FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

IPBRICK OS - Unified Communications Mobile App

Gonçalo Ferreira Duarte

FOR JURY EVALUATION

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Professor Miguel Monteiro

External Supervisor: Engenheiro Miguel Ramalhão

July 29, 2021

Resumo

O presente documento apresenta o projeto desenvolvido no âmbito da dissertação do Mestrado em Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto, desenvolvido em ambiente empresarial na IPBRICK S.A .

Os serviços da IPBRICK integram uma plataforma de comunicações comum para que os colaboradores de uma empresa possam comunicar entre si de forma rápida e intuitiva.

Nos últimos anos, os *smartphones* tornaram-se um aparelho essencial para a vida pessoal e profissional da maior parte dos cidadãos. Como tal, nos dias que correm, o sucesso de um serviço tecnológico pode depender da existência ou não de uma aplicação *mobile*. Atualmente, a IPBRICK tem uma aplicação publicada a ser distribuída aos seus clientes mas esta apesenta falhas que não permitem a sua utilização em ambiente empresarial.

O objetivo deste projeto é estudar e implementar uma nova solução mobile para o uso dos serviços de comunicação fornecidos pela IPBRICK.

Aplicações de comunicações em tempo real são provavelmente um dos tipos de aplicação mais desafiantes de desenvolver. Cada vez mais são impostas restrições no que toca a aplicações e serviços que correm no *background* de modo a salvaguardar a duração da bateria do dispositivo. Como é possível constatar ao longo documento, este foi um dos aspetos mais desafiantes do trabalho desenvolvido, que fez com que a solução proposta se fosse alterando ao longo do projeto.

Abstract

This document presents the work developed within the scope of the dissertation of the Master in Electrical and Computer Engineering of the Faculty of Engineering of the University of Porto, developed in a business environment at IPBRICK S.A .

Services provided by IPBRICK integrate a unified communications platform where collaborators from the same company can establish contact in a fast and intuitive fashion. In the last few years, smartphones have become an essential tool for the personal and professional lives of most citizens. Because of this, the success of a technological service can depend on the existence of a mobile solution, usually an app. As of today, IPBRICK has a published app that is being distributed to its clients but this app contains flaws that make it an unreliable work tool.

The main goal of this dissertation is to study and implement a new mobile solution for using the communications services provided by IPBRICK.

Real-time communications apps are probably one of the most challenging types of apps to develop. Restrictions when it comes to running apps and services in the background are increasingly being implemented to increase battery life. As demonstrated throughout the document, this was one of the most challenging aspects of the work developed which resulted in the proposed solution being altered during the semester.

Acknowledgements

Antes de mais agradeço à minha família e aos meus amigos pelo apoio ao longo destes 5 anos.

Quero também agradecer ao meu orientador , Professor Miguel Monteiro, docente na Faculdade de Engenharia da Universidade do Porto por ter demonstrado sempre disponibilidade para ajudar.

Agradeço à IPBRICK por me ter dado a oportunidade e confiança para desenvolver este projeto. Um obrigado ao meu orientador na empresa, Engenheiro Miguel Ramalhão pela ajuda e motivação. Agradeço ainda ao Engenheiro Hélder Santos por se ter mostrado sempre disponível na resolução de problemas mais técnicos ao longo do desenvolvimento do projeto.

Gonçalo Duarte

"Creativity is intelligence having fun."

Albert Einstein

Contents

List of Figures

List of Tables

Listings

Abbreviations and Symbols

API	Application programming interface
CSS	Cascading Style Sheets
HTML	HyperText Markup Language
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IP	Internet Protocol
IP PBX	Internet Protocol private branch exchange
MVVM	Model–View–ViewModel
OS	Operating System
REST	Representational State Transfer
SDK	Software Development Kit
SIP	Session Initiation Protocol
UCoIP	Unified Communications over IP
UI	User Interface
VoIP	Voice over IP
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

Chapter 1

Introduction

This chapter makes a brief introduction and contextualization to the problem addressed in this dissertation.

1.1 Context

Figure 1.1: Computer and phone app sharing

Communication between employees is essential for the success and growth of every organization. Bad communication can affect efficiency, processes, and every layer of a company. To keep all employees connected, companies can resort to unified communication platforms, a common place where collaborators can communicate in real and non-real time, strengthen relationships, and increase productivity.

Usually, when there is a need to communicate with someone, people tend to use the closest and most accessible device. Most of the time, this device is a mobile phone. This means it is very

important for unified communications platforms to integrate seamlessly its services with mobile phones.

This dissertation studies the implementation and adaptation of IPBRICK Unified Communications services to a reliable mobile phone solution.

1.2 IPBRICK

This master thesis was proposed by IPBRICK and was developed in a business environment. IPBRICK S.A is a company that specializes in solutions for businesses, with multiple collaborators and partners in Portugal and other parts of the world.

IPBRICK's main product is the IPBRICK.OS. IPBRICK.OS is a communications platform for companies, based on Linux Debian, that includes services such as Email and Groupware, Unified Communications, Document and process management, and Digital workplace. All these services are gathered and can be used in a social network, the CAFE social network.[3]

1.3 Motivation

As the title of the master thesis suggests, the work developed is related to the Unified Communication services. At the moment, all the Unified Communications services are hosted online on the CAFE social network and are easily accessed on a computer. The currently published CAFE mobile app has some problems related to performance and real-time communications. Because of this, a user needs to download more than one app in order to use the different provided services. This process will usually involve installing a soft-phone app for voice calls, a messaging client app for instant messaging, and use an email client for email. Besides the need to install multiple apps, this also means a challenging configuration of these apps which might not be that easy for the common IPBRICK user.

1.4 Goals

Knowing that using the IPBRICK services in a mobile phone is not a simple process for the common user, the main goal for this master thesis is to develop an Android and iOS mobile app to use UCoIP (Unified Communications over IP) services from IPBRICK.

The goals for this dissertation are:

- Study the different available solutions for developing mobile apps for iOS and Android;
- Study the different communication protocols used by IPBRICK services and how they can be implemented in a mobile app;
- Define the main requirements for the app to be developed;

- Choose the right solution for the app and develop an app that can later be published and distributed by the company;
- Test the app and validate its use in a business environment;

1.5 Document structure

This document presents 6 more chapters.

Chapter 2 tears down the communications protocols used by the IPBRICK services and analyzes the different available technologies for developing mobile apps. Chapter 3 displays the current problems in the IPBRICK environment and the motivation for its resolution. Chapters 4 and 5 relate to the two different phases of the developed work. Chapter 6 analyzes the results obtained in the end of the project and finally, chapter 7 makes a brief conclusion and points some future work.

Chapter 2

Communication Protocols and Mobile Development

This chapter describes the research made within the scope of mobile app development. It also takes a look at the communication protocols that are currently used in the IPBRICK communication stack.

2.1 Communication Protocols

This section takes a close look at the protocols used in IPBRICK UCoIP service so there is a better understanding when later implementing them in the app. In the next chapter, it is described how these protocols are used in the IPBRICK environment.

2.1.1 SIP - Session Initiation Protocol

SIP (Session Initiation Protocol) is a request-response-based protocol that is used to initiate and manage communication sessions between users. This protocol can have several uses such as VoIP (Voice over IP), presence sending, or Instant Messaging. It is used by IPBRICK UCoIP for managing VoIP calls. It is an open-source protocol and was standardized in 1999 by the IETF (Internet Engineering Task Force). [5][6]

This protocol can be split into two groups, for a better understanding: the SIP user agents and the SIP server.

A user agent is any device that registers itself in the SIP server and is used for starting and receiving calls. By registering itself in the server, a user agent is telling the server it is available for receiving calls.

The SIP server receives requests from user agents and redirects them to the correct receiver. Some examples of SIP requests are :

- Invite- request to invite a user to a call.
- Ack- confirmation that the other user has responded to a request.

- Update- used to modify a session.
- Cancel- cancel pending requests.
- Bye- end dialogues and calls.

Figure 2.1 shows the temporal events involved in a SIP call.

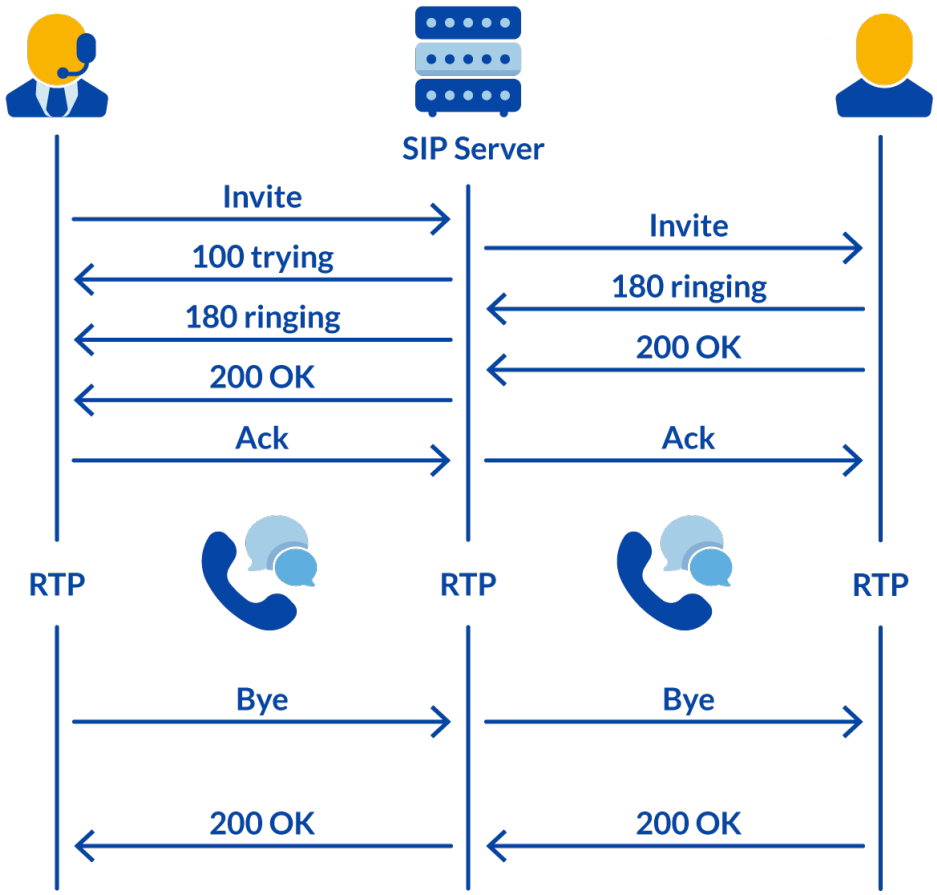

Figure 2.1: Temporal events involved in a SIP call [1]

2.1.2 XMPP - Extensible Messaging and Presence Protocol

XMPP (Extensible Messaging and Presence Protocol) is a real-time messaging protocol based on XML (Extensible Markup Language). At the time of its creation, the objective was to create an open, decentralized, and secure standardized protocol for instant messaging. It was first called Jabber but, in 2004, it was formalized by the IETF with the name Extensible Messaging and Presence Protocol (XMPP). [7] [8]

Like the SIP protocol, XMPP is divided into two main entities: the users and the server. A user sends messages to the server that redirects them to the correct user.

The XML messages exchanged in this protocol are called Stanzas and they are the main unit of messaging for this protocol:

Listing 2.1: Stanza message example

```
<message
    id="1234"
    from="gduarte@app.ucoip.pt"
    to="mramalhao@app.ucoip.pt">
    <body> Hello from XMPP </body>
</message>
```

This protocol follows a modular architecture that makes it possible to enable certain extensions based on the needs of the application. Although it is possible to develop custom extensions, there are several official extensions commonly used in most applications that are defined in various core specifications. [9]

Some examples of these core specifications are:

- XEP-0045 - Multi-User Chat;
- XEP-0313 - Message Archive Management;
- XEP-0144 - Roster Item Exchange;

2.1.3 Jitsi Meet

Jitsi Meet is an open-source JavaScript application used by the IPBRICK UCoIP services for video conferencing. It offers many features such as audio and video encrypted calls, screen sharing, remote control and can have up to 100 participants in one call. It offers its own interface and can be used in web browsers or with a mobile app. It also offers software that can be installed on Windows, Linux, and Mac OS (Operating System) computers. [10]

A unique link is attributed to every Jitsi Meet video room. By using this link, a user can connect to the meeting on any supported platform.

Being open-source, Jitsi Meet can be used as a service. This means it can be integrated and used as intended by developers. The IPBRICK UCoIP service takes care of creating and managing Jitsi Meet Rooms using them for scheduled video meetings and real-time video calls.

2.2 Mobile Development - Native vs Hybrid Apps

In the past, developing a mobile app for both Android and iOS would mean one thing: have two different apps, coded in two different native programming languages, one for each platform. In the current days, although this is still the preferred approach for most development teams, we have hybrid programming frameworks that allow us to maintain the same code base for the two platforms.

Both options have their advantages and disadvantages. When it comes to performance, a native app will, most of the time, have better performance since it runs on native code. For hybrid programming frameworks, the development will most of the time be easier, cheaper, faster and it is possible to port the app between different platforms. As the phone industry evolves, phones continue to improve their hardware and the performance issues between native and hybrid apps will become smaller.

This section analyzes the different technologies and solutions currently on the market for developing mobile apps.

2.2.1 Android App Development

Android is an Operating System (OS) based on Linux that was developed by Open Handset Alliance, a group of development companies where Google is the main contributor. This operating system is open-source and is mainly used in touchscreen mobile phones. Applications for this operating system are developed using the Android SDK (Software Development Kit) .[11]

The Android SDK is a set of development and debugging tools for Android app development. The Android SDK is bundled into Android Studio, the official IDE (Integrated Development Environment) for Android development. This IDE is maintained by Google and it is based on IntelliJ IDEA.[12] [13]

When using this SDK, we can choose to program our app in two different languages, Java or Kotlin. Optionally, C and C++ can also be mixed with these two languages.

Java is a class-based, object-oriented programming language developed by Oracle Corporation. Java was the preferred language for Android development for many years. In 2017 Google announced Kotlin as the new official language for Android App development.

Like Java, Kotlin is a class-based, object-oriented programming language and it was developed to introduce modern features related to mobile development into Java and to reduce its syntax. It is a concise programming language and is 100% compatible with Java. It was developed in 2010 by Jetbrains and it is now in version 1.5.10.[14][15][16][17]

2.2.2 iOS App Development

Simply put, iOS, previously iPhone OS, is the operating system used on Apple's mobile phones, the iPhone. iPod and iPad devices from Apple also run this OS. Unlike Android, iOS only runs on Apple devices and is not open-source. This operating system was announced in January 2007

along with the first iPhone. In the beginning, third-party apps were not supported on iOS but, in March 2008, Apple announced the iPhone SDK, now called iOS SDK.

iOS SDK is a set of tools developed by Apple Inc. that make app development for Apple's iOS operating system possible. This SDK is available for free for Mac personal computers. While you can program Android apps with the Android SDK on both Windows and Mac computers, the iPhone SDK is only available for Mac computers.

This SDK is usually used alongside Xcode, the official IDE from Apple, and apps are developed using officially supported programming languages. These include Swift and Objective-C.[18] [19] [20]

Until 2014, Objective-C was the official programming language for iPhone devices. Objective-C is an object-oriented programming language based on two distinct programming languages: C and Smalltalk. It was developed in the 1980s and was licensed by a company called NeXT Computer Inc. This company, founded by Steve Jobs, later developed the NeXTStep framework. This framework was the foundation for the first operating systems launched by Apple, the macOS, and iOS.

In 2014 Apple released Swift, its new official programming language for app development. Swift is very similar to Objective-C but it introduced many essential features that Objective-C lacks being the most important one bug safety. Besides protecting apps from programming mistakes, Swift is also faster and has better performance. It is currently in version 5.4.1 .

2.2.3 Hybrid Development Frameworks

In the last years, a set of new frameworks have changed the way we can develop mobile applications. Using these technologies, we can have the same base code for an app for both Android and iOS. Although there are more, this section will take a close look at the two most used and innovating hybrid frameworks in the market right now. These are React Native and Flutter.

React Native and Flutter convert the code to a near-native app which offers significantly better performance than older frameworks like Cordova and Ionic that use a WebView app approach. This makes them good candidates for the app to be developed.

2.2.3.1 React Native

React Native is an open-source mobile application framework developed by Facebook, Inc used to develop applications for Android,iOS, Android TV, macOS, tvOS, Web, and Windows by allowing developers to use the React framework with native capabilities.

React Native is a JavaScript framework that uses JSX to render components. JSX is a mix of HTML (HyperText Markup Language) and JavaScript on the same file that allows the developer to maintain the functional design and logic in the same file. Unlike the React framework, React Native does not use the typical HTML tags but rather tags related to mobile components, for

example, View, Flatlist, ScrollView, etc. Just like it does not use common HTML tags, it also does not use the common CSS (Cascading Style Sheets) file and proprieties. Instead, it uses its custom style pattern that, in the end, are just CSS proprieties wrote in CamelCase. [21]

React and therefore, React Native is developed with component hierarchy in mind. This means that an app is separated into pure components that are nested in each other and that are reusable in any part of the app. With this in mind, it is also possible to install extra packages to complement your app needs. There are thousands of extra packages online for React Native that are usually installed using a package manager such as npm or yarn [22] [23]. Figure 2.2 shows an example of a React Native component.

Figure 2.2: React Native Component example

React Native has a very strong online community and it is one of the most advanced and used hybrid framework for mobile development. Famous apps developed fully or partially in React Native include Facebook, Instagram, Tesla and Uber eats. Just this small list shows that the framework is mature enough to develop big-scale mobile apps.

2.2.3.2 Flutter

Flutter is an open-source UI (User Interface) toolkit developed by Google. This framework uses the Dart programming language to develop apps in Android, iOS, Windows, Mac, Linux, and Web. Just like React Native, Flutter uses a component-oriented architecture. In this framework, each component is called a widget and Flutter already offers some base components from scratch [24]. Figure 2.3 shows an example of a Flutter widget.

```
@override
Widget build(BuildContext context) {
  return Scaffold(
   ...
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Align(
            alignment: Alignment.centerLeft,
            child: Padding(
              padding: const EdgeInsets.only(left: 20.0),
              child: Text(
                'This is a Flutter Widget',
                style: TextStyle(fontSize: 20),
              ),
            ),
          ),
          SizedBox(
              width: 350,
              child: new RaisedButton(
                onPressed: doNothing,
                color: Colors.blue,
                padding: EdgeInsets.all(0),
                child: Text(
                  "I AM A BUTTON",
                  style: TextStyle(color: Colors.white),
                ),
              ))
        ],
      ),
    ),
  );
}
```

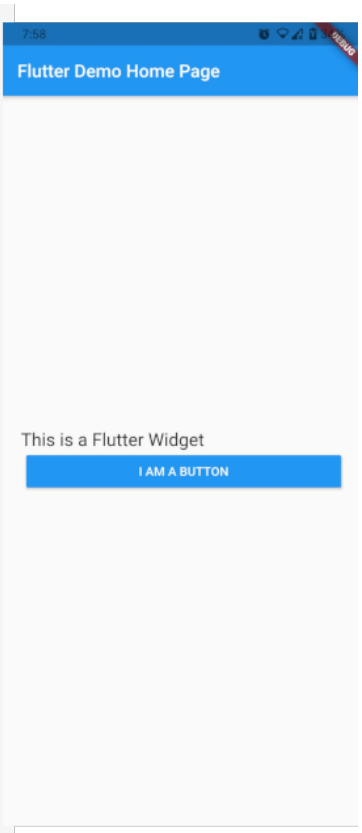

Figure 2.3: Flutter Widget example

Flutter is highly customizable allowing developers to build beautiful interfaces and is a fast development framework. Just like React Native, it is possible to add community packages to a Flutter app. This will usually be done using the pub.dev platform to search and install new packages.[25]

In these last years, Flutter has been gaining popularity among developers and it's thought to take React Native's place in the hybrid programming world. Some apps that use Flutter are eBay, The New York Times, and Google Assistant.

