Table of contents

1. Introduction

In the last years mobile market is growing explosively which had a lot of effect on our life. The ownership of smartphones are increasing year by year in most countries [1] and the habits of internet usage are continuously changing too [2]. People spend more time surfing on the internet on their phone than on other devices. This is also connecting to the fact that mobile market reached the tipping point [3] and there are already more smartphone users globally than laptop and PC users. That's why it is not surprising that the motto of the Mobile World Congress (MWC) in 2016 was: "Mobile is everything".

Based on the "mobile first" mentality smartphone applications have a major role in every kind of field now. It is well noticeable via the exponential growth of the number of the applications which can be found for example in Google Play and AppStore. [4] While in 2010 only 250 000 apps where available, this number was more than 3 million at end of 2015 based on the research of Vision Mobile. [5].

1.1. Motivation

Smartphones exist since almost 10 years and even if in this time the market went through drastic changes, the problem with segmented operation systems still remains. Although as time passed some of the operation systems slowly disappeared but the market still divided by the three major systems: Android, iOS and Windows Phone. This was problematic years ago just like today for companies and developers who does not have much resources and have to decide or prioritise one OS over on another. That's why there is a need for approaches which would give a solution for the multiplatform problem and allow to implement an app only once which afterwards can run on any kind of device. One way to reach this is to develop a web application which is technically not real application but a site which runs in the phone's browser. Although this approach improved a lot in the last years there are still certain limitations e.g. accessibility of all the features of the phone.

Fortunately there is a another approach between native and web application development called hybrid which combines the advantages of the previously mentioned methods. As every approach it also has disadvantages. In this thesis a main question is if it is an appropriate method for implementing a mobile application with real time communication.

1.2. Goals

The goal of this thesis is to analyze different approaches to write a real time communication application on mobile devices regarding to different criterias. On the example of the GeoGebra Groups application it will be shown which kind of platform will be the best choice regarding to the requirements.

GeoGebra Groups is providing opportunities for educators to collaborate with each other, create a virtual classroom for their students where they can share materials, write posts and comments, create and edit GeoGebra worksheets together, give tasks and feedback. Chapter 2 will cover this in more detail.

To reach this goal I will first examine the current hybrid development tools and compare it with a native platform, Android, then presenting the advantages, disadvantages and limitations in general. The comparison will be done from different perspectives based on the following criterias: maintainability, multiplatform support, perceived and device performance and features.

Another important component of the thesis is the real-time communication in mobile applications because it represents the base for the GeoGebra Groups app. I will research the currently most popular solutions (e.g. WebRTC [6], Socket.IO [7]) and their compatibility with different hybrid or native solutions. It is also important which kind of data can be transferred through those real time communication APIs (eg.: text, voice, video, files).

To show the result of the research and to examine the advantages and disadvantages I will implement prototypes using different technologies. The prototypes will provide the basic functionalities of the GeoGebra Groups. Based on them the most appropriate solution can be determined.

1.3. Limitations

The purpose of the prototypes is to demonstrate the differences between the used technologies and real time communication (RTC) methods. For the scope of this thesis, only the most relevant functionalities will be implemented such as:

- Groups
- Posting
- Commenting
- Direct Messaging
- Audio and video communication

Another limitation is that the native prototype will be implemented only for Android. The reason of the author's choice is based on two factors. First the usage of Android devices are over 65% on the market [8] and second - the accessibility of Android phones are higher among students based on the price.

2. GeoGebra Groups

GeoGebra is an interactive and dynamic application for teaching and learning mathematics which intended for schools and universities. It is available on several operating systems. The desktop application can be used on Windows, Linux and Mac OS X. It provides also applications for Android and iOS. Moreover it is also available from the browser through its web application [9].

In the applications it is possible to create materials using different views, such as: Geometry, Graphing, CAS, Spreadsheet, which can be saved afterwards.

GeoGebra Groups [10] is a platform which allows users to create groups and invite people. In a Group it is possible to share materials with other members, create and edit worksheets together and communicate via posts and comments. Besides that it also gives the opportunity to manage tasks and give feedbacks, which helps teachers to interact with a group of their students. GeoGebra Groups can be also used as a virtual classroom, or as a platform to collaborate with other people to work together on new materials.

3. Technological background

This chapter briefly covers the different technologies which are used in my thesis. First I will give a short introduction about the multi platform development in general (section 3.1) then present some of the existing hybrid development frameworks (section 3.2.). In section 3.3 I will show the two leading native operating systems - Android and iOS in more detail. As RTC is an important component of this thesis, I will give a general overview about it in the last section and show some of the most used solutions (section 3.4.).

3.1. Multi platform development

The diversity of mobile operation systems and devices has a big effect on the mobile application development. Implementing an app for different operating systems takes more time and money, because every system has its own API, uses its own programming language and following different concepts. Furthermore one may need different people who have knowledge in this area for developing applications for these operating systems. That's why it became a goal to find a more effective way where one codebase is enough and it's able to work on every device. There are different approaches of implementation which results different type of application. Currently the mobile application types can be classified as [11]:

- Native application
- Web application
- Hybrid application

Native applications are developed for a certain operation system using a specific programming language such as Java for Android and Objective C or Swift for iOS. The

advantages of the native applications are that they have a big variety of access of different sensors and features of the device which may be limited for other mobile application types.

Web applications are written by using web technologies such as HTML, CSS and Javascript. They are not really mobile applications in the traditional meaning, but websites which are running in the browser of the phone and are optimized for mobile devices. The main characteristic of this type is that they cannot be downloaded from the different stores like App Store or Google Play and (at the moment) has limited access to the sensors of the smartphone, but can have almost native look like.

Hybrid applications - as its name already shows - are a combination of the previous types. They are using web technologies but in contrast of web applications, don't need Internet to initially download the website. Another difference is that hybrid apps have access to the device's features through a native container in which they run in..

Figure 1 sums up which was shown in the previous chapter. Furthermore it makes once again clear that every type has its advantages and disadvantages. For example with a native app the developer get access to all features of the device but it will run just on one platform.

