

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Hernández** Jméno: **Oscar** Osobní číslo: **393390**
Fakulta/ústav: **Fakulta elektrotechnická**
Zadávající katedra/ústav: **Katedra počítačů**
Studijní program: **Otevřená informatika**
Studijní obor: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Multiplatformní horizontálně škálovatelný systém pro zasilání zpráv

Název diplomové práce anglicky:

Multi-platform horizontally scalable messaging system

Pokyny pro vypracování:

Prostudujte si doporučenou literaturu. Navrhněte a implementujte horizontálně škálovatelný systém umožňující zasilání zpráv na různé platformy pro velké množství zařízení. K implementaci použijte technologie kompatibilní s JVM. Nad systémem implementujte vzorovou aplikaci pro demonstraci a otestování funkcionality.

Seznam doporučené literatury:

- [1] Martin L. Abbott, Michael T. Fisher. The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise. Addison-Wesley Professional, May 23, 2015. ISBN 9780134031385
- [2] Cheng-Zhong Xu. Scalable and Secure Internet Services and Architecture. CRC Press, Jun 10, 2005. ISBN 9781420035209
- [3] Felipe Gutierrez. Spring Boot Messaging: Messaging APIs for Enterprise and Integration Solutions. Apress, May 3, 2017. ISBN 9781484212240
- [4] Craig Walls. Modular Java: Creating Flexible Applications With Osgi and Spring (Pragmatic Programmers). CreateSpace Independent Publishing Platform, 2014. ISBN 9781503001459
- [5] Bill Stonehem. Google Android Firebase: Learning the Basics. First Rank Publishing, Jun 29, 2016. ISBN 9781535004466

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Martin Mudra, Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **17.01.2018** Termín odevzdání diplomové práce: **25.05.2018**

Platnost zadání diplomové práce: **30.09.2019**

Ing. Martin Mudra
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

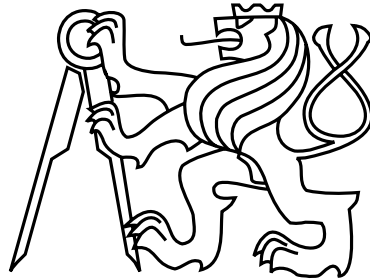
III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání

Podpis studenta

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Master's Thesis

Multi-platform Scalable Messaging System

Bc. Oscar Hernández

Supervisor: Ing. Martin Mudra

Study Program: Open Informatics

Field of Study: Software Engineering

May 25, 2018

Acknowledgements

I would like to thank my supervisor, Ing. Martin Mudra, for his advice and guidance. Whenever I was struggling he always showed me the right direction, helping me find the answer without giving it away.

I would also like to express my gratitude to my family and friends, who couldn't have been more supportive over the course of my studies.

Last, but definitely not least, I would like to thank my dear Zuzana, who helped me by proof-reading my work, supported me greatly, and took care of me as I worked day and night.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.

I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 25, 2018

.....

Abstract

This diploma thesis describes a horizontally scalable server-side application that allows sending messages across different platforms, with emphasis on modularity and easy expandability. The majority of the thesis and application is focused on the scalability aspect of distributing the system among a large number of computational nodes.

The application's purpose is to serve as a framework and foundation to be built upon and expanded for usage in modern applications.

Abstrakt

Tato diplomová práce popisuje horizontálně škálovatelnou serverovou aplikaci, která umožňuje zaslání zpráv napříč různými platformami s důrazem na modularitu a rozšiřitelnost. Drtivá většina práce a aplikace se zaměřuje na škálování za pomoci rozdělení systému na velké množství výpočetních zařízení.

Cílem aplikace je sloužit jako základ pro využití v moderních aplikacích.

Contents

| | | |
|----------|---|----------|
| 1 | Introduction | 1 |
| 1.1 | Motivation | 1 |
| 1.2 | Goals | 1 |
| 1.3 | Usage Scenarios | 2 |
| 2 | Analysis | 3 |
| 2.1 | Problem Analysis | 3 |
| 2.2 | Existing Similar Solutions | 5 |
| 2.2.1 | PubNub | 6 |
| 2.2.2 | Ably | 6 |
| 2.2.3 | Pusher | 7 |
| 2.2.4 | OneSignal | 8 |
| 2.2.5 | Similar solution comparison conclusion | 10 |
| 2.3 | Requirement Analysis (Core System) | 10 |
| 2.3.1 | Functional Requirements | 10 |
| 2.3.2 | Non-functional Requirements | 10 |
| 2.4 | Requirement Analysis (Sample Implementation) | 11 |
| 2.4.1 | Functional Requirements | 11 |
| 2.4.2 | Non-functional Requirements | 11 |
| 2.5 | Platform Analysis | 12 |
| 2.5.1 | Mobile platform | 12 |
| 2.5.1.1 | Android | 12 |
| 2.5.1.2 | iOS | 12 |
| 2.5.2 | Desktop and Server platforms | 14 |
| 2.5.2.1 | Java | 14 |
| 2.5.3 | Web | 14 |
| 2.6 | Analysis of Solutions for Implementation | 14 |
| 2.6.1 | Spring Boot | 14 |
| 2.6.2 | Testing Framework | 16 |
| 2.6.2.1 | JUnit | 16 |
| 2.6.2.2 | Spock Framework | 17 |
| 2.6.3 | Database technology | 17 |
| 2.7 | Analysis of Solutions for Sample Implementation | 18 |
| 2.7.1 | Firestore Cloud Messaging | 18 |
| 2.7.1.1 | FcmJava | 19 |

| | | |
|----------|--|-----------|
| 2.7.1.2 | Pushraven | 19 |
| 2.7.2 | Message Queue | 19 |
| 2.7.2.1 | RabbitMQ | 19 |
| 2.7.2.2 | Apache ActiveMQ | 20 |
| 2.8 | Scalability Analysis | 20 |
| 2.8.1 | Problematic Scenarios | 21 |
| 2.8.1.1 | Servicing a large amount of clients | 21 |
| 2.8.1.2 | A large amount of clients connects at the same time | 21 |
| 2.8.1.3 | A large amount of messages for a single client | 21 |
| 2.8.1.4 | A node dies | 21 |
| 2.8.1.5 | A message is sent to a user group with a large amount of users | 22 |
| 3 | Design | 23 |
| 3.1 | Scalability Design | 23 |
| 3.1.1 | Scalability Architecture Components | 24 |
| 3.1.1.1 | Nodes | 24 |
| 3.1.1.2 | Node Coordinator | 24 |
| 3.1.1.3 | Message Queue | 24 |
| 3.1.1.4 | Database | 26 |
| 3.1.2 | Problematic Scenarios and their Handling | 26 |
| 3.1.2.1 | Serving a large amount of clients | 26 |
| 3.1.2.2 | A large amount of clients connects at the same time | 26 |
| 3.1.2.3 | A large amount of messages for a single client | 27 |
| 3.1.2.4 | A node dies | 27 |
| 3.1.2.5 | A message is sent to a user group with a large amount of users | 27 |
| 3.2 | Architecture Design | 28 |
| 3.2.1 | Data Tier | 31 |
| 3.2.1.1 | Sample Implementation Data Tier | 31 |
| 3.2.2 | Business Tier | 31 |
| 3.2.3 | Collaboration Tier | 32 |
| 3.2.4 | Client Tier | 32 |
| 3.2.4.1 | Sample Implementation Client Tier | 33 |
| 3.3 | Modularity Design | 34 |
| 3.3.1 | Core Module | 35 |
| 3.3.2 | Database Modularity | 36 |
| 3.3.3 | Platform Modularity | 37 |
| 3.3.3.1 | Adapters | 38 |
| 3.3.4 | Message Queue Modularity | 40 |
| 4 | Implementation | 41 |
| 4.1 | Development platform | 41 |
| 4.2 | Code Structure | 42 |
| 4.2.1 | Msgr | 43 |
| 4.2.1.1 | Core Module | 43 |
| 4.2.1.2 | Message-Common | 44 |
| 4.2.1.3 | Fcm | 44 |

| | | |
|----------|---|-----------|
| 4.2.1.4 | Mysql | 44 |
| 4.2.1.5 | Websocket | 44 |
| 4.2.1.6 | Websocket-Common | 45 |
| 4.2.1.7 | ActiveMq | 45 |
| 4.2.1.8 | Node-Stats | 45 |
| 4.2.2 | Coordinator | 45 |
| 4.2.3 | Java-Client | 46 |
| 4.2.4 | Web | 46 |
| 4.2.5 | MsgrChattr | 46 |
| 5 | Testing | 49 |
| 5.1 | Automated Testing | 49 |
| 5.2 | Manual Testing | 49 |
| 5.2.1 | Testing Environment | 50 |
| 5.2.1.1 | Testing Environment Machine Specifications | 50 |
| 5.3 | Performance Testing | 51 |
| 5.3.1 | Tests | 51 |
| 5.3.2 | Test 1: Single Node, single client. 50 messages | 51 |
| 5.3.3 | Test 2: Single Node, single client. 200 messages | 51 |
| 5.3.4 | Test 3: 4 Nodes, single machine. 50 messages | 53 |
| 5.3.5 | Test 4: 5 Nodes, two machines. 50 messages | 53 |
| 5.3.6 | Test 5: Communication simulation. 50 messages, 20ms interval | 55 |
| 5.3.7 | Test 6: Communication simulation. 200 messages, 20ms interval | 56 |
| 5.3.8 | Test 7: Communication simulation. 200 messages, 100ms interval | 57 |
| 5.3.9 | Test 8: Communication simulation. 1000 messages, 100ms interval | 57 |
| 5.3.10 | Performance Test Conclusion | 59 |
| 6 | Conclusion | 61 |
| 6.1 | Goal Evaluation | 61 |
| 6.2 | Suggestions for Future Expansion | 61 |
| 6.2.1 | Authentication | 61 |
| 6.2.2 | System Administration | 61 |
| 6.2.3 | Encryption | 62 |
| | Bibliography | 63 |
| | A List of abbreviations | 67 |
| | B User Guide | 69 |
| B.1 | Building from Code | 69 |
| B.1.1 | System Requirements | 69 |
| B.1.2 | Software Prerequisites | 70 |
| B.1.3 | Building Shared Dependencies | 70 |
| B.1.3.1 | Coordinator/Common Module | 70 |
| B.1.3.2 | Msgr/Message-Common Module | 71 |
| B.1.4 | Building the Node and Node Coordinator Applications | 71 |

| | | |
|----------|---|-----------|
| B.1.5 | Building the Java-Client | 71 |
| B.1.6 | Building the MsgrChattr Android Application | 71 |
| B.1.7 | Building the PerformanceTester | 71 |
| B.2 | Running the Applications | 71 |
| B.2.1 | Running the Node Coordinator | 72 |
| B.2.2 | Running the Node Application | 72 |
| B.2.3 | Running the MsgrChattr Android Application | 74 |
| B.2.4 | Running the Chattr Web Application | 74 |
| B.2.5 | Running the PerformanceTester Application | 74 |
| C | Contents of CD | 75 |

List of Figures

| | | |
|------|---|----|
| 2.1 | Traditional client-server communication pattern | 4 |
| 2.2 | Realtime client-server communication pattern | 4 |
| 2.3 | Example situation of message delivery | 5 |
| 2.4 | Peak connections metric. Often used as a metric for pricing messaging services. Source[5] | 9 |
| 2.5 | Global mobile OS market share in sales to end users. Source[15] | 13 |
| 2.6 | Maven and Gradle build time comparison. Source[18] | 16 |
| 3.1 | Architecture design for horizontal scalability | 23 |
| 3.2 | Message flow for bound device | 25 |
| 3.3 | Message flow for unbound device | 25 |
| 3.4 | Message flow for group unfolding (flow after user processing omitted for readability) | 28 |
| 3.5 | Four Tier basic system architecture | 29 |
| 3.6 | Detailed Four Tier system architecture for sample implementation | 30 |
| 3.7 | Sample Implementation ORM Data Layer design | 31 |
| 3.8 | Flow of a message through the Business layer | 32 |
| 3.9 | Design of the main class components of the Java client library | 34 |
| 3.10 | Module structure design of sample implementation | 35 |
| 3.11 | Interfaces for database modules in the <i>core.db</i> package (Entity) | 36 |
| 3.12 | Interfaces for database modules in the <i>core.db</i> package (Repository) | 37 |
| 3.13 | Interfaces for platform modules in the <i>core.platform</i> package | 38 |
| 3.14 | Sequence diagram indicating the resolution process of Adapters | 39 |
| 3.15 | Interfaces for message queue modules in the <i>core.mq</i> package | 40 |
| 4.1 | Modules in the Msgtr project and their dependencies on each other | 42 |
| 4.2 | Node Coordinator monitor page | 46 |
| 4.3 | Chattr web application screenshot | 47 |
| 4.4 | MsgtrChattr Android application screenshot | 48 |
| 5.1 | Deployment diagram of the testing environment | 50 |
| 5.2 | Test 1: Delivery times | 52 |
| 5.3 | Test 2: Delivery times | 52 |
| 5.4 | Test 3: Delivery times | 53 |
| 5.5 | Test 4: Delivery times | 54 |
| 5.6 | Test 5: Delivery times | 55 |

| | | |
|-----|------------------------|----|
| 5.7 | Test 6: Delivery times | 56 |
| 5.8 | Test 7: Delivery times | 57 |
| 5.9 | Test 8: Delivery times | 58 |

List of Tables

| | | |
|-----|--|----|
| 2.1 | Similar solutions feature comparison | 9 |
| 5.1 | Testing environment machine specifications | 51 |
| 5.2 | Average message delivery time for Test 1 | 52 |
| 5.3 | Average message delivery time for Test 2 | 52 |
| 5.4 | Average message delivery time for Test 3 | 53 |
| 5.5 | Average message delivery time for Test 4 | 54 |
| 5.6 | Average message delivery time for Test 5 | 55 |
| 5.7 | Average message delivery time for Test 6 | 56 |
| 5.8 | Average message delivery time for Test 7 | 57 |
| 5.9 | Average message delivery time for Test 8 | 58 |

Chapter 1

Introduction

1.1 Motivation

Our world is becoming more interconnected than ever before, with over 10 billion devices connected to WLAN (Wireless Local Area Network) in 2017 and projections placing this number around 20 billion by 2021[39]. The further introduction of the Internet of Things (IoT) and many other smart devices such as home assistants, like Google Home¹ or Amazon Alexa², along with smart home devices ranging from thermostats and light bulbs to refrigerators and washing machines put these predictions at over 35 billion devices connected to the internet by 2021[38].

For the majority of people, being connected to the internet at every second of their day, directly or indirectly, has become a matter of fact. These developments have made the need for reliable and fast transmission of data more relevant than ever before.

The problem of real-time communication is increasingly important as bandwidth and connections speeds increase, allowing for more interactivity between users and computer systems. More and more systems require increasing amounts of communication in this age of an ever increasing number of web-based applications taking the place of traditional desktop ones. With services feeding customized content to a large number of different users in a more interactive and immediate manner, traditional methods prove too slow and cumbersome.

1.2 Goals

The goals of this thesis are to create a system that can serve as a framework and foundation to be built upon and expanded for usage in modern applications that need fast communication among potentially large numbers of devices, and a simple sample application in order to test the system's functionality.

¹Google Home <https://store.google.com/product/google_home/>

²Amazon Alexa <https://www.amazon.com/b/ref=gbpp_itr_m-2_2551_16067214?node=16067214011&ie=UTF8>

1.3 Usage Scenarios

The system designed and implemented as part of this thesis is meant to server as an extensible framework for modern applications, as mentioned above. Some of the many possible usages for the system can be instant messaging (IM), media (video or audio) streaming, communication within IoT systems, smart devices and more.

Chapter 2

Analysis

This chapter describes the problem that this thesis addresses, as well as its comparison to selected existing solutions. It also specifies the functional and non-functional requirements of the system to be implemented to address the specified problem, as well as a selection of platforms it will be implemented on in order to demonstrate its functionality.

2.1 Problem Analysis

Communication over networks is increasing greatly, not only user-to-user and user-to-machine interactions, but a massive growth has also occurred in machine-to-machine communication due to more automation, the usage of microservices architectures, system distribution and software and platform provided as a service (SaaS and PaaS, respectively).

These developments have led to changing requirements in the speed, volume, and reliability of data delivery. The need has risen for fast, lightweight, and secure real-time communication solutions.

The main difference between a traditional and real-time client-server¹ communication is that traditionally, the server would respond to a request from the client, be it synchronously or asynchronously. This communication happens over what is basically a one way channel. When the client would expect new data from the server, it would request it again, repeating the process.

During real-time communication, on the other hand, a two way channel is kept open between the client and server and data is sent and received both ways when needed, usually in an asynchronous fashion. See Figures [2.1](#) and [2.2](#).

¹Please note that client-server terminology is meant as someone who connects (client) to someone who provides data (server). The client can be another server-side application.

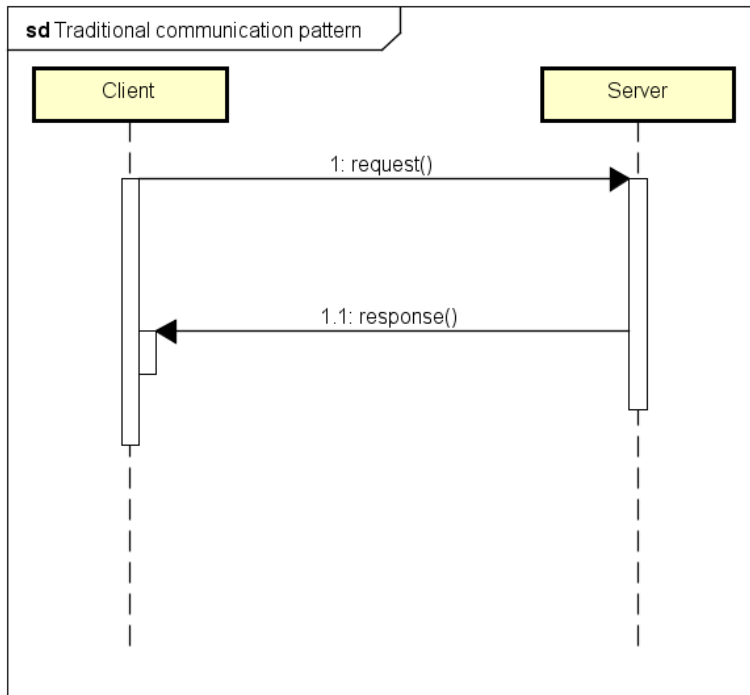


Figure 2.1: Traditional client-server communication pattern

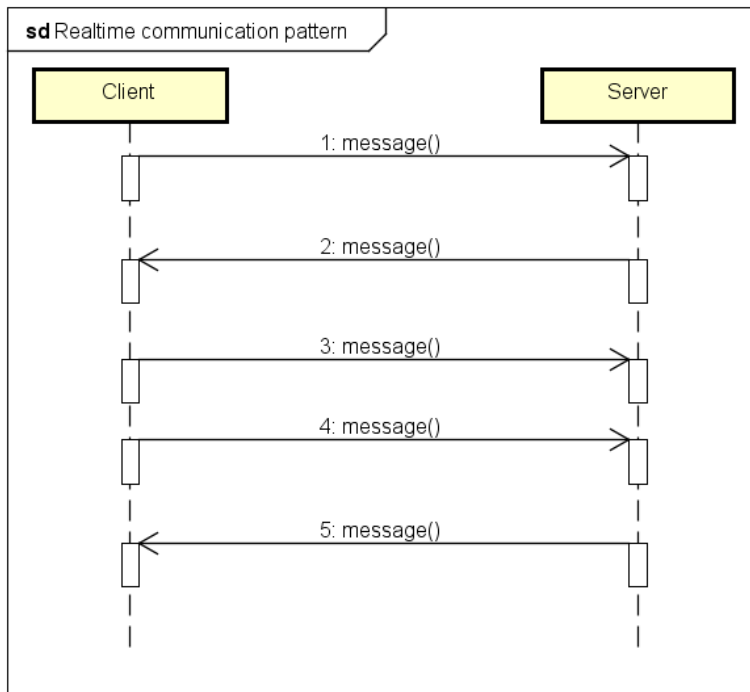


Figure 2.2: Realtime client-server communication pattern

While the basic pattern appears simple, the situation quickly begins to complicate once the facts that systems need to deliver realtime data to a large number of devices, often running on different platforms (eg. mobile or web platforms) and that devices may have periods of non-connectivity, such as a mobile device losing reception, are taken into account.

A very simple example of such a situation can be seen in Figure 2.3. A service needs to send two different messages to two groups of users, that are using different platforms and some of them even have multiple devices at once (eg. a web application in a browser on their computer and a smart phone) and the message needs to be delivered to all of these. For simplicity, only a one way message delivery is illustrated, but for true interactivity such a system must be able to also receive data, not just send it.

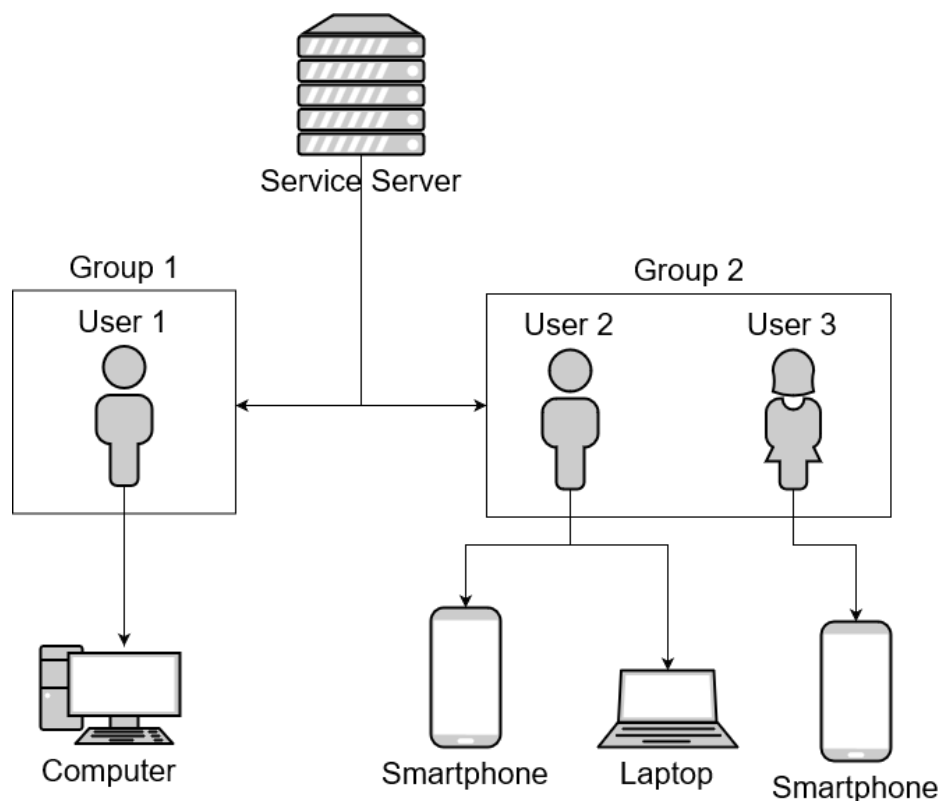


Figure 2.3: Example situation of message delivery

2.2 Existing Similar Solutions

At the moment there are several products on the market that provide messaging with differing capabilities and pricing strategies.

Henceforth will be described an overview of some of the most popular ones at the time of writing, based on information provided in marketing materials, documentations and FAQs (Frequently Asked Questions). An in-depth comparison of the inner workings of these is not included, since all of the solutions in question operate on a closed-source basis and do not disclose much information regarding the inner workings of their algorithms and data

structures. Furthermore, they are all provided as a service, without a possibility of self-deployment, so a comparison of their performance cannot be made on similar hardware.

2.2.1 PubNub

PubNub² is a commercial realtime messaging service based on a Publish-Subscribe pattern. One of the largest messaging service providers in the world, PubNub boasts, among others, secure end-to-end encryption, unlimited number of channels and 250ms latencies worldwide [28] and with over 70+ SDKs³ can be used on virtually any platform.

Key features:

- Unlimited Publish/Subscribe channels [28] (technically, though device and message amounts are limited with tier-based pricing)
- 250ms latency worldwide [28]
- Push notification support for Android, iOS and Windows
- Message delivery once target device comes online (catch-up) as long as message is still in queue. Messages are held in the queue by default for approximately 5 minutes or 100 messages (whichever is reached first), but can be extended using the Storage add-on. [25]

Pricing⁴ (monthly):

- Free: allows 100 daily active devices and 1 million total messages
- \$49: 500 daily active devices
- \$149: 1 500 daily active devices
- \$399: 5 000 daily active devices
- \$799: 20 000 daily active devices

2.2.2 Ably

Ably⁵ is a commercial real-time data delivery platform based on a Publish-Subscribe pattern, similarly to PubNub. Apart from several client libraries including ones for Javascript, Java, Python, PHP and others[4], Ably provides WebSocket and REST based APIs [3]. Just like PubNub, Ably also provides secure end-to-end encryption.