2.2.4 Native vs Hybrid: The pros and cons

When choosing between a native or hybrid solution there are some aspects to be analyzed:

- Operating system where the app should run.
- The type of app.
- Time and budget.

This subsection analyzes the pros and cons of native and hybrid development.

Advantages of hybrid mobile apps:

- **One code base for every platform**

  This is the biggest advantage of hybrid apps. A development team only has to write the code once and can run the app on any platform. There is no need to develop a different app for iOS and Android. Any small difference between the Android and iOS code can be conditionally implemented in the same app.

- **Development, maintenance and scalability efficiency**

  Using hybrid technologies, there is no need to hire more than one team of developers and spend lots of time developing several different apps. A small team can easily implement a production-grade mobile app in a short period of time. Since there is only one app, implementing new features becomes easier since only one app needs to be upgraded (Figure 2.4).

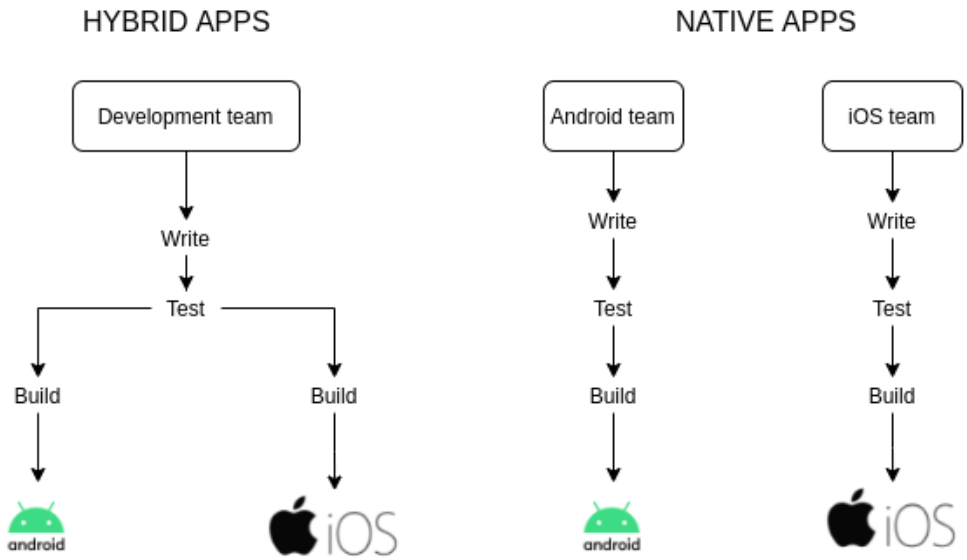

Figure 2.4: Hybrid vs Native - development team distribution example

- **Cost-effectiveness**

  Because of the previous point, it is cheaper to hire a small team of hybrid developers instead of maintaining multiple native teams. Besides this, hybrid frameworks have a smaller learning curve which makes it easy to migrate developers from other areas.

Disadvantages of hybrid mobile apps:

- **Performance**

  In general, complex apps with many features will run slower in hybrid implementations. Since hybrid frameworks need to convert the code to an app with native functionalities, the performance will depend on the phone hardware.

- **Limited accessibility to native components and device functionalities**

  This point will depend entirely on the used framework but some native functionalities such as camera, microphone and other sensors usage might not be available on some frameworks.

Advantages of native mobile apps:

- **Performance**

  Being written for a specific platform, the app will run smoother with a better performance since no conversion needs to be made.

- **Longer release cycles**

  Hybrid frameworks are a relatively new technology. The native development has been around for some time now and offers a more secure, well-tested, and reliable solution.

- **Access to hardware features**

  Unlike hybrid frameworks, there are no limitations when it comes to using device functionalities.

Disadvantages of native mobile apps:

- **Development cost and time**

  It takes longer and is more expensive to have a simultaneously iOS and Android app in development. Also, adding new features is more time-consuming since every new feature needs to be written, tested, and built twice.

- **Learning curve**

  For newcomers, native app development has a bigger and more complex learning curve comparing to hybrid frameworks.

Chapter 3

The IPBRICK environment and current drawbacks

This chapter describes the problem presented and the motivation for its resolution.

3.1 The IPBRICK environment

Each IPBRICK client (company or organization) has its own dedicated IPBRICK server. Each company has its needs and, therefore, the services provided for one client might not be the exact same for another.

Services provided by IPBRICK are divided into several applications [2] (Figure 3.1) :

- **IPBRICK.UCoIP:** Unified Communications over IP. These include:
  - A XMPP server using the ejabberd framework.
  - A SIP IP PBX (Internet Protocol private branch exchange) using the Asterisk framework.
  - Video chat rooms using the Jitsi Meet framework.
- **IPBRICK.CAFE:** a social network for the company that is also integrated with all the Unified Communications from IPBRICK.
- **iPortalDoc:** integrated solution for document uploading and management;
- **IPBRICK.Mail and Groupware:** email service with groupware tools (calendar, notes, tasks)

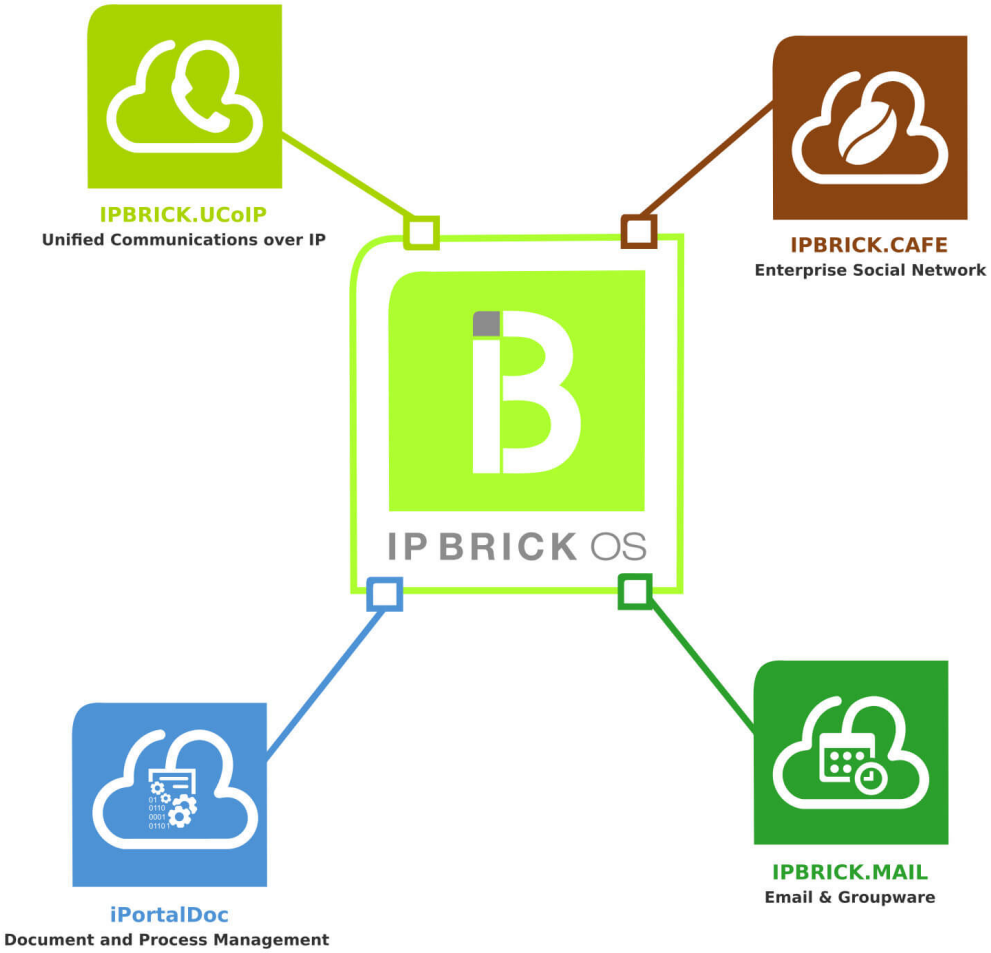

Figure 3.1: The IPBRICK environment [2]

3.2 UCoIP and CAFE

Although the UCoIP service is the one responsible for establishing communications between users, most of the time, users will use the CAFE network to establish contact with each other. Besides being a fully capable social network for the organization, the CAFE network integrates the UCoIP services and makes it possible to make and receive calls, send messages, join and create video chat rooms and use email. On the right side of figure 3.2, it is possible to see a list of all the users of the same organization and how to interact with them.

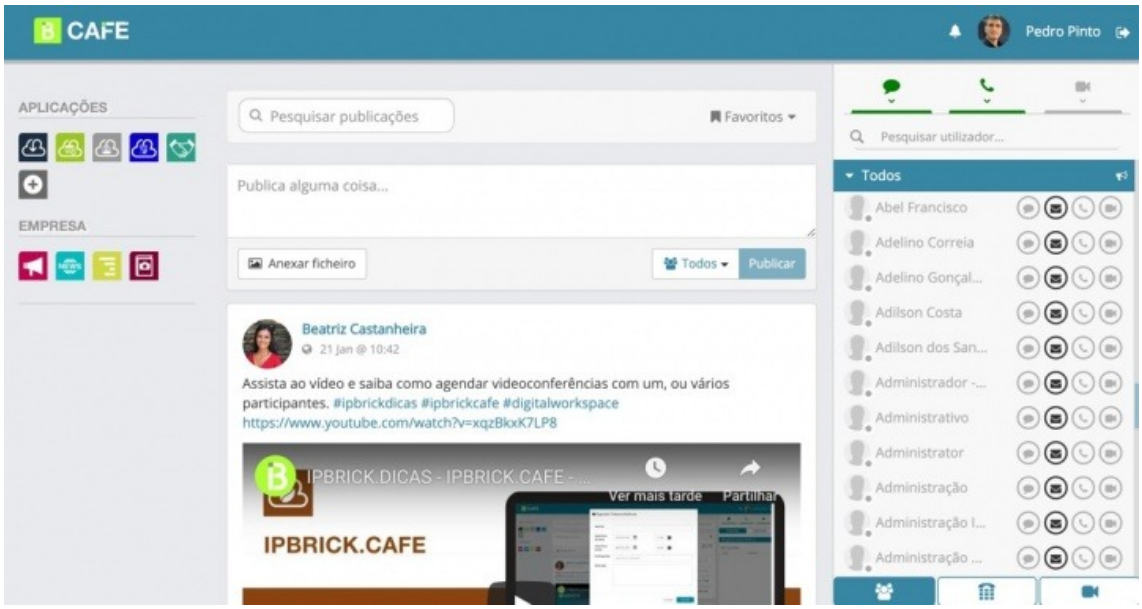

Figure 3.2: The IPBRICK CAFE social network [3]

For making voice calls, CAFE uses the SIP protocol already covered in 2.1.1. As long as a user is registered in the SIP server, it can receive SIP calls. In some cases, it is also possible to make calls to external numbers (Figure 3.3).

For sending and receiving text chats, CAFE uses the XMPP protocol 2.1.2 (Figure 3.4).

For the video calls, CAFE uses the Jitsi Meet framework 2.1.3. The Jitsi Meet framework only takes care of the media sessions (Figure 3.5). These video calls still need to be signaled. For this, CAFE uses the XMPP connection for signaling incoming video calls using a custom Stanza. This Stanza includes a URL containing the Jitsi Meet (Listing 3.1).

Listing 3.1: Custom Video Call Stanza

```
<custom
    type="ucoip-webrtc"
    email="hsantos@app.ucoip.pt" url="https://ucoip.app.ucoip.pt/rtc/?
        tk=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1N..."/>
```

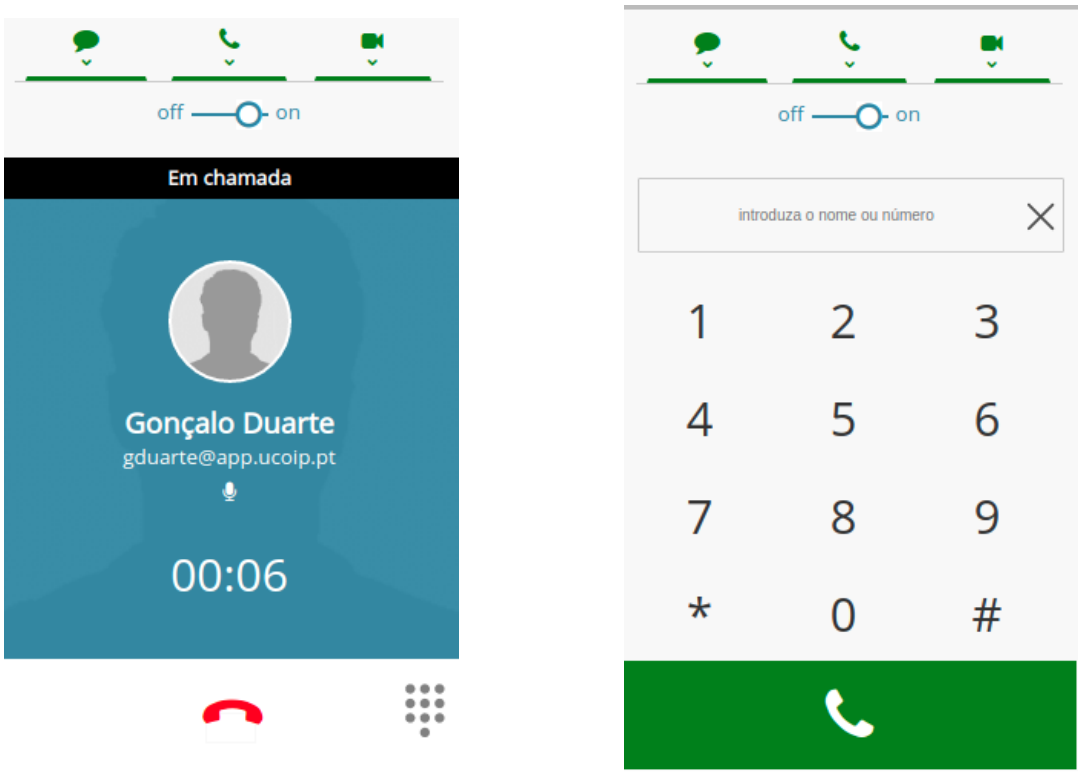

Figure 3.3: Voice calls using SIP

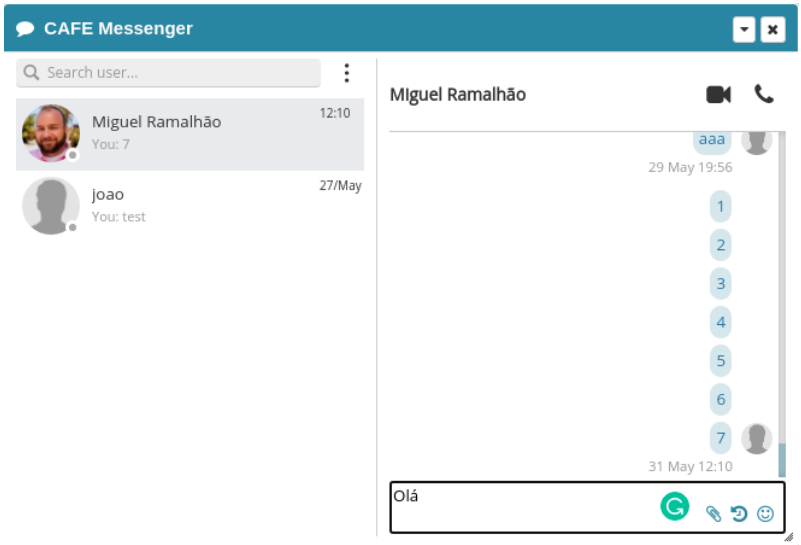

Figure 3.4: CAFE Messenger

Figure 3.5: Jitsi Meet Conference

3.3 Problem Statement, current solutions, and drawbacks

At the time of writing, IPBRICK does not provide a fully working solution for using CAFE and its UCoIP services in a mobile phone. There is currently a CAFE mobile app on the market but it has some problems. The app consists of a simple WebView that runs the CAFE social network website. A WebView is similar to a browser window that is limited to certain domains (Figure 3.6). Being a WebView, the performance is not the best and, as soon as the app is closed, it is impossible to receive any type of communication.

Since the CAFE app does not guarantee communications persistence, there is another solution users usually resort to. Being SIP and XMPP open protocols that are not owned by IPBRICK, it is possible to connect them to third-party apps (Figure 3.7).