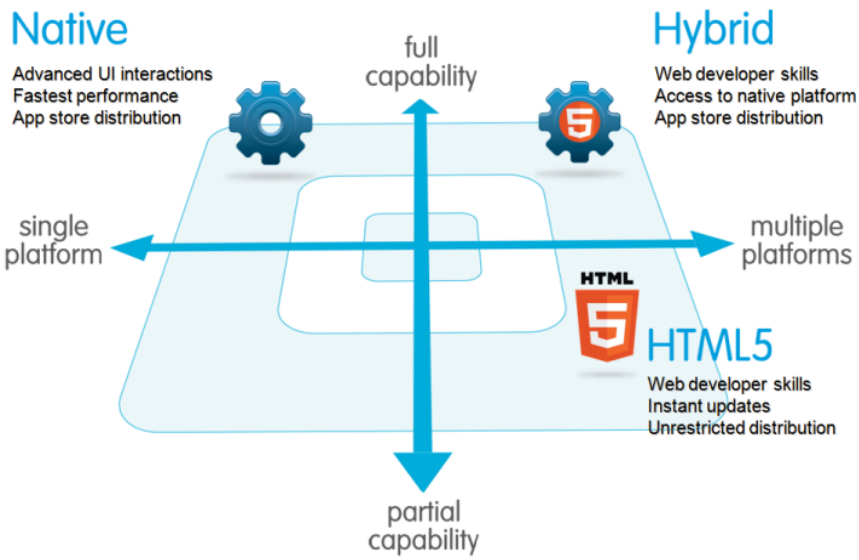

Figure 1. System of the different mobile development approaches [12]

3.2. Hybrid mobile development

Chapter 3.1. already discussed that a hybrid app is commonly defined as web application which runs locally on the device. It is using HTML, CSS, Javascript or other scripting languages and runs in a WebView in a native container. The app is called hybrid because it is not a complete web app but neither a native one. Device's features and sensors can be accessed through the native container and different plugins not like in web apps. Figure 2 shows how WebView, NativeUI and the Container interact with each other. In contrast to native apps the layout is rendered via the WebView instead of the platform's own UI framework. Another main difference is that hybrid apps can be downloaded from app stores which is important when the goal is to reach the target audience of the product in the most efficient way.

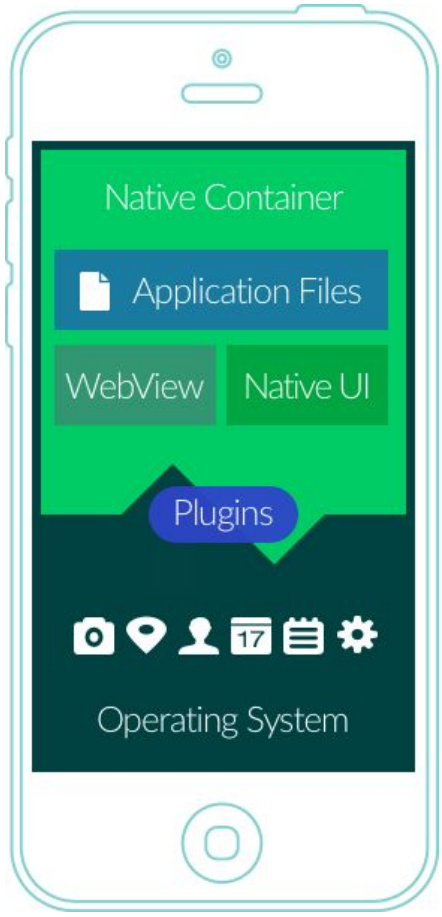

Figure 2. Structure of a hybrid application [13]

As mentioned already earlier in section 3.1. there are not just advantages but also disadvantages like for other type too. One of the most important is the performance regarding to the speed of the application. Although it is much faster than a web app which runs in the browser there are still significant differences compared to a native application. Of course the difference of the performance depends on the purpose of the application. A complex design of the UI or heavily relying on sensors makes the product slower. That's why there are typical fields where hybrid apps can provide really good native like-and-feel such as business applications or content delivery. [14]

As it was described in the previous section the motivation behind the hybrid approach is "code less - reach more". It basically means to find the most effective way to implement mobile apps from development and also from a business point of view. As a hybrid app uses one code base for every platform it is matching with those requirements. Developers don't need to know the platform specific programming languages and also the app has to be done only once. It reduces the development cost and time dramatically. Of course it is not so straightforward which approach is the best, there are lot of factors which has to be examined before deciding in which direction to go. I will present those factors in detail in chapter 6.

3.2.1. Apache Cordova

Apache Cordova is one of the leading platforms [15] for writing and building hybrid applications. The official documentation defines it as: "Apache Cordova is an open-source mobile development framework. It allows you to use standard web technologies - HTML5, CSS3, and JavaScript for cross-platform development. Applications execute within wrappers targeted to each platform, and rely on standards-compliant API bindings to access each device's capabilities such as sensors, data, network status, etc." [16]

This definition is matching the definition of hybrid apps which is presented in section 3.2. The app is developed in HTML, CSS and JavaScript. For accessing device specific sensors there are plugins which have to be implemented in a native way for every platform.

The development process with Cordova is the following. The developer implements the web application then with the Cordova tool and the certain SDKs packaging it to a native application runtime container for each target platform as it is shown in Figure 3. [17]