²PubNub <<https://www.pubnub.com/>>

³PubNub SDK full list <<https://www.pubnub.com/docs>>

⁴As of January 3 2018

⁵Ably <<https://www.ably.io/>>

Key features:

- Presence awareness, ie. notification when a device becomes online or offline
- Messages are stored for redelivery for 2 minutes by default. It can however be expanded with the channel message history where messages are stored for up to 24-72 hours [6] with persisted history enabled.
- Binary encoded messages help reduce bandwidth and streamlines processing time for encoding and decoding messages [2]
- Message and worker queues
- Reliable message ordering [7] - devices are guaranteed to receive messages in the order they were sent
- WebHooks, which are essentially HTTP callbacks
- Simple WebSocket and REST APIs allow for easy client implementation for platforms other than officially supported
- Protocol adapters providing interoperability between other real-time and queuing protocols [1]

Pricing⁶ (monthly):

Unlike PubNub's tier-based monthly pricing system, Ably provides a more flexible monetization model:

- Free: 100 peak connections(see Figure 2.4) and channels, 3 million monthly messages
- Self-service: \$12.50 per thousand peak connections or channels, \$1.25 per million messages. Volume discounts possible
- Enterprise: tailored package with premium support and no hard limits

2.2.3 Pusher

Pusher⁷ is a commercial real-time messaging service, also based on a Publish-Subscribe pattern. Pusher provides WebSocket and HTTP APIs for message publishing. Like all previously mentioned services, Pusher also supports end-to-end encryption.

Pusher provides official SDKs for both sending and receiving messages for several languages and frameworks including Go, Java, Node.js, Javascript, Swift, PHP, Python and others[30].

There is also a range of community developed and maintained libraries including clients for languages and frameworks such as Grails, Flash, ActionScript, Arduino, Haskell and more[29].

⁶As of January 3 2018

⁷Pusher <<https://pusher.com/>>

Key features:

- WebSockets with fallbacks in case they are not available
- Client events. These include when a device becomes online or offline
- Android and iOS support
- Status API for retrieving information such as occupied channels, number of connected devices, etc[31]
- Webhooks

Pricing⁸ (monthly):

Pusher offers a tier-based model similar to PubNub

- Free: 100 peak connections, unlimited channels, 200 000 messages per day (for comparison with Ably, this equals between 5.6 and 6.2 million messages per month, based on the number of days in said month)
- \$49: 500 peak connections, 1 million messages per day (28-31 million messages per month)
- \$99: 2 000 peak connections, 4 million messages per day (112-124 million messages per month)
- \$299: 5 000 peak connections, 10 million messages per day (280-310 million messages per month)
- \$499: 10 000 peak connections, 20 million messages per day (560-620 million messages per month)
- Tailored: a custom pricing plan made to fit

2.2.4 OneSignal

OneSignal⁹ is a commercial closed-source high volume push notification delivery service. Compared to PubNub, Ably or Pusher, OneSignal specializes in push notifications for mobile apps, though it does also support web notifications. This fact restricts the amount of platforms OneSignal can be used on in a significant manner.

While all over-the-network communication with OneSignal and then Apple/Android servers is done over HTTPS, compared to the aforementioned services it does not support end-to-end encryption out of the box[23]. However, this can be implemented on a server-client basis, ie. encrypt message before sending to OneSignal and then decrypt in app on target device, although this method couldn't be used to send notifications through OneSignal's useful dashboard.

OneSignal provides SDKs for many cross-platform mobile development environments such as Unity, PhoneGap, React Native, Xamarin, and others.

⁸As of January 3 2018

⁹OneSignal <<https://onesignal.com/>>

Key features [22]:

- A/B Test Messages
- Scheduled notifications
- Android, iOS and WebPush notifications support
- Simple dashboard for managing notifications and users
- Default time to live for notifications when device is offline is 72 hours[24]
- Free

Pricing:

Unlike PubNub, Ably or Pusher, OneSignal offers unlimited devices and notifications for free. Its monetization strategy is based on providing premium support.

Main feature comparison

| Feature | PubNub | Ably | Pusher | OneSignal |
|---|------------|-------------|------------|-----------|
| Publish/Subscribe channels | YES | YES | YES | NO |
| Receive message during short disconnects | YES | YES | NO | YES |
| Time message is held if device is offline | 5 minutes | 24-72 hours | N/A | 72 hours |
| End-to-End encryption | YES | YES | YES | NO |
| Scheduled messages | NO | NO | NO | YES |
| Device online/offline events | YES | YES | YES | NO |
| Pricing | Tier based | Usage based | Tier based | Free |
| Can be self hosted | NO | NO | NO | NO |
| Open Source | NO | NO | NO | NO |

Table 2.1: Similar solutions feature comparison

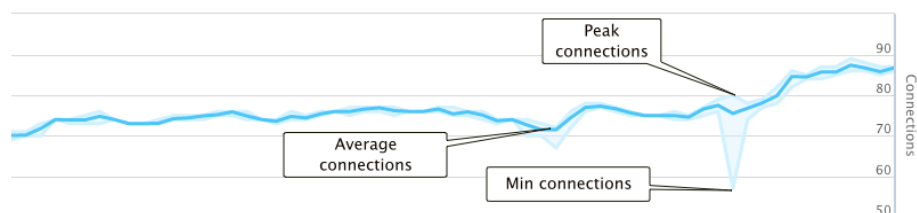


Figure 2.4: Peak connections metric. Often used as a metric for pricing messaging services. Source[5]

2.2.5 Similar solution comparison conclusion

All four aforementioned provide a service that allows sending a high volume of messages to many connected devices, with a similar list of additional features and different language or framework support and pricing. All have slightly different use cases and offer varying degrees of flexibility and support options.

PubNub, Ably and Pusher provide different restrictions based on the price of their service. Both Ably and Pusher use a peak connections metric, that is defined by the maximum amount of concurrent connected devices (see Figure 2.4). PubNub, on the other hand, has stopped using the peak connection metric[27] in favour of using Daily Active Devices. This metric refers to the total amount of connected devices in a 24-hour period.

However, all of the compared solutions are closed source and provide no means of self hosting. This leads to an intrinsic dependency on the companies that develop and maintain these platforms. This can be a problem for applications using these services if, for example, the pricing was suddenly changed, or the service was shut down entirely. A perfect example of this is when GoInstant was shut down and its customers were forced to switch to a different technology, one of these being PubNub, who offered a guide for migration[26].

2.3 Requirement Analysis (Core System)

After thorough analysis of what is expected of the system, the following requirements are put in place. These requirements are split into two categories, functional and non-functional requirements.

2.3.1 Functional Requirements

Functional requirements define the behaviour of the system.

- The system must be able to transmit messages between devices
- The system must be able to deliver a single message to a single device
- The system must be able to deliver a single message to several devices
- The system must be able to receive messages from devices

2.3.2 Non-functional Requirements

Non-functional requirements define the properties of the system.

- The server side of the system must be easily horizontally scalable, ie. scaling by adding new instances of the application
- The server side of the system must be modular, so that methods of receiving and sending messages can be easily replaced or added
- The server side of the system must be testable

- The server side of the system must be implemented so that it may run on the JVM¹⁰ platform
- The system must include a client library for representatives of the following platforms: web, mobile, desktop and server
- The web platform client must support most modern browsers, including the following: Microsoft Edge 16+, Mozilla Firefox 52+, Google Chrome 65+ and Safari 11+
- Third party software, such as libraries, used by the system must be open source

2.4 Requirement Analysis (Sample Implementation)

In order to prove the system meets its requirements and works properly, a sample implementation of the highly modular parts of the system must be provided. This sample implementation must meet the following functional and non-functional requirements:

2.4.1 Functional Requirements

Functional requirements define the behaviour of the system.

- The sample implementation must be able to deliver notifications and messages onto a mobile platform
- The sample implementation must be able to deliver notifications and messages onto the Web platform
- The sample implementation must be able to deliver notifications and messages onto the a desktop platform
- The sample implementation must be able to deliver notifications and messages onto the a server platform
- The sample implementation must contain a simple application to showcase the functionality of the system

2.4.2 Non-functional Requirements

Non-functional requirements define the properties of the system.

- The sample implementation must include a module implementing database functionality for a relational database, eg. MySQL, PostgreSQL, MariaDB.
- The sample implementation must include a module implementing the message queue functionality for a chosen platform, eg. ActiveMQ, RabbitMQ.

¹⁰Java Virtual Machine

2.5 Platform Analysis

While the system is designed in such a way that adding or removing supported platforms is simple, for the purposes of this thesis support is to be implemented for representatives of the mobile, web, server and desktop platforms, one each.

2.5.1 Mobile platform

Mobile devices, such as smart phones and tablets are taking over many of the functions that used to be exclusive to desktop computers. With this, it is a growing platform that cannot be overlooked by any modern application.

This section compares the two most popular mobile platforms, Android and iOS, and elaborates on the choice for the mobile platform representative implemented as part of this thesis.

2.5.1.1 Android

Android¹¹ is a mobile operating system developed by Google¹² widely used in smart phones, tablets, televisions, wearables such as smart watches, and even automobiles. With around 80% market share (see Figure 2.5) between mobile operating systems, it is indisputably one of the most important platforms on the market.

While development is lead mostly by Google, Android is an open source project. Its wide adoption by many manufacturers for different purposes is a direct result of this. The Android platform is based on a Linux kernel and apps are run using Android Runtime (ART)¹³ created specifically for Android and uses the Dalvik Executable (Dex) bytecode specification.

Development for Android can be done using the Android SDK, which allows the use of Java and Kotlin languages. While ART and the Android SDK closely mimic the Java Runtime Environment (JRE) and Java Development Kit (JDK) respectively, it has some slight differences.

2.5.1.2 iOS

iOS¹⁴ is a mobile operating system developed by Apple¹⁵ used in smart phones, tablets, wearables such as smart watches, and other devices. Unlike Google's Android, iOS is a completely proprietary and closed source platform and can only be found on devices directly developed and sold by Apple. It is the second most popular mobile operating system (see Figure 2.5).

¹¹Android <<https://www.android.com/>>

¹²Google <<https://www.google.com/>>

¹³ART <<https://source.android.com/devices/tech/dalvik/>>

¹⁴iOS <<https://www.apple.com/lae/ios/ios-11/>>

¹⁵Apple <<https://www.apple.com/>>

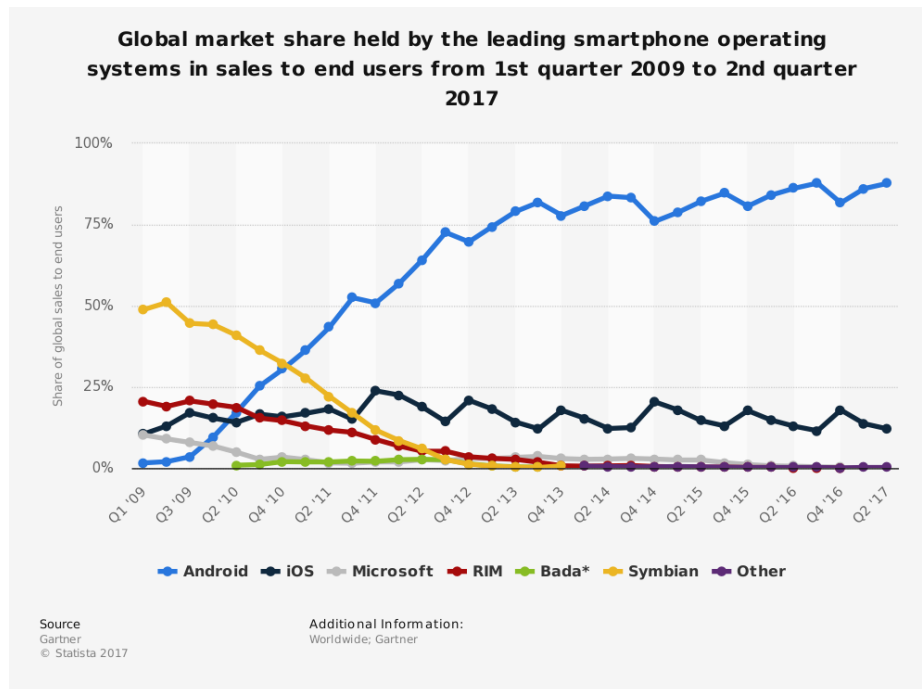


Figure 2.5: Global mobile OS market share in sales to end users. Source[15]

Development for iOS can be done using Xcode, Apple’s Integrated Development Environment (IDE). While previously iOS apps were built mostly using Objective-C, Apple has been pushing forward their Swift¹⁶ language as the future of iOS (and MacOS) app development.

Choice of mobile platform for implementation

The mobile platform chosen for the implementation is Android. Not only does Android possess the majority share of the market, its accessibility and openness to developers is a great advantage. In order to develop an application for Android, the developer only needs to download the Android Studio IDE and Android SDK, which are both available for Windows, Mac, and Linux. No physical device is needed for running the application during development, as the SDK contains a powerful emulator. For developing on a physical Android device, the developer can easily turn the device into development mode in the device’s settings.

Developing Android applications and distributing them through own means is completely free. However, most applications are published to Google’s official marketplace, Google Play, for which Google has a one time \$25 USD registration fee[16], after which the developer may publish any number of applications on the marketplace.

Developing applications for iOS, however, is much more complicated due to Apple’s closed ecosystem. iOS applications may only be developed on Mac devices[9] and an iOS device is needed to run the application during development. Apple also requires all developers to be

¹⁶Swift <<https://developer.apple.com/swift/>>

enrolled in their Apple Developer Program, which has an annual fee of \$99 USD, or \$299 USD for their Apple Developer Enterprise Program[10].

2.5.2 Desktop and Server platforms

2.5.2.1 Java

Java¹⁷ is a popular language owned by Oracle¹⁸ that is compiled into Java bytecode, which can be run on any Java Virtual Machine (JVM), regardless of the underlying platform that the JVM is running on.

Although nowadays it is one of the most popular platforms for developing enterprise server applications, owing to its large community support including extensive libraries, frameworks and platform independence, it can also be used to develop traditional desktop applications, both console based and with a Graphical User Interface (GUI).

2.5.3 Web

Frontend web applications based on HTML, CSS and JavaScript¹⁹ as more full fledged and interactive applications, compared to the static web sites of the past, have been gaining on popularity. This is partly due to modern browsers and faster internet bandwidths and speeds and partly due to a boom in powerful JavaScript-based frameworks and libraries facilitating the development of extensive, interactive applications that run inside a user's web browser.

A large number of applications have been moving some and in cases even most of their application logic from backend servers to clients in web browsers. In order for these applications to be responsive and interactive, it is key to have real-time reliable communication with any components that are on a remote server.

2.6 Analysis of Solutions for Implementation

In order for the implementation of the core system to meet all of its requirements, a careful and well-informed choice of the proper tools is paramount. This section describes the chosen tools as well as elaborates on the reasons as to why these tools were chosen.

2.6.1 Spring Boot

Spring Boot²⁰ is a JVM based framework for creating stand-alone production-grade applications based on the popular Spring Framework²¹. Unlike most other enterprise Java frameworks, Spring Boot does not need a container (such as Tomcat or Glassfish) to be present on the machine to run in, as it includes an embedded one. This allows for easy and simple JAR (Java ARchive) based deployment.

¹⁷Java <<https://www.java.com/>>

¹⁸Oracle <<https://www.oracle.com/>>

¹⁹HTML <https://www.w3.org/wiki/The_web_standards_model_-_HTML_CSS_and_JavaScript/>

²⁰Spring Boot <<https://projects.spring.io/spring-boot/>>

²¹Spring <<https://spring.io/>>

Spring Boot is an opinionated framework, building on the idea of *Convention over Configuration*. In essence, the idea behind this is to reduce the amount of configuration needed by having sensible defaults and using rules, or conventions, in naming and structure so that the framework may assume, based on these conventions, what it is supposed to do. A typical exam of this would be that a class called *WelcomeController* would map to the `"/welcome"` URL[35].

Spring Boot, being based on the Spring Framework, has powerful Inversion of Control (IoC) capabilities, also known as Dependency Injection (DI). IoC "... is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern." [36]

By delegating the creation and management of objects to the framework, IoC reduces the dependency of components on one another while still allowing them to interact and allows for more modularity, as it is the framework at runtime who decides which instances will be injected, typically by building what is called a Dependency Graph.

Spring Boot projects can be managed by either of the two most popular JVM build automation and dependency management tools, Maven and Gradle.

Maven

Maven²² is one of the most popular JVM build automation and dependency management tools. Maven's configuration is based on XML files. It manages the project's dependencies (third-party modules and libraries) and defines the build and execution order of different tasks. Maven also downloads the project's dependencies from online repositories, which are defined in the configuration, and caches them on the local machine.

Maven is distributed as open source under the Apache License, Version 2.0²³.

Gradle

Gradle²⁴ has, over the past few years, become a strong competitor to Maven and gained great popularity[11]. Like Maven, Gradle is a build automation and dependency management tool. However, it uses a Groovy-based DSL (Domain-Specific Language) for its configuration. This leads to shorter and more readable configuration files, while at the same time providing more flexibility and even scripting options. Like Maven, Gradle also downloads dependencies from online repositories and stores them on the local machine.

When it comes to performance, thanks to its advanced and modern techniques, Gradle builds are much faster compared to Maven (see Figure 2.6).

²²Apache Maven <<https://maven.apache.org/>>

²³Apache License 2.0 <<https://www.apache.org/licenses/>>

²⁴Gradle <<https://gradle.org/>>

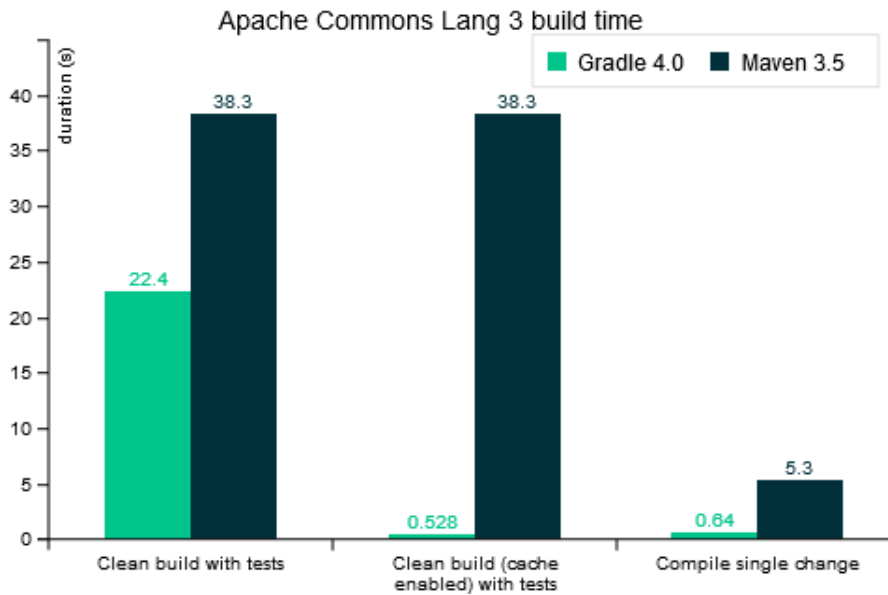


Figure 2.6: Maven and Gradle build time comparison. Source[18]

Thanks to Gradle’s learning curve, ease of use and advanced features, Gradle is the default build tool for Google’s Android OS.

Both Gradle and Spring Boot are distributed as Open Source software under the same license as Maven, the Apache License (ASL), Version 2.0 ²⁵.

Build Tool Conclusion

After careful evaluation of the available build and dependency management tools, including their features, drawbacks and other factors, such as active community support, extensibility, and thorough documentation among others, the selected tool for the implementation part of this thesis is Gradle.

2.6.2 Testing Framework

In order to create automated tests for the system’s code, a powerful testing framework is needed. Automated tests help to ensure that all components work as intended, even after small changes to the underlying code. Testing is indispensable for a large application to remain maintainable.

2.6.2.1 JUnit

JUnit²⁶ is one of the most popular JVM-based testing framework. Since the JUnit 5 version, it has been split into three main modules, The JUnit Platform, JUnit Jupiter and JUnit Vintage.

²⁵Apache License 2.0 <<https://www.apache.org/licenses/>>

²⁶JUnit <<http://junit.org/junit5/>>

The JUnit Platform "serves as a foundation for launching testing frameworks on the JVM. It also defines the TestEngine API for developing a testing framework that runs on the platform. Furthermore, the platform provides a Console Launcher to launch the platform from the command line and build plugins for Gradle and Maven as well as a JUnit 4 based Runner for running any TestEngine on the platform."^[21]

On top of providing the basis for running other testing frameworks, JUnit also provides its own solution for writing tests, which resides in the JUnit Jupiter module.

JUnit is Open Source software released under the Eclipse Public License (EPL) 1.0²⁷.

2.6.2.2 Spock Framework

Spock²⁸ is a powerful all-round testing framework for the JVM platform. Based on the Groovy²⁹ language and JUnit, it aims to put together the plethora of test libraries available into a comprehensive and easy to use framework.

Thanks to its use of Groovy DSL, Spock boasts an easy to understand and expressive specification language. On top of basic testing, some of its more advanced features include powerful Mocking APIs, class Stubbing, Data Driven Testing and Interaction Driven Testing, and its Spring Module provides seamless integration with the Spring TestContext Framework^[14].

Spock Framework is Open Source software distributed under the Apache License (APL) 2.0³⁰

2.6.3 Database technology

Since the system needs to persistently store, alter and access data, a database system is a proper solution. One of the most commonly used methods of accessing database storage in Java applications is the usage of an ORM (Object-Relational Mapping) framework, arguably the most popular one being Hibernate³¹.

Hibernate provides abstraction from the concrete database implementation, along with its own query language, HQL (Hibernate Query Language), which allows for more flexibility and interchangeability when it comes to the database software in use. Hibernate's ORM implementation is also an implementation of the Java Persistence API (JPA)^[20].

On top of powerful ORM for SQL databases, Hibernate also provides Hibernate OGM, a powerful JPA implementation for NoSQL database systems, including first party implementations for Infinispan, MongoDB and Neo4j and community-maintained dialects for Cassandra, CouchDB, EhCache, Apache Ignite and Redis^[19].

Hibernate is Free Software distributed under the GNU Lesser General Public License (LGPL) 2.1³² or Apache License (ASL), Version 2.0³³ licenses.

²⁷EPL 1.0 <<https://opensource.org/licenses/EPL-1.0>>

²⁸Spock Framework <<http://spockframework.org/>>

²⁹Apache Groovy <<http://groovy-lang.org/>>

³⁰APL 2.0 <<https://www.apache.org/licenses/LICENSE-2.0>>

³¹Hibernate <<http://hibernate.org/>>

³²LGPL 2.1 <<https://www.gnu.org/licenses/old-licenses/lgpl-2.1.html/>>

³³ASL 2.0 <<https://www.apache.org/licenses/LICENSE-2.0.html/>>

The aim of the system implemented as part of this thesis is to provide as much freedom, flexibility and modularity to the its users³⁴. For this reason, the system should be designed in such a way that it is easy to switch out the end database layer, including support for both SQL and NoSQL databases, and create custom implementations.

2.7 Analysis of Solutions for Sample Implementation

In order to prove the system meets its requirements and works properly, a sample implementation of the highly modular parts of the system must be provided. The requirements for the sample implementation are listed in Section [2.4 Requirement Analysis \(Sample Implementation\)](#).

2.7.1 Firebase Cloud Messaging

Firebase Cloud Messaging (FCM)³⁵ is a multi-platform messaging solution by Google³⁶. FCM is the successor to the Google Cloud Messaging (GCM)³⁷ platform, that was aimed mainly at the Android, iOS, and Google Chrome platforms. On top of the platform support of GCM, FCM adds support for the Web, NodeJS, C++, and Unity platforms.

FCM also provides powerful HTTP and XMPP (Extensible Messaging and Presence Protocol) APIs, which can be used to create clients for other platforms.

Key Features[13]

- Support for Android, iOS, Web, C++ and Unity platforms.
- Unlimited number of messages, with up to 4kB of data each.
- Support for Notification messages, which display a notification on the target device to the user.
- Support for Data messages, which are only handled in the background of the app on the target device.
- Normal and High priority message setting.
- Message Time to Live (TTL).
- HTTP and XMPP APIs.
- Free of charge.

³⁴Users here meaning those who would self-host the system, not the end users of any application using the system

³⁵FCM <<https://firebase.google.com/docs/cloud-messaging/>>

³⁶Google <<https://google.com>>

³⁷GCM <<https://developers.google.com/cloud-messaging/>>

Firebase Cloud Messaging is used in the sample implementation to relay messages onto the Android platform.

In order to simplify the use of FCM from the system a proper client library should be used. While there are no first party Java libraries, there are several community-maintained ones, two of which are compared below.

2.7.1.1 FcmJava

FcmJava³⁸ is a community-maintained Java library for communication with the FCM API. It provides object-oriented encapsulation of the FCM APIs. At the time of selecting the tools for the sample implementation (14. 1. 2018), FcmJava does not support the new FCM HTTP v1 API, but uses the Legacy HTTP Cloud Messaging, however support for the FCM HTTP v1 API is planned in the 3.0 milestone[12].

