. DOI: 10.2991/isca-13.2013.20. URL: http://www.atlantis-press.com/php/paper-details.php?id=9566 (visited on 01/25/2019).

[73] *The Legion of the Bouncy Castle Java Cryptography APIs*. URL: https://www.bouncycastle.org/java.html (visited on 07/28/2019).

[74] *The Most Secure Collaboration Platform · Wire*. URL: https://wire.com (visited on 12/22/2019).

[75] *The Open GApps Project*. URL: https://opengapps.org (visited on 01/06/2020).

[76] Nik Unger et al. "SoK: Secure Messaging". In: *2015 IEEE Symposium on Security and Privacy*. 2015 IEEE Symposium on Security and Privacy (SP). San Jose, CA: IEEE, May 2015, pp. 232–249. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.22. URL: https://ieeexplore.ieee.org/document/7163029/ (visited on 01/25/2019).

[77] Matthias Wachs, Quirin Scheitle, and Georg Carle. "Push Away Your Privacy: Precise User Tracking Based on TLS Client Certificate Authentication". In: *2017 Network Traffic Measurement and Analysis Conference (TMA)*. 2017 Network Traffic Measurement and Analysis Conference (TMA). June 2017, pp. 1–9. DOI: 10.23919/TMA.2017.8002897.

[78] I. Warren et al. "Push Notification Mechanisms for Pervasive Smartphone Applications". In: *IEEE Pervasive Computing* 13.2 (Apr. 2014), pp. 61–71. ISSN: 1536-1268. DOI: 10.1109/MPRV.2014.34.

[79] *What Is Instance ID?* URL: https://developers.google.com/instance-id (visited on 01/24/2020).

[80] *WhatsApp Nutzer weltweit bis 2018*. URL: https://de.statista.com/statistik/daten/studie/285230/umfrage/aktive-nutzer-von-whatsapp-weltweit/ (visited on 12/22/2019).

[81] *Wire Statement on Code Contributions*. Dec. 21, 2019. URL: https://github.com/wireapp/wire#contributing-to-the-code (visited on 12/22/2019).

[82] *Wire Swiss GmbH on Github*. 2019. URL: https://github.com/wireapp (visited on 12/22/2019).

[83] *Wireapp/Wire-Android on Github*. Wire Swiss GmbH, Dec. 20, 2019. URL: https://github.com/wireapp/wire-android (visited on 12/22/2019).

[84] *Wireapp/Wire-Server on Github*. Wire Swiss GmbH, Dec. 20, 2019. URL: https://github.com/wireapp/wire-server (visited on 12/22/2019).

[85] *Wireapp/Wire-Webapp on Github*. Wire Swiss GmbH, Dec. 20, 2019. URL: https://github.com/wireapp/wire-webapp (visited on 12/22/2019).

[86] Yuchen Yang et al. "A Survey on Security and Privacy Issues in Internet-of-Things". In: *IEEE Internet of Things Journal* 4.5 (Oct. 2017), pp. 1250–1258. ISSN: 2327-4662, 2372-2541. DOI: 10.1109/JIOT.2017.2694844.

[87] Y. S. Yilmaz, B. I. Aydin, and M. Demirbas. "Google Cloud Messaging (GCM): An Evaluation". In: *2014 IEEE Global Communications Conference*. 2014 IEEE Global Communications Conference. Dec. 2014, pp. 2807–2812. DOI: 10.1109/GLOCOM.2014.7037233.

[88] Peter Zimmermann. "A Survey and Taxonomy on Privacy Enhancing Technologies". UT Vienna, 2014. URL: http://katalog.ub.tuwien.ac.at/AC12146819 (visited on 01/25/2019).