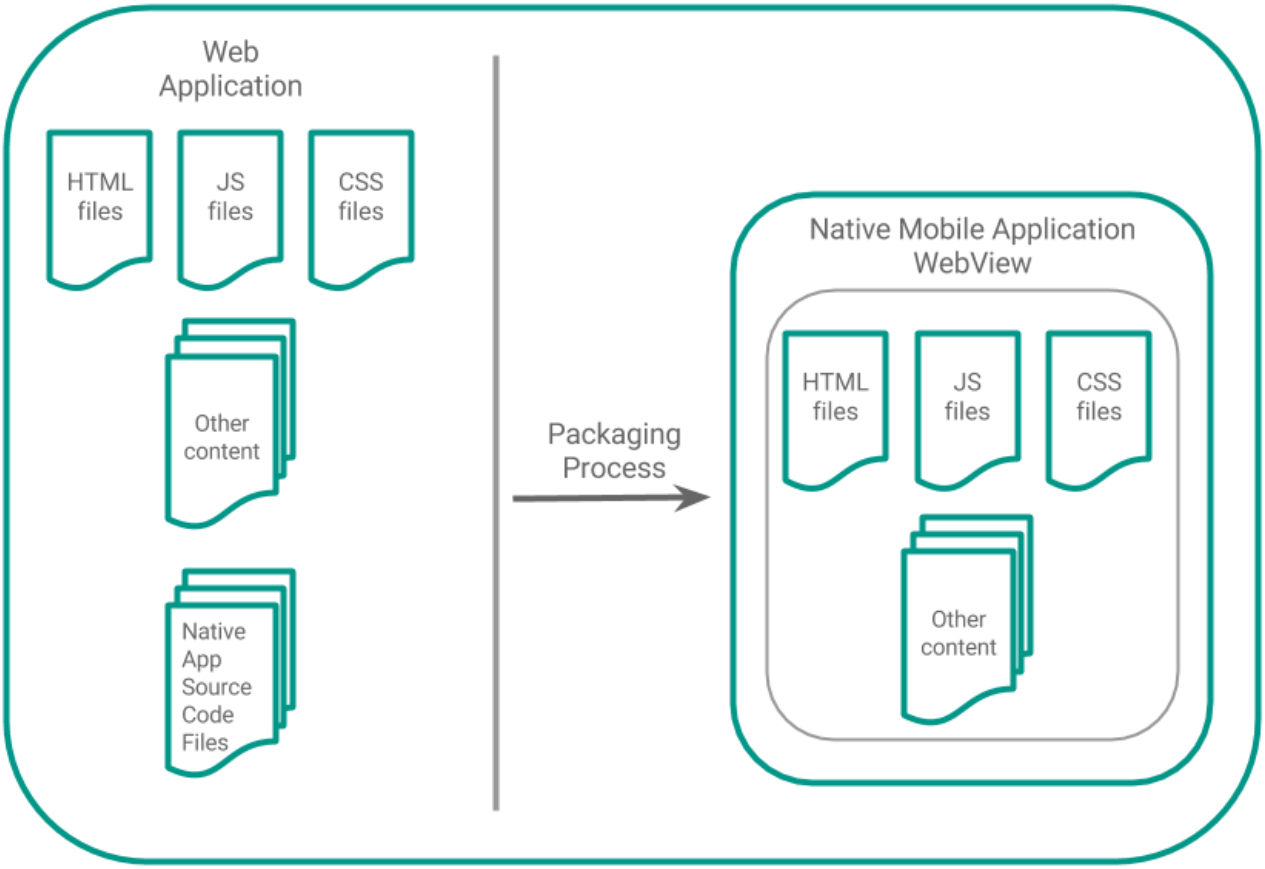

Figure 3. Cordova packaging process [11]

The hybrid app will contain only a WebView which loads the start screen of the application (usually the index.html). The web application in this native container runs without modification, so the source code is not translated to the platform's native language. After the app started the user can interact with the different components like in any other application. If the application is done well then the user will not feel significant difference in the look-and-feel compared to a native application.

One of the most important part of Apache Cordova are the plugins. They are providing access to the native operation system functionality and device capabilities using Javascript. These set of plugins are called Cordova Core Plugins [18], such as:

- Sensors
- Connection
- File Transfer
- Events
- Status Bar
- Notifications etc.

Besides those other third party plugins exist too for features which are not necessarily available in every OS.

3.2.2. Ionic

Ionic is an open source HTML5 mobile app development framework to build hybrid applications [19] which became very popular in the last few years. The framework exists since 2013 (alpha version) but just in 2015 more than 1.3 million apps were created with it.

Ionic provides a set of front end components which - in contrast to responsive frameworks - are native style UI elements for all the different supported platforms. This is one of the main advantages of Ionic, mobile applications which are developed with it will

have the same look-and-feel like native ones which is very important from the user experience point of view.

The framework was built on top of AngularJS, which is a JavaScript framework for developing single-page applications. It is very powerfull via providing a lot of AngularJS directives. These directives extend HTML attributes starting with “ng” prefix. Ionic also requires Cordova as a native wrapper to run the applications. The structure of a hybrid app created with Ionic is shown in Figure 4. It shows that with HTML, CSS, JavaScript and Angular Ionic apps are written which are running on top of Apache Cordova. That’s why the Cordova Plugins can be used to get access to features which would be not accessible from the WebView itself.

Figure 4. Ionic development structure

3.3. Native platforms

3.3.1. Android

Android is a Linux based open source operating system which was developed by Google and the OHA (Open Handset Alliance). It was developed for mobile devices such as phones and tablets and further for smartwatches and smart TVs too.

The platform was designed for the purpose to have an unified open source system for smart phones. The main idea is to have a system which is built on Linux with modifications to be able to handle the specific device hardware. This changed after Google bought the company Android Inc. in 2005 and decided to add a virtual machine above the Linux kernel which is responsible for running the apps and handling the user interface. It was necessary because the device manufacturers can use a lot of different hardware in their devices and it is not sure if an app written in native code (like C) would run the same way on every device. To avoid the problem it needed a Java compiler and for this the Dalvik Virtual Machine (DVM) was designed.

Running an application on Android works in the following way. Applications written in Java first compiled to Java byte code then they are converted from .class to .dex files which is the Dalvik Executable file format. This is an optimized file format which was designed for devices with limited resources.

The architecture of the Android system contains several layers which are shown in Figure 5. The base layer is the Linux kernel, where the different drivers can be found. The kernel communicates through those drivers with the hardware elements such as camera or WiFi. On the next layer the libraries which are necessary for the system can be found. These libraries are written in C++ and they can be reached through the Application framework. The Android Runtime layer contains the Java libraries and the Dalvik Virtual Machine. The last two layers are the Application Framework through the application can

reach the hardware, send notification or run services and the Applications where Android apps can be found.