2.7.1.2 Pushraven

Pushraven³⁹ is a community-maintained Java library for communicating with the FCM API. It provides a nicely designed object-oriented encapsulation of the FCM APIs, that is simple to use and very easy to read. On 1. 12. 2017, the main author of Pushraven, Raudius, released a fully updated version of the library with support of the new FCM HTTP v1 API[40].

FCM library conclusion

Due to its good design, ease of use, fast update intervals and support of the more modern FCM HTTP v1 APIs, the library used in the sample implementation was decided to be Pushraven.

2.7.2 Message Queue

The design of the architecture of the system features a message queue for passing messages between the individual instances of the application (see Chapter 3 Design). This section presents several of the most popular message queue brokers, discusses their differences and presents a choice for the sample implementation.

2.7.2.1 RabbitMQ

RabbitMQ⁴⁰ is an Open Source message queue broker, licensed under the Mozilla Public License (MPL)⁴¹. RabbitMQ supports deployment in a distributed cluster, allowing for easy scaling, can be deployed on a wide variety of systems including Windows and Linux, and provides client libraries for languages such as Java, .NET, PHP, Python, Javascript, Ruby, Go and others[32], as well as integration with frameworks such as Spring.

³⁸FcmJava <<https://github.com/bytedfish/FcmJava>>

³⁹Pushraven <<https://github.com/Raudius/Pushraven>>

⁴⁰RabbitMQ <<https://www.rabbitmq.com/>>

⁴¹MPL <<https://www.mozilla.org/en-US/MPL/>>

While RabbitMQ is not a JMS provider, it includes a plugin that enables support for the JMS queue and topic messaging models[33].

Key features:

- Support for both message queues and publish/subscribe pattern topics
- Delivery acknowledgement
- Routing based on wildcards

2.7.2.2 Apache ActiveMQ

Apache ActiveMQ ⁴² is an Open Source message queue broker, licensed under the Apache 2.0 License (APLv2)⁴³. ActiveMQ provides a plethora of clients and protocols for languages such as Java, C++, C#, Ruby, Perl, Python, PHP, and others[8], as well as several different communication protocols, such as AMQP, MQTT, OpenWire and STOMP[8]. ActiveMQ also has support for frameworks such as Spring and unlike RabbitMQ is a JMS provider.

ActiveMQ also provides message queue data persistence and scaling via distribution and clustering.

Key features:

- Support for both message queues and publish/subscribe pattern topics
- Delivery acknowledgement
- Routing based on wildcards

Message Queue broker conclusion

RabbitMQ and Apache ActiveMQ provide extremely similar functionality and therefore are equivalent solutions for the implementation. However, based on the fact that ActiveMQ itself is based on the JVM and is a native JMS provider, as well as prior personal experience with the platform, the message queue broker used in the implementation is Apache ActiveMQ.

2.8 Scalability Analysis

In order to achieve a system that is horizontally scalable, the system must be able to easily run distributed among different machines, let us call every such instance of the back-end application a *node*. The system must be able to run across multiple nodes, distributing the computational load among these and it must be easy to add or remove nodes from the system while it maintains full functionality. This section describes the requirements on such a system and various problematic scenarios that may occur and the system must be able to handle.

⁴²Apache ActiveMQ <<http://activemq.apache.org/>>

⁴³APLv2 <<http://www.apache.org/licenses/LICENSE-2.0.html>>

2.8.1 Problematic Scenarios

2.8.1.1 Servicing a large amount of clients

This is the most basic use case for a scalable system. The system must be able to service a large amount of clients at the same time without any significant performance loss or becoming overloaded and stop servicing them at all.

This is a problematic scenario for applications as the infrastructure an application runs on has limited resources. An increasing amount of clients can be serviced by vertically scaling the application, i.e. adding more resources, such as CPU speed and/or cores and RAM memory, but this approach is feasible only to some extent, which is where horizontal scaling comes into play. By scaling the system horizontally, onto multiple machines, the load can be better distributed.

This scenario is expected to appear very often, as a large amount of clients are to be using the system simultaneously.

2.8.1.2 A large amount of clients connects at the same time

This scenario may occur when a large amount of clients attempts to connect to the system simultaneously. The difference between this scenario and the scenario described in Section [2.8.1.1 Servicing a large amount of clients](#) is that a large number of clients must not only be serviced at the same time, but they are also initializing their connection to the system simultaneously, placing great strain onto the entry point of the system.

This scenario is expected to occur very rarely on a very large scale and in minor scales during peak times, for example if an application utilizing the system would connect when the user starts using their phone a peak of connections can be expected in the morning, when people wake up and start their day.

2.8.1.3 A large amount of messages for a single client

This case describes a scenario where a large number of clients are all sending messages to a single addressee client. The possible problem is that a client is connected to a single node, therefore the messages that are to be delivered to it cannot be scaled across multiple nodes.

This scenario may happen in cases where the system is used to link a server to its clients, for example let us imagine a tracking app for food delivery. The individual clients would be the apps on the phones of each delivery driver, while the client receiving all their messages would be the server running the tracking application. All the drivers constantly stream their position data to the "server" through the system, resulting in a large number of messages from various clients, all addressed to a single recipient client.

2.8.1.4 A node dies

This scenario describes a situation where one of the nodes dies or is disconnected. The system must be able to respond to such a situation by reconnecting any clients that were connected to this node to a different node or set of nodes and manage to reroute any messages that are addressed to these clients.

This scenario is expected to happen rarely, with a cause either due to technical issues or for example upgrading the application on a node to a new version.

2.8.1.5 A message is sent to a user group with a large amount of users

This scenario describes a situation where the system contains a group with many users, for example an application's group of all its users, and a message is sent to this group. In the aforementioned example that could be used as a broadcast message to all the app's users. This scenario is problematic as the number of users in the group may be large enough that unfolding the group into all its users may put extreme stress on a node and its memory, possibly causing it to crash.

This scenario is expected to occur depending on the application using the system.

Chapter 3

Design

This chapter provides an in-depth description of the system architecture and its individual components, based heavily on the conclusions drawn from Chapter 2 [Analysis](#). It will cover the separation of responsibility of each component as well their interactions.

3.1 Scalability Design

This section describes the architecture proposed in order to achieve a horizontally scalable system, as well as how the proposed architecture deals with possible problematic scenarios described in Section 2.8.1 [Problematic Scenarios](#).

The proposed design of the system architecture to achieve horizontal scalability is shown below in Figure 3.1. The design can be split into four main components: **Nodes**, **Node Coordinator**, **Message Queue** and **Database**, the connections between these are shown in the Figure in different colours. The responsibilities, as well as the way in which the different components are connected and communicate with each other, are described in detail in the following sections.

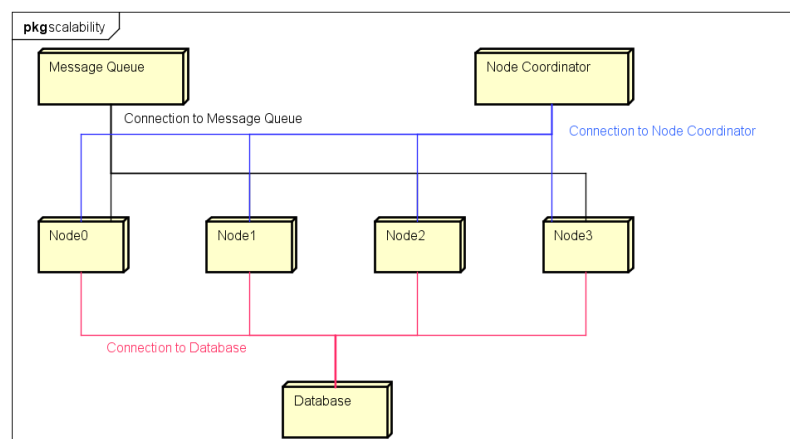


Figure 3.1: Architecture design for horizontal scalability

3.1.1 Scalability Architecture Components

3.1.1.1 Nodes

The Nodes form the core of the system. A node is an instance of the application containing all of the business logic. A Node's main responsibility is to dispatch messages to client devices as well as resolving groupings (such as *Group* or *User*, see Section 3.2.1 Data Tier). The Nodes are also the system's main scalability point, as the amount of load the system can process is proportional to the number of active Nodes in the system.

Nodes are aware of each other, as they receive a list of Nodes ordered by least load from the Node Coordinator (see Section 3.1.1.2 Node Coordinator).

3.1.1.2 Node Coordinator

The Node Coordinator is a simple component, whose main responsibility is to keep track of all Nodes in the system, as well as perform periodical health checks on them. Each Node should connect and announce its presence to the Node Coordinator upon start-up. The Node Coordinator also provides its connected Nodes with a list of n least loaded Nodes.

In case of a large number of Nodes connected to the Coordinator, if Nodes requested the list of least loaded Nodes upon every request, the Coordinator could quickly become overloaded. Therefore, Nodes should cache this list for a short amount of time and update it when necessary. This is achieved by the Node only requesting a new list of least loaded Nodes after the caching time expires.

3.1.1.3 Message Queue

The Message Queue component is an implementation of a message queue which is inherently scalable. For protocols that are bound to a concrete Node, for example a websocket connection, the message queue channels would correspond to individual devices connected to the Node, for example if a device with id *device1* would connect to the Node, the Node would then subscribe to a channel called *websocket.device1*. Any messages addressed to this device would then be pushed into this channel. This flow can be seen in Figure 3.2.

For devices which are not bound to a concrete Node, for example mobile devices connected through FCM, the messages for the device would be pushed into the channel *FCM*, from which any Node can dequeue and process it. This flow can be seen in Figure 3.3.

As unfolding a user into their respective devices is also an operation which does not require a specific Node to process, these are pushed to a user channel, from which the messages get distributed among all available Nodes. The equivalent can be done for groups.

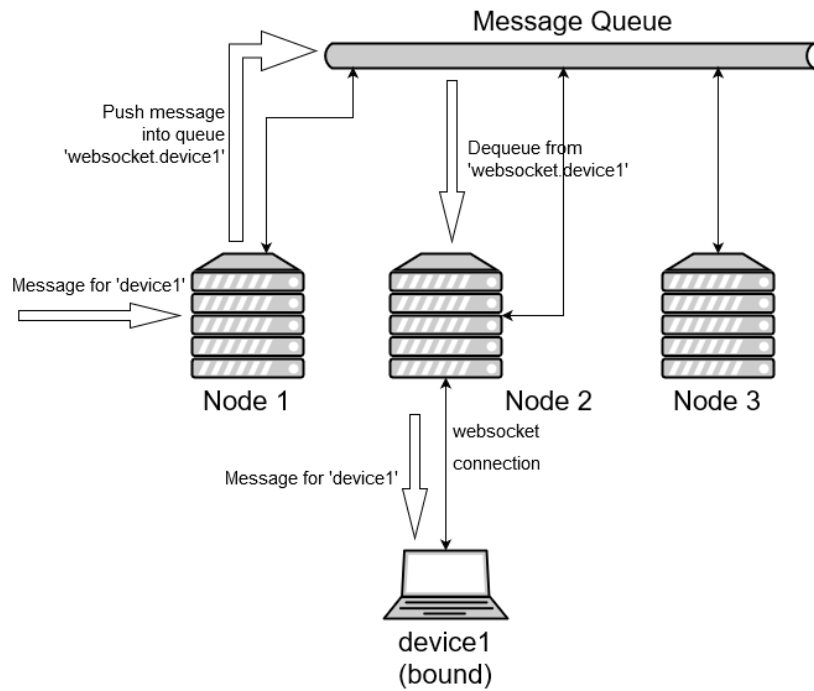


Figure 3.2: Message flow for bound device

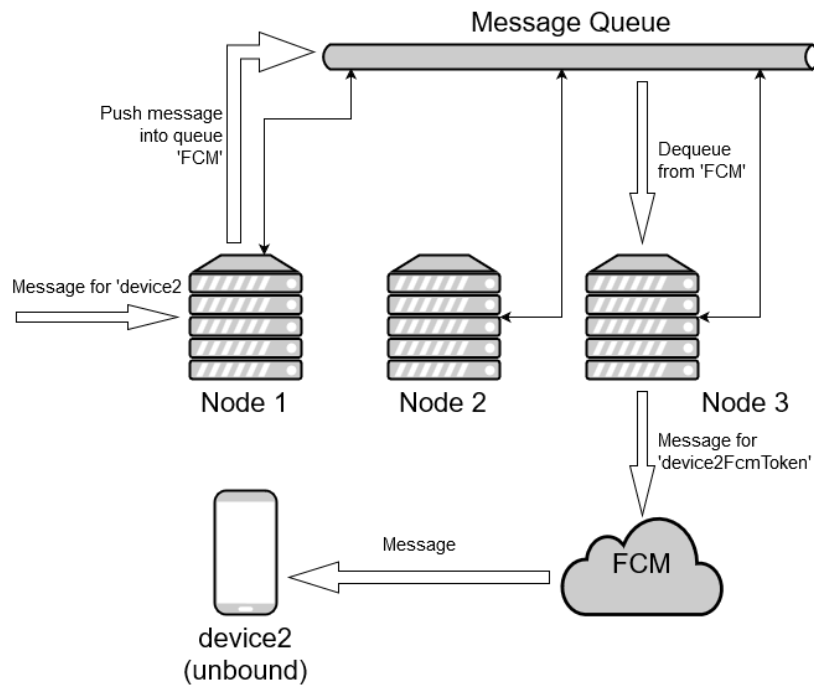


Figure 3.3: Message flow for unbound device

3.1.1.4 Database

The Database component is a database layer shared between Nodes, containing all persistent data. As the database layer is fully modular, the end implementation used is up to the discretion of whoever is building the system (see Section [3.3.2 Database Modularity](#)). However, in order to create a system that is fully scalable a scalable database solution (e.g. database cluster) is recommended.

3.1.2 Problematic Scenarios and their Handling

The system can encounter several problematic scenarios, as described in Section [2.8.1 Problematic Scenarios](#). This section describes how the proposed architecture design handles these scenarios.

3.1.2.1 Serving a large amount of clients

In this scenario, a large amount of clients is being served by the system. In order to prevent the system from becoming overwhelmed, the load must be distributed among the available Nodes in such a way that all maintain a healthy load.

The proposed architecture deals with this scenario by spreading the load using the coordination of the Nodes and the Node Coordinator. When a client tries to connect to a Node, in case the Node is above a certain load threshold, it sends the list of the least loaded Nodes to the client. The client then attempts to connect to a random Node from this list.

This resolves the issue by offloading incoming traffic onto less loaded Nodes.

3.1.2.2 A large amount of clients connects at the same time

This scenario is very similar to the one described in Section [3.1.2.1 Serving a large amount of clients](#). However, the significant difference is that in this scenario, a large number of clients are attempting to connect to the system at the same time.

The proposed architecture design deals with this scenario mostly in the same way as with the scenario [3.1.2.1](#). On top of that, a load balancer or DNS-level balancing should be put in place before the entry point to the system. By placing a load balancer into the system entry point or using multiple IP addresses for the DNS (Domain Name System) entry of the system, the initial connection requests can be distributed among different Nodes.

The system can handle this scenario this way also thanks to the randomization of the client connection after receiving the list of least loaded Nodes. If the clients attempted to connect to the least loaded Node, then in the case that a large number of clients connects at the same time, all would be redirected to the same Node, therefore placing it under large load. By randomly choosing between the n least loaded Nodes, the connections are distributed among the Nodes.

3.1.2.3 A large amount of messages for a single client

This scenario describes a situation where a large number of messages are addressed to a single client.

The system handles this scenario by using a message queue which is inherently scalable. The messages for the client are expected to be sent onto different Nodes (from all the other clients), which then place them in the Message Queue. The Node to which the target client is connected to then takes these messages and passes them onto the client.

A possible bottleneck in this scenario may be the connection speed between the Node and the client, but that is unsolvable by scaling the system. However, in order to prevent the Node from getting blocked by this client and stop serving other clients connected to it, it must be able to handle each client in parallel.

3.1.2.4 A node dies

This scenario describes a situation in which a Node becomes disconnected from the system, either by crashing or connectivity issues. The system discovers that a Node has died by the Node failing a health check from the Coordinator.

The proposed design handles this situation on two levels:

Client side

When the client loses connection to the Node it is connected to, it attempts to reconnect to it. If the reconnecting attempt also fails, it assumes the Node is no longer reachable and begins the connection process anew, connecting to a different Node.

Server side

When a Node fails a health check, it is assumed dead and the Coordinator removes it from its connected Nodes. Any messages meant for clients connected to the now dead Node stay in the Message Queue and if the clients reconnect to a new Node within a certain message timeout period, the messages are then passed onto the clients.

3.1.2.5 A message is sent to a user group with a large amount of users

This scenario describes a situation where the system contains a group with many users. So many users in fact, that their unfolding from the group could cause an overload on the node, possibly causing it to crash.

The system handles this scenario by splitting the group into user pages of a size that can be easily handled, pushing references to these into the message queue, from where they are split across multiple nodes that process them. This flow can be seen in Figure 3.4.

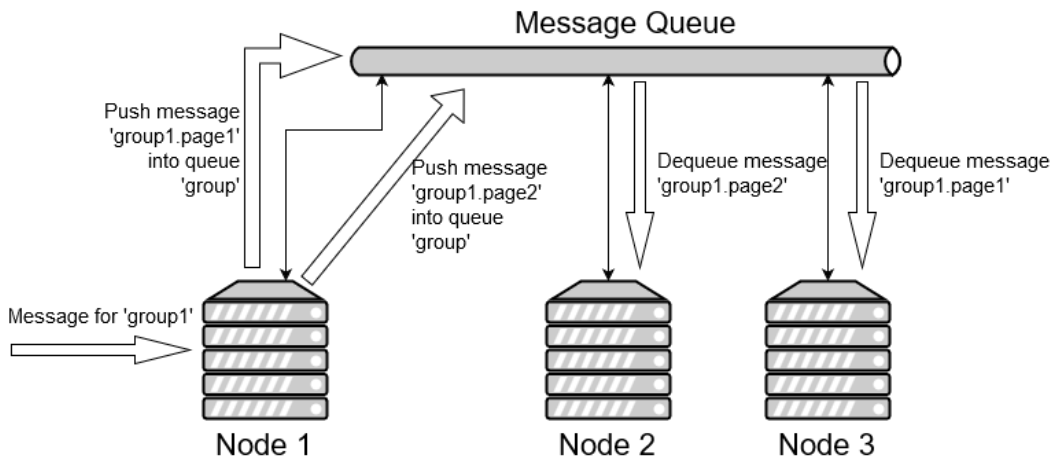


Figure 3.4: Message flow for group unfolding (flow after user processing omitted for readability)

3.2 Architecture Design

The system can be divided into multiple main parts¹, the highest-level of division can be into a basic four tier (or four-layered) architecture containing the following four tiers:

- Client Tier (Presentation Tier)
- Collaboration Tier
- Business Tier (Business Process Tier)
- Data Tier

Figure 3.5 shows these four layers and the components belonging in each. Due to the high modularity requirements of the system, only most of the Business and part of the Collaboration Tiers belong into the core system, the Presentation and Data Tiers are to be easily interchangeable and therefore are part of the sample implementation.

A more concrete version of the tiered architecture for the full sample implementation can be seen in Figure 3.6.

¹Note: These do not necessarily overlap with the modular aspects of the system.

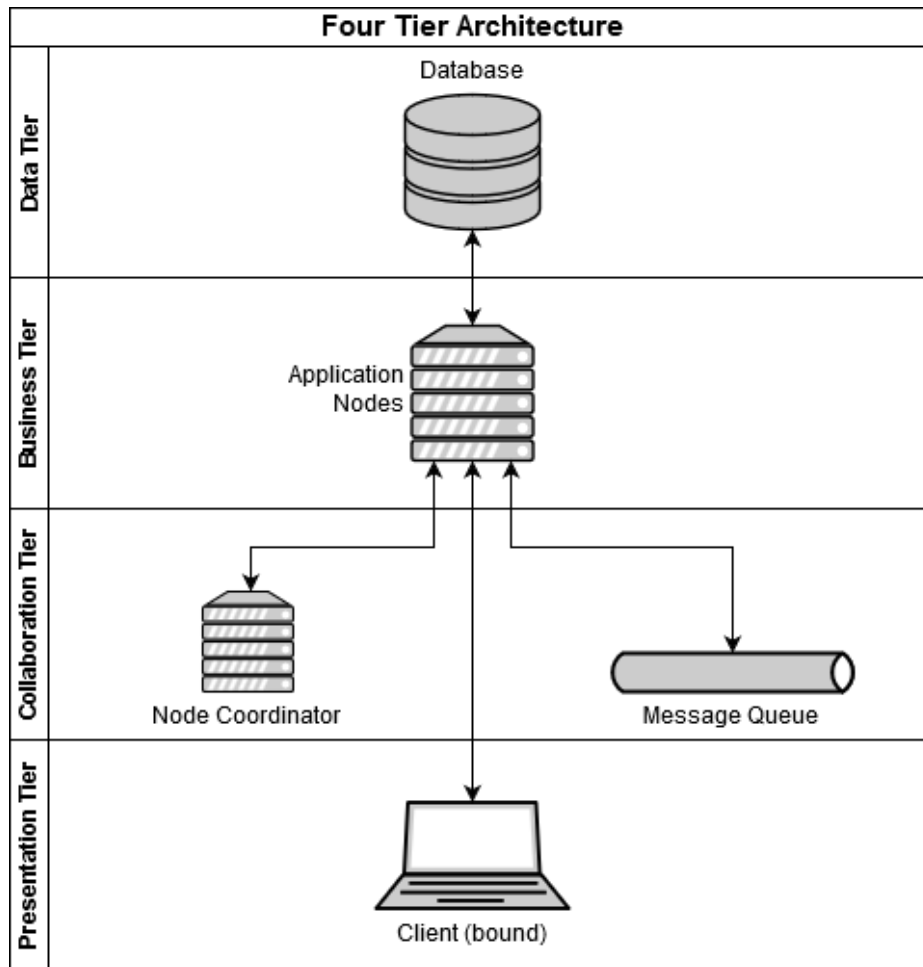


Figure 3.5: Four Tier basic system architecture

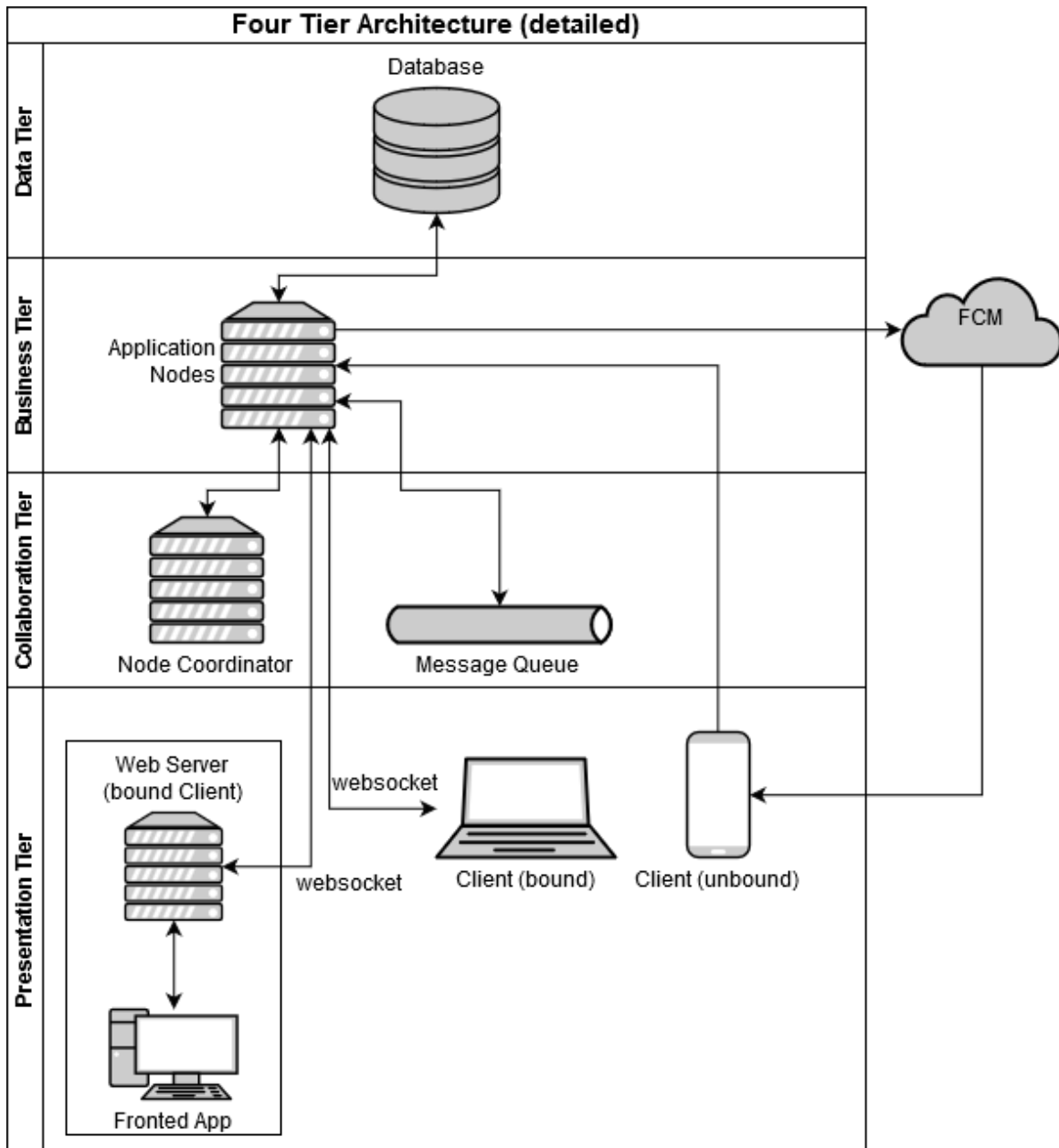


Figure 3.6: Detailed Four Tier system architecture for sample implementation

3.2.1 Data Tier

The *Data Tier* is used to persistently store data describing how messages should be handled and how they are to be delivered to their intended recipient. After analysing the necessary requirements, the data to be stored has been divided into the following entities.