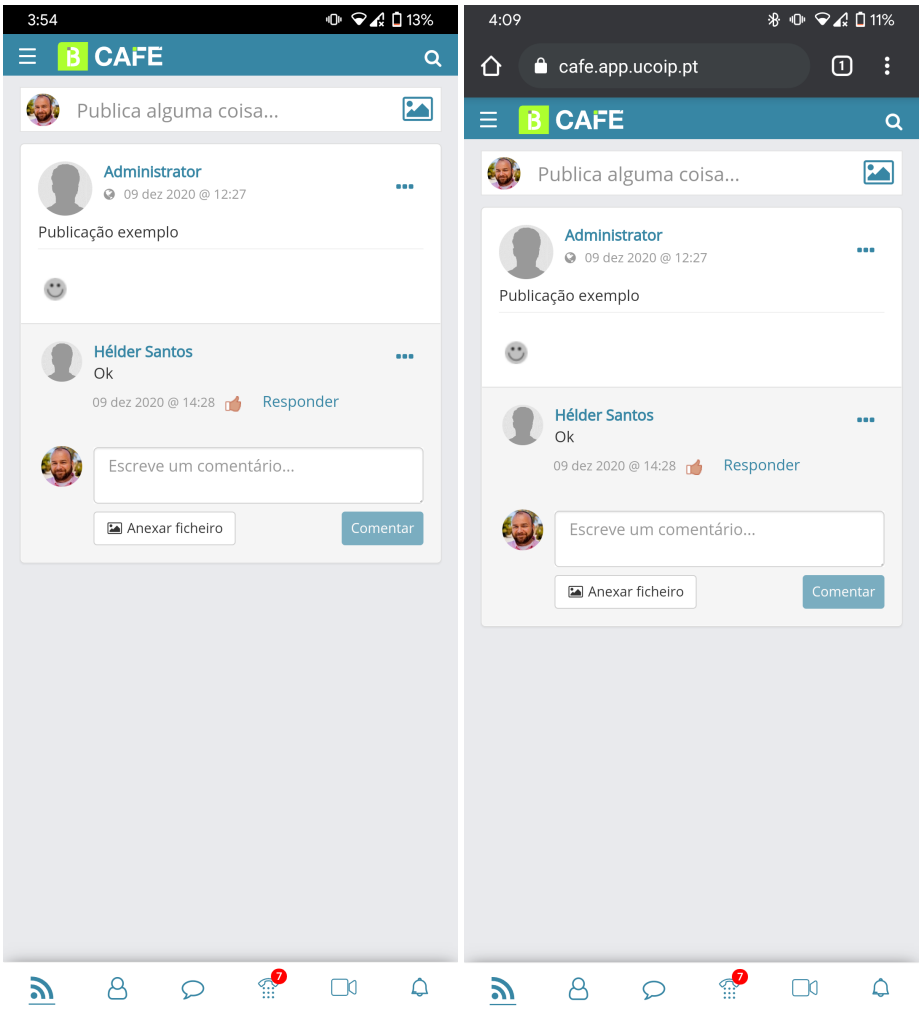

Figure 3.6: Current CAFE Mobile App vs CAFE in a browser

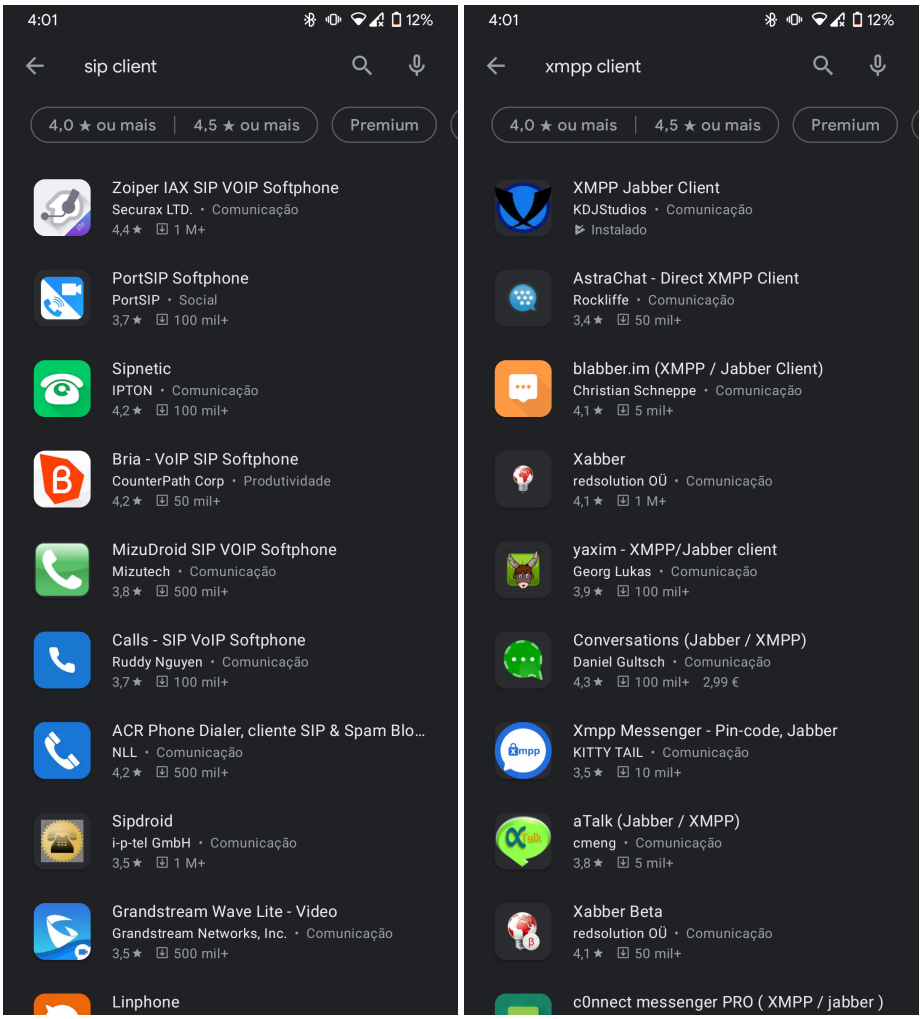

Figure 3.7: Third-party app solutions

There are several problems that come with third-party apps:

- These apps usually need an initial configuration that might not be the easiest for the common user.
- They do not allow to take full advantage of the UCoIP such as real-time Jitsi video calls.
- Most of the apps still cannot guarantee communications persistence at all times.

3.4 Goals

Facing the problems in the current solution, the main goals of the app developed are:

- Develop a new solution where IPBRICK clients can use the IPBRICK UCoIP services in a fast, lightweight, and modern mobile app;

- The app must have the least configuration possible and have acceptable performance on most phones;

- The app must be developed or prepared for Android and iOS platforms.

3.5 Solution Requirements

Based on the main goals described above, these are the main requirements of the app:

3.5.1 Functional Requirements

- Display contacts from the CAFE social network;

- The app developed must establish a connection to both SIP and XMPP IPBRICK servers and allow a user to send and receive chat messages, SIP voice calls, and Jitsi video calls;

- The app must also be able to display local contacts stored in the phone so the app can use them to place voice calls;

- As long as the phone has an available internet connection, the phone must receive any incoming communications even when the app is in the background or killed state. As the development of the app evolved, this showed to be the most challenging requirement.

3.5.2 Technological Requirements

- The app must have low resources consumption;

- The app must be easily maintainable. This means the app structure and code must be well organized and be easily read by other developers;

- The app must be easily scalable. This means the app must be ready to implement new functionalities in the future;

- The app must be developed following the recommended technologies and approaches as much and possible.

3.6 Proposed solution

To choose the right programming language for the app, advantages, disadvantages and compatibility with the communication protocols must be analyzed.

Native Android Development

- **SIP Protocol:** the Android SDK includes the SipManager. A Native module to handle and manage SIP calls;
- **XMPP Protocol:** Smack Library: An Android implementation for using the XMPP protocol;
- **Jitsi Meet:** Jitsi Meet Android SDK;
- **Advantages:** Performance in comparison to hybrid frameworks; Life time support; Online support and community;
- **Disadvantages:** Development time; Two code bases, one for each platform.

Native iOS Development

- **SIP Protocol:** Several available SIP SDKs such as vaxvoip, sofia-sip, exosip;
- **XMPP Protocol:** robbiehanson XMPPFramework, FluuxIO XMPP;
- **Jitsi Meet:** Jitsi Meet iOS SDK;
- **Advantages:** Performance in comparison to hybrid frameworks; Life time support; Online support and community;
- **Disadvantages:** Development time; Two code bases, one for each platform.

React Native

- **SIP Protocol:** Community React-native-sip module;
- **XMPP Protocol:** StanzaJS library;
- **Jitsi Meet:** React Native Jitsi Meet module;
- **Advantages:** Very strong online community; Faster development; Same code for both platforms; Mature framework; Already used in major apps;
- **Disadvantages:** Lower performance in comparison to native development; Supported by Facebook, cannot guarantee life time support.

Flutter

- **SIP Protocol:** sip_ua library;
- **XMPP Protocol:** xmpp_stone library;
- **Jitsi Meet:** Jitsi Meet library;

- **Advantages:** Growing framework; Faster development; Same code for both platforms; Developed by Google;
- **Disadvantages:** Lower performance in comparison to native development; Newer to the market; Smaller community.

As seen above, any of the options would, in theory, work and meet the main requirements of the app. In order to facilitate the development of the app, the first approach was trying to implement the app in React Native. Comparing with native development, it speeds up the development and the learning curve is also shorter. Comparing to Flutter, it is a more mature framework and probably more stable. Besides that, React Native has a very strong online community which makes it easier when it comes to solving bugs. The decision was not made final as we could encounter problems that could result in changing the framework.

Figure 3.8 shows an initial mockup of the app to be developed. This includes the following screens:

- Inbox Screen - Landing page of the app that displays the last messages of the user;
- Dialer Screen - A screen to input a number to place a SIP call;
- Contacts Screen - A screen displaying contacts from the CAFE social network and contacts saved on the user's phone;
- Private Message Screen - A screen to directly chat with a specified user;
- Call Screen - from the private message screen or from the dialer screen, a user can start a SIP or Video call. This screen is where the calls will take place.

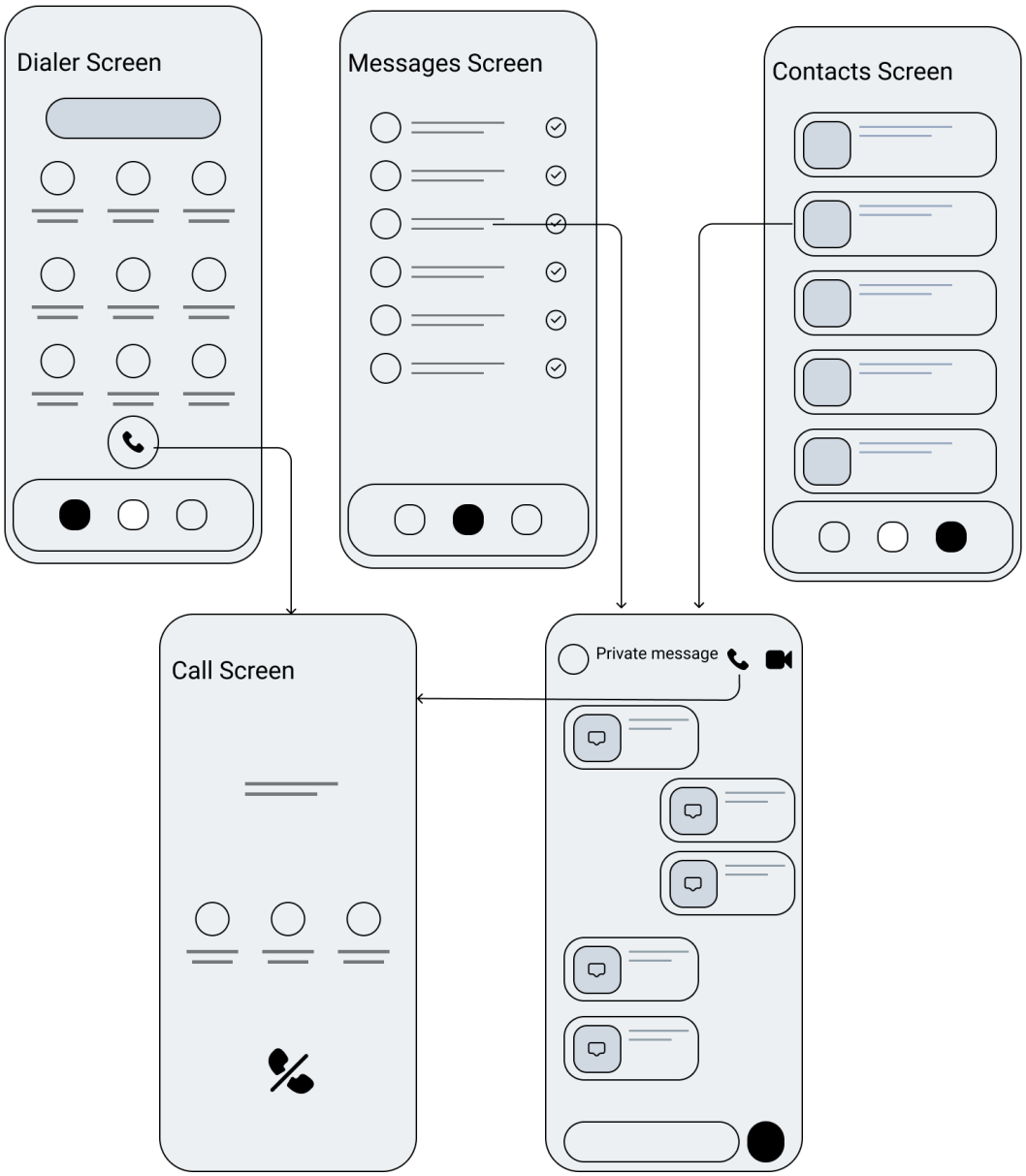

Figure 3.8: Initial app Mockup

Chapter 4

First approach and difficulties encountered

For the development of the project, a IPBRICK test server was allocated to conduct any tests and experiments related to the app development.

During the preparation stage of the master thesis, a prototype app was developed using React Native. This was a very basic app that included only a SIP and XMPP connection. Shortly after the beginning of the project, this hybrid app showed problems working in the background. The connections would only work when the app was opened and in the foreground.

To work around this connection problem, the original hybrid app implementation was abandoned and a native Android implementation was started. This chapter describes this implementation and gives a brief description of the Android environment and development.

4.1 Project Setup

For this app, Kotlin was used since it is the official programming language recommended by Google. Besides using Kotlin, a Beta library called Jetpack Compose was used.[26]

Jetpack Compose is a modern UI toolkit library developed by Google that changes the original way of implementing the UI part of our app. Originally, user interfaces and the business logic are done in separated files. Usually, the interfaces are placed in an XML file that is then linked to an activity of the app. This process is usually quite complicated and there are several ways to go about it.

Jetpack Compose uses declarative UI meaning that graphical components are declared in the main activity file which makes it easy to develop the interaction between UI and business logic.

This library was only used in order to facilitate the development of the app and focus more on the business logic, *i.e.* the connections to the CAFE services. At any time of the development, Jetpack Compose could be dropped and replaced by the original and recommended XML UI files implementation keeping the business logic.

A simple app with a button that shows a toast message was made to show how Jetpack Compose can facilitate the development (Figure 4.1) .

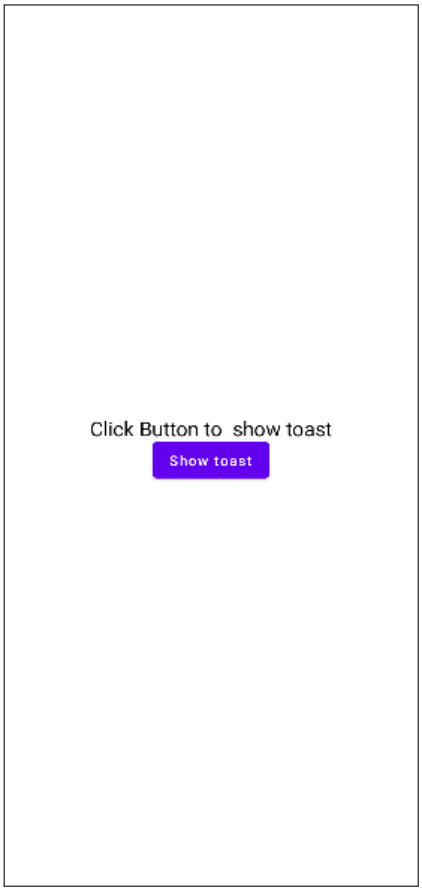

Figure 4.1: Example Toast Button App screen

In Jetpack Compose this screen can be implemented in a single file (Listing 4.1).

Listing 4.1: Jetpack Compose solution

```
@Composable
fun ButtonScreen() {
    Column(...) {
        Text(text = "Click␣Button␣to␣show␣toast")
        Button(onClick = { showToast() }) {
            Text(text = "Show␣toast")
        }
    }
}
```

For a conventional solution, there would need two be at least two files, the layout file (Listing 4.2) and an activity file (Listing 4.3).

Listing 4.2: XML layout file

```
...
<Button
        android:id="@+id/toastButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Show toast"
        tools:layout_editor_absoluteX="158dp"
        tools:layout_editor_absoluteY="367dp" />

...
```