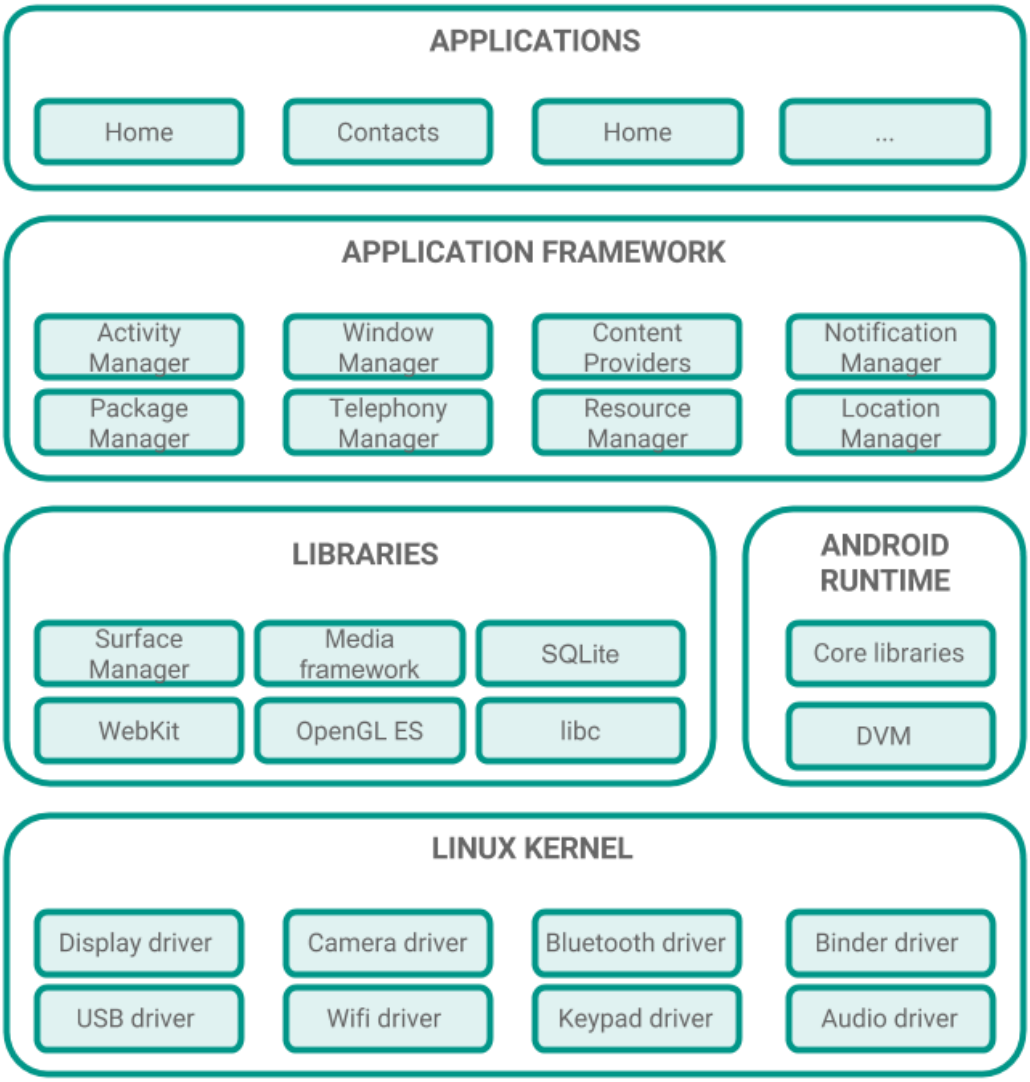

Figure 5. Android architecture

3.3.2. iOS

IOS is the mobile operation system from Apple which was developed for their own products, such as iPad, iPhone and iPod. The operating system exists since 2008, and is the second most popular OS behind Android.

As Figure 6 shows, the architecture of iOS is split up in four layers and starting with the Core OS layer as the base, which provides the core low-level functionalities of the operating system such as Bluetooth, External Accessory Framework and Network Extension Framework. The next layer is the Core Services which contains the basic services for the applications. These services are for example location, iCloud, SQLite and address book. Above this the Media layer can be found. As its name says this layer is responsible for graphics, audio and video services. On top of the architecture is the Cocoa Touch layer which contains frameworks which are responsible for the look-like of the apps. It also provides support for technologies like multitasking, touch input and notifications. [20]

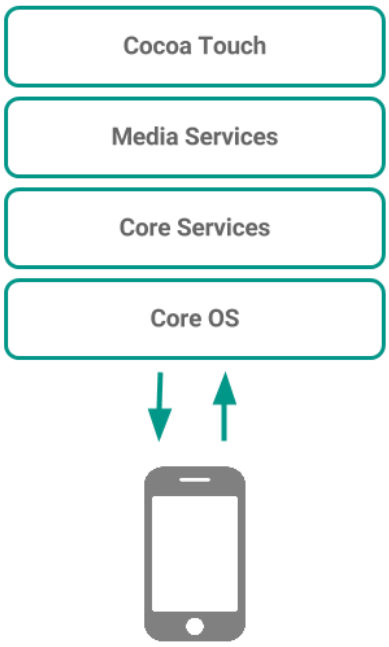

Figure 6.: IOS architecture

3.4. Real time communication

3.4.1. WebSocket

WebSocket is a protocol for bidirectional communication between client and server via a single TCP socket. [21] It is an independent TCP-based protocol which only needs HTTP for the WebSocket Handshake process to opening a connection. Figure 7. shows this handshake process in detail. A client who wants to establish a connection with the server first needs to send an HTTP request. The only difference between this and a regular HTTP request is the *Upgrade* header (shown in Listing 1) which tells the server that the client would like to upgrade the connection to a WebSocket connection.

```
GET ws://websocket.example.com/ HTTP/1.1
Origin: http://example.com
Connection: Upgrade
Host: websocket.example.com
Upgrade: websocket
```

Listing 1. Request of client to upgrade to WebSocket connection

If the server supports this protocol, it will send back a response as shown in Listing 2 which informs the client that the WebSocket connection request was accepted and then the connection between the server and the client is upgraded to WebSocket which uses the same TCP/IP connection.