- **Group:** The Group entity represents a collection of Users, aggregated for some reason, eg. all users of an application, members of a chatroom, etc.
- **User:** The User entity represents an end user of the application using the system. A User can belong to Groups and has Devices.
- **Device:** The Device entity represents an end device onto which messages will be delivered. Every Device belongs to a User and is on a certain Platform.
- **Platform:** The Platform entity represents the platform on which a Device is running and therefore determines how a message is to be delivered to that Device, eg. websocket, FCM, etc.

Due to the high modularity design required of the system, the core of the system only contains interfaces and instructions for the implementation of the data layer.

3.2.1.1 Sample Implementation Data Tier

In the sample implementation, which is part of this thesis, a relational database system will be used. The Object-Relational Mapping (ORM) implementation design of the data layer interface can be seen in Figure 3.7

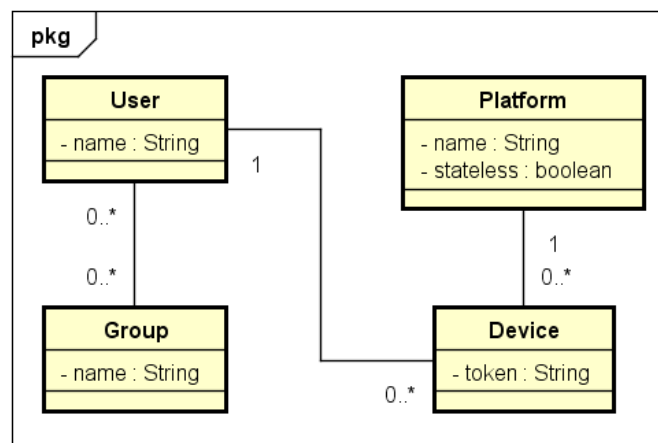


Figure 3.7: Sample Implementation ORM Data Layer design

3.2.2 Business Tier

The *Business Tier*, also commonly known as the *Business Process Tier* or *Business Logic Layer*, is where all the main functions of the system are done. Here the messages are received,

processed and passed along to either the message queue or the respective adapters for the target device's platform².

Figure 3.8 shows a simplified representation of the flow of a message through the Business Tier. The message starts in the *Client Tier*, where one of the clients sends it. After being received through the system's API it is processed and passed to the *MessagingService*, which communicates with the *Data* layer to find information on the target group, user or device and the platforms associated with them. After this, the message is passed into the message queue in the *Collaboration Tier*, from which individual *Nodes*, subscribed to their relevant channels, receive the messages into the proper platform adapters. From here, the message is sent out to the end devices, in the *Client Tier*.

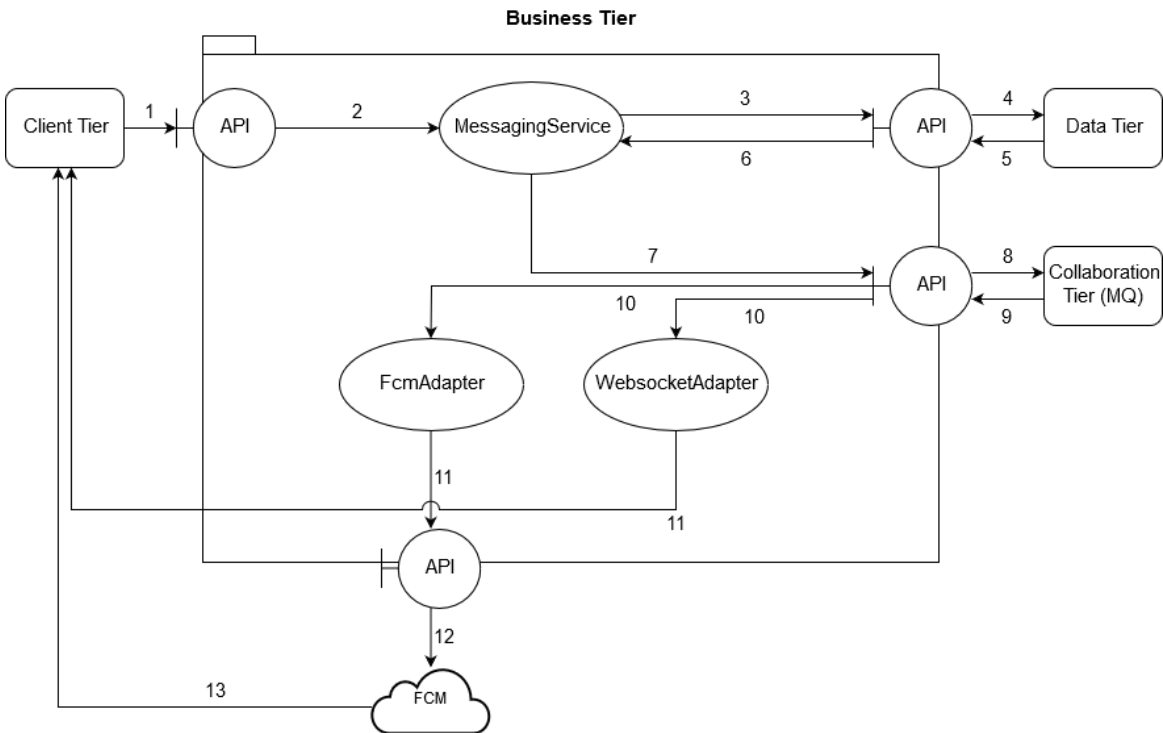


Figure 3.8: Flow of a message through the Business layer

3.2.3 Collaboration Tier

The *Collaboration Tier* is responsible for managing information sharing among multiple instances of the *Business Tier*. The *Collaboration Tier* includes the *Node Coordinator*, which serves to keep track of active *Nodes* in the system, and the *Message Queue*, which provides a way to communicate between *Nodes* and distributing messages among them.

3.2.4 Client Tier

The *Client Tier*, also commonly referred to as the *Presentation Tier* includes all client libraries and applications that can receive and/or send messages from/to the system. Because

²More on adapters in Section 3.3.3.1

the *Client Tier* must be designed with high modularity in mind, none of it is part of the main system and the specifics of its API depend on the individual adapters³ that go with it.

3.2.4.1 Sample Implementation Client Tier

The sample implementation that is part of this thesis includes an implementation of the *Client Tier*, aka the client libraries, for the platforms stated in Section 2.4.1 [Functional Requirements](#).

Dektop, Server, and Mobile

The client library for the Desktop and Server Java platforms is a Java Archive (JAR) file, that can be included in a Java application's classpath and provides interfaces to send and receive messages to and from the system, respectively. It will be independent of any framework, such as Spring, so that it works with any Java application. Since applications for the Android mobile platform can be written in Java, the Java client library is designed in such a way that it can be used on the Android platform as well.

With real-time communication in mind, the design of the client library was made using an Event Driven architectural pattern for receiving messages through a websocket. However, keeping in mind that end applications might use a different way of consuming the messages, for example in Android's case through Firebase Cloud Messaging, connection using websockets is optional.

Figure 3.9 shows an overview of the classes the library contains along with their external interface. For simplicity, private members of the classes have been omitted.

The most important class being *MsgrClient*, which is the main point of interaction for the application using the library. It contains methods for setting up the client, sending messages, and setting up a message listener, which is an instance of the *Consumer* class, from Java 8's functional API. Because in Android's ART the Groovy JSON Builder cannot be used, a *Function* instance can be also set, to use custom behaviour for constructing JSON strings.

The *WsSocket* class is an internal class of the library, used by the main *MsgrClient* in order to process incoming and outgoing messages through a websocket connection.

Data container classes are not important to the overall architecture design, they serve simply as POJOs (Plain Old Java Objects) that envelop data, along with providing getters and setters, in order to pass the data between components in a simple, encapsulated object-oriented manner. These include classes such as *DataMessageDto*, *NotificationDto*, etc.

³More on adapters in Section [3.3.3.1](#)

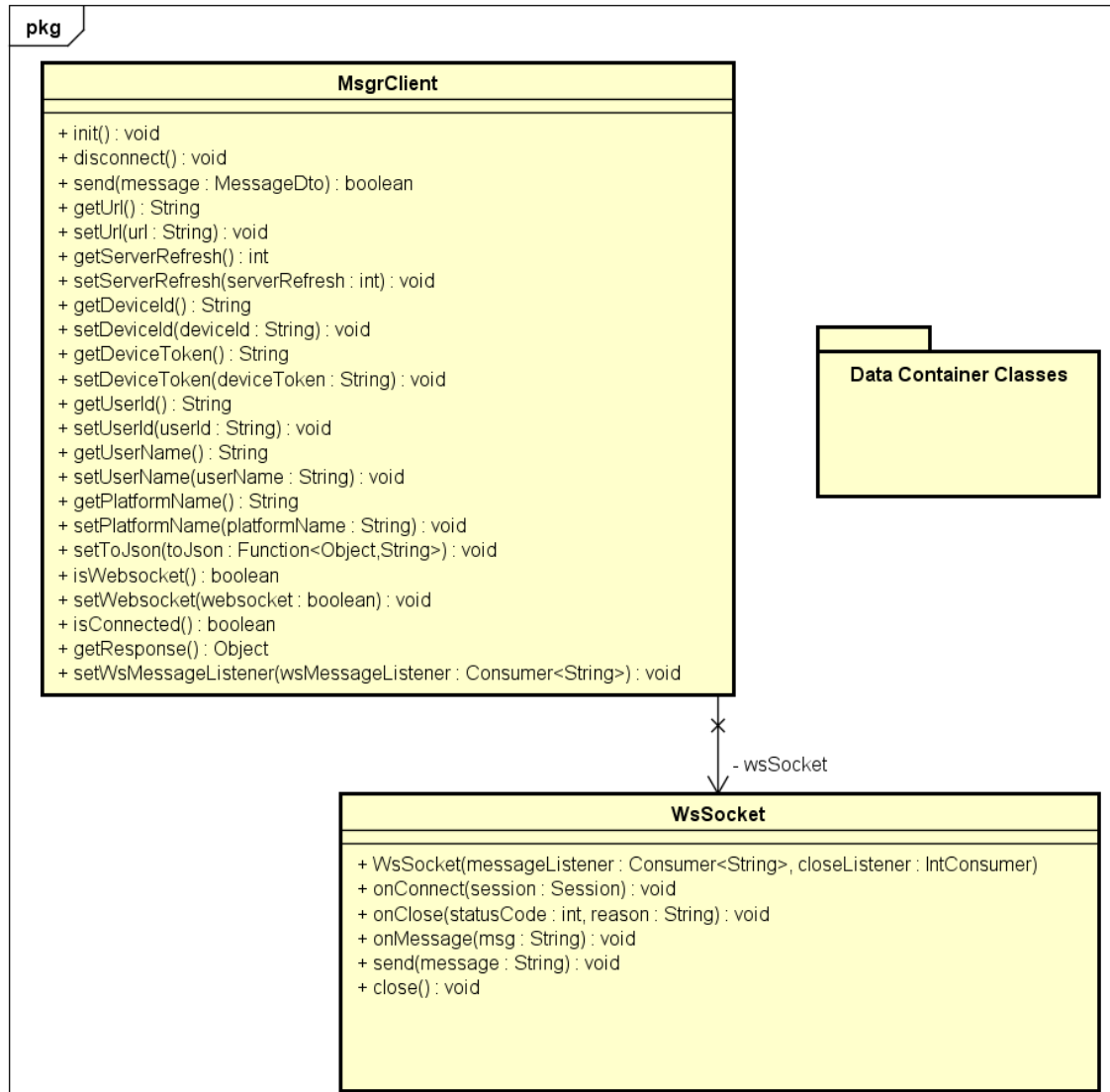


Figure 3.9: Design of the main class components of the Java client library

Web

The Web client is a Javascript library application, its overall design very similar to the design of the Java client library described in Section 3.2.4.1 *Dektop, Server, and Mobile* and uses websockets to communicate with the system. The Javascript library is compatible with most modern web browsers (versions: Chrome 50+, Firefox 44+, Opera Mobile 37+)[17].

3.3 Modularity Design

A critical part of the requirements put onto the system is its high flexibility and adaptability, based on high modularity. For this reason, the system has been designed in order to reach

maximum modularity. In order to achieve this, the main system is contained in a Core module, which will be the basis for any full applications with the system. Functionality and components that may be interchangeable depending on the applications' needs are contained in individual modules, which can be added to the full application as further dependencies (together with the Core module). An example of an application with different modules can be seen in Figure 3.10, which shows the basic module structure used in the Sample Implementation.

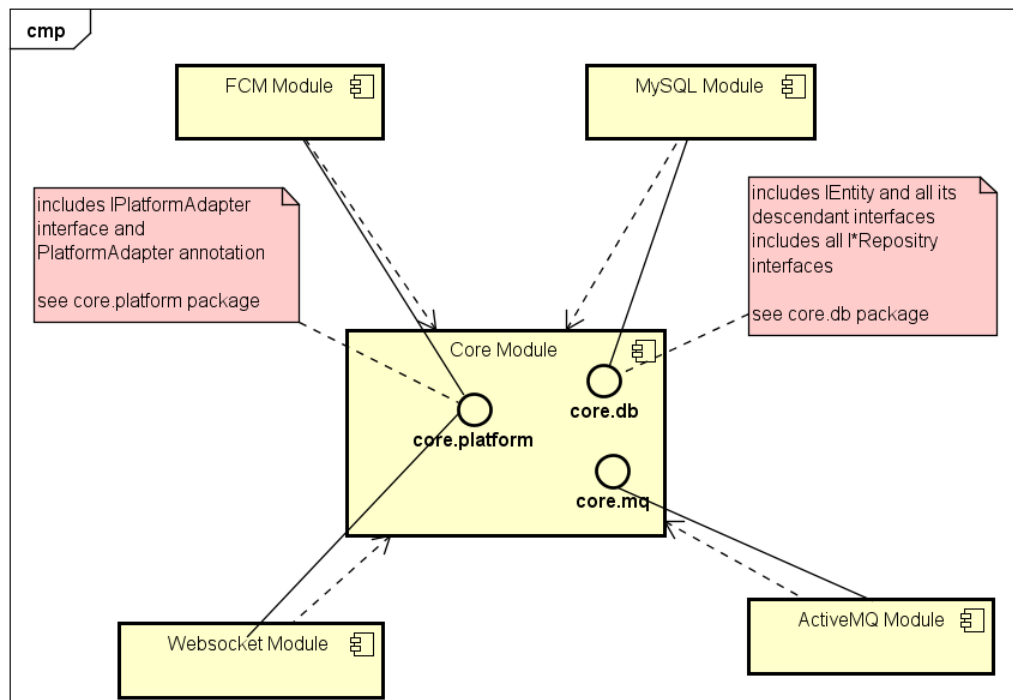


Figure 3.10: Module structure design of sample implementation

3.3.1 Core Module

The Core module, or Core library, is the heart of the entire system. The Core module provides the interfaces needed for other modules to implement the Spring Managed Beans (from now on referred to simply as Beans) that will be used across the application.

Beans are objects whose lifecycle is managed by the Spring's IoC Container's Application Context, which means the Spring Container initializes and configures them and where needed, allows them to be injected[34]. These Beans provide the main business logic of the application as well as manage the cooperation between all modules. The design of these interactions is further elaborated upon in the following sections.

The Core module also provides the business logic for basic message processing, Node health status, and core Node APIs.

3.3.2 Database Modularity

The system's access to the *Data Tier* has been designed in a fully modular way, so that the end deployment is not dependent on any one type or provider of database. The result of this effort for maximum freedom and interchangeability are the interfaces in the *core.db* package, as seen in Figures 3.11 and 3.12.

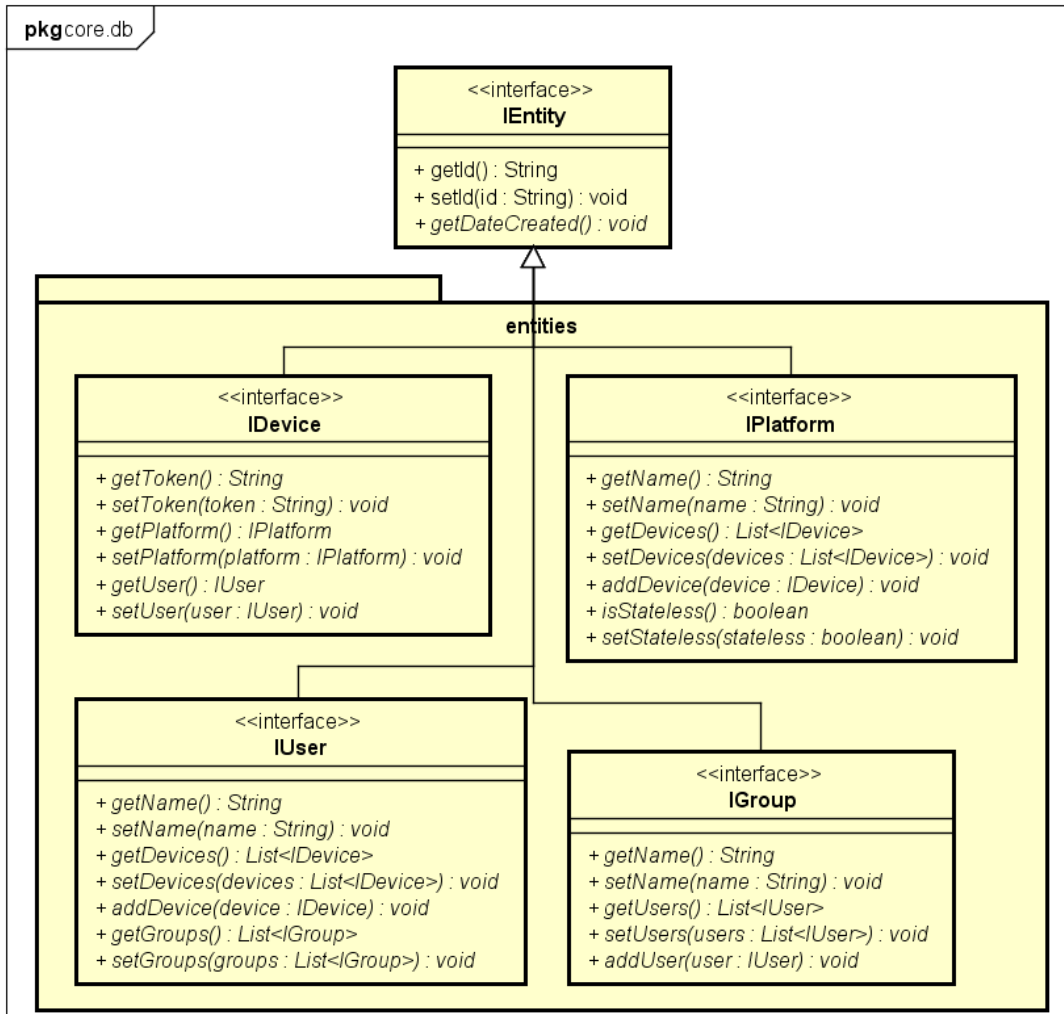
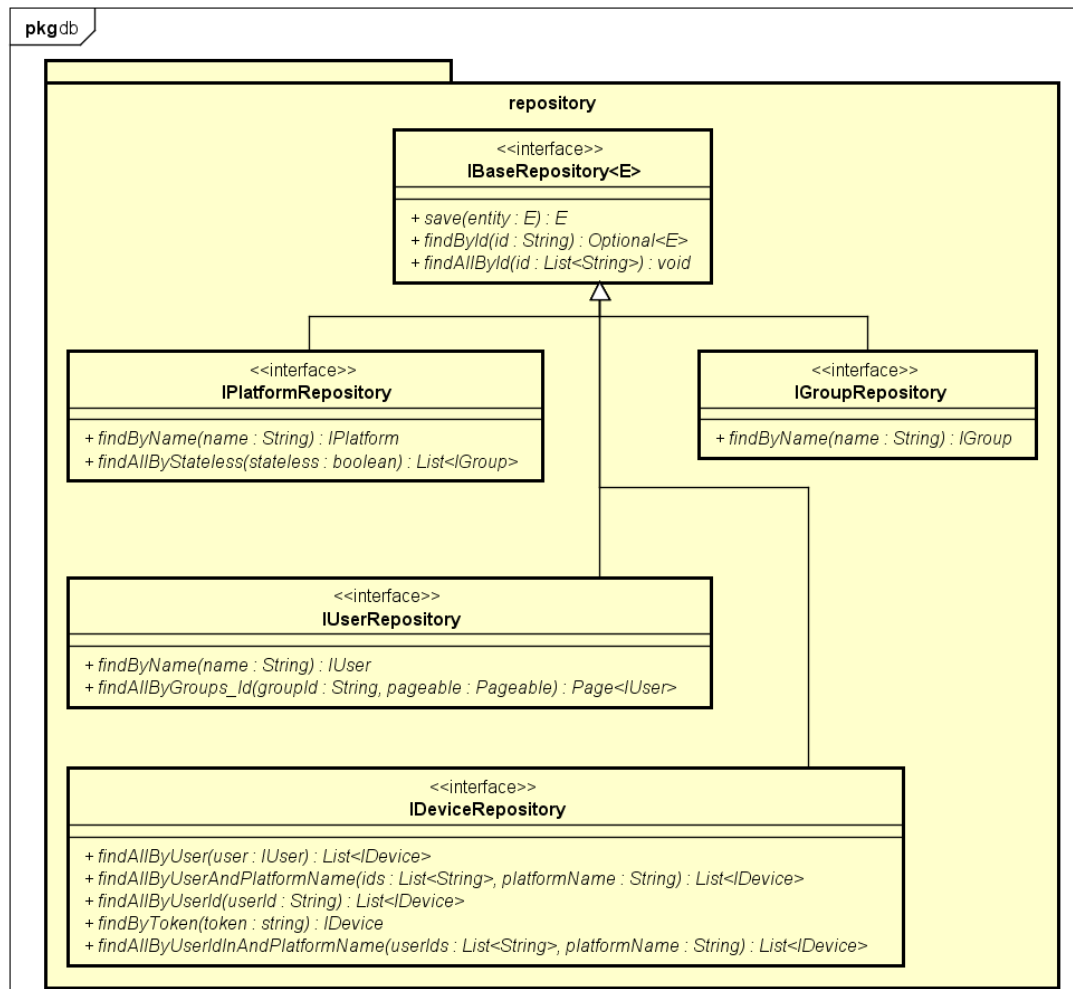


Figure 3.11: Interfaces for database modules in the *core.db* package (Entity)

Any module that wishes to implement access to a database must create JPA Entities implementing the entity interfaces and create interfaces that extend the repository interfaces and extend Spring Data's *CrudRepository* interface. This will indicate to the Spring Context to instantiate repository Beans based on these interfaces[37], which the Context will then inject into the Beans where they are used.

Figure 3.12: Interfaces for database modules in the `core.db` package (Repository)

3.3.3 Platform Modularity

One of the key features of the system is its multi-platform support. In order to be able to support the widest possible range of platforms, the platform portion of the system had to be designed with complete modularity in mind. The results of the design choices made based on these requirements led to the creation of the `IPlatformAdapter` interface and `@PlatformAdapter` annotations, which can be seen in more detail in Figure 3.13.

Any module that wishes to implement support for a new platform must contain at least one Bean implementing the `IPlatformAdapter` interface and annotate it with the `@PlatformAdapter` annotation, which takes the platform's name as a parameter.

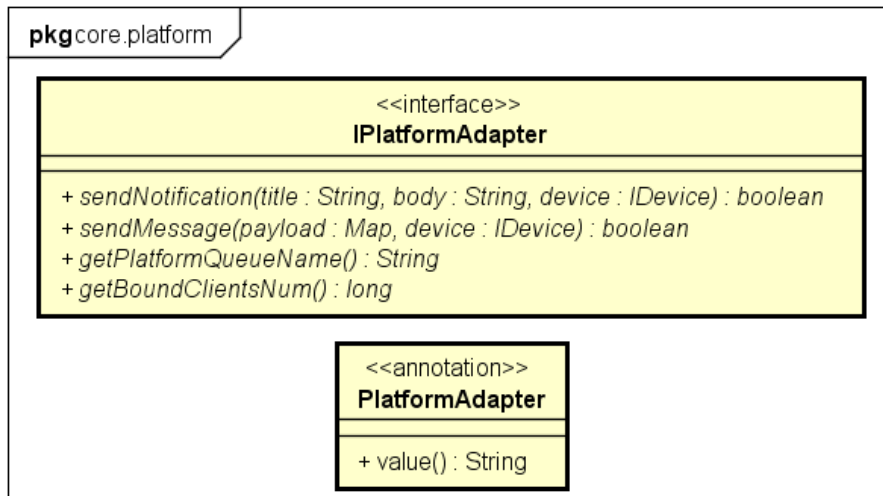


Figure 3.13: Interfaces for platform modules in the *core.platform* package

3.3.3.1 Adapters

Platform Adapters are Spring Managed Beans that implement the Core module's *IPlatformAdapter* interface and are annotated with the *@PlatformAdapter* annotation, which takes the platform's name as a parameter.