Listing 4.3: Kotlin activity

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val toastButton = findViewById<Button>(R.id.toastButton)
        toastButton.setOnClickListener {
            showToast()
        }
    }
}
```

On a larger application, having to do binding between all the layout components and the Android activity can be quite complicated and time-consuming. Besides that, Jetpack Compose navigation system and component re-usability are much faster to implement than conventional development.

To use Jetpack Compose, Android Studio Canary has to be used in order to access this beta feature. [27]

4.2 App Architecture

In Android development, there are several ways to develop an app when it comes to its architecture. Usually, this will vary from team to team and in some cases, custom solutions are implemented. A common rule of thumb is when in doubt, use the recommend architecture by Google.

As stated by Google, when choosing between app architectures there are usually two concepts to keep in mind:

- **Separation of concerns:** the several elements of the app should be separated. These elements usually include business logic, data, and views (interfaces). This will keep the code clean, the development easy and the app scalable.

- **Drive UI from a model:** this means that the data (model) of the app is not view aware. This way the data will not be affected by life cycle events of the app or lost when the app is destroyed or not responding.

At the time of this dissertation, the recommended app architecture by Google is Model–View–ViewModel (MVVM) [28].

4.2.1 Model–View–ViewModel (MVVM)

MVVM is a software architectural pattern that makes it easy to separate the UI and business logic of an app.

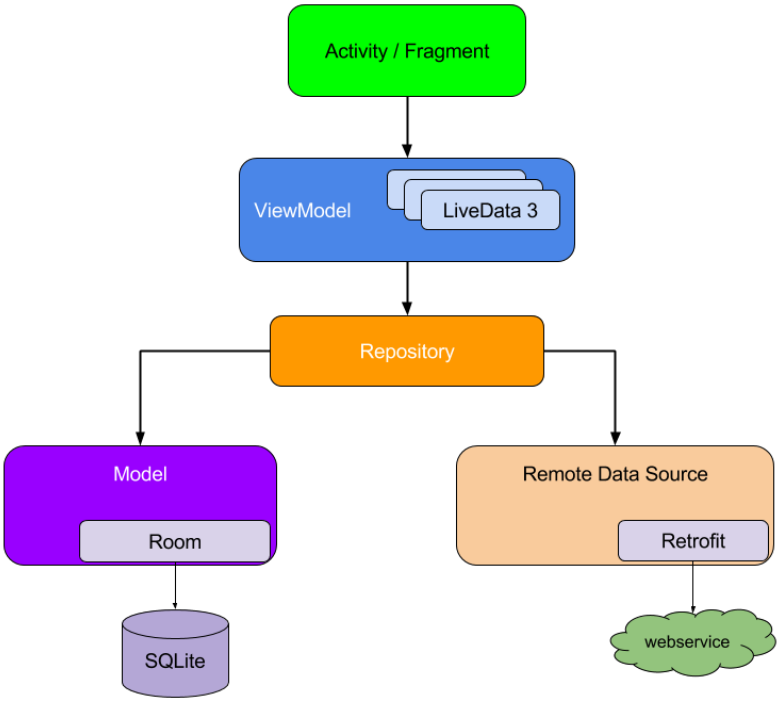

Figure 4.2: Model–View–ViewModel (MVVM)

As seen in figure 4.2, the several parts of the architecture are:

- **Activity/Fragment:** These represent the View part of the architecture. These components represent what a user sees on the screen and handle user interaction for example with clicks. The View interacts with the data using the ViewModel, the layer below.

- **ViewModel:** The ViewModel is a simple interface that facilitates the interaction between the view (user interface) and the model (data). We can have several ViewModels in our app and reuse them whenever we want so we can interact with the data.

- **Model:** The model layer represents the data or objects that we want to access in our app. Since this data can come from several places (local storage, network, etc) this layer is usually wrapped with a repository which makes it easy to interact with data from different sources.

4.3 IpCompose

Since this app is using Jetpack Compose, we decided to call this app IpCompose. In this section, it is described how the app works and how the connections to the CAFE services were implemented.

4.3.1 Foreground Services

As discussed before, one of the main requirements of the app is to keep the connections to the CAFE services alive. This way, users can be alerted in real-time for incoming communications. On Android development, one way to implement this functionality is to use foreground services.

In an Android app, a service is a component that can perform long-running tasks in the background. A foreground service is a service that is noticeable to the user (usually by showing a notification) and that is allowed to keep running even when the app is in the background or in a killed state. A common place to see this type of service is in music players and time-watch apps.[29]

In this app, a foreground service contains the XMPP and SIP connections so that the phone stays connected to the CAFE services even when the app is closed. Figure 4.3 shows the foreground service opening and when the app is in the background.

When it comes to MVVM, there is no recommended place to implement foreground services. This is a topic of great discussion and there are many opinions about this. Since the connections in the service will mainly affect the data model of the app (*e.g.*, for storing new incoming messages) the foreground service only interacts with the repository class of the app.

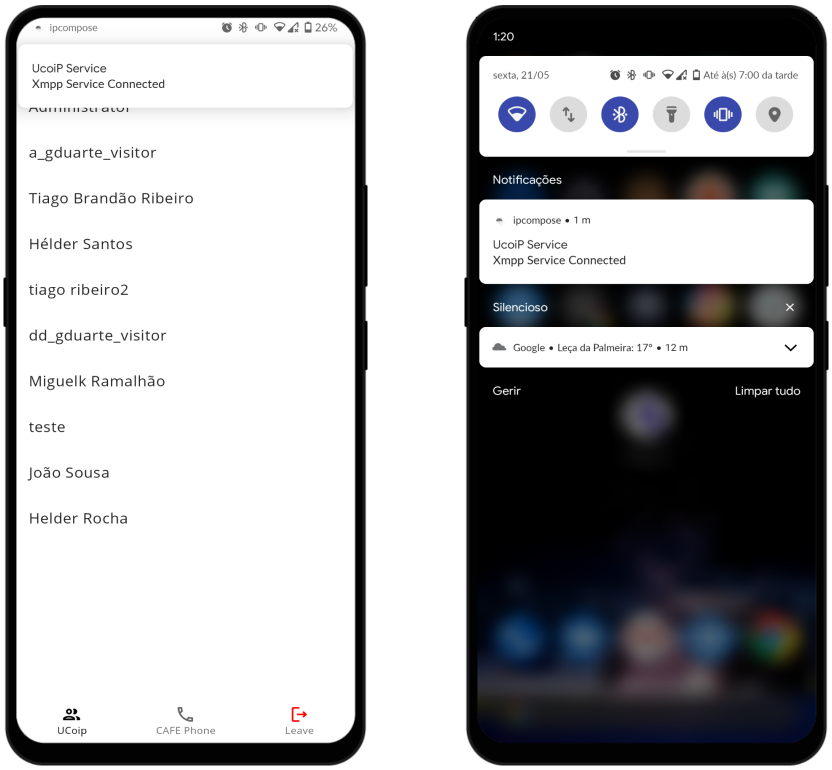

Figure 4.3: IpCompose - Foreground service starting and in the background

4.3.2 SIP

As seen before in section 3.6, Android includes a Sip Manager for developing VoIP apps [30]. This connection lives in the foreground service of the app. When the service starts, the Sip Manager registers a profile with the credentials and domain from CAFE and starts listening for incoming calls (Listing 4.4).

Listing 4.4: IpCompose - SIP registering

```
...
val builder = SipProfile.Builder(username, domain).setPassword(password)

sipProfile = builder.build()
val intent = Intent(this, IncomingCallService::class.java)
val pendingIntent = PendingIntent.getService(this, 0, intent)

sipManager.open(sipProfile, pendingIntent, null)
...
```

By passing a pending intent to the sipManager, we tell the Sip manager to fire that intent when a call is received. This intent then triggers an activity to come to the foreground that is responsible for managing the call state, for example answering or declining the call (Figure 4.4). The flowchart in figure 4.5 represents this chain of events.

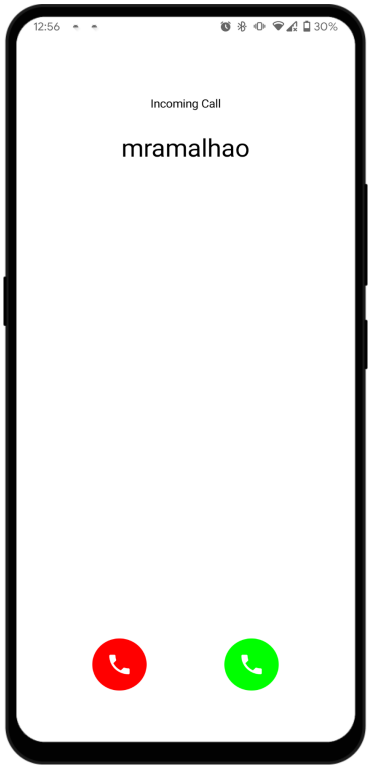

Figure 4.4: IpCompose - Incoming SIP Call

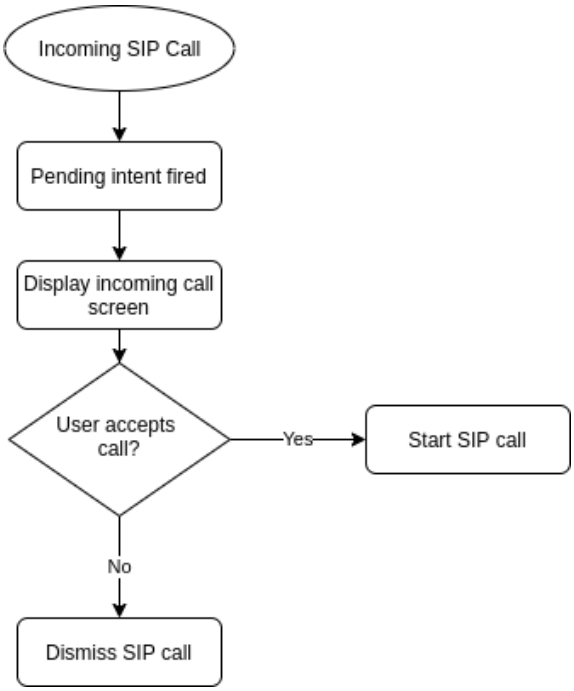

Figure 4.5: IpCompose - Incoming SIP Call flowchart

4.3.3 XMPP

For connecting to the XMPP server, the app uses the Smack library [31]. This library developed by igniterealtime is rich in features and, although it was developed to be used with Java, it is 100% compatible with Kotlin. Just like for the SIP connection, when the service is started, the XMPPTCPConnection object is created and tries to connect to the CAFE server. It also implements several listeners related to connection state and for listening for incoming and outgoing messages (Listing 4.5).

Listing 4.5: XMPP Connection

```
        ...
val config = XMPPTCPConnectionConfiguration.builder()
            .setUsernameAndPassword("gduarte", *****)
            .setXmppDomain("mobile-com.app.ucoip.pt")
            .setPort(5222)
            .setSendPresence(true)
            .build()
conn = XMPPTCPConnection(config)
       ...
chatManager = ChatManager.getInstanceFor(conn)
chatManager.addOutgoingListener(...)
chatManager.addIncomingListener(...)
        ...
conn.addConnectionListener(this)
```

```
try {
    conn.connect().login()
}
```

The chatManager object is responsible for listening for incoming and outgoing messages. These listeners add received and sent messages to the local storage of the device so they can be displayed in the other views. These listeners interact with the repository object to add items to the local database (Listing 4.6).

Listing 4.6: XMPP Incoming message listener

```
        ...
override fun newIncomingMessage(message, ...) {
        ...
    xmppRepository.addMessage(
        ...
            message.from.asBareJid().toString(),
            message.to.asBareJid().toString(),
            message.body,
            sending = false,
            sent = false
        )
    )
}
         ...
```

The connection listener is responsible for alerting the user about the connection state and performing actions when the user first connects to the server, such as fetching the contact list and old messages.

After being added to the database of the app, messages can then be used by the ViewModels to display messages on the views of the app (Figure 4.6).

Figure 4.6: IpCompose - XMPP Chat Screen

4.3.4 Jitsi Meet

As seen before, Jitsi Meet offers an SDK for using alongside Android [32]. It is also known that the Jitsi video calls are signaled using XMPP. Using the XMPP connection already established in the foreground service, a listener is set to intercept incoming video calls. When receiving a video call, the app shows an activity to deal with the management of said call, *i.e.* answering or declining (Figure 4.8). If the user accepts the call, the url included in the custom XMPP Stanza is passed to the Jitsi SDK and a Jitsi Meet call is launched (Listing 4.7). The flowchart in figure 4.7 represents this chain of events.

Listing 4.7: Using the Jitsi SDK

```
                ...
val options = JitsiMeetConferenceOptions.Builder()
    .setServerURL("https://webrtcproxy.app.ucoip.pt")
    .setRoom("MTYxNDYyMTY2Nw")
    .setToken("eyJ0eXAiOiJKV1QiLCJh...")
    .build()
JitsiMeetActivity.launch(this, options)
```

. . .

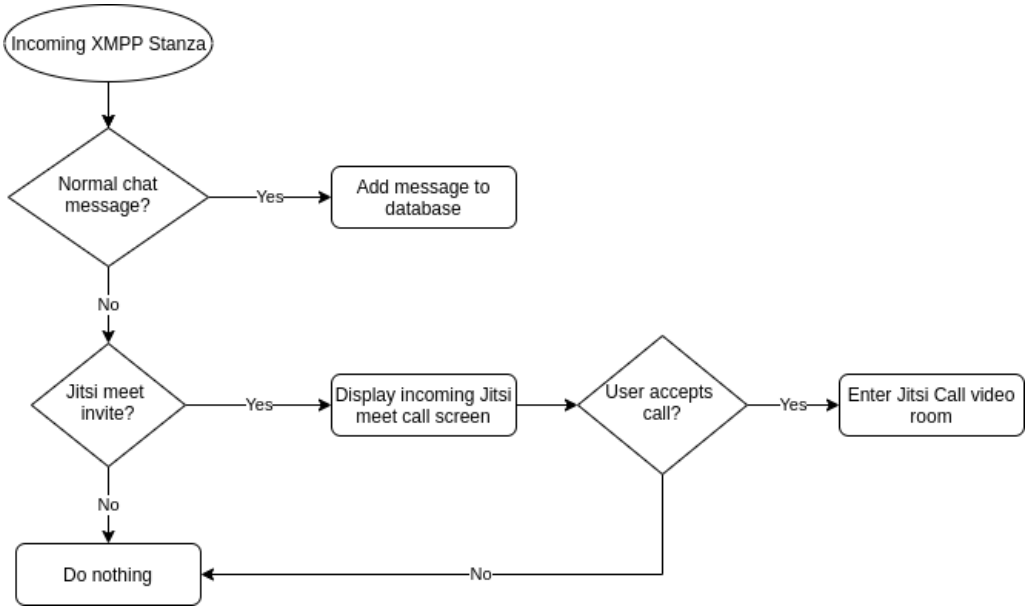

Figure 4.7: IpCompose - Incoming XMPP Stanza flowchart

Figure 4.8: IpCompose - Jitsi Meet

4.3.5 Final product and testing

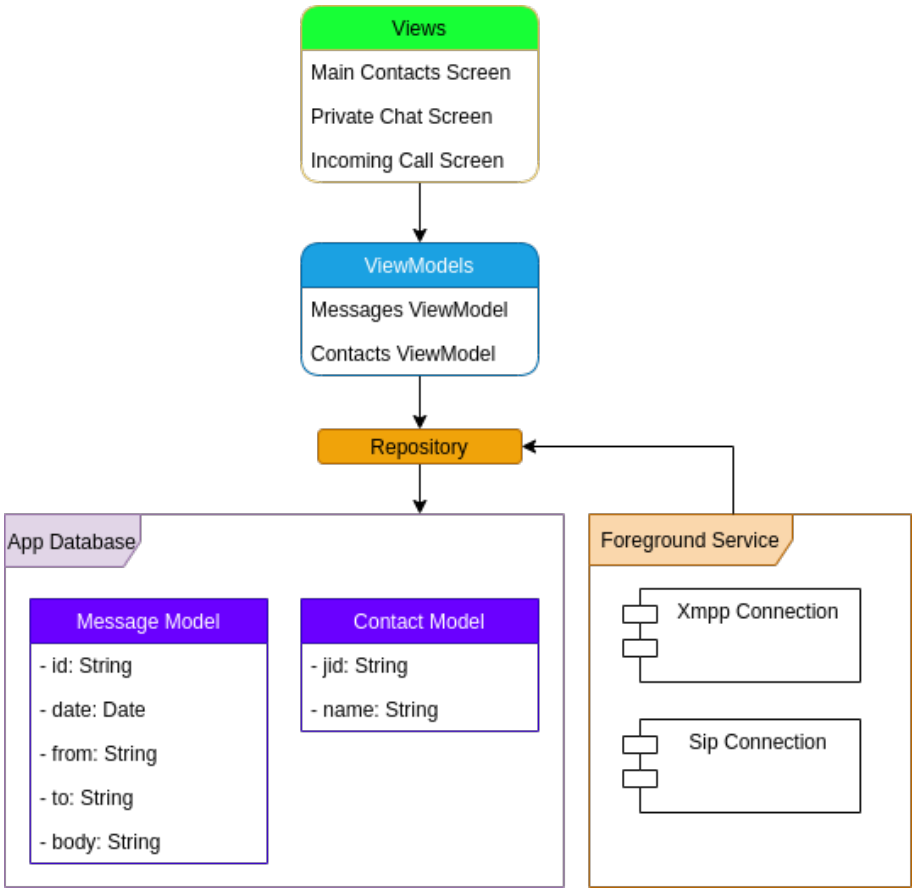

Figure 4.9: IpCompose - Final MVVM Architecture

In the end of this phase, we had a basic native app with SIP, XMPP, and Jitsi video calls implemented. Figure 4.9 describes the final app architecture.

The Sip Manager SDK showed some bugs where sometimes the account would not register and the phone would have to be restarted in order to fix it. Outgoing SIP and video calls were not implemented. Table 4.1 summarizes the achieved functionalities.

Functionality	Native Android Implementation
SIP Incoming calls	implemented with some bugs
SIP Outgoing calls	not implemented
Send/receive XMPP messages	implemented
Jitsi Meet incoming calls	implemented
Jitsi Meet outgoing calls	not implemented
Persistence	unstable

Table 4.1: Native Android functionalities

As stated several times, one of the main concerns for the app is the constant connection to the servers. To test this, the app was installed on two different phones. Both phones were of the brand Xiaomi whose phones are known for the optimal battery performance. To test how the foreground service behaved on both phones, the foreground service was left running on both phones. The results obtained were very inconclusive.

On one phone the connection would sometimes be kept alive for an entire night but other times would take less than one hour to be shut down. On the second phone, the more recent one, the connection would most of the time not be kept alive for more than 20 minutes. These results brought two things to attention:

- The connection state in the foreground services is unreliable to implement in an app that must serve devices from any brand since most of the times these services are stopped by the OS. Besides that, as the Android OS updates, the more battery optimizations there are so this problem would get even worse.
- We realized that besides the app adapting to the CAFE services, also the CAFE services would need to adapt to be integrated with a mobile app since a simple connection to the SIP or XMPP server are not enough to maintain the communications at all times. This started the next phase of the work.

Chapter 5

Reformulated solution using push notifications

With the Android and foreground service implementation, we came to the conclusion that push notifications would need to be implemented to make the app receive incoming communications when in the background or killed state. Only this way we can guarantee that the app receives incoming notifications at any time. This chapter analyzes push notifications solutions and how they were integrated with a new reformulated React Native app.

5.1 Push Notifications

Nowadays, one of the main optimizations performed by phone manufacturers is battery duration. To extend battery life, the operating system will most of the time kill background services. This behavior is unpredictable for every phone and most of the time will depend on several factors such as battery level, type of network connection, free RAM, and others. With this in mind, the persistence of the native app could not be guaranteed with foreground services.

Since the operating system is responsible for this problem, it also gives developers a reliable solution. The solution to this problem is push notifications. Push notifications are a communication channel that is enabled on every phone and that can be used by any app to receive notifications at any time. [33]

The most common push notifications service is firebase cloud messaging from Google. It is a free push notification service that can be integrated with all the frameworks seen before. Most of the time, push notifications services will work on the same basis. Each phone has a unique token that can be used to receive targeted notifications. The service can also be used as a pub-sub service, so every phone that subscribes to a specific topic can receive notifications. Although they are called push notifications, this service can be used to cause a small sync in the background which doesn't mean automatically a notification shown to the user. [34]

Even though most push notifications services are free, management of registration tokens and sending of push notifications falls on the development team. For this project, the firebase push

notifications service by Google was used. Besides being free, they offer an SDK for clients to receive the notifications and an SDK for the server to trigger the push notifications.

5.2 REST API

API (Application Programming Interface) is one of the most important parts of any modern web application. In most modern systems there are two main parts of the software architecture: the front-end and the back-end.

The front-end is any device where the system may run. This usually includes mobile phones, computers, or any other supported electronic devices. In short words, it is an interface the user sees. The back-end, usually called the server, is where all the system logic will live and where the data will be usually stored and handled. Separating these two parts allows the system to centralize all the data and business logic in one single back-end server and include any new front-end device if needed.

An API is an interface that makes possible the communication between these two parts of a system. It acts as an interface between the front-end and the back-end.

There are several types of APIs. To assist the mobile application and to manage the registration tokens for the push notifications a REST (Representational State Transfer) API was implemented. A REST API is a specific type of API created to interact with web Services that uses HTTP as a communication protocol. By using the HTTP protocol we can make use of many requests depending on the operation. The most common ones are GET, POST, PUT, and DELETE. HTTP requests also make it possible to send data from the device to the server using the request body.

Representational state transfer (REST) means that when a request is made to this API, a response is sent back containing a particular state of the system. This state can be sent back in several formats but the most common one is JSON (JavaScript Object Notation). Even though having JavaScript in the name, JSON can be read by most modern programming languages and is easily readable by both humans and machines. The API response will also include a response code. This code is usually used to handle errors:

- Informational responses (100–199)
- Successful responses (200–299)
- Redirects (300–399)
- Client errors (400–499)
- Server errors (500–599)

Figure 5.1 shows an example of a front-end and back-end interaction using a REST API.

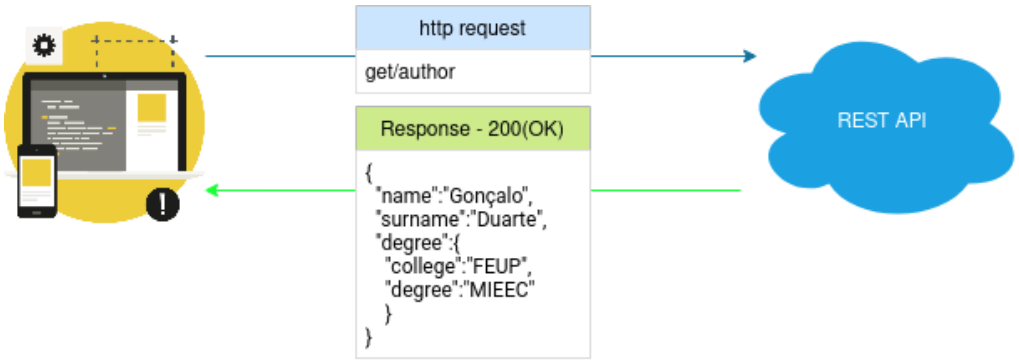

Figure 5.1: REST API Request-Response example

5.3 Next.js

Next.js is a JavaScript framework developed by Vercel that makes the development of fully capable web applications fast and easy. It uses the React framework as a base and allows us to have front-end pages and REST API endpoints in the same server. Next.js was used to develop a dashboard to manage the saved tokens in the system and to create a REST API to be used by the mobile app.[35]

A Next.js project contains a folder for pages and a folder for the API endpoints. The framework will take care of the routing automatically. For example:

The file **pages/author.js** (Listing 5.1) makes available a web page at [serverDomain]/author (Figure 5.2).

Listing 5.1: Next.js example - Author Page