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Wed, 12 Oct 2016 16:13:22 GMT
Connection: Upgrade
Upgrade: WebSocket
```

Listing 2. Response of server to upgrade to WebSocket connection

Afterwards it provides a persistent connection between the two parties and through this they can send messages any time. Another advantage compared to traditional HTTP

requests is that the parties can send as much data as they want within the same connection. This works because the WebSocket protocol uses a frame-based messaging system, where every message is wrapped into one or more frames. This is better from a performance point of view because compared to a traditional HTTP-requests one doesn't have to send the HTTP-Header with every request. It also means that an *"onMessageReceived"* callback will be called only when the whole message is available for the client.

The connection is alive until either the server or the client is not closing the connection. When this happens the endpoint who wants the disconnect will send a *Close* frame to the other party, which has to send back a *Close* frame immediately. After both sent and received a Close frame the underlying TCP connection will be closed.

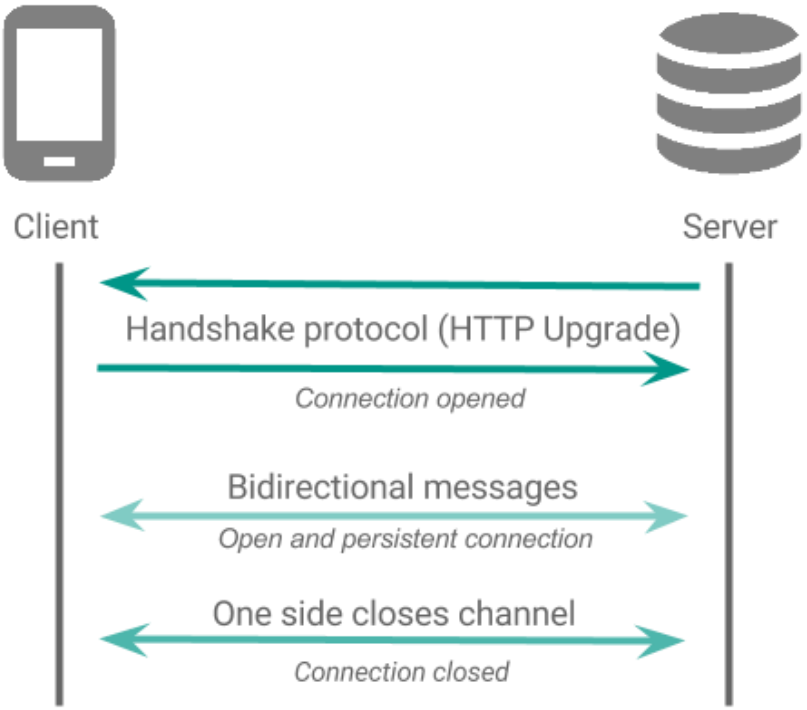

Figure 7 : Connection lifecycle of WebSocket

3.4.2. Socket.IO

Socket.IO is a Javascript library for realtime communication between client and server which mainly using the Websocket protocol. It has two parts, the client side library which is running in a browser and the server side library for Node.js. [22]

Socket.IO is currently very popular which has several reasons. First of all both the server and the client side API is easy to use and easy to set up which makes the development much faster. However Socket.IO is used in a lot of case as a WebSocket wrapper there is much more in it because Socket.IO provides cross-browser compativility which is another important reason why it is so widely used nowdays. Besides its WebSocket support it supports other transport mechanism too. For example AJAX long polling, AJAX multipart streaming and JSON polling.

The communication method is very simple in Socket.IO. It uses so called “rooms” which are different data channels, where the clients can join. When a user is connected he/she joins automatically to a room which is identified by its session id and is unique. Through this a message can be sent to this client.. After a user is connected only two cases are possible, either the server is sending a message through the opened socket to the client, or the client is sending data to the server. Listing 3 below shows how it is done on the server side. The *socket.emit()* method will be triggered when a new user is connected, and it's emitting a message to them.

```
io.sockets.on('connection', function (socket) {
        socket.emit('message', 'You are connected!');
});
```

Listing 3. Emitting message from the server side when a user is connected

On the client side we can receive the message as shown in Listing 4 from the server tagged with the same string identifier like ‘message’ in the example.