Adapters are automatically found at system startup by the Core module's *AdapterService* using Spring's Application Context and registered based on the platform's name, as specified in the *@PlatformAdapter* annotation. The sequence of these events including an example message being sent can be seen illustrated in Figure 3.14

The platform adapters need to subscribe to their relevant queues/topics in the message queue, using the *IReceiver* bean (see Section 3.3.4 Message Queue Modularity). The timing of when to subscribe is up to the adapter, as it may differ based on the platform's needs. For example for the FCM platform, the adapter subscribes after its initialization, while the websocket adapter subscribes after a device is bound to it.

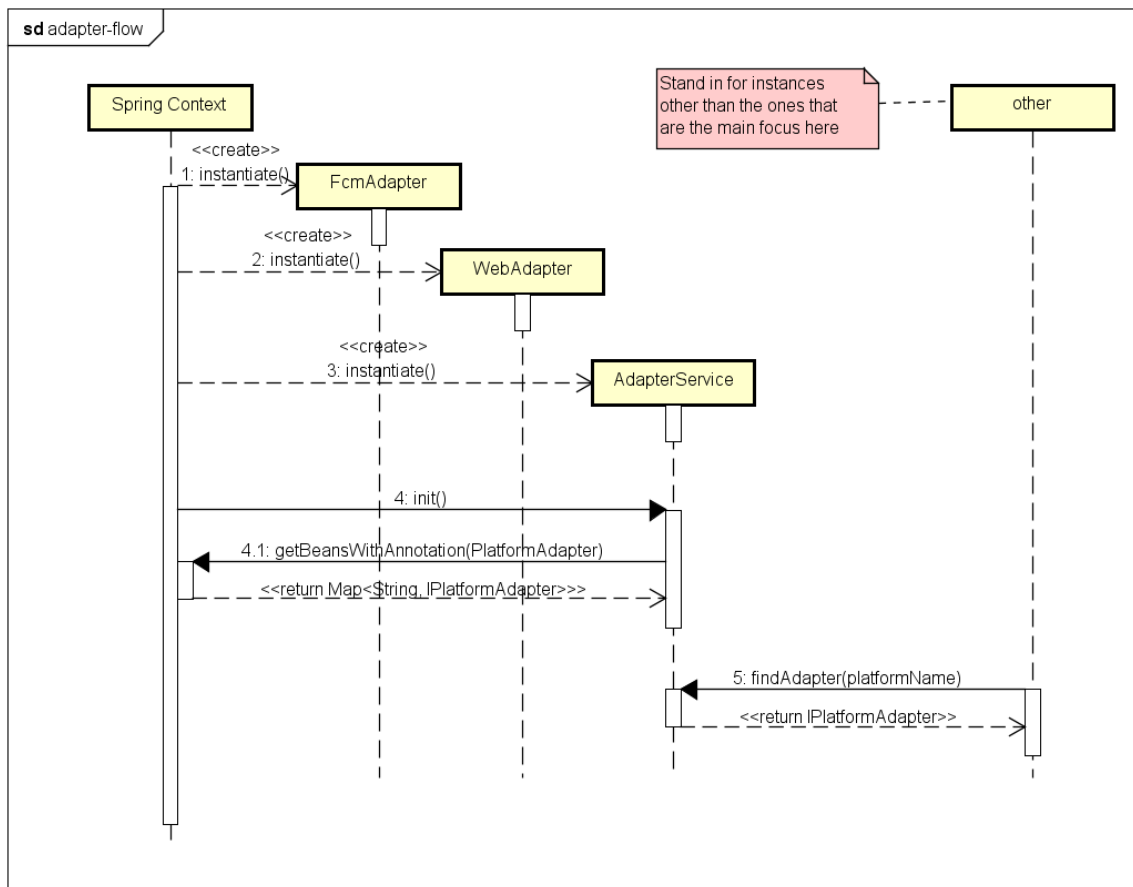


Figure 3.14: Sequence diagram indicating the resolution process of Adapters

3.3.4 Message Queue Modularity

The message queue portion of the application has also been designed in a modular way, so the best solution for each specific scenario can be used. Modules that would implement the message queue functionality need to implement the interfaces defined in the *core.mq* package, which can be seen in Figure 3.15.

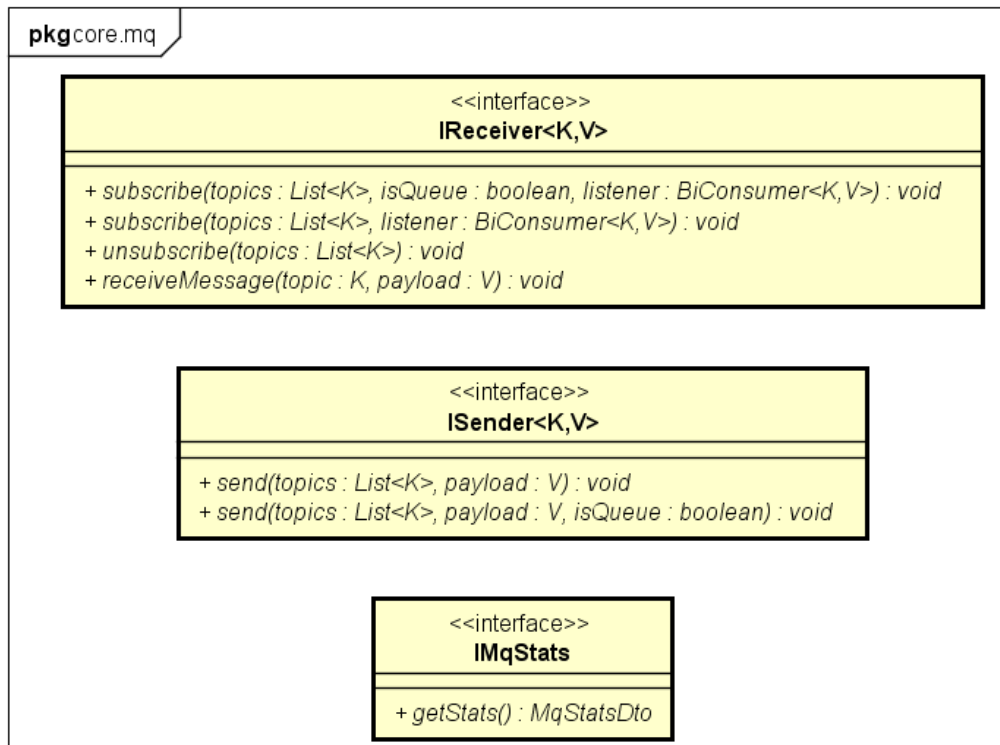


Figure 3.15: Interfaces for message queue modules in the *core.mq* package

The message queue core package consist of three main interfaces, the *ISender*, *IReceiver*, and *IMqStats*. The interfaces use Java generics, where **K** stands for the type of the message queue key. In most message queue implementations, this would be a `String`. The **V** generic stands for the type that is pushed into the message queue. While the sample implementation uses a `String` containing a JSON object, the use of generics in the interface allows implementations of the message queue model to use other types, for example an array of bytes as a direct serialization of Java objects.

Chapter 4

Implementation

This chapter describes the development environment and tools used to implement the system, as well as the sample solution. The detailed structure of the implementation project is presented and explained.

4.1 Development platform

The entire system is developed on Microsoft's Windows 10 platform. Based on the conclusions made from in-depth analysis (see Chapter 2 [Analysis](#)), the primary development language chosen is Groovy, a powerful dynamic Object-Oriented language based on the Java platform, and the following set of applications for development:

Software used for the development of the system core

- **Java Development Kit (JDK) 1.8¹** as the Java runtime and development kit
- **JetBrains IntelliJ Idea 2018.1 (Ultimate Edition)²** as the Integrated Development Environment (IDE)
- **Apache Groovy 2.4³** as the Groovy language compiler
- **Atlassian Sourcetree 2.5⁴** as a source control for GIT

Software used for the development of the sample application

On top of the software used for the development of the system core, the following software was also used in the development of the sample implementation.

¹JDK 1.8 <<http://www.oracle.com/technetwork/pt/java/javase/downloads/jdk8-downloads-2133151.html>>

²JetBrains IntelliJ Idea <<https://www.jetbrains.com/idea/>>

³Apache Groovy <<http://groovy-lang.org/>>

⁴Atlassian Sourcetree <<https://www.sourcetreeapp.com/>>

- **Android Studio 3.0**⁵ as the IDE for developing the sample Android application
- **MySQL 5.6**⁶ as the database provider
- **Apache ActiveMQ 5.15**⁷ as the message queue broker
- **Postman 6.0**⁸ for sending requests to REST endpoints
- **Mozilla Firefox 62**⁹ as web browser and Javascript debugger

4.2 Code Structure

The code of the implementation is structured into 5 groups, henceforth called *projects*, each encasing a complete application. As some applications share parts of their code, in order to avoid duplication, some project modules are designed to be packaged into a JAR library, which is used in other projects. This section describes these projects and their internal structure. The 5 top-level projects are *Msgr*, *Coordinator*, *Java-Client*, *MsgrChattr*, and *Web*.

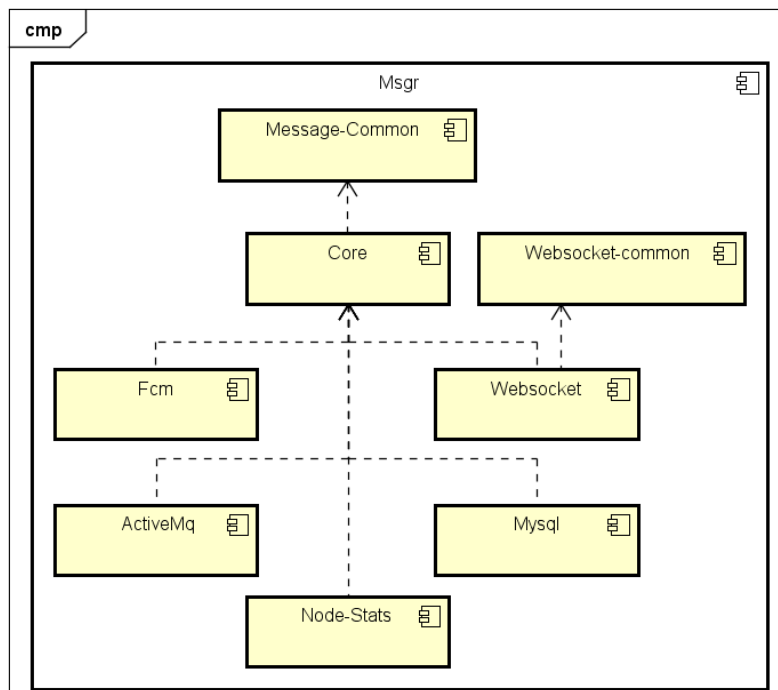


Figure 4.1: Modules in the Msgr project and their dependencies on each other

⁵Android Studio <<https://developer.android.com/studio/>>

⁶MySQL <<https://www.mysql.com/>>

⁷Apache ActiveMQ <<http://activemq.apache.org/>>

⁸Postman <<https://www.getpostman.com/>>

⁹Mozilla Firefox <<https://www.mozilla.org/en-US/firefox/new/>>

4.2.1 Msgr

The Msgr project represents the Node application, containing both the system core and modules containing the sample implementation of the modular components of the system.

The Msgr root project is just a simple Gradle project containing only a Spring Boot application, but no other code. The root project puts together all the individual modules of the application by declaring dependency on its subprojects, the module projects. The project tree, including dependencies between modules can be seen in Figure 4.1. The following subsections further break down and describe the modules.

4.2.1.1 Core Module

The Core Module contains all the interfaces for other modules to implement, which are explained in detail in Sections 3.3.2, 3.3.3, and 3.3.4, the application's core business logic, described in Section 3.2.2, and several controllers providing core APIs. This section contains a brief description of a selection of the classes contained in the Core Module.

Controllers

The controller package contains three classes:

- **ConnectionController**, which serves as the entry point into the system by exposing the `/connect` endpoint. The controller registers the connecting device, based on the `ConnectionRequest` object in the request's payload, and responds with the device's full registration information and a list of addresses of least loaded Nodes.

Example response payload:

```
{
  "addresses": [
    "node1.example.com",
    "node2.example.com",
    "node3.example.com"
  ],
  "deviceData": {
    "userId": "user1",
    "userName": "user name",
    "deviceId": "device1",
    "deviceToken": "device 1"
  }
}
```

- **MessageController**, which provides an API for sending messages and notifications through HTTP REST calls by exposing the `/notification` and `/message` endpoints.

- **HealthController**, which exposes the `/health` endpoint that is used by the Node Coordinator to check the Node's health and receive Node status data.

Example response payload:

```
{
  "load":0.4,
  "memory":58.7
}
```

Queue Processors

The queue package contains classes that perform the processing of messages dequeued from general queues, such as the group and user queues.

CoordinatorConnector

The *CoordinatorConnector* class represents a bean that, at the start of the Spring Application Context, announces the Node to the Node Coordinator, and then caches and refreshes the list of least loaded Nodes from the Coordinator.

Services

The services package contains the *AdapterService* and *MessagingService* classes, which provide most of the core business logic and are described in detail in Section [3.2.2 Business Tier](#).

4.2.1.2 Message-Common

The *Message-Common* module is a simple collection of Data Transfer Objects (DTOs), which serve to encapsulate data sent over APIs, and a few utility methods for working with them. The *Message-Common* module is meant to be used as a library in other projects which use these same classes, such as the *Java-Client*.

4.2.1.3 Fcm

The *Fcm* module contains the adapter implementation for the FCM platform.

4.2.1.4 Mysql

The *Mysql* module contains implementations for the Data Tier modularity interfaces, as described in Section [3.2.1 Data Tier](#).

4.2.1.5 Websocket

The *Websocket* module contains the adapter implementations for the Websocket platform.

4.2.1.6 Websocket-Common

The *Websocket-Common* module contains classes that are shared between the *Websocket* module and the *Java-Client* project.

4.2.1.7 ActiveMq

The *ActiveMq* module contains the implementations of the message queue interfaces for the ActiveMQ platform, along with all message queue related logic.

4.2.1.8 Node-Stats

The *Node-Stats* module exposes the */stats* endpoint, which provides detailed information on the current state of the Node, including the number of connections to the message queue, bound devices, and more.

Example response payload:

```
{
  "system": {
    "load": 0.272,
    "memory": 37.92
  },
  "totalBoundClients": 0,
  "platformClients": {
    "websocket": 0
  },
  "mq": {
    "totalConnections": 1,
    "sessions": 3,
    "origins": [
      "queue://FCM",
      "queue://q.group.*",
      "queue://q.user.*"
    ]
  }
}
```

4.2.2 Coordinator

The Coordinator project represents the Node Coordinator application, which maintains a list of all connected Nodes and performs regular health checks on them. The Coordinator also exposes */free-nodes* endpoint, which provides the list of *n* least loaded nodes, and a monitor page containing the current status and health of the Coordinator and all its connected Nodes (see Figure 4.2).

The Coordinator further includes the *Common* module subproject, which contains utility classes for obtaining statistics about the system, such as CPU load and free memory. This module is used as a library in the *Core* module of the *Msg* project.

| Msgr NodeCoordinator status | | |
|------------------------------------|-----------------|-----------------------|
| CPU Load (%) | Free Memory (%) | Memory (free/total) |
| 0.4 | 64.97 | 244565232/376438784 B |






| Msgr Node status monitor | | | | | | |
|---------------------------------|-------------------|------------|--------------|-----------------|-------------------------------|---|
| Total connected nodes: 5 | | | | | | |
| ID | Address | Responding | CPU Load (%) | Free Memory (%) | Last Successful Check | More |
| Node001 | 192.168.1.2:8080 | ✓ | 35.0 | 50.64 | Wed May 23 23:48:49 CEST 2018 |  |
| Node-server | 192.168.1.42:8091 | ✓ | 1.9 | 19.11 | Wed May 23 23:48:49 CEST 2018 |  |
| Node004 | 192.168.1.2:8083 | ✓ | 34.9 | 52.34 | Wed May 23 23:48:49 CEST 2018 |  |
| Node003 | 192.168.1.2:8082 | ✓ | 35.0 | 53.15 | Wed May 23 23:48:49 CEST 2018 |  |
| Node002 | 192.168.1.2:8081 | ✓ | 35.0 | 53.65 | Wed May 23 23:48:49 CEST 2018 |  |

Figure 4.2: Node Coordinator monitor page

4.2.3 Java-Client

The Java-Client project contains the classes forming the Java platform client library, which is described in detail in Section 3.2.4.1 [Dektop, Server, and Mobile](#).

4.2.4 Web

The Web project contains the sample implementation for the Web platform, which consists of a Javascript client library and a web application, called Chattr, which functions as an IM chat that can be used to send and receive messages (see Figure 4.3).

4.2.5 MsgrChattr

The MsgrChattr project contains the Android application that is part of the sample implementation for the mobile platform. The app consists of a single screen, or Activity, as they are called in Android terminology, which is able to receive and send messages, showing them in an IM chat fashion (see Figure 4.4). The application uses Google's Firebase Cloud Messaging to receive Android notifications and messages and the Java-Client library to access the system and send messages.

Chattr messages

Device:

Username:

User id: ed4b0830-756f-4646-8fde-fbc86316e8ad

Status: Connected

Messages:

- 5/24/2018, 12:53:43 AM @Server: Connected to node 192.168.1.2:8080
- 5/24/2018, 12:54:36 AM @test-user: Hello!
- 5/24/2018, 12:54:50 AM @oscar: Oh hi there!
- 5/24/2018, 12:54:57 AM @oscar: How are you?
- 5/24/2018, 12:55:09 AM @test-user: I'm great, thanks

Write a message:

Target Type:

Target:

Message :

Figure 4.3: Chattr web application screenshot

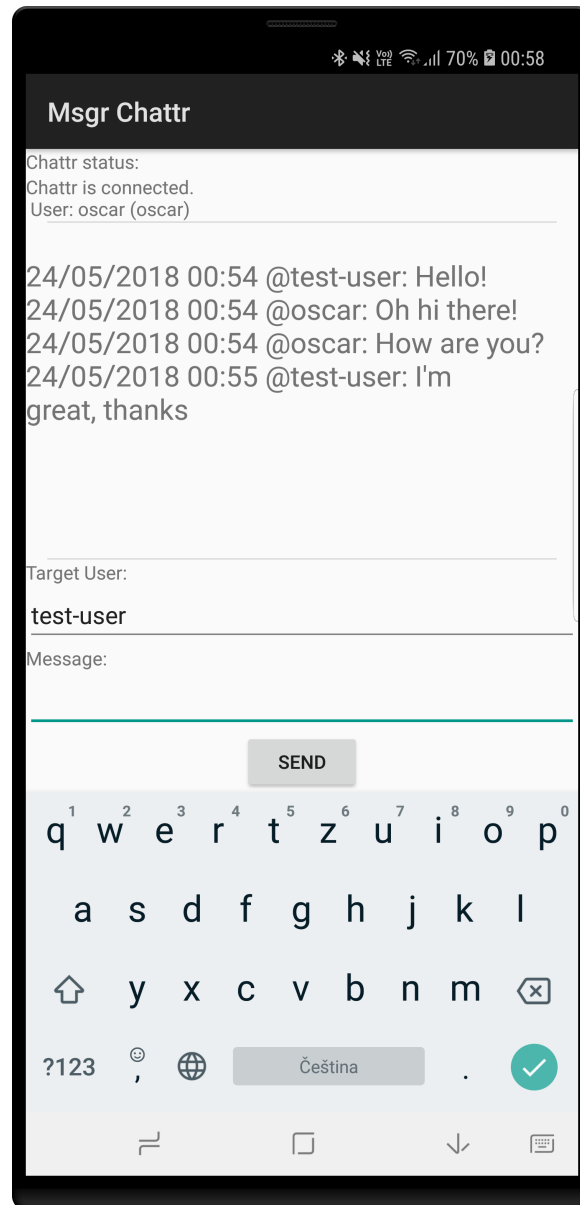


Figure 4.4: MsgrChattr Android application screenshot

Chapter 5

Testing

Testing is an inseparable component of software development and is an integral part of the code writing process. In order to ensure the designed and implemented system works correctly and satisfies the requirements placed upon it, three different approaches to testing and verification were chosen: automated tests, manual tests and performance tests. This chapter describes the performed tests, the environments in which they were performed, and their outcomes along with derived conclusions.

5.1 Automated Testing

Automated tests for the application are written using the Spock framework, described in Chapter 2.6.2.2 [Spock Framework](#). The Spock framework has a powerful mocking API, which is used in order to test application logic in isolation in the form of unit tests. Spock also provides support for easily implemented Data Driven and Interaction Based testing, both of which were used.

Where needed, automated Spock's Spring plugin is used in order to run integration tests, which are used for verification of logic that relies on the Spring Application Context to be loaded. For integration tests, an embedded H2¹ database is used, allowing for tests to easily be run automatically by continuous integration tools, without the need of supplying an external test database.

5.2 Manual Testing

Due to the nature of the system and the complexity of testing a distributed system, the sample implementation application, Chattr, was created to verify the proper behaviour of the system. Using this, a large part of the tests of the functionality of the system as a whole was done using manual tests, based on use-case scenarios. This section describes the test scenarios and the environment in which they were performed.

¹H2 Database Engine <<http://www.h2database.com/html/main.html>>

5.2.1 Testing Environment

In order to approximate a distributed environment, the testing environment consists of two machines, on which the system is run, utilizing each machine to run multiple applications and Nodes. The deployment diagram of the system testing environment can be seen in Figure 5.1.

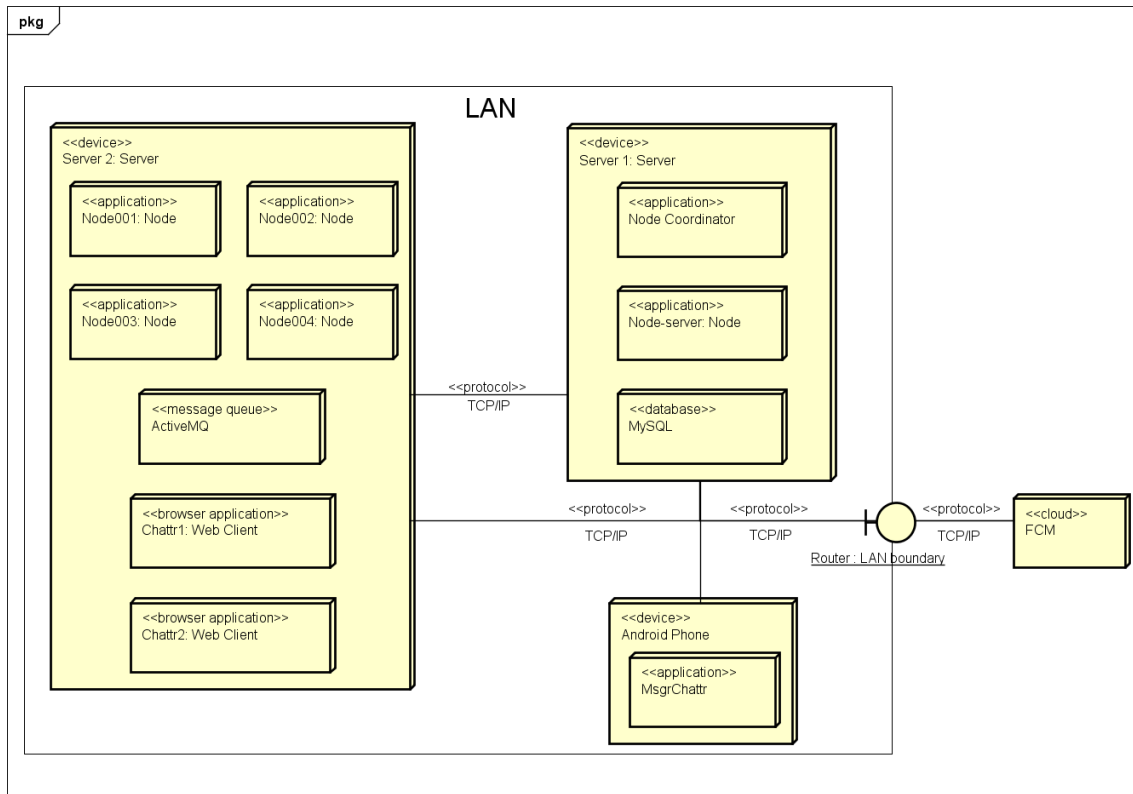


Figure 5.1: Deployment diagram of the testing environment

The machine called *Server2* runs four Node instances, an ActiveMQ instance and two instances of the Web client, Chattr. The second machine, called *Server1*, runs a Node instance, the Node Coordinator, and a MySQL database. Both machines share the same Local Area Network (LAN), with an Android smart phone also connected to the same network. The LAN is connected to the internet, where the Firebase Cloud Messaging service resides in the cloud.