```
export default function author() {
  return (
    <div>
      <h1>Goncalo Duarte</h1>
      <p>Feup MIEEC</p>
    </div>
  );
}
```

Listing 5.2: Next.js example - Author Endpoint

```
export default (request, response) => {
  response.status(200).json({
    name: "Goncalo",
    surname: "Duarte",
    degree: {
      college: "FEUP",
      degree: "MIEEC",
    },
```

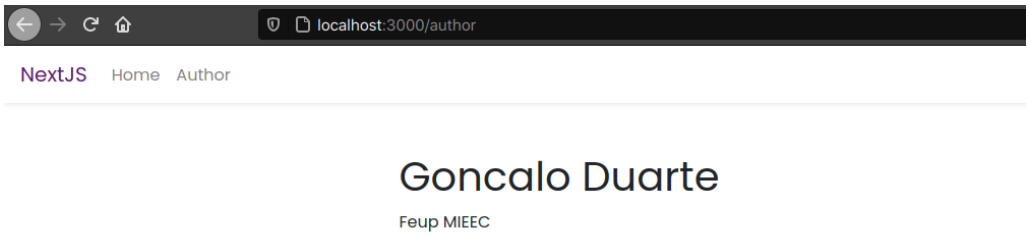

Figure 5.2: Next.js example - Author page

```
    });
};
```

The file **pages/api/author.js** (Listing 5.2) makes an endpoint available at "[serverDomain]/api/author" that can be used by any device to make HTTP requests (Figure 5.3).

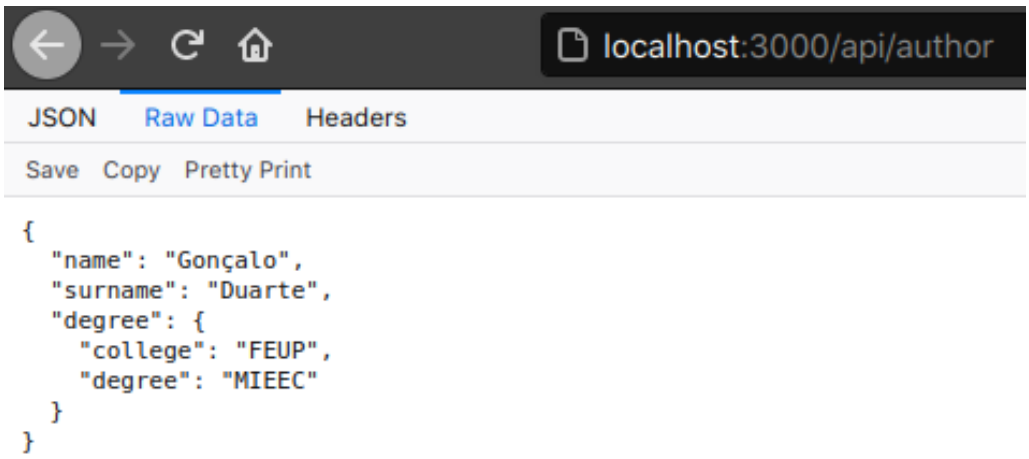

Figure 5.3: Next.js example - Author API endpoint

Note that in listing 5.2, the function takes two parameters, the request, and the response. The request object can be used to intercept the request made and act accordingly if anything is wrong. For example, if the request is of the incorrect allowed method (GET, POST, etc) the server must respond with an error message. The request can also be used to send data from the device to the server using the request body. The response object is used to send responses back to the device that made the request. The response will contain an HTTP code and any relevant data for the response.

5.4 The IpPush web application

To assist the mobile app when it comes to push notifications, a Next.js web application was developed. We decided to call it IpPush.

To store the users and the push notifications tokens, the Next.js application needs a database to store the data. Next.js is compatible with most database solutions. To make it easy, a free mongoDB was used. mongoDB is a noSQL, document-oriented database. Besides having a free hosting plan, mongoDB makes it easy to communicate with the Next.js framework without the need to write complex SQL queries since a mongoDB table stores its items as JSON objects (Figure 5.4).[36]

Figure 5.4: mongoDB table example

To query this table the app only needs to call "db.collection("usertokens").find()".

To send push notifications from the IpPush web application to our mobile app the server uses de firebase admin SDK available for JavaScript [37]. The firebase admin SDK is a group of libraries that makes it possible to interact with the firebase services from the server. To send a push notification to one or several devices the server must use the tokens stored in the database and pass them to firebase SDK with the desired type of notification (Listing 5.3).

Listing 5.3: Firebase sdk example

```
var message = {
    notification: {
      title: "Ipbrick Messenger",
      body: "You received a message",
    },
    tokens: tokenList,
  };
 ...
  await admin.messaging().sendMulticast(message)
```

Auth0 was used to integrate a login system in our app. It is a service that allows protecting pages and API endpoints so our app stays protected from the open internet. [38]

Below is a description of the major endpoints present in the IpPush web application:

- **.../api/posttoken:** This endpoint is used to store a user token on the database. It is used by the mobile app to send the device token to the database. This endpoint uses the POST method (Figure 5.5).

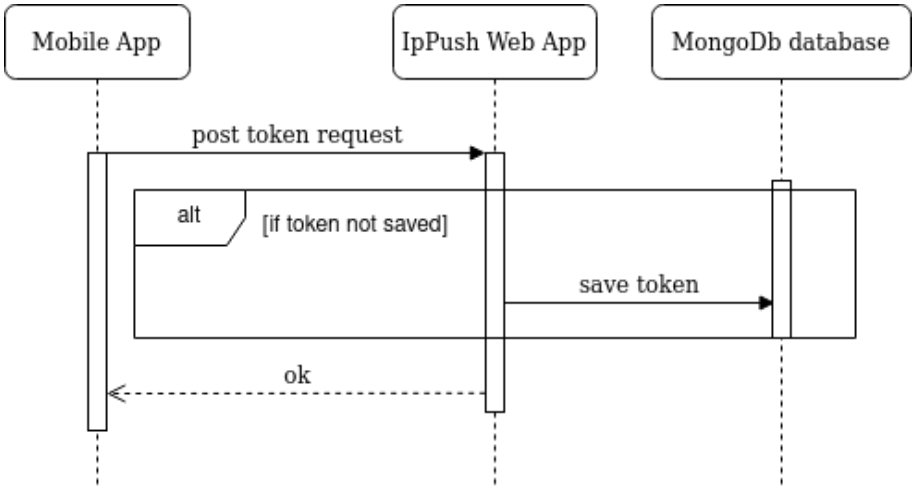

Figure 5.5: Post token endpoint sequence diagram

- **.../api/push:** This endpoint is used by the XMPP server to tell the IpPush web application to send a push notification to a specific user. If the target user is stored in the database, a push notification is sent to its tokens using the firebase admin SDK (Figure 5.6).
- **.../api/inbox:** During the development of the hybrid and native app, XMPP showed one major fault. As said before, XMPP is a modular protocol that allows adding modules depending on the needs of the service. However, XMPP lacks a module to implement an inbox. An inbox is just a list of the last conversations a user had. To overcome this, this endpoint retrieves a list of the last 10 messages a user has had with any user on their contact list. To implement this, the IpPush web application needs to connect to one CAFE database that stores the messages for every user and queries the last 10 messages. Please note that this database is different than the mongoDB used to store user tokens (Figure 5.7).

Besides the endpoints, the IpPush web application allows the user to see a list of saved users in the database and its tokens (Figure 5.8).

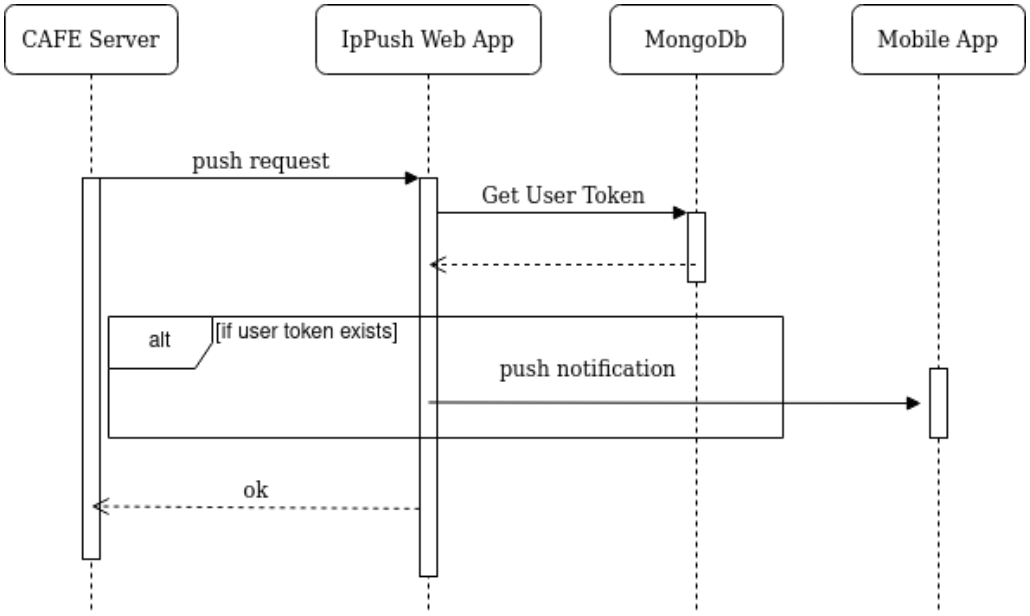

Figure 5.6: Push notification endpoint sequence diagram

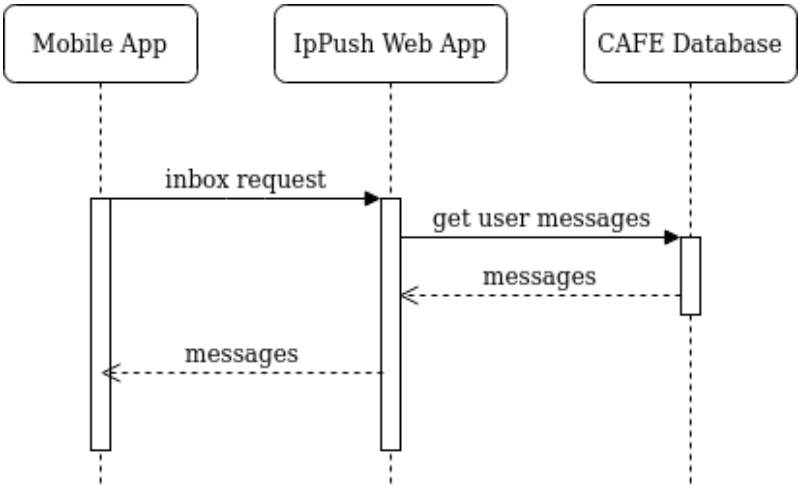

Figure 5.7: Inbox endpoint sequence diagram

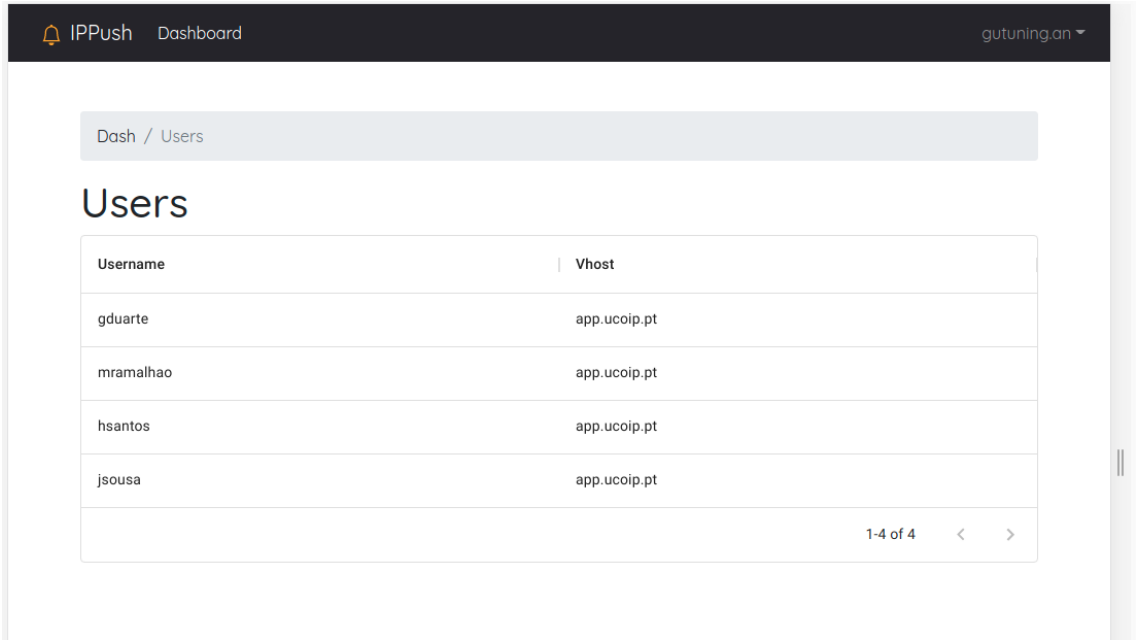

Figure 5.8: IpPush dashboard screen

5.5 CAFE services adaption

As discussed before, the CAFE services are not ready to integrate with a mobile app using push notifications. To overcome this, we had to integrate the IpPush web application with the XMPP and SIP CAFE services.

5.5.1 Ejabberd XMPP server

For the XMPP implementation, a custom module named "mod custom push" had to be integrated into the ejabberd server. This is an open-source module that suffered some small changes to work with the latest ejabberd version. When the XMPP server starts, this module hooks a listener for offline messages. When a message is sent to an offline user, this module makes an API call to a specified path on the internet. On the configuration file of this module, this path is set to the push endpoint of our IpPush web application (Figure 5.6). This way, when a message is sent to an offline user, the IpPush web application receives a request containing the message and the receiver user. It then proceeds to send a push notification to that user if it is stored in the database.

5.5.2 SIP calls and Jitsi Meet video calls

For the SIP server, one similar module has to be implemented in the Asterisk SIP server. Due to time restrictions and its complexity, this module was not implemented. To test push notifications for a SIP call, manual push notifications are sent to the device for testing purposes. The same happens for the Jitsi video calls. Since Jitsi video calls are signaled using a custom Stanza and not regular messages, a similar module has to be implemented either in the XMPP server or in the CAFE social network.

5.6 React Native Implementation

Now that there is a platform to assist the mobile app when it comes to push notifications, the problem with the first React Native implementation was solved and we decided to go back to a React Native hybrid app. This section goes deeper in the React Native environment and how the final app was implemented.