```
socket.on('message', function(message) {
        //message received from the server
})
```

Listing 4. Receiving message on the client side

Figure 8. sums up how the message broadcasting is working. The client sends the message to the server which emits it to all receivers, eg. everyone in the same room, except the sender.

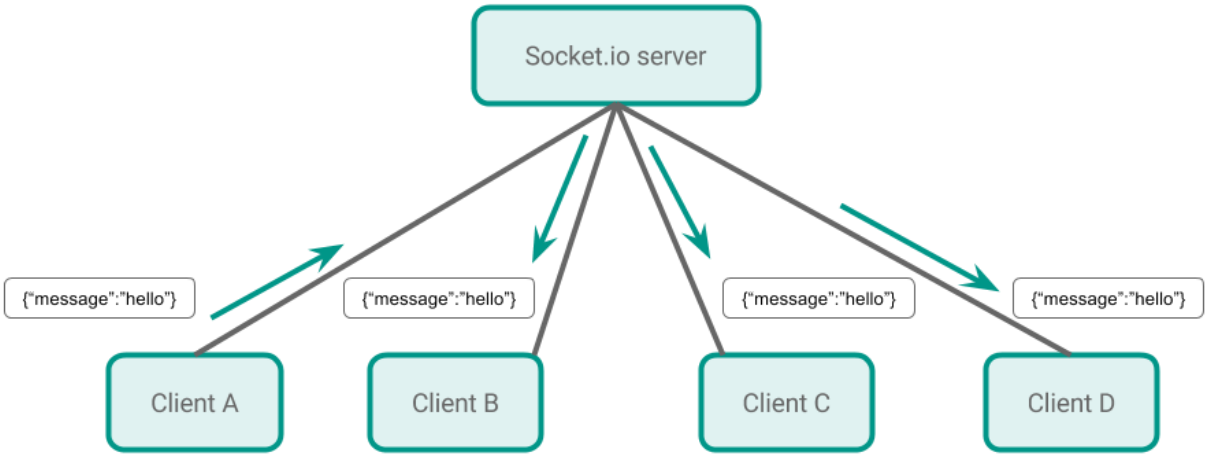

Figure 8. Message broadcasting method in Socket.io

3.4.3. WebRTC

WebRTC is an open source project which provides the opportunity for video, audio and file sharing through web browsers without any additional plugins. It is a set of communication protocols that enables real-time communication over a peer-to-peer connection. [23] Currently the W3C (World Wide Web Consortium) and the Internet Engineering Task Force (IETF) is working on the WebRTC standard. [24]

Although the WebRTC communications such as video chat or conference call is through peer-to-peer connections it still needs a web server to establish this peer-to-peer connection. Figure 9 shows a simplified communication flow between two communication partners. Before the communication between the two clients begins it is necessary to identify each other. This process is called signaling. The first client sends a message to the server with an unique id of the second client with whom the connection should be established. The server forwards this offer to the second client. In case the second client accept the offer, it will send back an answer through the server. Through this signaling both peers can identify each other and establish a peer to peer connection. In Chapter 5. the detailed communication flow of WebRTC is described.

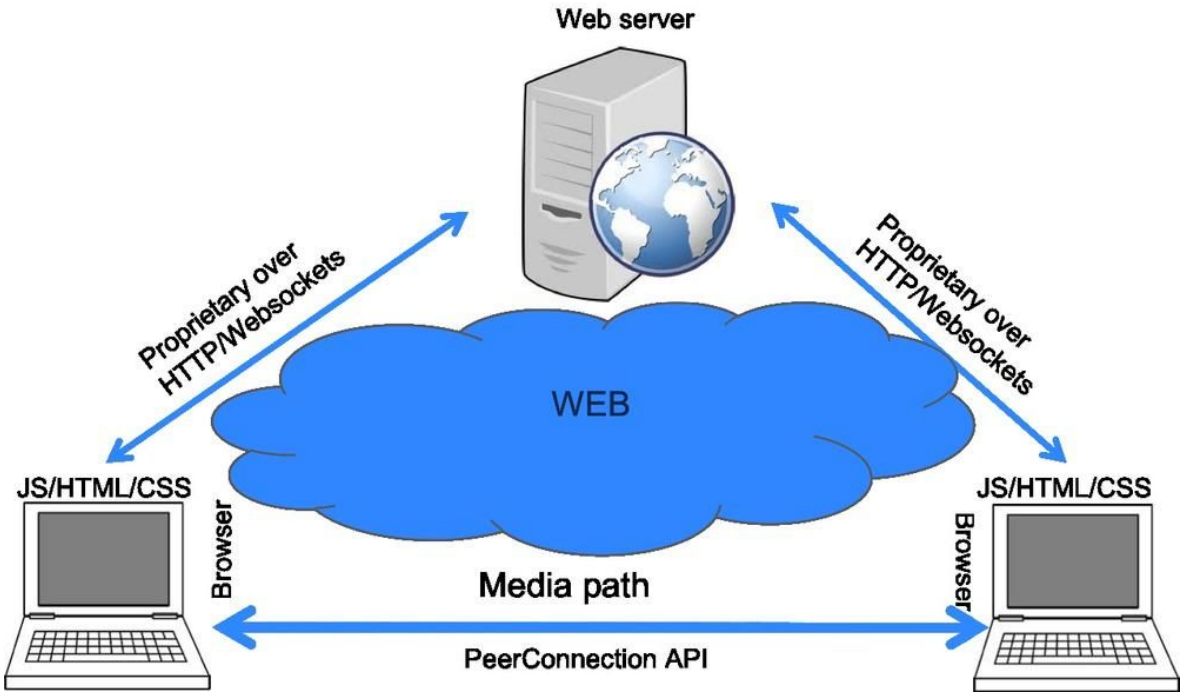

Figure 9. WebRTC communication flow [25]

3.4.4. PhoneRTC

PhoneRTC is a plugin for Cordova which provides a cross-platform implementation of WebRTC. It works on Android, iOS and Browser platforms. The need of using this plugin for hybrid mobile application development is because the Safari browsers still not supporting the WebRTC protocol at the moment. [26]

The advantages of PhoneRTC is the flexibility and that it is easy to setup and use. It does not require one to use a third party server but gives the opportunity to set up your own signaling and TURN (Traversal Using Relays around NAT) servers. The plugin can be easily added to the project with the command below:

```
cordova plugin add https://github.com/alongubkin/phonertc.git
```

Afterwards it can be used on Android, but for iOS platform it needs further modification. This will be covered in more detail in Chapter 6..

3.4.5. FCM

FCM (Firebase Cloud Messaging) or formerly known as Google Cloud Messaging (GCM) is a notification service for mobile and web applications. It allows to send push messages from third party servers through FCM to a client. The advantage of this method is that it is not necessary anymore to send requests from the client to the server periodically and check if there is new data which is a lot more efficient from a performance point of view. It is also important that all apps on the phone which use FCM are using the same connection. This means that the apps don't have to keep alive their connections to their own server which is again saving a lot of resources because just one connection is open and the phone is not constantly waking up and checking if there are new messages.

Figure 10 shows the workflow of FCM. First of all the clients need a Registration ID if they don't own already one. For this they send a request to the FCM-Server and then

get the id in the response. Afterwards this id, which is an unique string, has to be sent to a server which is called *My Server* in the figure and where the server-side logic of the application runs. My Server can use this id to send messages to the client. Then every time the server wants to send a message to the client, it sends a message to the FCM-Server with the registration id of the receiver and with the content itself. These messages will be afterwards forwarded to the target mobile device.

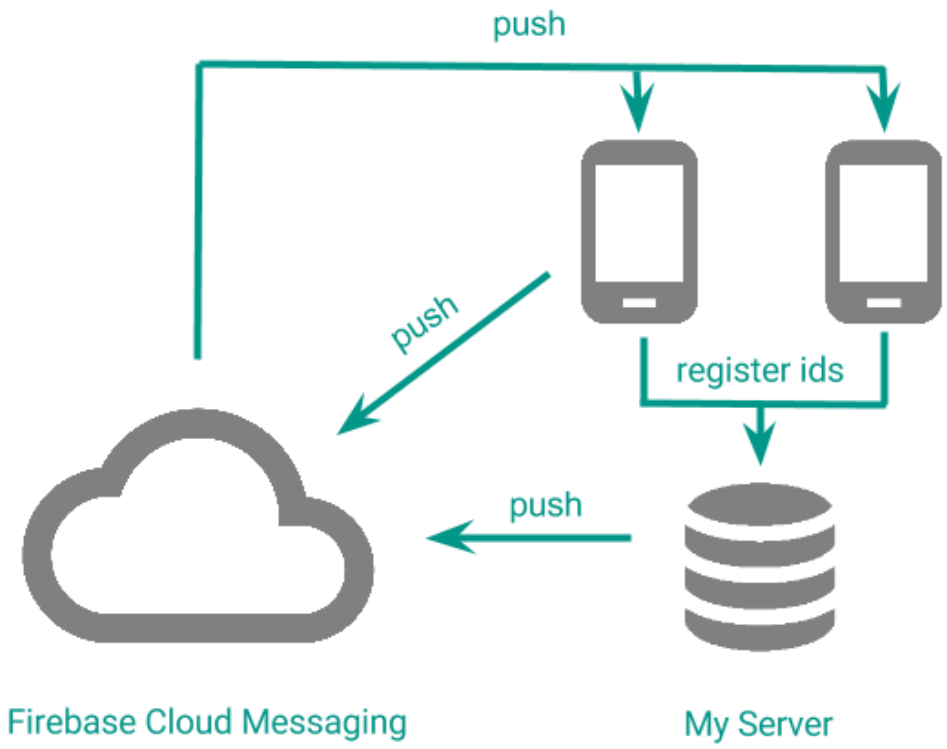

Figure 10. Workflow of FCM (GCM)

4. Prototypes

This chapter will give more information about the prototypes which were implemented for this thesis. Sections 4.1 will show the goal of the prototypes. Afterwards Section 4.2 will give some technical overview about the mobile applications and the server side. The rest of this chapter will describe in more detail the applications for the different platforms, their structure and functionalities.

4.1. Goals

The goal of the prototypes is to show the differences between a hybrid and native implementation of the same application and to demonstrate how they are working with real-time communication.

The main advantage on the side of a hybrid implementation is that there is only one codebase for more platforms which saves time for the developer and also money. Based on this argument it would be really easy to say that this method is obviously the better choice. That's why the purpose of the prototypes is to examine them from different point of views through the implementation to give an overview which helps to give a more sophisticated picture about this two mobile development methods. The results are shown in Chapter 6.

The other goal of the prototypes was to show how real-time communication is working in mobile applications. Because real time communication is representing the main part of the applications, that's why it was important to examine if there is any difference in implementation and usability in the hybrid and native application.

4.2. Project overview

4.2.1. Server side

For the server side development Node.js [27] was chosen, because it was easy to integrate the Socket.IO into it. This had an important role in the decision, as Socket.IO is used for broadcasting post messages in the groups, and also to emitting the direct messages between the users. As an other reason next to the Node.js, it was also very simple to build a REST API with it as it brings already a lot of functionalities for this.

As a storage MongoDB was selected, because it is a NoSQL database which is using BSON (Binary JSON) documents. It makes it easier to persist JSON data, which is used for the communication between the server and client.

The server is hosted on OpenShift [28], which is a PaaS (Platform as a Service), where one can host small projects for free. It also makes the development process more comfortable, because it is possible to push a whole Git repository there and which will be automatically deployed afterwards. It also provides encrypted connection via HTTPS which is important from privacy point of view in a real time communication app.

The most important functionalities of the server side are the following:

- Storing data
- Handling WebSocket connection to communicate directly with the apps
- Providing data to the clients through REST calls
- Signaling for the WebRTC (audio/video conversations)

4.2.2 Client side

The applications are made as prototypes for the GeoGebra Groups. They are not complete applications, only the most relevant functionalities are implemented, which can

be the critical points from a technical point of view of the two types of implementation. These are the real time communication related features, and those which using the native functionalities of the platform such as using the camera, add notifications and opening files with an other native app. The implemented functionalities are the following

- Login
- Join and select Group
- Write posts and comment
- Start direct messaging with a group member
- Start audio/video call with a group member
- Open shared GeoGebra materials
- Local notifications about events

4.3. Setup of the development environment

4.3.1. Hybrid application

For implementing the hybrid application Cordova and Ionic were used. The author decided to use those technologies because they are widely used on the market in the last time. Cordova is currently the most popular cross-platform developer tool [10], and Ionic is also a very popular UI framework for hybrid mobile apps at the moment. [29] It provides a modern user interface and a lot of UI components which gives a native look-and-feel on Android and iOS.

4.3.1.1. Development requirements for Android

Developing Cordova apps for Android needs the Android SDK to be installed on the computer. Android SDK can be installed on the most commonly used operating systems, such as Windows, Linux and OS X. If it is necessary for the project to use native Android tools to be used then the best choice is to download Android Studio, otherwise the

standalone SDK is enough too. [30] For setting up the development environment the following steps are needed:

1) Download Oracle JDK and set up the environment variables
2) Install the Android SDK Tools
3) Download from the SDK Manager the appropriate build-tools, SDK platform and Support Repository for the target API level
4) If it's needed, setup a virtual device in the AVD Manager

4.3.1.2. Project setup.

First one need to create a new application with the start command and then specify which template should be used for it. The currently available Ionic Templates are the following: *blank*, *tabs* and *sidemenu*. Afterwards the project is initialized. Android as a platform should be added, then one can build and start the new project. If an Android device is connected the app will run there by default, anyway it will use a previously configured virtual device. The following commands show how to start an Ionic app on Android.