5.2.1.1 Testing Environment Machine Specifications

The hardware and operating system specifications of the two machines used in the testing environment are detailed in Table 5.1.

| Machine | Operating System | CPU | RAM |
|---------|-----------------------------|--|-------------------|
| Server1 | Ubuntu Server 16.04 LTS x64 | Intel Core i5-7500T, 4 cores @ 2.7GHz | 8GB DDR3L 1600MHz |
| Server2 | Windows 10 Home x64 | Intel Core i7-3610QM, 4 cores @ 2.3GHz | 16GB DDR3 1600MHz |

Table 5.1: Testing environment machine specifications

5.3 Performance Testing

As the system is meant to support large-scale real-time applications, throughput and message processing performance are critical. For the purposes of gauging the system's message throughput in different scenarios, a performance benchmark application was created. The *PerformanceTester* application is a console application written in Groovy, which uses the *Java-Client* library to communicate with the system.

This Section describes the performance tests performed, presents the obtained experimental data and uses it to formulate conclusions on the system's performance.

5.3.1 Tests

All tests were performed in the testing environment described in Section 5.2.1 [Testing Environment](#), with messages being addressed to a user. This means the message goes through the following flow:

The message is sent to a Node, where it gets processed and pushed into the user message queue, from which another Node dequeues it, processes it, and as it is a message addressed for a user, unfolds the user into their devices, pushing the message into the devices respective message queues. After this, the Nodes that each of these devices is bound to dequeues the message and passes it to its client.

Every test was repeated four times in order to average out any abnormalities caused by other processes on the machines.

The precision of the time measurements is in milliseconds (ms).

5.3.2 Test 1: Single Node, single client. 50 messages

In this test, the system consist only of *Node001*, running on the same machine as the *PerformanceTester* application.

The *PerformanceTester* connects a single client to the Node and sends 50 messages addressed to itself. As all of the messages are sent almost simultaneously, with no delay between them, this test floods the system with a large number of messages at the same time.

5.3.3 Test 2: Single Node, single client. 200 messages

The scenario for this test mirrors Test 1, except this time 200 messages are sent simultaneously.

| Measurement # | Average message delivery time (ms) |
|---------------|------------------------------------|
| 1 | 0.3 |
| 2 | 0.42 |
| 3 | 0.48 |
| 4 | 0.18 |

Table 5.2: Average message delivery time for Test 1

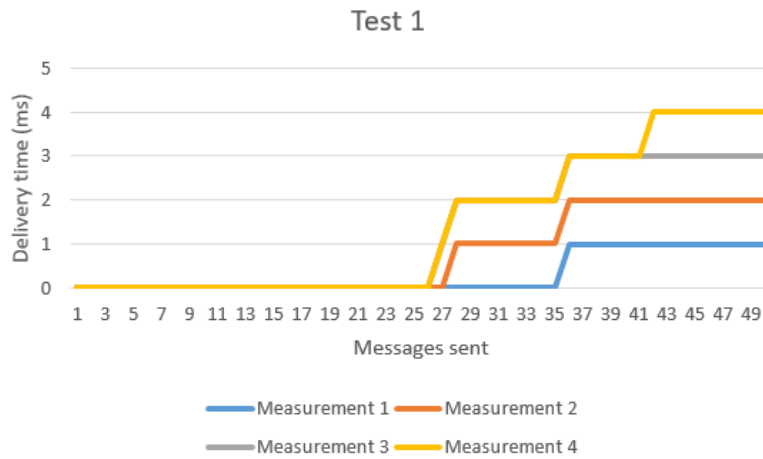


Figure 5.2: Test 1: Delivery times

| Measurement # | Average message delivery time (ms) |
|---------------|------------------------------------|
| 1 | 1.205 |
| 2 | 1.125 |
| 3 | 1.105 |
| 4 | 1.32 |

Table 5.3: Average message delivery time for Test 2

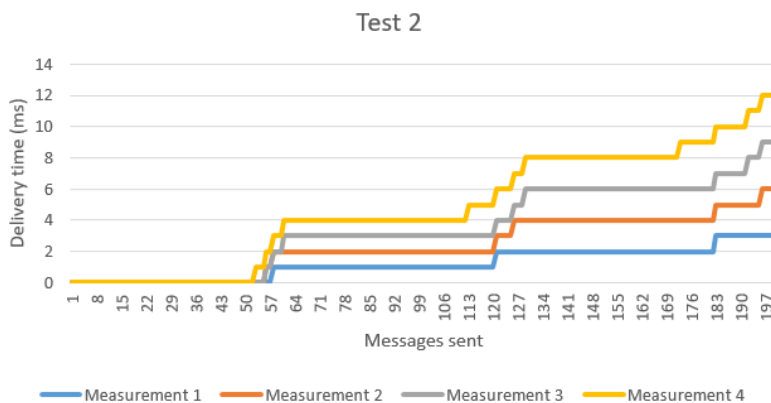


Figure 5.3: Test 2: Delivery times

The data from Tests 1 and 2 shows a clear connection between the number of messages sent simultaneously and their delivery time. This is most likely due to the congestion of the websocket communication channel, as all messages are pushed into it at once with little to no spacing between them.

5.3.4 Test 3: 4 Nodes, single machine. 50 messages

The scenario for this test is similar to the scenarios in Tests 1 and 2, but enhanced by the addition of Node instances to the system, so the *Server2* machine now contains four Nodes and the PerformanceTester is running a separate websocket client for each.

For every message to be sent, a random client is chosen and the message is sent as a broadcast to all the clients, including the sender. This effectively means that while 50 messages are sent out, 200 messages in total are received - 50 for each client.

| Measurement # | Average message delivery time (ms) |
|---------------|------------------------------------|
| 1 | 2.97 |
| 2 | 2.92 |
| 3 | 1.975 |
| 4 | 2.785 |

Table 5.4: Average message delivery time for Test 3

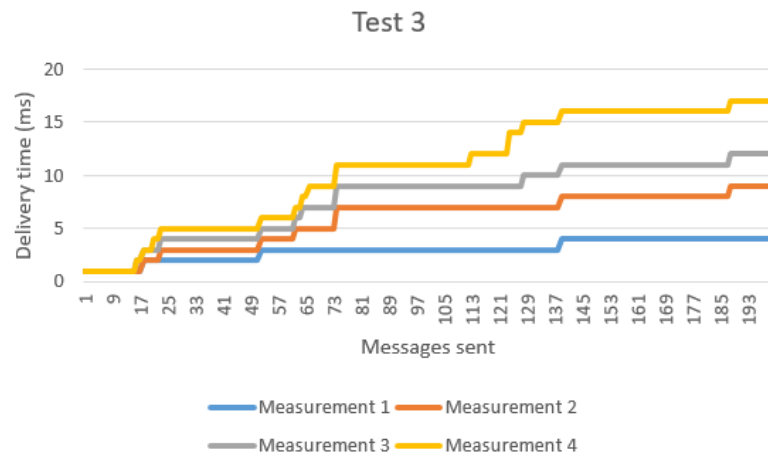


Figure 5.4: Test 3: Delivery times

5.3.5 Test 4: 5 Nodes, two machines. 50 messages

The scenario for this test further enhances the previous scenarios by adding yet another Node instance, the *Node-server* on machine *Server1*, therefore scaling the system onto a second machine.

As there are now 5 nodes, it also means 50 more messages need to be delivered, making it a total of 250 delivered messages.

| Measurement # | Average message delivery time (ms) |
|---------------|------------------------------------|
| 1 | 3.296 |
| 2 | 3.448 |
| 3 | 2.852 |
| 4 | 3.2 |

Table 5.5: Average message delivery time for Test 4

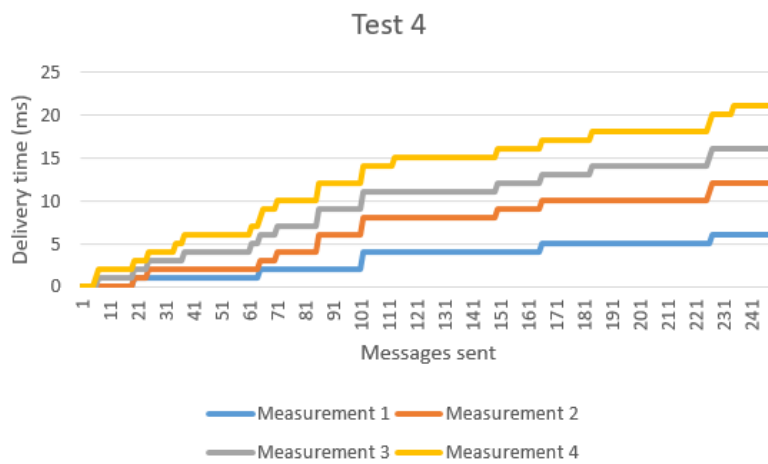


Figure 5.5: Test 4: Delivery times

The data from Tests 3 and 4 continue to support the trend from Tests 1 and 2, where the larger the amount of messages pushed through the communication channel at one time, the longer the message delivery time. However, for the same amount of delivered messages (200), we can see that Test 3 has higher average delivery time than Test 2. As Test 3 distributes the load among multiple Nodes, this further supports the hypothesis that the bottleneck that is being congested is the websocket connection, not the system itself.

5.3.6 Test 5: Communication simulation. 50 messages, 20ms interval

This scenario aims to simulate a real-time communication between two clients, with a two-way constant stream of data from one to the other and vice versa. One client is connected to Node *Node001* on machine *Server2* and the other to Node *Node-server* on machine *Server1*. The individual messages are separated with 20ms intervals, mimicking a real communication closer than the previous tests, which pushed all the messages at once.

| Measurement # | Average message delivery time (ms) |
|---------------|------------------------------------|
| 1 | 0.42 |
| 2 | 0.42 |
| 3 | 0.32 |
| 4 | 0.22 |

Table 5.6: Average message delivery time for Test 5

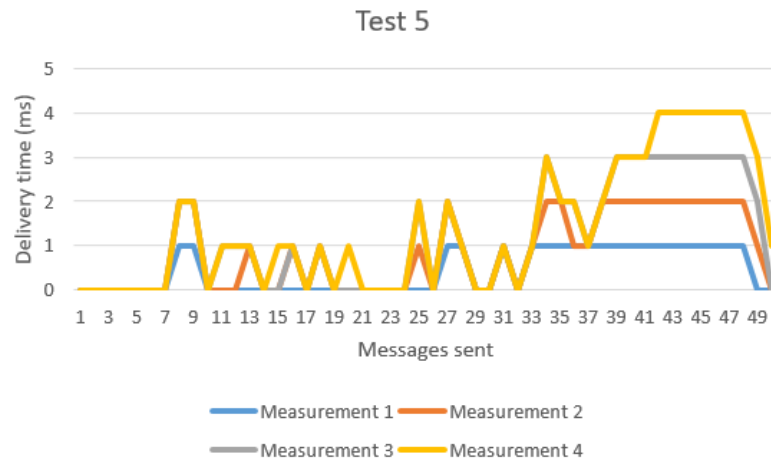


Figure 5.6: Test 5: Delivery times

5.3.7 Test 6: Communication simulation. 200 messages, 20ms interval

This scenario is an extension of the scenario in Test 5, increasing the number of sent messages from 50 to 200.

| Measurement # | Average message delivery time (ms) |
|---------------|------------------------------------|
| 1 | 2.17 |
| 2 | 2.19 |
| 3 | 1.21 |
| 4 | 1.47 |

Table 5.7: Average message delivery time for Test 6

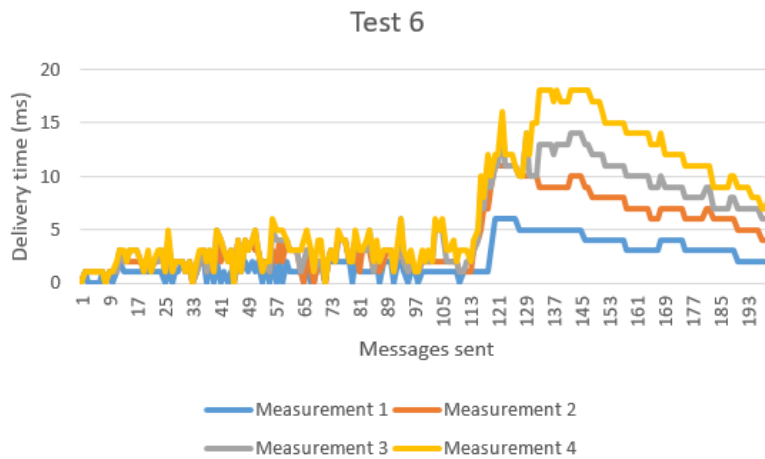


Figure 5.7: Test 6: Delivery times

The data from Tests 5 and 6 immediately shows a drastic decrease in delivery time as congestion on the websockets is relieved and processing load is distributed over two machines. The times are higher than in Tests 1 and 2, but that is likely due to the fact that messages have to travel over the network to *Node-server*, while Tests 1 and 2 were executed completely on a single machine and the PerformanceTester only had to maintain a single client.

5.3.8 Test 7: Communication simulation. 200 messages, 100ms interval

This scenario builds upon the findings of Tests 5 and 6, where a more realistic interval between messages decreased the congestion on the websocket channel, and further increases this interval up to 100ms. This scenario simulates a real-time application that sends ten messages per second.

| Measurement # | Average message delivery time (ms) |
|---------------|------------------------------------|
| 1 | 0.115 |
| 2 | 0.125 |
| 3 | 0.115 |
| 4 | 0.095 |

Table 5.8: Average message delivery time for Test 7

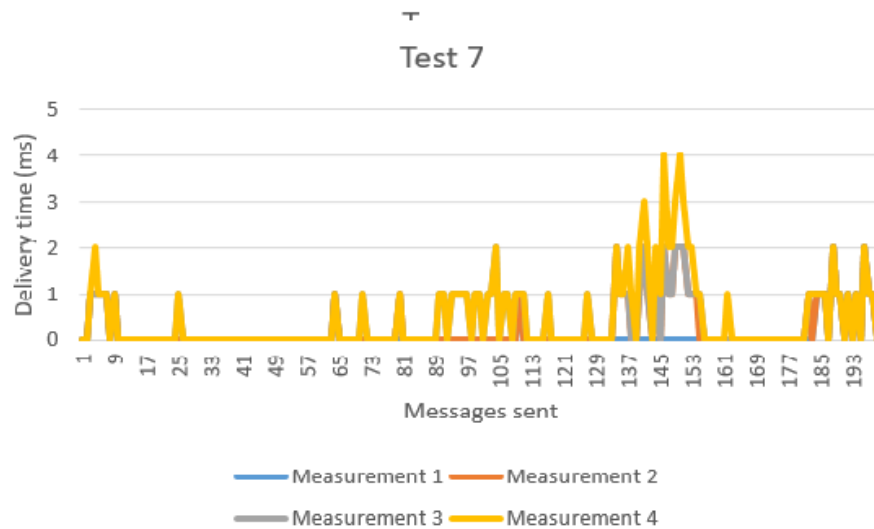


Figure 5.8: Test 7: Delivery times

5.3.9 Test 8: Communication simulation. 1000 messages, 100ms interval

Based upon the hypothesis that the bottleneck shown in Tests 1-5, this scenario keeps the 100ms interval of Tests 7, but sends a five times larger volume of data, 1000 messages.

The expected outcome of this test is that the average message delivery time will be comparable to the one in Test 7, as the lower congestion of the websocket channel allows messages to flow and the system has no trouble processing the higher volume of messages.

| Measurement # | Average message delivery time (ms) |
|---------------|------------------------------------|
| 1 | 0.189 |
| 2 | 0.061 |
| 3 | 0.082 |
| 4 | 0.52 |

Table 5.9: Average message delivery time for Test 8

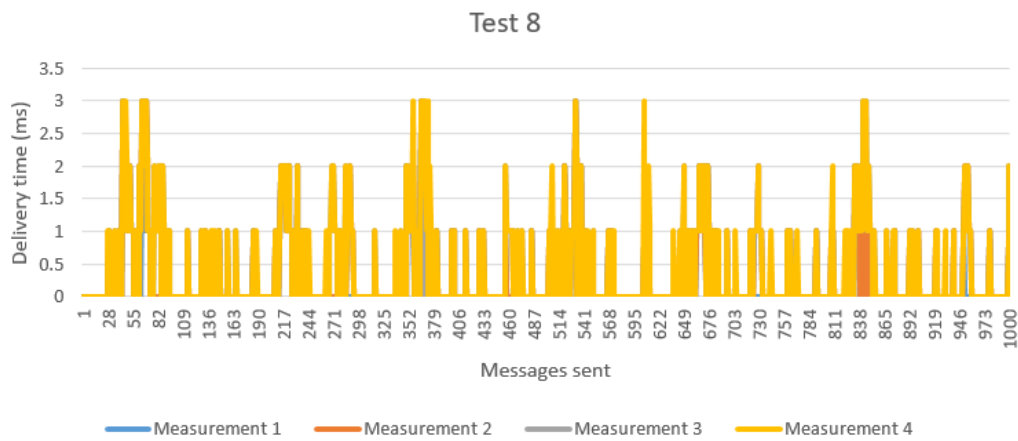


Figure 5.9: Test 8: Delivery times

5.3.10 Performance Test Conclusion

The data from Tests 7 and 8 shows that most messages are delivered under 1ms, even with a large volume of messages. Combining this with the data from Tests 1-6, it can be assumed that the largest bottleneck of the system is the websocket channel between Nodes and clients and is based solely on connectivity.

Chapter 6

Conclusion

6.1 Goal Evaluation

The goals of this thesis were to design and implement a system that can serve as a framework and foundation to be built upon and expanded for usage in modern applications that need fast communication among a potentially large numbers of devices.

The application, which is the culmination of this thesis, is a highly modular and extensible framework, capable of running distributed across multiple instances and machines with capability for easy horizontal scaling by way of adding new instances.

In order to demonstrate the functionality of the system, as well as provide an example of the implementation of its highly modular components, a sample application was developed, showcasing the system on representatives of the Web, Mobile, Desktop and Server platforms. The application was also subjugated to multiple forms of testing.

6.2 Suggestions for Future Expansion

This thesis provides a framework that is designed to be further built upon, improved, and expanded. This Section provides some ideas that were outside the scope of this thesis.

6.2.1 Authentication

The system in its current state already provides a logical division of devices between users and allows for aggregation of users into groups. A possible improvement is evident: add support for user authentication.

6.2.2 System Administration

This improvement suggestion goes hand in hand with the one mentioned above, in Section [6.2.1 Authentication](#). Once user authentication is possible, the system can be extended with administration interfaces, allowing users with special privileges to manage devices, users and groups.

6.2.3 Encryption

While at its current state, the system could easily be used to send data with End-to-End encryption by encrypting and decrypting it before sending and after receiving, respectively, optional End-to-End encryption could be built-in in the client libraries.

Bibliography

- [1] Ably. Ably protocol adapters. <https://www.ably.io/adapters/>. 3. 1. 2018.
- [2] Ably. Do you binary encode your messages for greater efficiency? <https://support.ably.io/support/solutions/articles/3000047365-do-you-binary-encode-your-messages-for-greater-efficiency-/>. 3. 1. 2018.
- [3] Ably. Documentation. <https://www.ably.io/documentation/>. 3. 1. 2018.
- [4] Ably. Download. <https://www.ably.io/download/>. 3. 1. 2018.
- [5] Ably. How does ably count peak connections? <https://support.pubnub.com/support/solutions/articles/14000043668-how-are-peak-connections-counted-/>. 9. 1. 2018.
- [6] Ably. How long are messages stored for? <https://support.ably.io/support/solutions/articles/3000030059-how-long-are-messages-stored-for-umentation/>. 3. 1. 2018.
- [7] Ably. Reliable message ordering for connected clients. <https://support.ably.io/support/solutions/articles/3000044641-guaranteed-message-ordering-for-connected-clients/>. 3. 1. 2018.
- [8] A. ActiveMQ. Cross Language Clients. <http://activemq.apache.org/cross-language-clients.html/>. 22. 5. 2018.
- [9] Apple. Enrollment. <https://developer.apple.com/support/enrollment/>. 22. 5. 2018.
- [10] Apple. Purchase and Activation. <https://developer.apple.com/support/purchase-activation/>. 22. 5. 2018.
- [11] Baeldung. Build tools market share. <http://www.baeldung.com/java-in-2017#build/>. 11. 1. 2018.
- [12] bytefish (via GitHub). Any plan to add support for new http v1 api? <https://github.com/bytefish/FcmJava/issues/37/>. 14. 1. 2018.
- [13] G. Firebase. About fcm messages. <https://firebase.google.com/docs/cloud-messaging/concept-options/>. 14. 1. 2018.

- [14] S. Framework. Spock framework reference documentation. http://spockframework.org/spock/docs/1.1/all_in_one.html/. 14. 1. 2018.
- [15] Gartner. Global mobile os market share in sales to end users from 1st quarter 2009 to 2nd quarter 2017. <https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/>. 9. 1. 2018.
- [16] Google. How to use the Play Console. <https://support.google.com/googleplay/android-developer/answer/6112435?hl=en/>. 22. 5. 2018.
- [17] Google. Set up a javascript firebase cloud messaging client app. <https://firebase.google.com/docs/cloud-messaging/js/client/>. 15. 1. 2018.
- [18] Gradle. Maven vs gradle. <https://gradle.org/maven-vs-gradle/>. 11. 1. 2018.
- [19] Hibernate. Hibernate ogm. <http://hibernate.org/ogm/>. 14. 1. 2018.
- [20] Hibernate. Hibernate orm. <http://hibernate.org/orm/>. 14. 1. 2018.
- [21] JUnit. JUnit 5 user guide. <http://junit.org/junit5/docs/current/user-guide/>. 14. 1. 2018.
- [22] OneSignal. Product overview. <https://documentation.onesignal.com/docs/>. 3. 1. 2018.
- [23] OneSignal. Security. <https://documentation.onesignal.com/v4.0/docs/security/>. 8. 1. 2018.
- [24] OneSignal. Sending notifications. <https://documentation.onesignal.com/v3.0/docs/sending-notifications/>. 8. 1. 2018.
- [25] PubNub. Can i use catchup to retrieve older messages? <https://support.pubnub.com/support/solutions/articles/14000043538-can-i-use-catchup-to-retrieve-older-messages-/>. 5. 12. 2017.
- [26] PubNub. Goinstant. <https://www.pubnub.com/goinstant/>. 8. 1. 2018.
- [27] PubNub. How are peak connections counted? <https://support.pubnub.com/support/solutions/articles/14000043668-how-are-peak-connections-counted-/>. 9. 1. 2018.
- [28] PubNub. Realtime Pub/Sub Messaging. <https://www.pubnub.com/products/realtime-messaging/>. 5. 12. 2017.
- [29] Pusher. Community libraries. <https://pusher.com/docs/libraries#community-libraries/>. 9. 1. 2018.
- [30] Pusher. Libraries. <https://pusher.com/docs/libraries#rest-libraries/>. 9. 1. 2018.
- [31] Pusher. Querying application state. https://pusher.com/docs/server_api_guide/interact_rest_api#querying-application-state/. 9. 1. 2018.

- [32] RabbitMQ. Clients & Developer Tools. <https://www.rabbitmq.com/devtools.html/>. 22. 5. 2018.
- [33] RabbitMQ. RabbitMQ JMS Client. <https://www.rabbitmq.com/jms-client.html/>. 22. 5. 2018.
- [34] Spring. Bean overview. <https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-definition/>. 16. 1. 2018.
- [35] Spring. Convention over configuration. <https://docs.spring.io/spring/docs/3.0.0.M3/reference/html/ch16s10.html/>. 11. 1. 2018.
- [36] Spring. The ioc container. <https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html/>. 11. 1. 2018.
- [37] Spring. Working with spring data repositories. <https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html/>. 16. 1. 2018.
- [38] Statista. Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions). <https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>. 22. 5. 2018.
- [39] Statista. Number of wireless local area network (WLAN) connected devices worldwide from 2016 to 2021 (in billions). <https://www.statista.com/statistics/802706/world-wlan-connected-device/>. 22. 5. 2018.
- [40] R. (via GitHub). HTTP v1 API. <https://github.com/Radius/Pushraven/commit/ca4356c6fb034042dcd576ac2364c4386a9bfe6f/>. 14. 1. 2018.