5.6.1 The flux pattern and Redux

As seen before, one of the principles that make React development so easy and organized is component hierarchy. This component hierarchy brings one problem to the table.

Take the example:

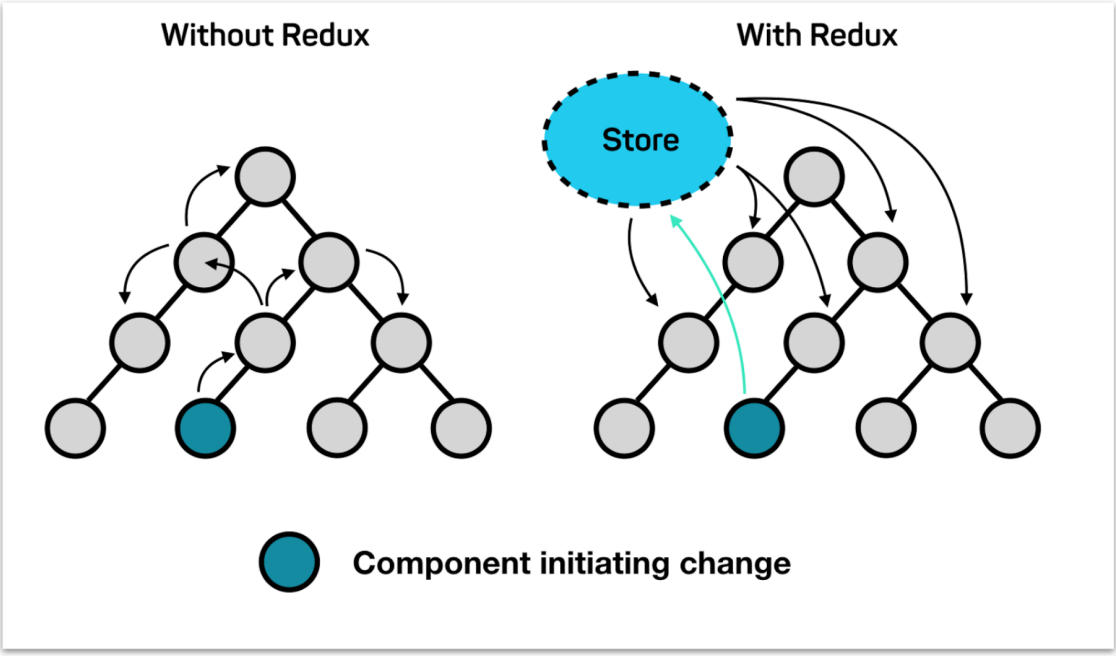

Figure 5.9: Redux tree example [4]

As seen in figure 5.9, when there is a complex structure, sharing state between components can become quite complicated, especially when these components are deeply nested or parallel. This is where the flux pattern comes in handy. By creating a common store to the app, there is no need to deep link all the components in the app. There are several libraries that can be installed in a React Native project that implement the flux pattern but Redux is the most common choice [39].

Redux is composed of three main components: Actions, Reducers, and Store (Figure 5.10).

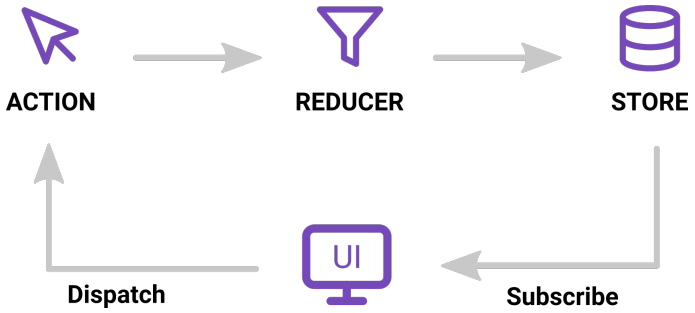

Figure 5.10: Redux main components

The store is where the data lives. By wrapping this store around the app, any component on the app has access to the store and any changes in this store trigger a re-render of the UI where needed.

Actions are responsible for modifying the state. Each declared action has a different function. Each function has a propriety Type and Payload. The action Type is used to uniquely identify the action. The payload is data that can be used by the reducers to mutate the store depending on the action type.

Reducers are the interface between the store and actions. A reducer listens for all the dispatched actions and changes the store accordingly to the action type and the action payload. An app can have several reducers and different reducers can respond to the same action.

In React Native, there is not an option to launch a service with XMPP and SIP connections like in the Android native app. To solve this, we need to discuss one more additional element of the Redux library, the middleware (Figure 5.11).

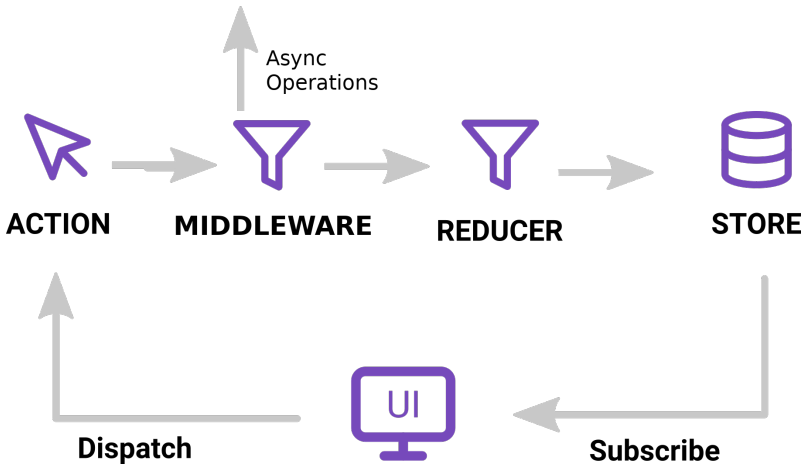

Figure 5.11: Redux middleware

A middleware is a filter that intercepts actions and launches asynchronous operations without blocking the Redux state. Besides this, middlewares can also dispatch actions themselves. This is where the XMPP and SIP connections live, each on its middleware.

5.6.2 SIP

For connecting to the SIP server the React Native app uses the React-Native-sip package [40]. A custom SIP middleware is implemented in the app. When the app is first launched, this middleware intercepts the login action and registers the SIP account. After this, it implements listeners for incoming calls. For showing incoming and outgoing calls, the react-native-sip package is used together with React Native callkeep [41]. This package allows us to show the native call handler for receiving and making calls (Figure 5.12).

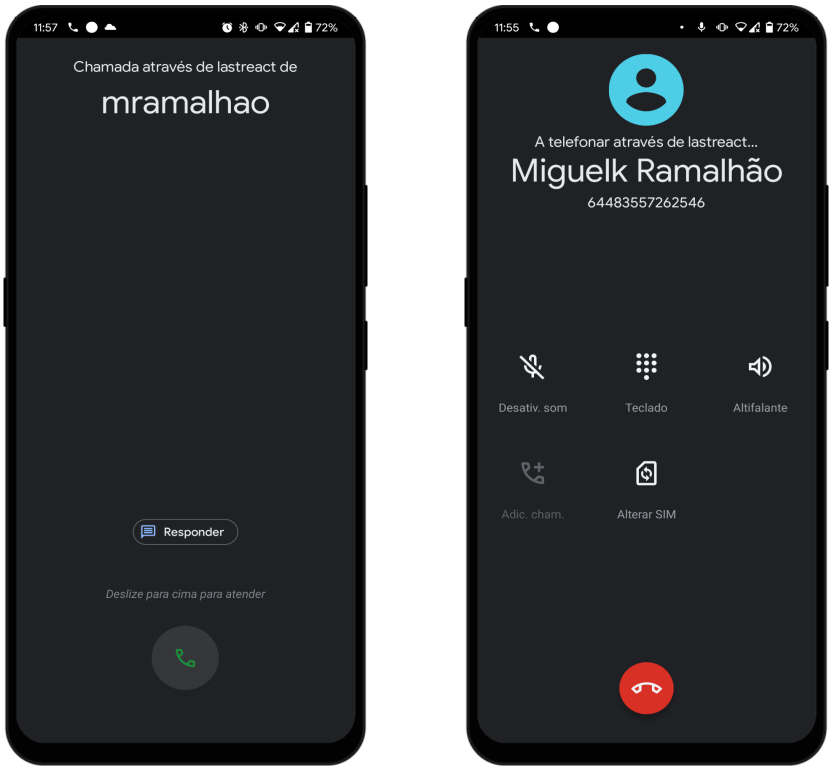

Figure 5.12: React Native implementation - SIP calls

Besides the CAFE contacts, users can place calls to users saved on their contact list. For this, a package called React-Native-contacts was used to access the local contacts [42]. It is also possible to make SIP calls to a number using a dialer (Figure 5.13).

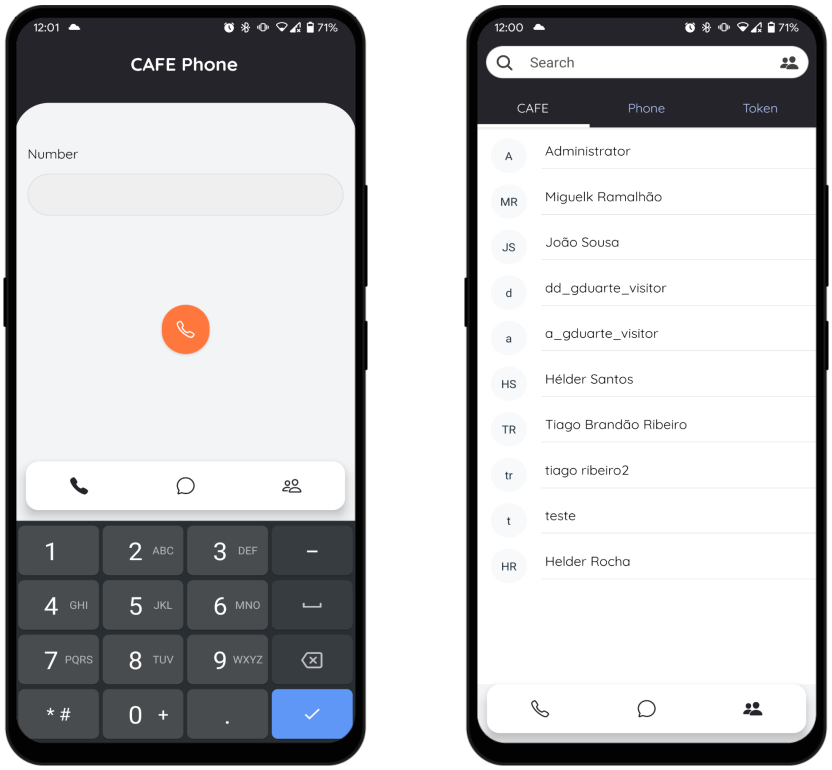

Figure 5.13: React Native implementation - Dialer and contacts

5.6.3 XMPP

For XMPP, a package called StanzaJS was added to the app [43]. Just like the SIP connection, this connection lives in a middleware. When this middleware receives the login action, it connects to the CAFE servers and implements listeners for incoming messages. At the same time, it also queries the IpPush web application for the user inbox.

For the messages screen, a package called React Native gifted-chat was used [44]. This package implements a chat screen where the app only needs to pass the messages stored. It also allows us to customize the chat interface to make it fit the app better (Figure 5.14).

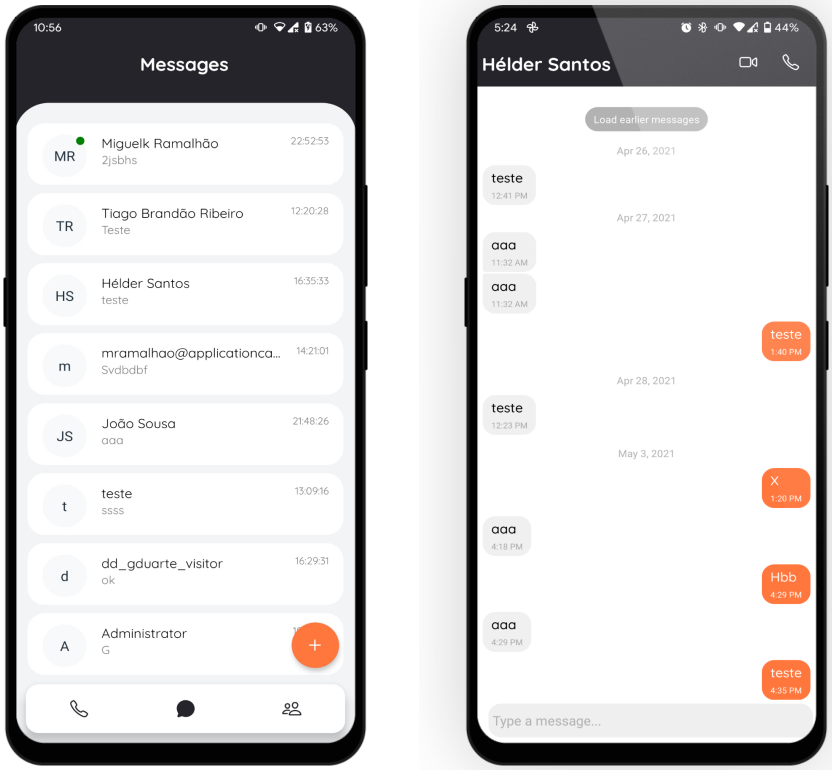

Figure 5.14: React Native implementation - messaging

5.6.4 Jitsi Meet video calls

For intercepting incoming Jitsi Meet video calls a custom filter was added to the XMPP connection in the XMPP middleware. When a call arrives, the React Native callkeep package, also used for the SIP calls, is used to warn the user about the incoming call. For the Jitsi Meet video call, a different approach was implemented. Although React Native has a package for Jitsi Meet, there is a problem when using this package. The CAFE social network wraps a browser window around the Jitsi Meet video calls. What would happen when using the Jitsi SDK whether in React Native or the native Android implementation, is that the call would start in the mobile app but the CAFE user would not be alerted of the start of the call and the call would not appear as answered on the browser. To overcome this, instead of using the Jitsi SDK, the app uses the link sent in the XMPP Stanza and uses a WebView to enter this link. This way, the user enters the call in the mobile app, and, at the same time, the CAFE social network is alerted of the start of the call and the two users can start communicating (Figure 5.15).

Figure 5.15: React Native implementation - Jitsi Meet calls

5.6.5 Integration with the IpPush web application

For receiving push notifications, the firebase React Native SDK was added to the app [45].

For an XMPP message, the push notification message is of the type notification. This means the app does not need to do anything and the notification will be displayed automatically to the user (Figure 5.16).

For SIP calls, the notification is of type data. This means that no notification will be displayed and that the app must handle the notification and decide what to do. For this, the app filters the type of the incoming push notification and, if it is of the type "SIP CALL", the app dispatches a login action that makes the phone connect to the SIP server (Listing 5.4). After this, any incoming SIP call is received by the app.

Figure 5.17 displays an ideal SIP implementation regarding push notifications. Since there was no time to implement such operating mode, the SIP call testing was done manually. This means sending manually a push notification of type "SIP CALL", wait for the app to receive the push notification, and place a SIP call using the CAFE social network.

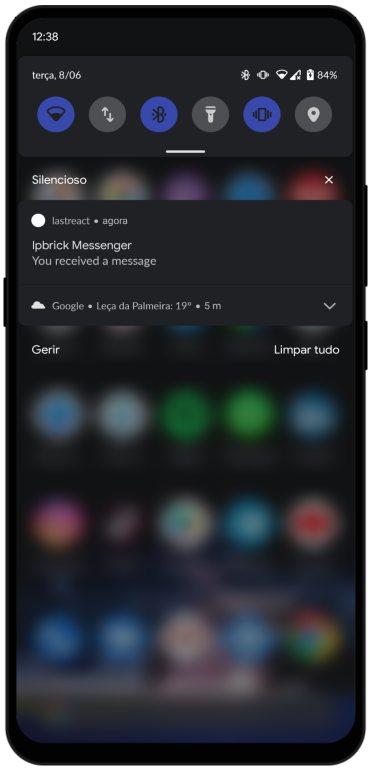

Figure 5.16: Push notification received for new XMPP message

Listing 5.4: Push notification listener