Appendix A

List of abbreviations

WLAN Wireless Local Area Network

IoT Internet of Things

IM Instant Messaging

API Application Programming Interface

JVM Java Virtual Machine

SaaS Software as a Service

PaaS Platform as a Service

FAQ Frequently Asked Questions

SDK Software Development Kit

REST Representational State Transfer

HTTP Hypertext Transfer Protocol

HTTPS HTTP Secure

ART Android Runtime

Dex Dalvik Executable

JRE Java Runtime Environment

JDK Java Development Kit

OS Operating System

IDE Integrated Development Environment

GUI Graphical User Interface

HTML Hyper Text Markup Language

| | |
|-------------|--|
| CSS | Cascading Style Sheets |
| JAR | Java ARchive |
| URL | Uniform Resource Locator |
| IoC | Inversion of Control |
| DI | Dependency Injection |
| XML | Extensible Markup Language |
| DSL | Domain-Specific Language |
| SQL | Structured Query Language |
| GPL | General Public Licence |
| HA | High Availability |
| ORM | Object-Relational Mapping |
| HQL | Hibernate Query Language |
| JPA | Java Persistence API |
| ASL | Apache Licence |
| LGPL | GNU Lesser General Public Licence |
| FCM | Firestore Cloud Messaging |
| TTL | Time to Live |
| XMPP | Extensible Messaging and Presence Protocol |
| POJO | Plain Old Java Object |
| DNS | Domain Name System |
| DTO | Data Transfer Object |
| LAN | Local Area Network |
| CPU | Central Processing Unit |
| RAM | Random Access Memory |

Appendix B

User Guide

This chapter aims to provide a comprehensible guide to building, deploying and using the system.

The steps described in this guide have been tested on 64bit versions of Windows 10 and Ubuntu 16, but should be compatible with most modern versions of Windows, Linux, and MacOS operating systems.

B.1 Building from Code

This Section describes the steps needed to build the applications forming the system from the source code provided.

B.1.1 System Requirements

Minimum System Requirements

- 1GHz processor
- 2GB RAM
- 500MB free hard drive space

Recommended System Requirements

- 64bit Windows 10 or Ubuntu 16 operating system
- 2GHz multi-core processor
- 8GB RAM
- 1GB free hard drive space

B.1.2 Software Prerequisites

In order to build and the applications, the following software must be installed on the user's machine:

- Java JDK 1.8+
- Gradle 4.0+

(May be replaced by using the projects' Gradle wrappers. To use the Gradle wrappers, replace commands `gradle <command>` with `./gradlew <command>` on Linux and `gradlew.bat <command>` on Windows.)

You can verify their presence by running the the `java -version` and `gradle -version`, which should give a result similar to the following:

```
$ java -version
java version "1.8.0_101"
Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode)
```

```
$ gradle -version
```

```
-----
Gradle 4.7
-----
```

```
Build time: 2018-04-18 09:09:12 UTC
Revision:   b9a962bf70638332300e7f810689cb2feb4a6c

Groovy:    2.4.12
Ant:       Apache Ant(TM) version 1.9.9 compiled on February 2 2017
JVM:      1.8.0_101 (Oracle Corporation 25.101-b13)
OS:       Linux 4.4.0-43-Microsoft amd64
```

B.1.3 Building Shared Dependencies

Some applications depend on libraries built from other applications' submodules. This Section describes the order in which these dependencies need to be built and the folders where the built libraries are to be placed.

B.1.3.1 Coordinator/Common Module

In order to build this module, navigate to the `messagingSystem/coordinator/common` folder and run `gradle assemble`. If successful, the command output should contain the following:

```
BUILD SUCCESSFUL in 9s
2 actionable tasks: 2 up-to-date
21:57:57: Task execution finished 'assemble'.
```

Next, move the assembled JAR file from *messagingSystem/coordinator/common/build/libs/common-1.1.jar* to *messagingSystem/msgr/core/lib*.

B.1.3.2 Msgr/Message-Common Module

Navigate to *messagingSystem/msgr/message-common* and run `gradle assemble`.

Move the assembled JAR file from *messagingSystem/msgr/message-common/build/libs/message-common-1.0.jar* to *messagingSystem/java-client/lib*.

B.1.4 Building the Node and Node Coordinator Applications

If the previous steps have been completed, navigate to *messagingSystem/msgr* for the Node application or *messagingSystem/coordinator* for the Node Coordinator.

Run the `gradle assemble` command, which will build the project, along with all its submodules and produce a JAR file, which can be found in the *./build/libs* folder. This file is a runnable JAR file. For information on how to configure and run the Node and Node Coordinator, please see Sections [B.2.2 Running the Node Application](#) and [B.2.1 Running the Node Coordinator](#).

B.1.5 Building the Java-Client

In the *messagingSystem/java-client* folder, running the `gradle shadowJar` command will produce the complete library JAR file, which includes all its dependencies. The JAR file will be located in *messagingSystem/java-client/build/libs/java-client-1.0-SNAPSHOT-all.jar*

B.1.6 Building the MsgrChattr Android Application

In order to build the MsgrChattr Android Application, the Java-Client JAR must be placed in the *messagingSystem/MsgrChattr/app/libs* folder. For instructions on how to build the Java-Client and where to find the JAR file, see Section [B.1.5 Building the Java-Client](#).

For building the Android app itself, refer to the official documentation depending on which type of build is needed: <https://developer.android.com/studio/build/building-cmdline>.

B.1.7 Building the PerformanceTester

To build the PerformanceTester, navigate to the *messagingSystem/PerformanceTester* and run the `gradle shadowJar` command.

B.2 Running the Applications

This Section describes how to run and configure the individual applications that are part of the system. This Section assumes that all applications have been built into runnable JAR files, as described in Section [B.1 Building from Code](#)

B.2.1 Running the Node Coordinator

In order to run the Node Coordinator, start the runnable JAR file using `java -jar coordinator-1.0.jar` and the application will begin to boot.

While the Node Coordinator application uses some sensible default configuration values, it is possible to change these by using a configuration file. The configuration file must be placed in `config/application.properties`, relative to where the JAR file is being run from. The options configurable in the properties file are described in the following properties template:

```
# cron expression to use for healthcheck, if null,
  coordinator.healthcheck.seconds will be used
coordinator.healthcheck.cron=
# number of seconds between healthchecks, used only if
  coordinator.healthcheck.cron is null. Defaults to 5
coordinator.healthcheck.seconds=

# number of milliseconds for request timeout when getting healthcheck. Defaults to
  1000
coordinator.healthcheck.connection.timeout=

# number of least loaded nodes to return, defaults to 10
coordinator.list.node.amount=
# maximum load on a node to be offered in the node list (vals 0.0-1.0), defaults
  to 0.9
coordinator.list.node.max-load=

# number of seconds since last successful healthcheck to proclaim node dead,
  defaults to 15
coordinator.node.unhealthy.timeout=

# number of seconds to cache list of least loaded nodes, defaults to 5
coordinator.node.cache.time=

# logging settings
# you can set logging level based on packages or individual classes
# available log levels are ERROR, WARN, INFO, DEBUG, TRACE
# packages example:
logging.level.org.springframework.data=
logging.level.cz.cvut.fel.hernaosc=
# class example:
logging.level.cz.cvut.fel.hernaosc.service.CoordinatorService=

# configure the port on which the application will run
server.port = 8090
```

B.2.2 Running the Node Application

The Node application can be started by running the assembled JAR file using `java -jar msgr-1.0.jar`. However, unlike the Node Coordinator, the Node Application requires the following software in order to be able to run:

- MySQL Server 5.5+
- ActiveMQ 5+

In order for the Firebase Cloud Messaging adapter to be able to connect to FCM and send messages, a JSON file containing your Firebase account credentials (instructions on how to obtain it can be found here: https://firebase.google.com/docs/admin/setup#add_firebase_to_your_app) has to be provided to the application.

The Node Application provides reasonable defaults for most configuration, however if needed, these can be overwritten using the `config/application.properties` file, using the following template:

```
# MySQL DB settings
spring.jpa.hibernate.ddl-auto=
spring.datasource.url=jdbc:mysql://
spring.datasource.username=
spring.datasource.password=

spring.jpa.properties.hibernate.enable_lazy_load_no_trans=true

# FCM adapter settings
msgr.adapter.fcm.projectId=
msgr.adapter.fcm.serviceAccountFilename=

# MSGR Node settings
# node ID to use when connecting to Coordinator. Defaults to random UUID
msgr.node.id=
# node address (including any port other than 80) that the Coordinator will use
#   to health check and coordinate traffic
msgr.node.address=
# Address (including any port other than 80) for the Node Coordinator
msgr.coordinator.address=

# number of seconds to cache list of least loaded nodes, defaults to 5
msgr.node.cache.time=

# number of users to include in a page when breaking down group, defaults to 50
msgr.group.user.page.size=

# ActiveMQ settings
spring.activemq.broker-url=
spring.activemq.user=
spring.activemq.password=
# maximum number of connections between Node and ActiveMQ. Max 500 sessions per
#   connection. Defaults to 1
msgr.activemq.max.connections=
# message time to live in ActiveMQ, in milliseconds. Default 108000ms (30 minutes)
msgr.activemq.message.ttl=
```

B.2.3 Running the MsgrChattr Android Application

For instructions on how to run the Android application, please refer to official Android documentation at <https://developer.android.com/studio/build/building-commandline>.

B.2.4 Running the Chattr Web Application

To run the Chattr web application, open the `messagingSystem/web/chattr/chattr.html` file in any supported browser (see Section [2.3.2 Non-functional Requirements](#)).

B.2.5 Running the PerformanceTester Application

The PerformanceTester application runs as a command line Java application. Trying to run the JAR file without any arguments gives the following output:

```
$ java -jar PerformanceTester-1.0-all.jar
Invalid arguments. Usage: MsgrPerf <cmd> <args>
Available commands:
For details on each command run MsgrPerf help <cmd>
testConnection
multiMessage
multiNodeMessage
communication
help
```

The `help` command shows which inputs are required for each command. Example output:

```
$ java -jar PerformanceTester-1.0-all.jar help multiMessage
Usage: multiMessage <url> <target> <numMessages>
Valid targets: 'device', 'user'
```

Appendix C

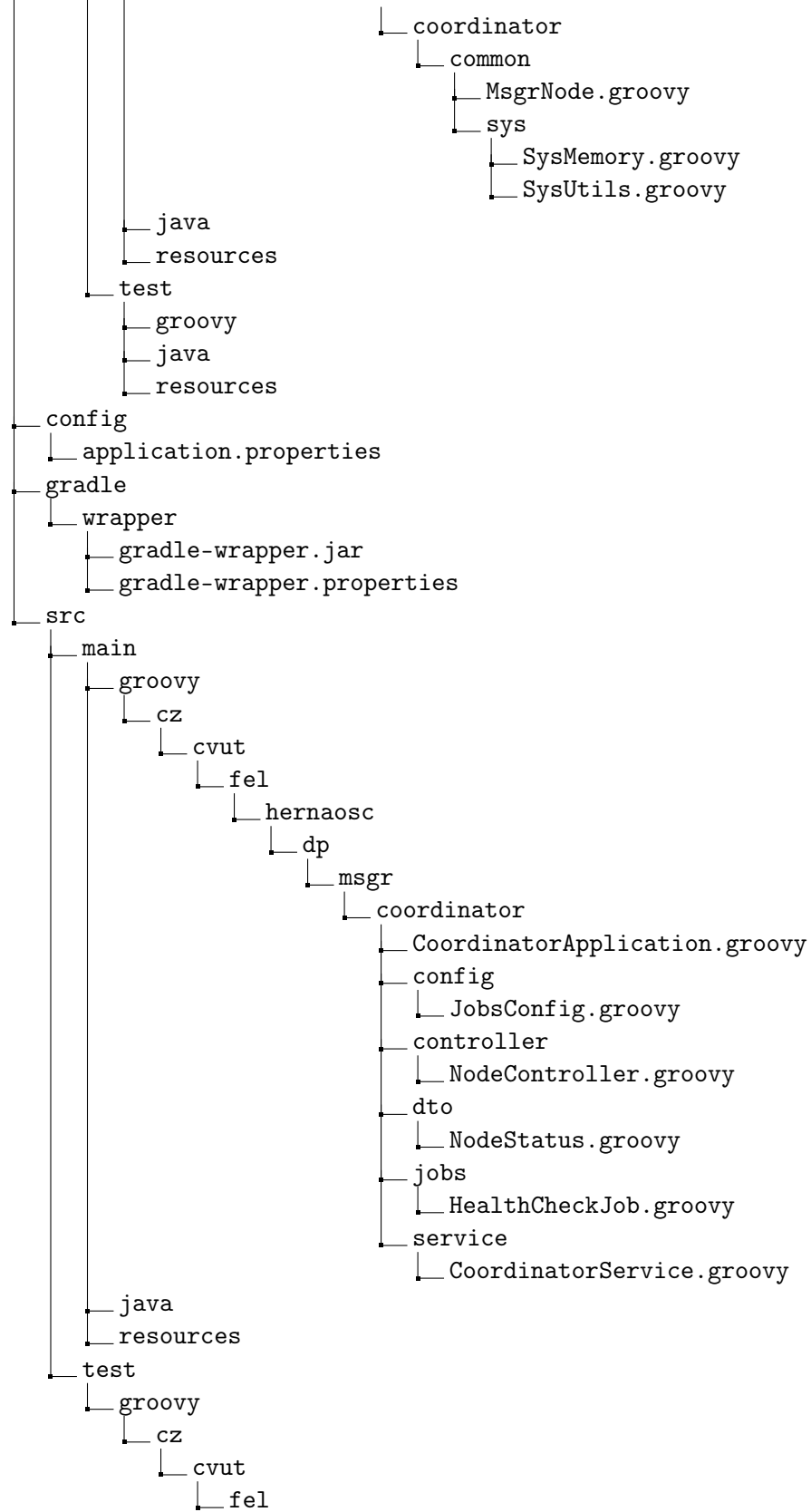
Contents of CD

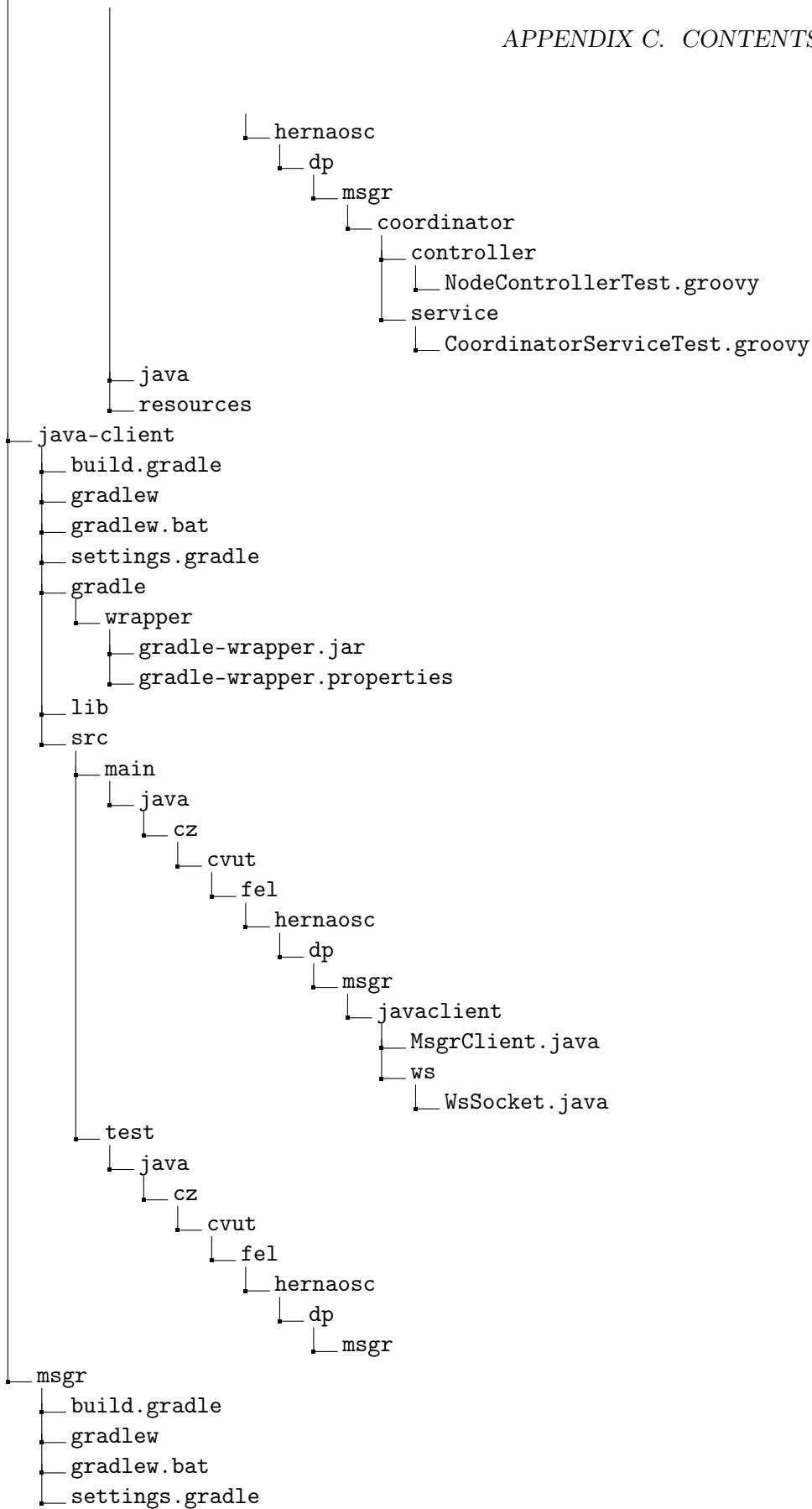
```
/mnt/c/Users/Osa-S/Documents/school/fel/ing/diplomka/dp-cd/  
├── Multi_platform_Scalable_Messaging_System.pdf  
├── doc  
│   ├── chattr-android.png  
│   ├── chattr-web.png  
│   ├── maven-gradle-speed.png  
│   ├── mobile-market.png  
│   ├── monitor-page.png  
│   ├── peak-conns.png  
│   └── diagrams  
│       ├── 20180426_133118.jpg  
│       ├── adapter-flow.asta  
│       ├── adapter-flow.png  
│       ├── basic-arch.asta  
│       ├── basic-arch.png  
│       ├── basic-arch_0.png  
│       ├── core-db-module-repo.png  
│       ├── core-db-module.png  
│       ├── core-mq-module.png  
│       ├── core-platform-module.png  
│       ├── example-situation.png  
│       ├── java-client-classes.asta  
│       ├── java-client-classes.png  
│       ├── layer-arch-detail.png  
│       ├── layer-arch.png  
│       ├── mq-fcm-device.png  
│       ├── mq-group.png  
│       ├── mq-ws-device.png  
│       ├── msg-flow.asta  
│       ├── msg-flow.png  
│       ├── msgr-project.png  
│       └── orm.png
```

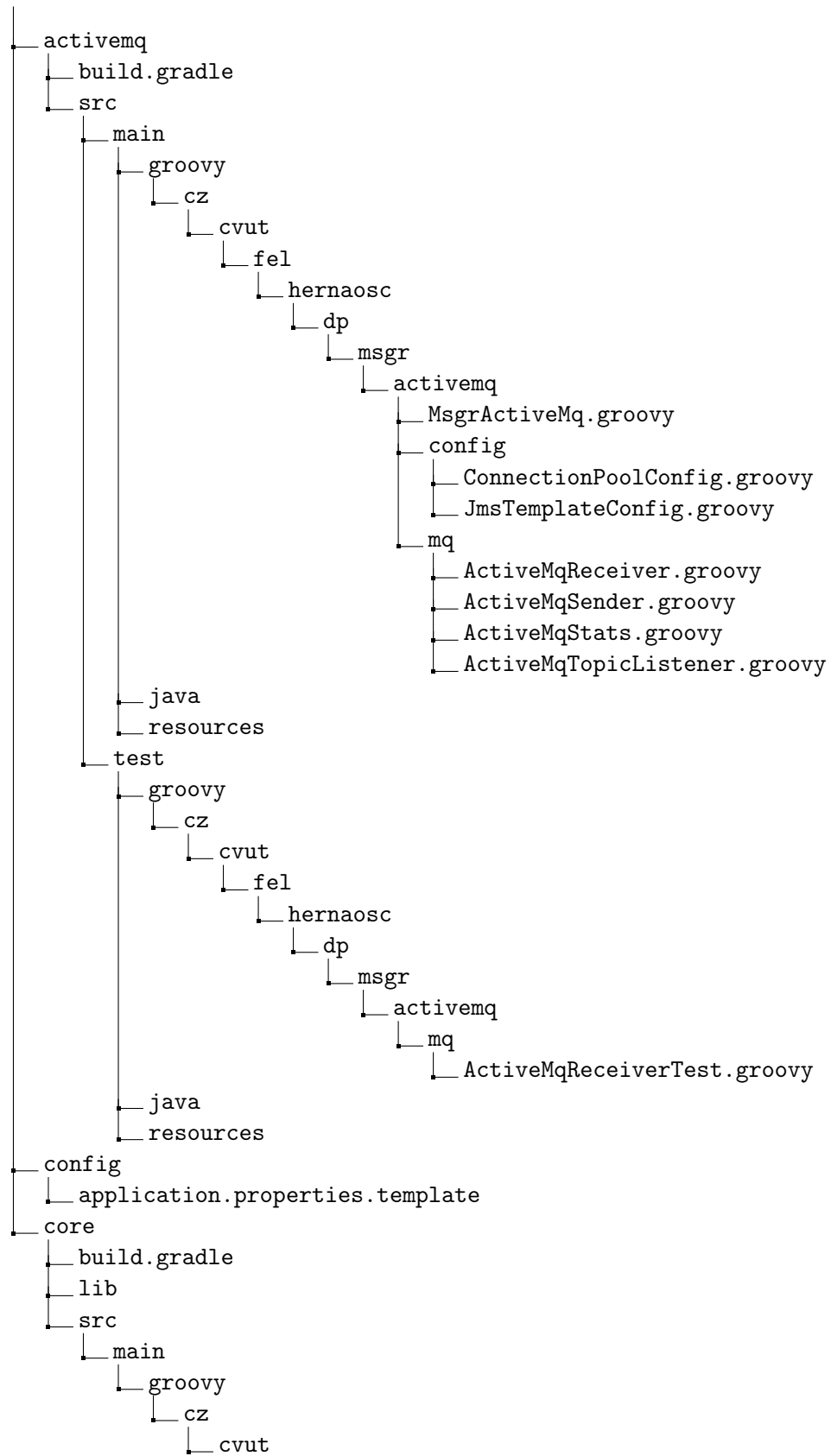
```

├── Realtime-communication-pattern.png
├── s-impl-arch.png
├── s-impl-comps.asta
├── s-impl-comps.png
├── scalability-architecture.png
├── test-deploy-dia.asta
├── test-deploy-dia.png
├── Traditional communication pattern.asta
├── Traditional-communication-pattern.png
├── img
│   ├── android-logo-transparent-background.png
│   ├── Apple-Logo-black-png-transparent.png
│   ├── commons-wiki-User_icon_BLACK-01.png
│   ├── firebase.png
│   ├── google_PNG19630.png
│   ├── KSDyuoBGRcmvqnL3ozKi_1024px-Apple_iOS_new.svg.png
│   ├── microsoft-windows-22.png
│   ├── microsoft_PNG14.png
│   └── windows_logos_PNG31.png
├── test
│   ├── test-perf1.png
│   ├── test-perf2.png
│   ├── test-perf3.png
│   ├── test-perf4.png
│   ├── test-perf5.png
│   ├── test-perf6.png
│   ├── test-perf7.png
│   └── test-perf8.png
├── messagingSystem
│   ├── perf-results.xlsx
│   └── coordinator
│       ├── build.gradle
│       ├── gradlew
│       ├── gradlew.bat
│       ├── settings.gradle
│       └── common
│           ├── build.gradle
│           └── src
│               ├── main
│               └── groovy
│                   ├── cz
│                   └── cvut
│                       ├── fel
│                       └── hernaosc
│                           ├── dp
│                           └── msgr

```





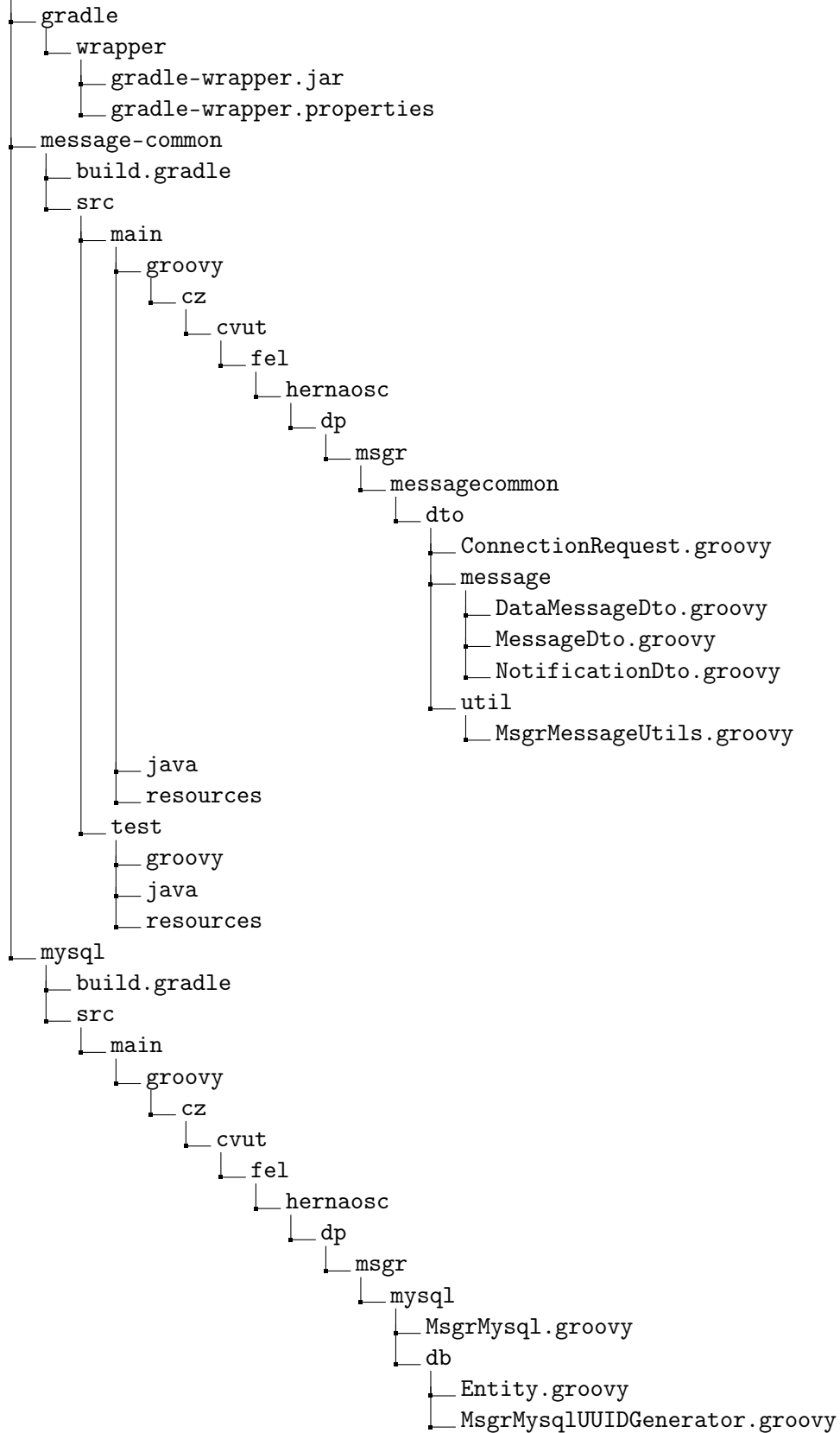


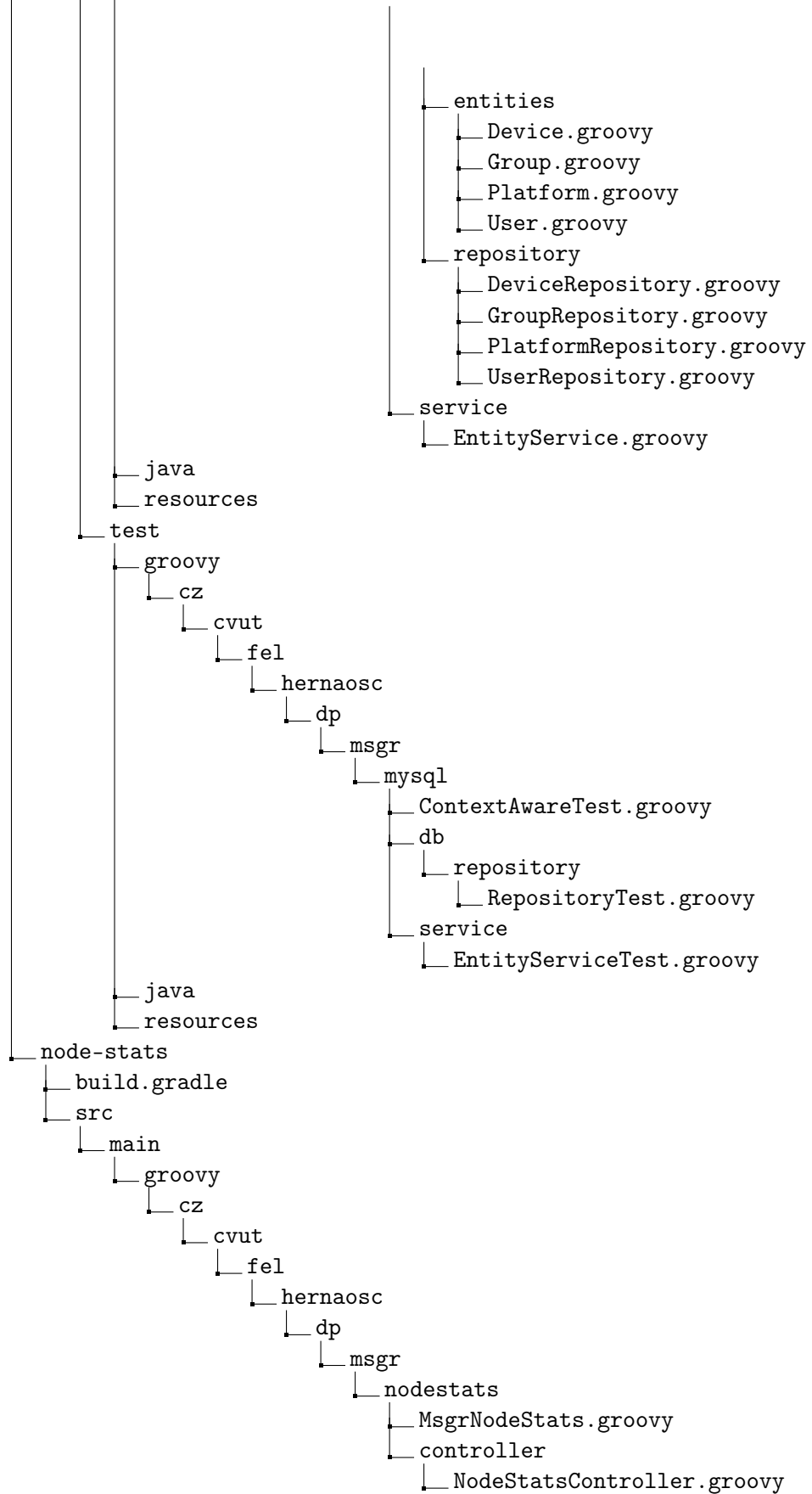
```

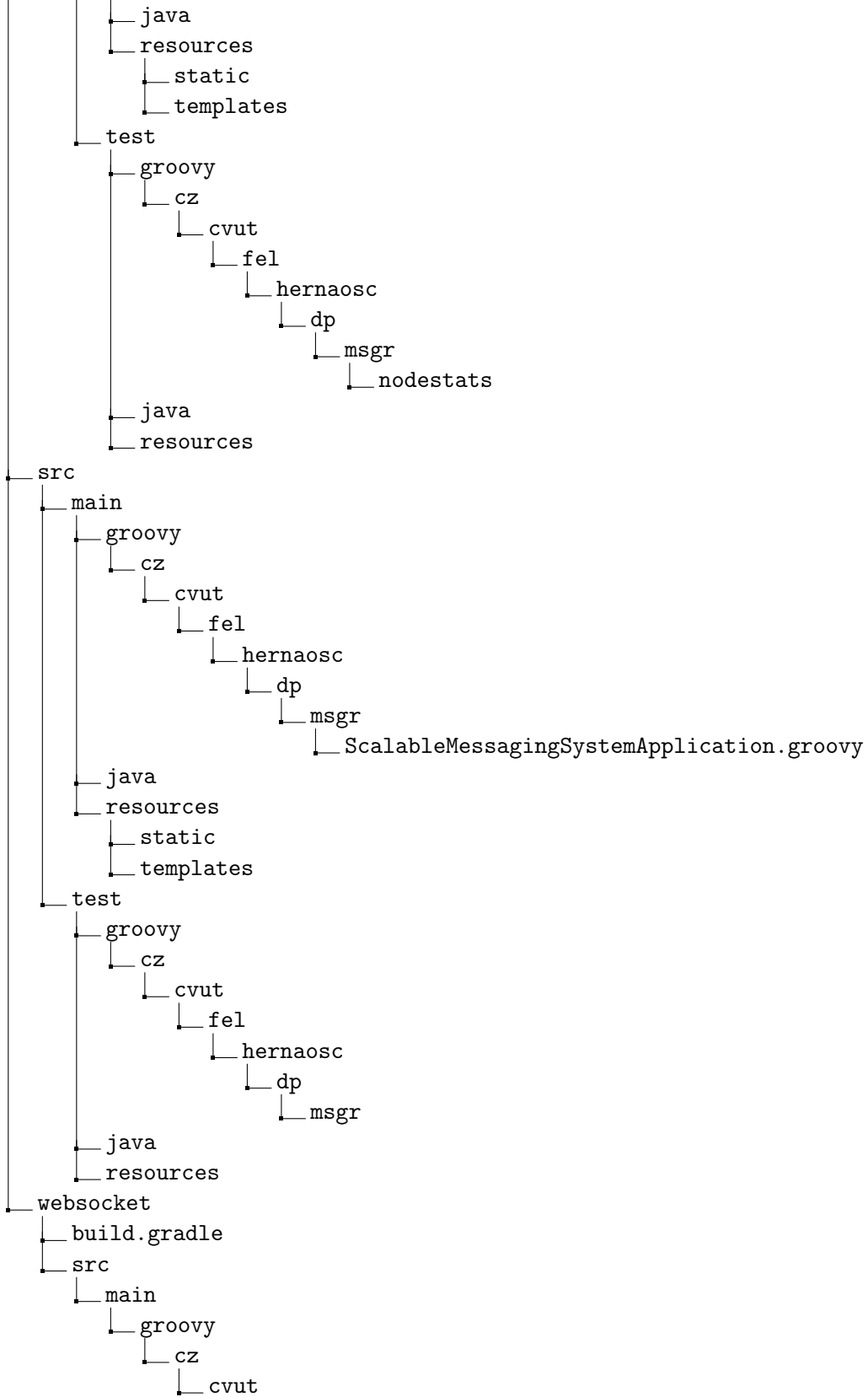
├─ fel
│   └─ hernaosc
│       └─ dp
│           └─ msgr
│               └─ core
│                   └─ CoordinatorConnector.groovy
│                   └─ MsgrCore.groovy
│                   └─ config
│                       └─ CorsConfig.groovy
│                   └─ controller
│                       └─ ConnectionController.groovy
│                       └─ HealthController.groovy
│                       └─ MessageController.groovy
│                   └─ db
│                       └─ IEntity.groovy
│                       └─ entities
│                           └─ IDevice.groovy
│                           └─ IGroup.groovy
│                           └─ IPlatform.groovy
│                           └─ IUser.groovy
│                       └─ repository
│                           └─ IBaseRepository.groovy
│                           └─ IDeviceRepository.groovy
│                           └─ IGroupRepository.groovy
│                           └─ IPlatformRepository.groovy
│                           └─ IUserRepository.groovy
│                   └─ dto
│                       └─ MqStatsDto.groovy
│                   └─ mq
│                       └─ IMqStats.groovy
│                       └─ IReceiver.groovy
│                       └─ ISender.groovy
│                   └─ platform
│                       └─ IPlatformAdapter.groovy
│                       └─ PlatformAdapter.groovy
│                   └─ queue
│                       └─ GroupProcessor.groovy
│                       └─ UserProcessor.groovy
│                   └─ service
│                       └─ AdapterService.groovy
│                       └─ IAdapterService.groovy
│                       └─ IEntityService.groovy
│                       └─ IMessagingService.groovy
│                       └─ MessagingService.groovy
│                   └─ util
│                       └─ MsgrUtils.groovy

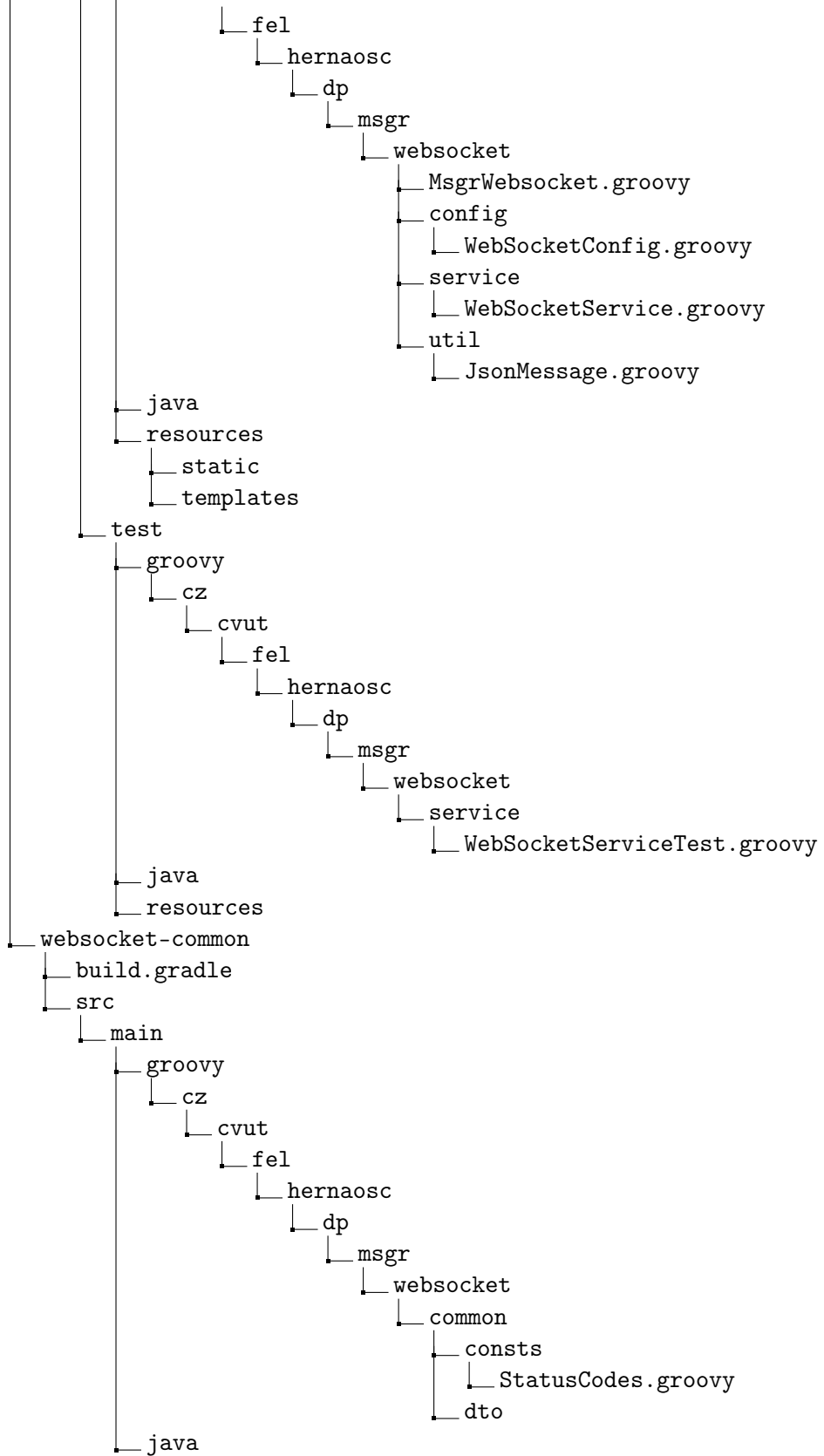
```

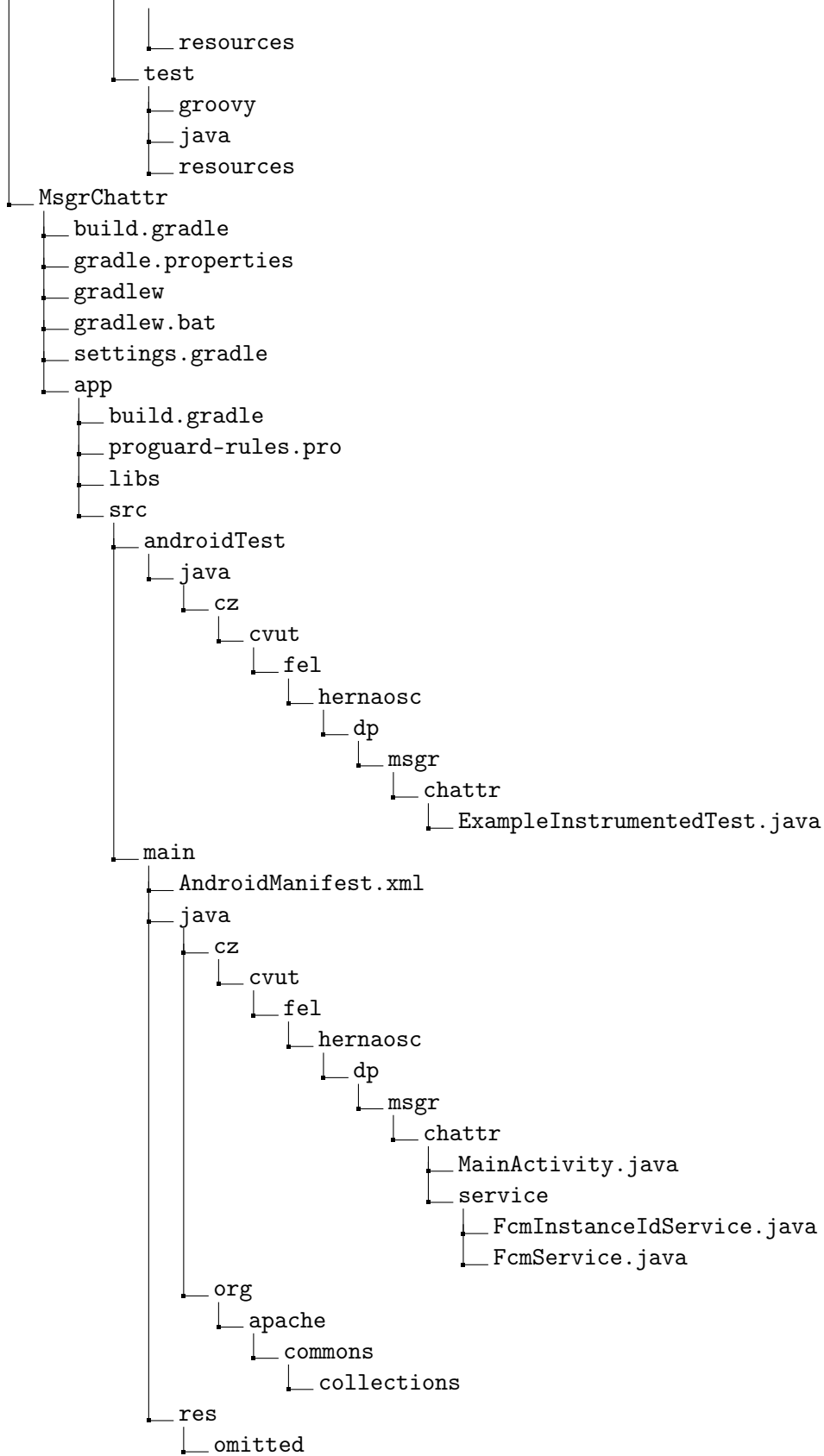


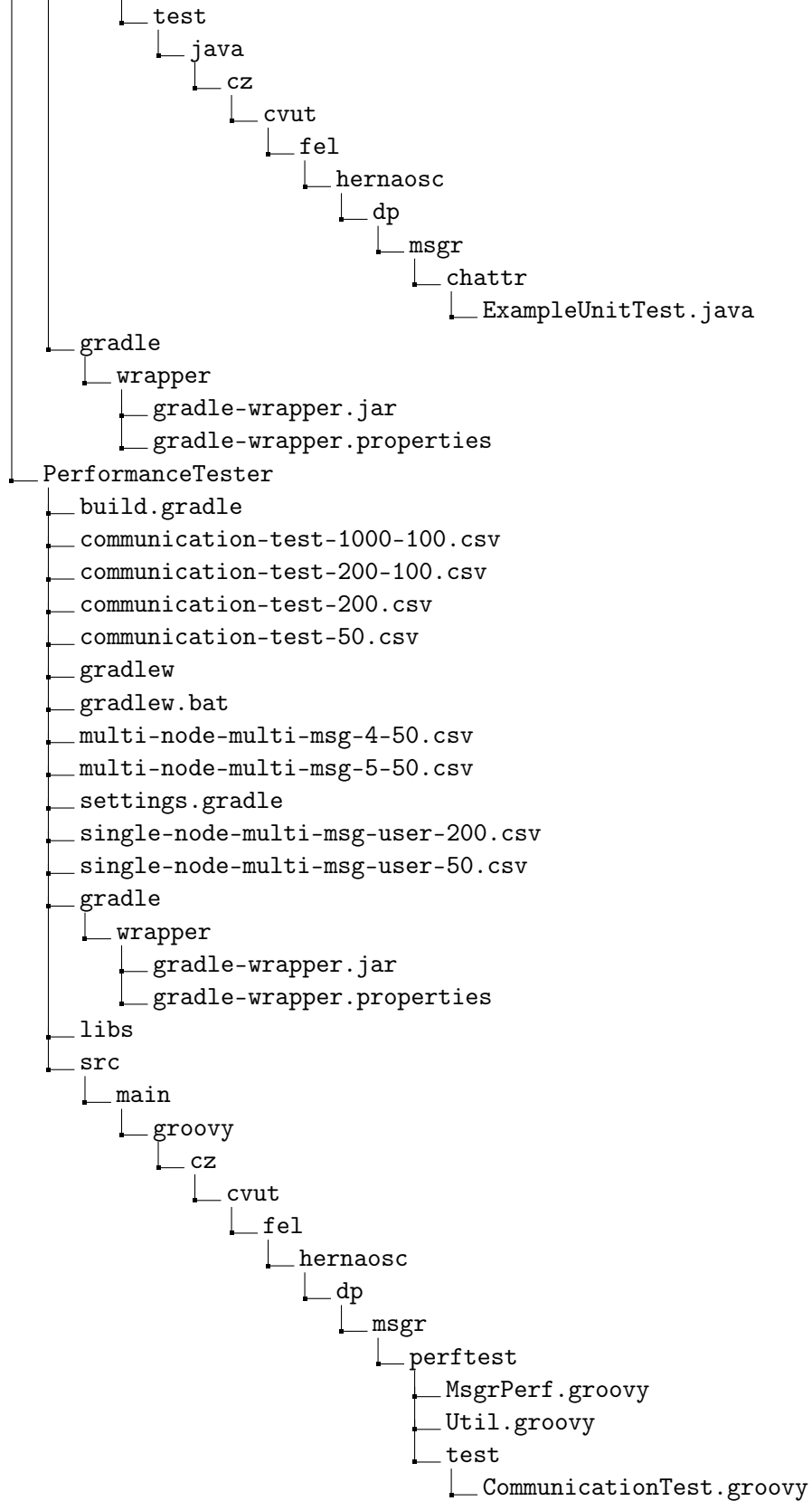












```

├── ConnectionTest.groovy
├── MultiNodeMultiMessageTest.groovy
├── SingeNodeMultiMessageTest.groovy
├── web
│   ├── chattr
│   │   └── chattr.html
│   ├── msgr-client-js
│   │   └── msgr-client.js
├── tex
│   ├── hernaosc_master_thesis.tex
│   ├── hyphen.tex
│   ├── reference.bib
│   ├── chapters
│   │   ├── 01_introduction.tex
│   │   ├── 02_analysis.tex
│   │   ├── 03_design.tex
│   │   ├── 04_implementation.tex
│   │   ├── 05_testing.tex
│   │   ├── 06_conclusion.tex
│   │   ├── AX_abbrev.tex
│   │   ├── AX_cd.tex
│   │   └── AX_user-guide.tex
│   ├── figures
│   │   ├── LogoCVUT.eps
│   │   ├── LogoCVUT.pdf
│   │   ├── 02_analysis
│   │   │   ├── example-situation.png
│   │   │   ├── maven-gradle-speed.png
│   │   │   ├── mobile-market.png
│   │   │   ├── peak-conns.png
│   │   │   ├── Realtime-communication-pattern.png
│   │   │   └── Traditional-communication-pattern.png
│   │   └── 03_design
│   │       ├── adapter-flow.png
│   │       ├── basic-arch.png
│   │       ├── core-db-module-repo.png
│   │       ├── core-db-module.png
│   │       ├── core-mq-module.png
│   │       ├── core-platform-module.png
│   │       ├── java-client-classes.png
│   │       ├── layer-arch-detail.png
│   │       ├── layer-arch.png
│   │       ├── mq-fcm-device.png
│   │       ├── mq-group.png
│   │       ├── mq-ws-device.png
│   │       └── msg-flow.png

```

```
├── orm.png
├── s-impl-arch.png
├── s-impl-comps.png
├── scalability-architecture.png
├── 04_implementation
│   ├── chattr-android.png
│   ├── chattr-web.png
│   ├── monitor-page.png
│   └── msgr-project.png
├── 05_testing
│   ├── test-deploy-dia.png
│   ├── test-perf1.png
│   ├── test-perf2.png
│   ├── test-perf3.png
│   ├── test-perf4.png
│   ├── test-perf5.png
│   ├── test-perf6.png
│   ├── test-perf7.png
│   └── test-perf8.png
├── misc
│   └── k336_thesis_macros.sty
```