```
messaging().setBackgroundMessageHandler(async remoteMessage => {
    if (remoteMessage.data.type === 'SIP_CALL') {
      store.dispatch(login());
   }
});
```

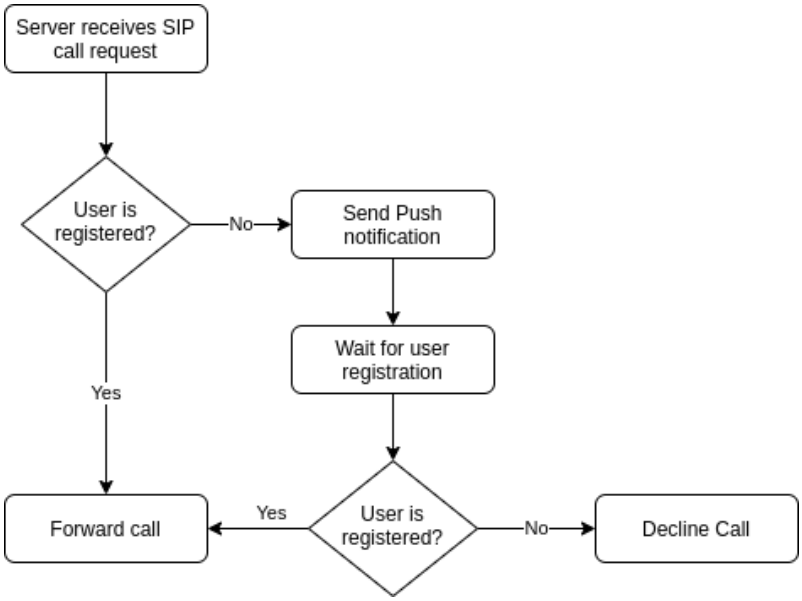

Figure 5.17: Ideal SIP push notifications implementation

5.6.6 Final Architecture

Figure 5.18 shows the final app architecture.

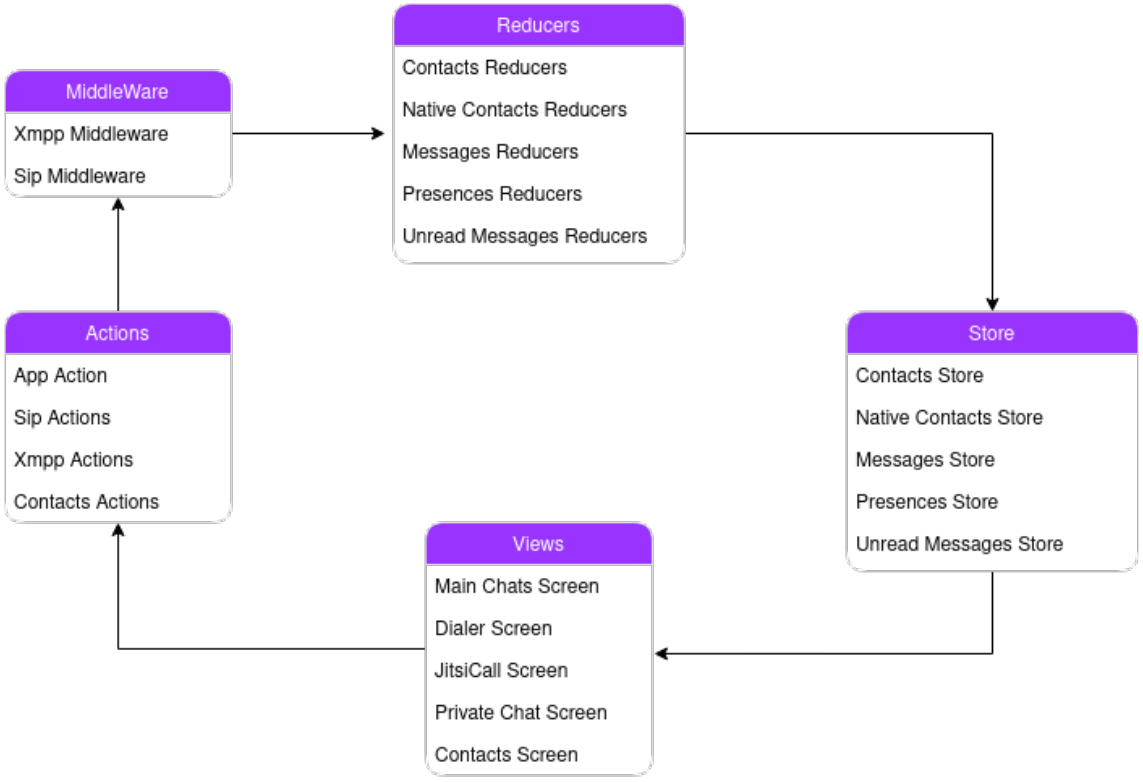

Figure 5.18: React Native app architecture

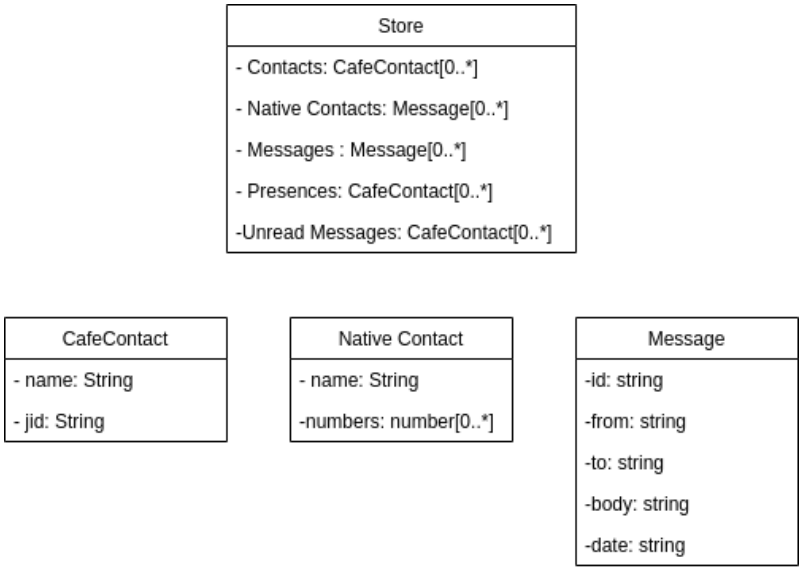

Figure 5.19: Redux Stores

As seen in Figure 5.19 the main Redux Store is composed by the following sub-stores:

- Contacts Store - Stores the contacts from the CAFE social network;
- Native Contacts Store - Stores a list of contacts saved on the phone;
- Messages Store - Stores a list of messages received;
- Presences Store - Tracks the presences of the contacts list, whether a user is online or offline; If a user is in this store, the user is online;
- Unread Messages Store - Stores a list of unread messages for alerting the user. If a user is in this store, there is an unread message from this user.

Figures 5.20 and 5.21 represent the behavior of the XMPP and SIP middlewares for different actions.

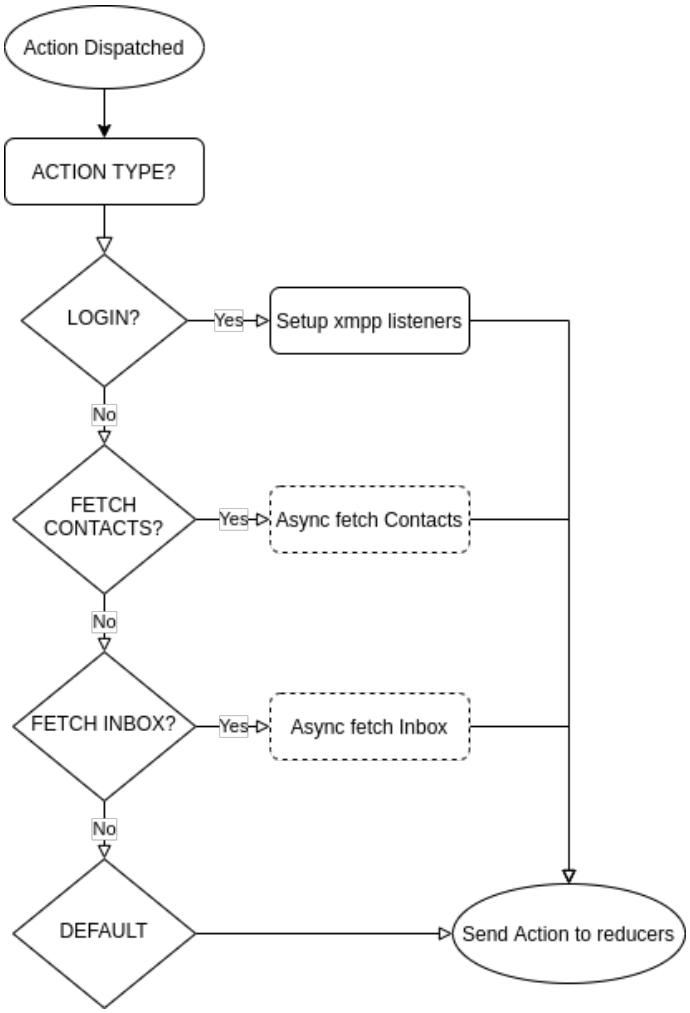

Figure 5.20: XMPP middleware flowchart

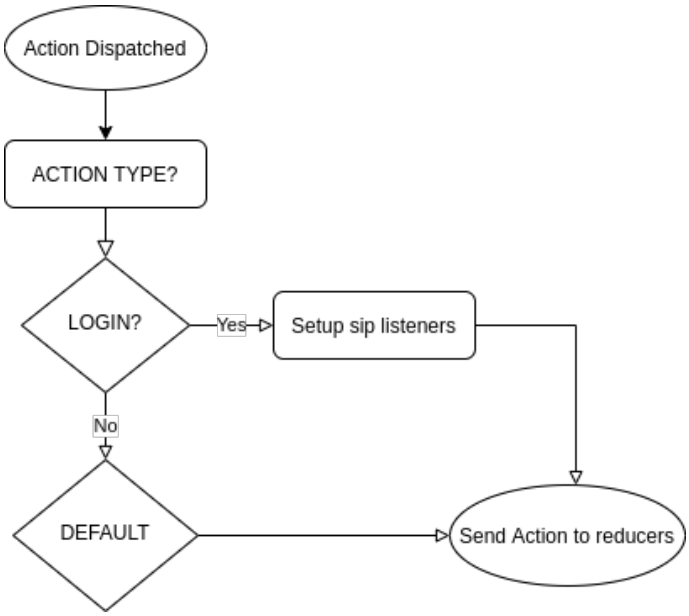

Figure 5.21: SIP middleware flowchart

Please note that asynchronous operations in the middleware do not block the state. The asynchronous operation starts and at the same time, the action is sent to the reducers.

Figures 5.22 to 5.26 show the behavior of the app reducers and how they mutate the different stores depending on the action type.

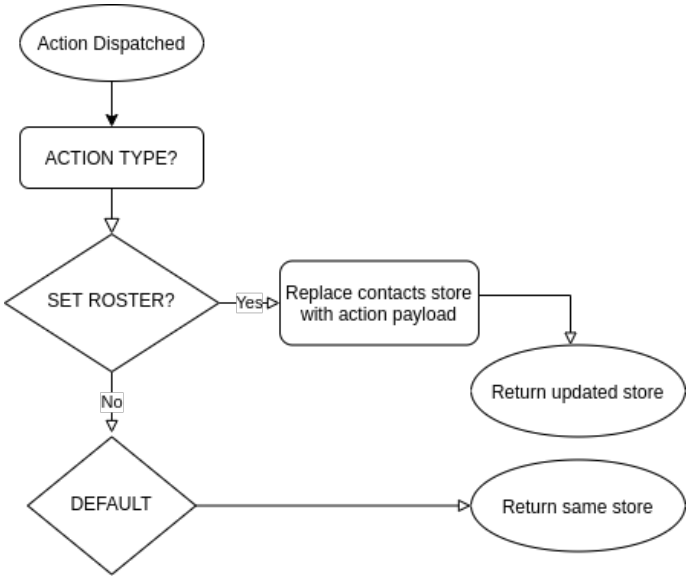

Figure 5.22: Contacts Reducer flowchart

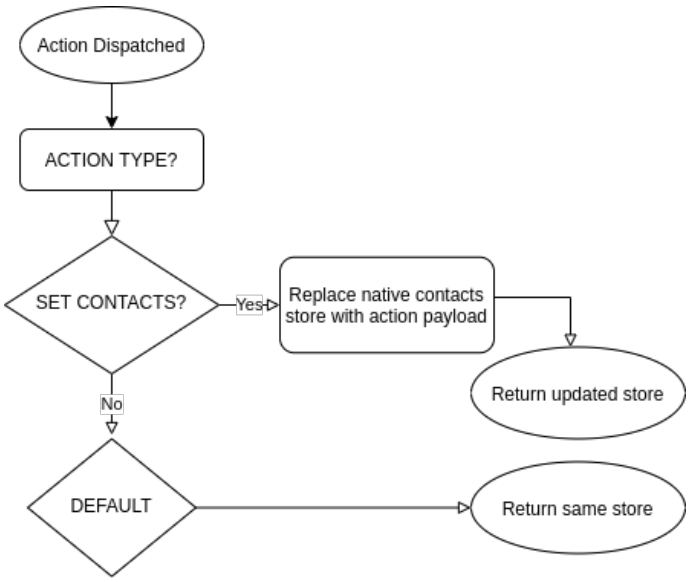

Figure 5.23: Native Contacts Reducer flowchart

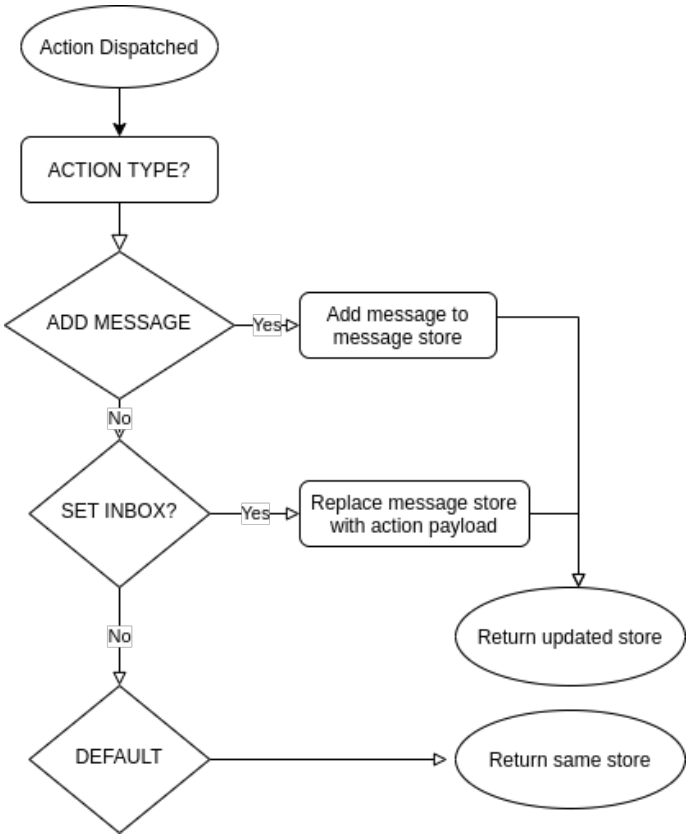

Figure 5.24: Messages Reducer flowchart

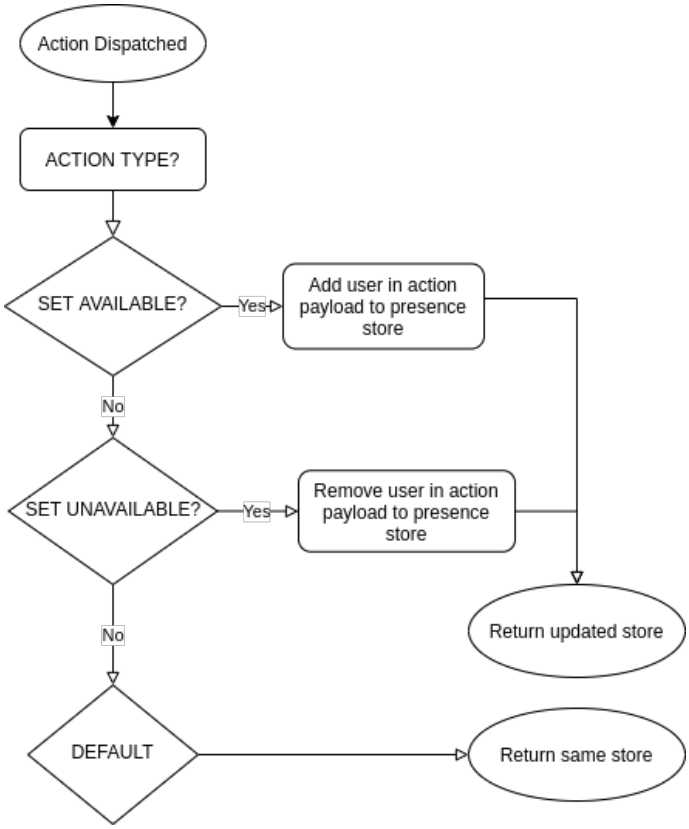

Figure 5.25: Presences Reducer flowchart

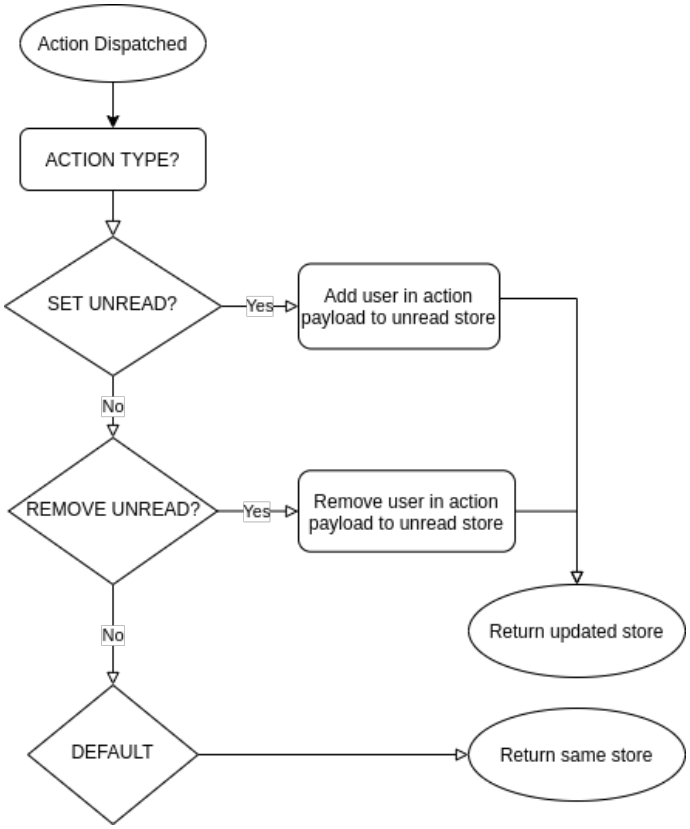

Figure 5.26: Unread Messages flowchart

Chapter 6

Results

This chapter describes the final state of the project and analyzes results regarding performance and the original requirements.

6.1 Final Architecture and operation mode

The final solution is composed by the React Native app and the Next.js web application described in the previous chapter (Figure 6.1). The IpPush web application was published in Vercel, a free hosting platform for testing purposes.[46]

In the previous chapter, it was already described how the systems interact with each other. Simply put, we can distinguish two different operating modes (Figure 6.2):

App online - As long the app is opened and connected to the internet, the CAFE services communicate directly with the app. When the app is open it only communicates with the IpPush web application for querying the messages inbox (Left screen of Figure 6.3).

App offline - When the app is offline, CAFE services communicate directly with the IpPush web application that tries to notify the user about incoming communications using firebase push notifications (Right screen of Figure 6.3).

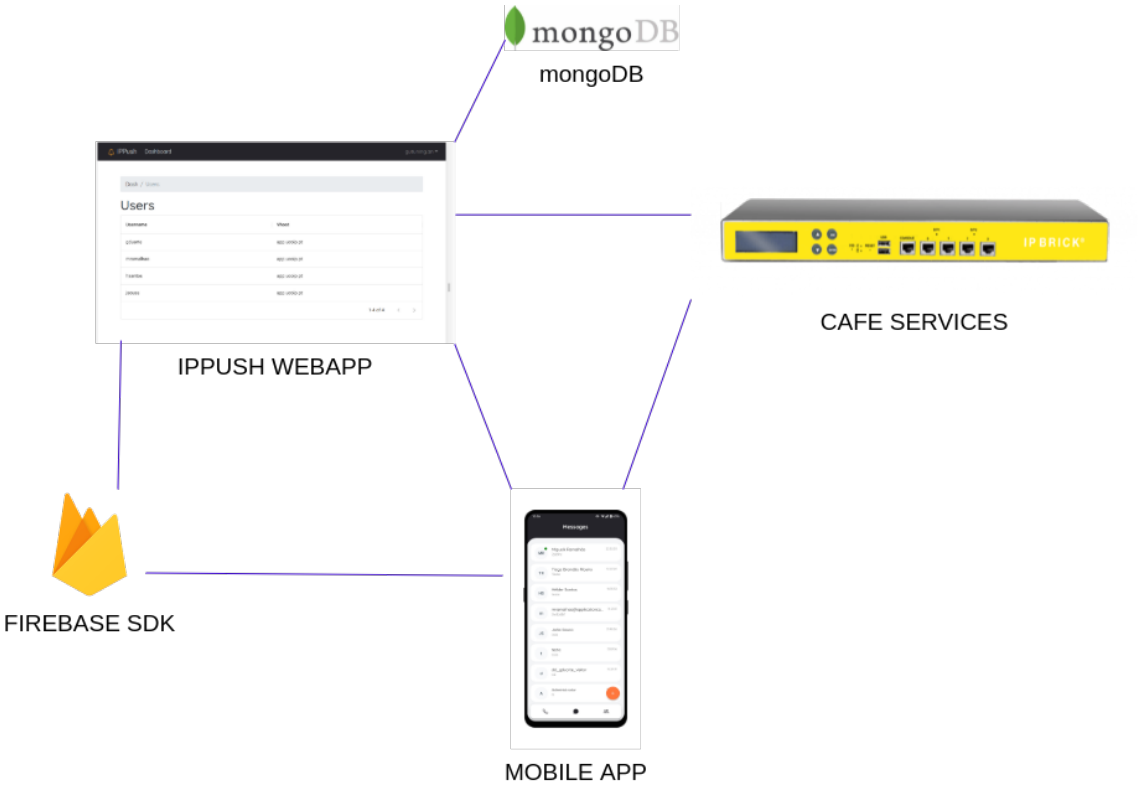

Figure 6.1: Final Solution structure

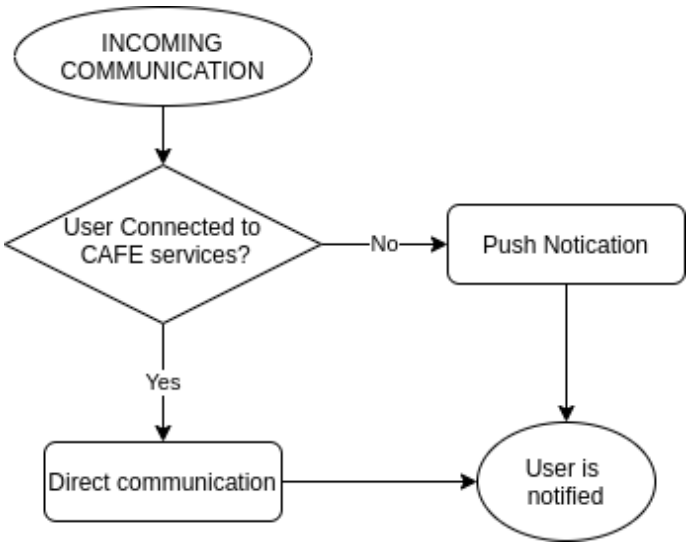

Figure 6.2: General working flow

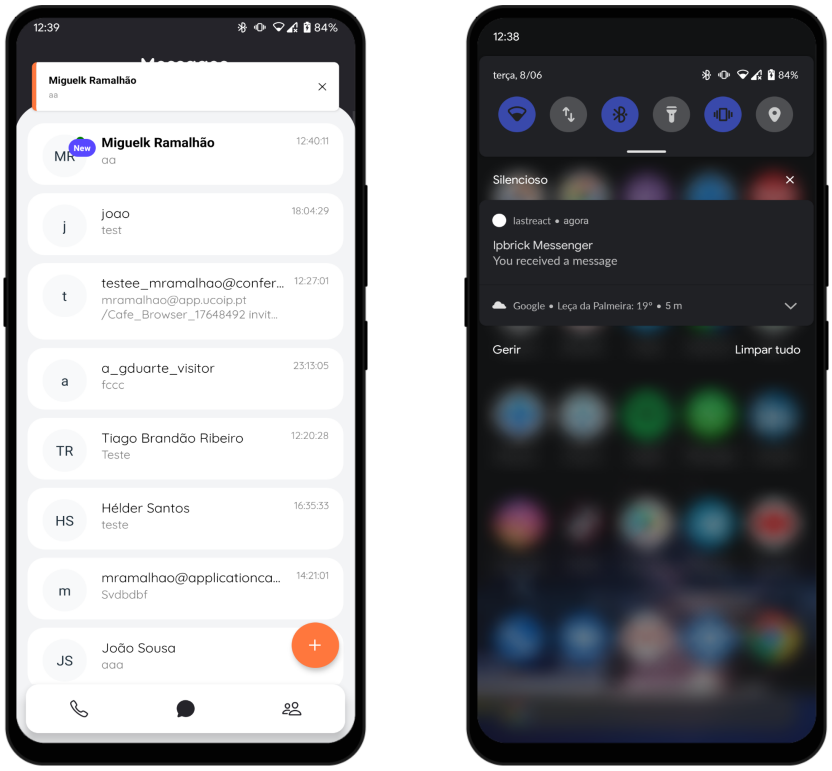

Figure 6.3: XMPP messages received with app opened and closed

6.2 Testing

For testing purposes, the app was installed on three different phones. The phones were left on standby for periods of 1 hour, 3 hours, and 6+ hours. At the end of each period, one XMPP message was sent from CAFE. Table 6.1 shows the results of this test.

	Phone 1	**Phone 2**	**Phone 3**
1 hour	received	received	received
3 hours	received	received on second message	receivced
6 hours	received	received on second message	received on second message

Table 6.1: XMPP Push notifications test results

The app behaved like intended and, on the 3 phones, a push notification was received every time. Sometimes the push notification would only be triggered when sending two consecutive messages. This behavior is most certainly caused by the free hosting platform that automatically puts the web application in doze mode when it is not used for long periods of time.

Note that these tests were only performed for the XMPP messages. Since the SIP and Jitsi Meet calls will work on the same basis, the results will eventually be the same for those services.

As for performance, the app was tested on two phones and, using a third-party app for measurement, the app maintained 60fps on both phones, which is the max.

The app was also tested using the Android Profiler, a tool part of the Android Studio IDE [47]. The profiler results can be seen in figure A.1 of the appendix. During the profiling, the test order was the following:

- 1- Send 2 messages;
- 2- Change app screens randomly;
- 3- Make a SIP call;
- 4- Make a Jitsi call.

6.3 Results discussion

At the end of the project, the overall work developed was considered very positive.

React Native turned out to be a reliable solution with all the connections to CAFE services working. The final app was only tested in Android but all the packages used announce compatibility with iOS. This means it will not be hard to make the app work for iOS as well.

Table 6.2 shows the implemented requirements in the final React Native app also comparing them to the native Android app.

As seen in the profiling results (A.1), towards the end of the experiment, there is a slight increase in resource consumption. This relates to the Jitsi Meet call. In general, WebViews tend to

	Native Android Implementation	**React Native App**
SIP Incoming calls	implemented with some bugs	implemented
SIP Outgoing calls	not implemented	implemented
Send/receive XMPP messages	implemented	implemented
Jitsi Meet incoming calls	implemented	implemented
Jitsi Meet outgoing calls	not implemented	implemented
Persistence	unstable	stable with push notifications

Table 6.2: Native Android and React Native comparison

have bad performance. For the current implementation of CAFE, this is the only reliable solution for Jitsi Meets. Besides that, the app seems to have a good performance overall with low resource consumption. The app did not use more than 500Mb of RAM and CPU usage stayed below 50% most of the time

The testing on the app was quite limited. The push notification system and CAFE connections appear to be stable but ideally, the app should have been distributed in a beta release for a real-world test. Since the final solution came so late in the semester, there was no time for such test.

The app is also not ready for production yet. Some key features still need to be implemented, such as a login system.

As for the support platform, the IpPush web application, this solution must be reformulated and deployed for production. Since each IPBRICK client (company or organization) has its own server, this platform can be implemented in two different ways. One solution is two have a central server that serves all the IPBRICK clients (Figure 6.4). Other solution is for each client to have its own platform on its server (Figure 6.5).

Although this subject must be studied further down, it seems reasonable to go for the latter solution since a centralized solution might be unreliable due to the high number of clients. Please note that the firebase service must be the same since there is only one app.

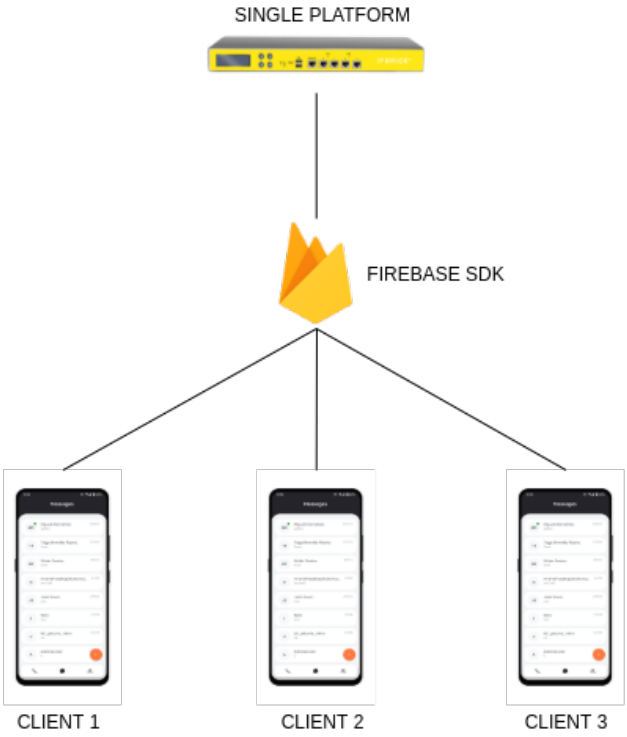

Figure 6.4: Support platform solution 1

Figure 6.5: Support platform solution 2

Chapter 7

Conclusions and future development

7.1 Summary of development and achievements

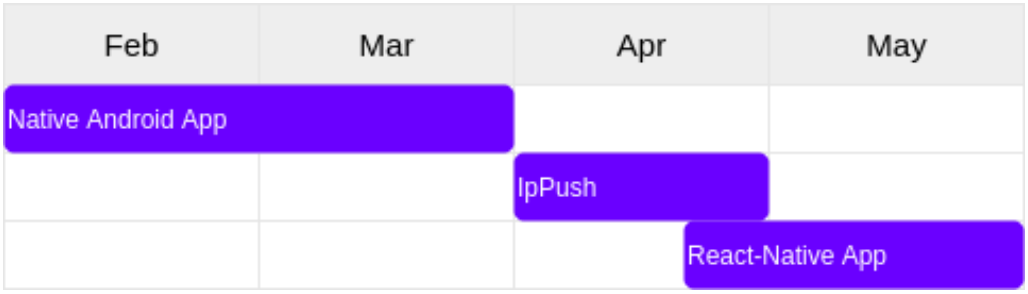

Figure 7.1: Time diagram of the work developed (approximately)

Looking at figure 7.1, the React Native development took almost half the time of the native Android. In that time, the React Native app developed was much more stable and included a lot more functionalities than the Android one. This translates to the experienced complexity of native development. The native development had a much steeper learning curve and brought a lot more problems and bugs. Even though it is a more reliable solution, a team of experienced developers is needed to develop a production-grade native mobile app.

Even though the native Android development was discarded, the time spent developing it was very important. When transferring to the hybrid solution, there was a much more strong base of knowledge related to the CAFE services and to the protocols used.

As seen during the semester, the biggest challenge was to keep the flow of communication at all times. Right from the beginning of the work, everything indicated that push notification would need to be implemented. The foreground service was an attempt to depart from that implementation since push notifications would also mean a change in the CAFE environment. After some time, we came to the conclusion that these push notifications needed to be implemented and there was no way to avoid it whether in a native or hybrid solution.

The module added to XMPP server was obtained in an internet forum. Since developing XMPP modules is a complex subject, this module might contain some bugs since the module was not fully tested. Also, implementing a push notification system on the SIP server is a very complex task since it can break the current working operations. Because of this, SIP calls using push notifications were not fully implemented but are working as a proof of concept in the final app.

7.2 Further needed developments

As stated before, the final app is just a prototype. To go into production, the app must be rebuilt from the start and implement some extra functionalities such as:

- Login system - The app must be able to verify and login a user of any organization;
- Design - During the development of the app, the design aspect was ignored. To bring the app for production, a design that fits with the IPBRICK ecosystem must be implemented.

As for the assisting platform, in our case the IpPush web application, this platform must also be rebuilt and moved into a production-level product. As seen in figures 6.4 and 6.5 of the previous chapter, there are several ways to implement this platform. As for the endpoints of this API, the minimum recommended methods are the following:

- Login method - The API must have a method for the app to validate the user credentials;
- Token management - The API must be able to add and remove user tokens when a user logins and logouts of the app;
- Push notification triggering - The API must have methods to trigger push notifications for XMPP messages, SIP calls, and Jitsi Meet video calls;
- Message retrieval - Since the XMPP protocol does not offer an inbox solution, the API must be able to retrieve a list of messages to be used by the app.

7.3 Future evolution

The app developed in this project is mainly focused on the communications aspect of the IPBRICK services. Related to the UCoIP services, some functionalities can be added such as group chatting, display of scheduled meetings, and email functionalities. If done right, a React Native app is easily scalable and, with the help of an assisting platform, these functionalities can be easily implemented in the app. When there is a reliable and stable solution to use the communication services, the app can scale even more and implement all the aspects of the CAFE social network such as posts and picture sharing.

References

[1] Julie Bai. An Introduction to SIP Protocol: Definition, Features, More, 2019. URL: https://www.nextiva.com/blog/sip-protocol.html.

[2] IPBRICK OS - IPBRICK. URL: https://www.ipbrick.com/ipbrick-os.

[3] IPBRICK.CAFE Rede Social Corporativa - IPBRICK. URL: https://www.ipbrick.com/pt-pt/ipbrick-cafe-rede-social-empresarial.

[4] Developing modern offline apps with ReactJS, Redux and Electron - Part 3 - ReactJS + Redux - codecentric AG Blog. URL: https://blog.codecentric.de/en/2017/12/developing-modern-offline-apps-reactjs-redux-electron-part-3-reactjs-redux-

[5] Session Initiation Protocol - Wikipedia. URL: https://en.wikipedia.org/wiki/Session_Initiation_Protocol.

[6] rfc3261. URL: https://datatracker.ietf.org/doc/html/rfc3261.

[7] XMPP | About XMPP. URL: https://xmpp.org/about/.

[8] Extensible Messaging and Presence Protocol – Wikipédia, a enciclopédia livre. URL: https://pt.wikipedia.org/wiki/Extensible_Messaging_and_Presence_Protocol.

[9] XMPP | Specifications. URL: https://xmpp.org/extensions/.

[10] Jitsi Meet. URL: https://meet.jit.si/.

[11] Android (operating system) - Wikipedia. URL: https://en.wikipedia.org/wiki/Android_(operating_system).

[12] SDK Tools release notes | Android Developers. URL: https://developer.android.com/studio/releases/sdk-tools.

[13] Meet Android Studio | Android Developers. URL: https://developer.android.com/studio/intro.

[14] Java (programming language) - Wikipedia. URL: https://en.wikipedia.org/w/index.php?title=Java(programminglanguage)&oldid=991899959.

[15] Kotlin (programming language) - Wikipedia. URL: https://en.wikipedia.org/wiki/Kotlin_(programming_language).

[16] Matheus Flauzino, Júlio Veríssimo, Ricardo Terra, Elder Cirilo, Vinicius H.S. Durelli, and Rafael S. Durelli. Are you still smelling it? A comparative study between Java and Kotlin language. In *ACM International Conference Proceeding Series*, pages 23–32. Association for Computing Machinery, sep 2018. doi:10.1145/3267183.3267186.

[17] Kotlin VS Java - Android Development | Hacker Noon. URL: https://hackernoon.com/kotlin-vs-java-android-development-qh6z329j.

[18] iOS SDK - Wikipedia. URL: https://en.wikipedia.org/w/index.php?title=IOSSDK&oldid=992170097.

[19] What's New in iOS - Apple Developer. URL: https://developer.apple.com/ios/whats-new/.

[20] Xcode - Suporte - Apple Developer. URL: https://developer.apple.com/pt/support/xcode.

[21] React Native · Learn once, write anywhere. URL: https://reactnative.dev/.

[22] npm. URL: https://www.npmjs.com/.

[23] Home | Yarn - Package Manager. URL: https://yarnpkg.com/.

[24] Flutter - Beautiful native apps in record time. URL: https://flutter.dev/.

[25] Dart packages. URL: https://pub.dev/.

[26] Jetpack Compose | Android Developers. URL: https://developer.android.com/jetpack/compose?gclid=Cj0KCQjw_dWGBhDAARIsAMcYuJwIr9rddlxGX8gqIqxahzKSW2tAxxjn_Kg0LUMhCe8YF3iav0JIi4waAk0DEALw_wcB&gclsrc=aw.ds.

[27] Android Studio Preview | Android Developers. URL: https://developer.android.com/studio/preview.

[28] Guide to app architecture | Android Developers. URL: https://developer.android.com/jetpack/guide.

[29] Foreground services | Android Developers. URL: https://developer.android.com/guide/components/foreground-services.

[30] SipManager | Android Developers. URL: https://developer.android.com/reference/android/net/sip/SipManager.

[31] Ignite Realtime: Smack API. URL: https://www.igniterealtime.org/projects/smack/.

[32] Android SDK · Jitsi Meet Handbook. URL: https://jitsi.github.io/handbook/docs/dev-guide/dev-guide-android-sdk.

[33] What is a Push Notification and Why It Is Important. URL: https://buildfire.com/what-is-a-push-notification/.

[34] Firebase Cloud Messaging. URL: https://firebase.google.com/docs/cloud-messaging.

[35] Next.js by Vercel - The React Framework. URL: https://nextjs.org/.

[36] The most popular database for modern apps | MongoDB. URL: https://www.mongodb.com/.

[37] Adicionar o Firebase a seu projeto JavaScript. URL: https://firebase.google.com/docs/web/setup?hl=pt.

[38] Auth0: Secure access for everyone. But not just anyone. URL: https://auth0.com/.

[39] Redux - A predictable state container for JavaScript apps. | Redux. URL: https://redux.js.org/.

[40] GitHub - florindumitru/react-native-sip: SIP module for React Native. URL: https://github.com/florindumitru/react-native-sip.

[41] GitHub - react-native-webrtc/react-native-callkeep: iOS CallKit framework and Android ConnectionService for React Native. URL: https://github.com/react-native-webrtc/react-native-callkeep.

[42] GitHub - morenoh149/react-native-contacts: React Native Contacts. URL: https://github.com/morenoh149/react-native-contacts.

[43] GitHub - legastero/stanza: Modern XMPP, with a JSON API. URL: https://github.com/legastero/stanza.

[44] GitHub - FaridSafi/react-native-gifted-chat: The most complete chat UI for React Native. URL: https://github.com/FaridSafi/react-native-gifted-chat.

[45] React Native Firebase | React Native Firebase. URL: https://rnfirebase.io/.

[46] Develop. Preview. Ship. For the best frontend teams – Vercel. URL: https://vercel.com/.

[47] The Android Profiler | Android Developers. URL: https://developer.android.com/studio/profile/android-profiler.

Appendix A

Results

A.1 Profiling results

Figure A.1 displays the profiling results of the final React Native app. The app was tested using the Android Profiler, a tool part of the Android Studio IDE. During the profiling, the test order was the following:

- 1- Send 2 chat messages;
- 2- Change app screens randomly;
- 3- Make a SIP call;
- 4- Make a Jitsi video call.

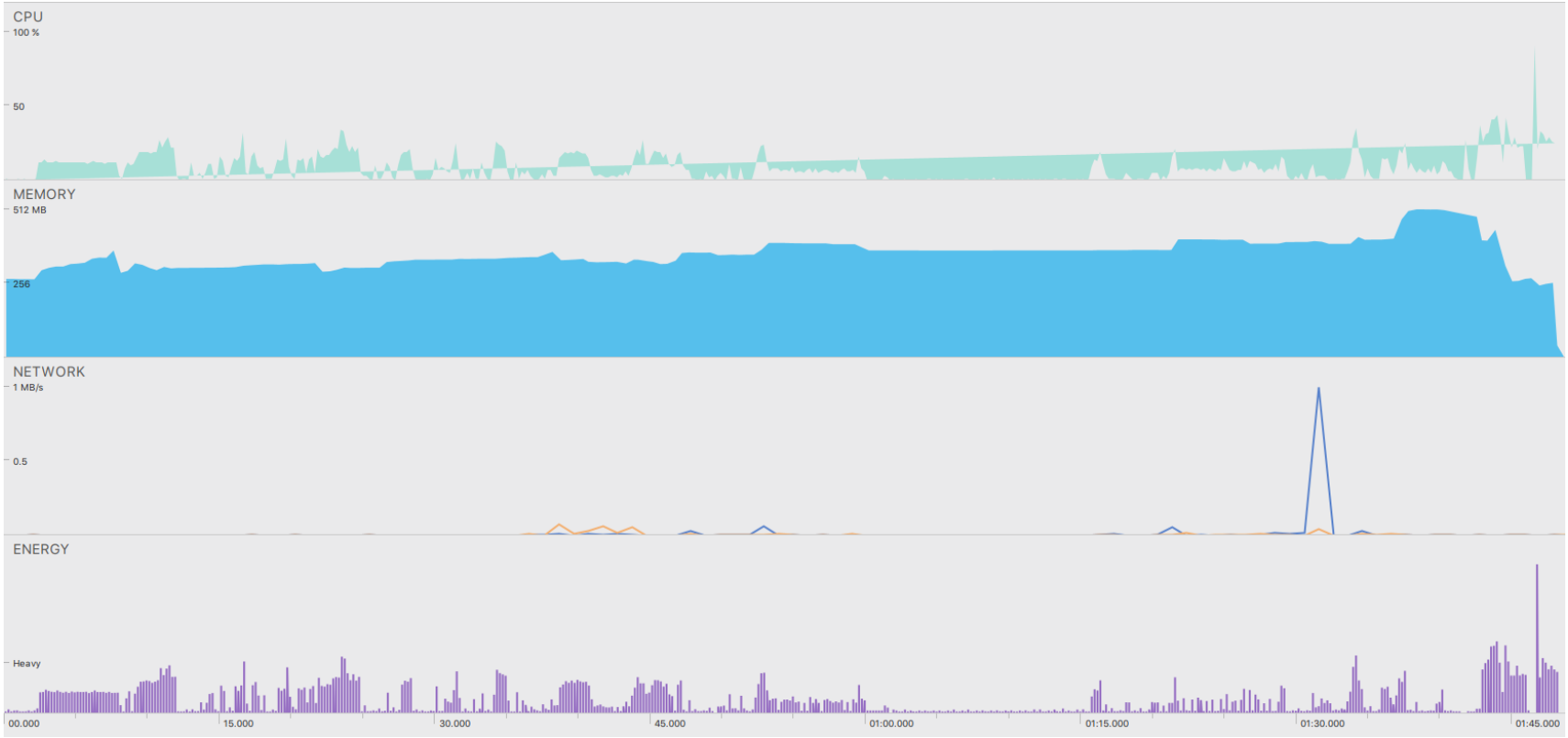

Figure A.1: Android profiler results