```
$ ionic start example blank
$ ionic platform add android
$ ionic build android
$ ionic emulate android
```

4.3.1.3. Development requirements for iOS

Developing Cordova applications for iOS is only possible on OS X operating systems because the SDK is just available for Apple's operating systems. Once a device runs OS X version 10.9 or higher which includes the OS X Software Development Kit

(SDK) XCode can be used as an Development Environment or any other IDE. [31] The workflow of creating and setting up a new project is the same like for Android which was described in the previous section (4.3.3.3.) The only difference is that iOS has to be added as platform, and then the project can be built on a simulator or on a real device.

4.3.2. Native application

For native Android development the Android SDK is needed which is also needed for Cordova for Android, as already discussed in section 4.3.1. Similar to that the development can be done on Windows, Linux or Mac OS X because the Development SDK is available for theses operating systems. For programming the best choice is to use the Android Studio IDE, as it is the officially supported IDE for Android development which provides a lot of features. Through Android Studio it is easy to download the necessary tools for the development from the integrated SDK Manager.

The setup of the development environment is pretty simple, and similar to the installation process which was described section 4.3.1.1. The only difference is, that the second step is, to install Android Studio instead of downloading the standalone Android SDK. After it is installed, a wizard helps to set it up.

Creating a new Android project is also very straightforward. By clicking on File>New Project, a wizard window opens and leads through several steps where one can customize it. At this point for example the API-Level can be chosen, and also the UI layout of the *MainAcitivty*, such as tab layout, navigation drawer or a Login Activity.

4.4. User Interface

In this section the user interface will be described with the most important views parallel for the native and the hybrid implementation. On the left side the hybrid

application can be seen which is running on Android, and on right side the native Android app. The purpose is to lead through the functionalities of the prototypes and to show the differences between the simple UI elements of both approaches.

The app starts with a simple *Login* screen which is shown in Figure 11 where the user has to give the credentials, which are later stored in the system. After the login was successful the user get to the list, where one can choose a GeoGebra Group to join. Here those groups are listed in which the user were already invited.

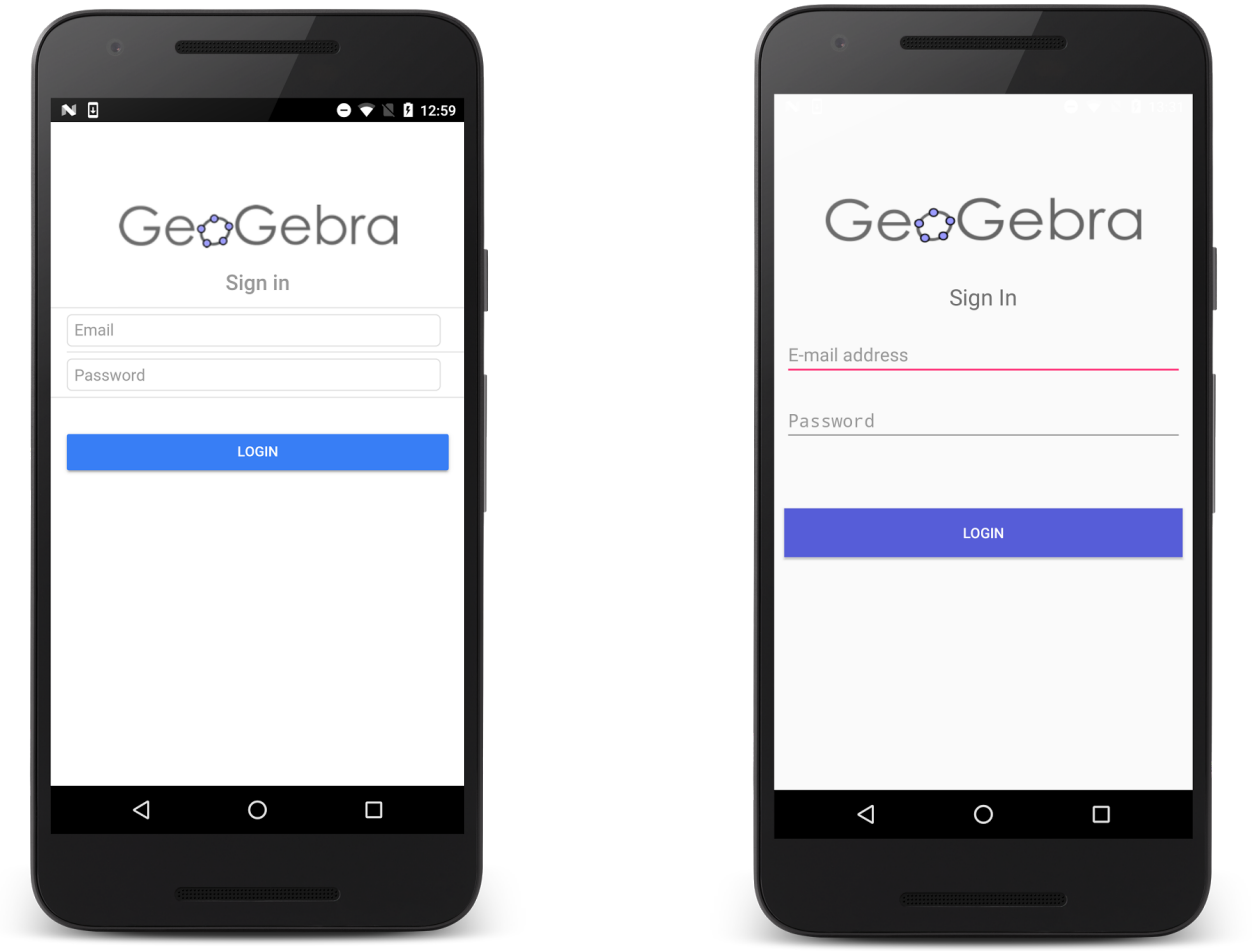

Figure 11. Login screen of the hybrid (left) and of the native Android (right) app

After the user joined to a group, the *Posts* tab is opened where all post messages and comments are shown from all members. (Figure 12.) On the top of this view the user can also post a message by clicking in the input field. After the user clicked in there, it is extended similar to how it works on the web version of GeoGebra.

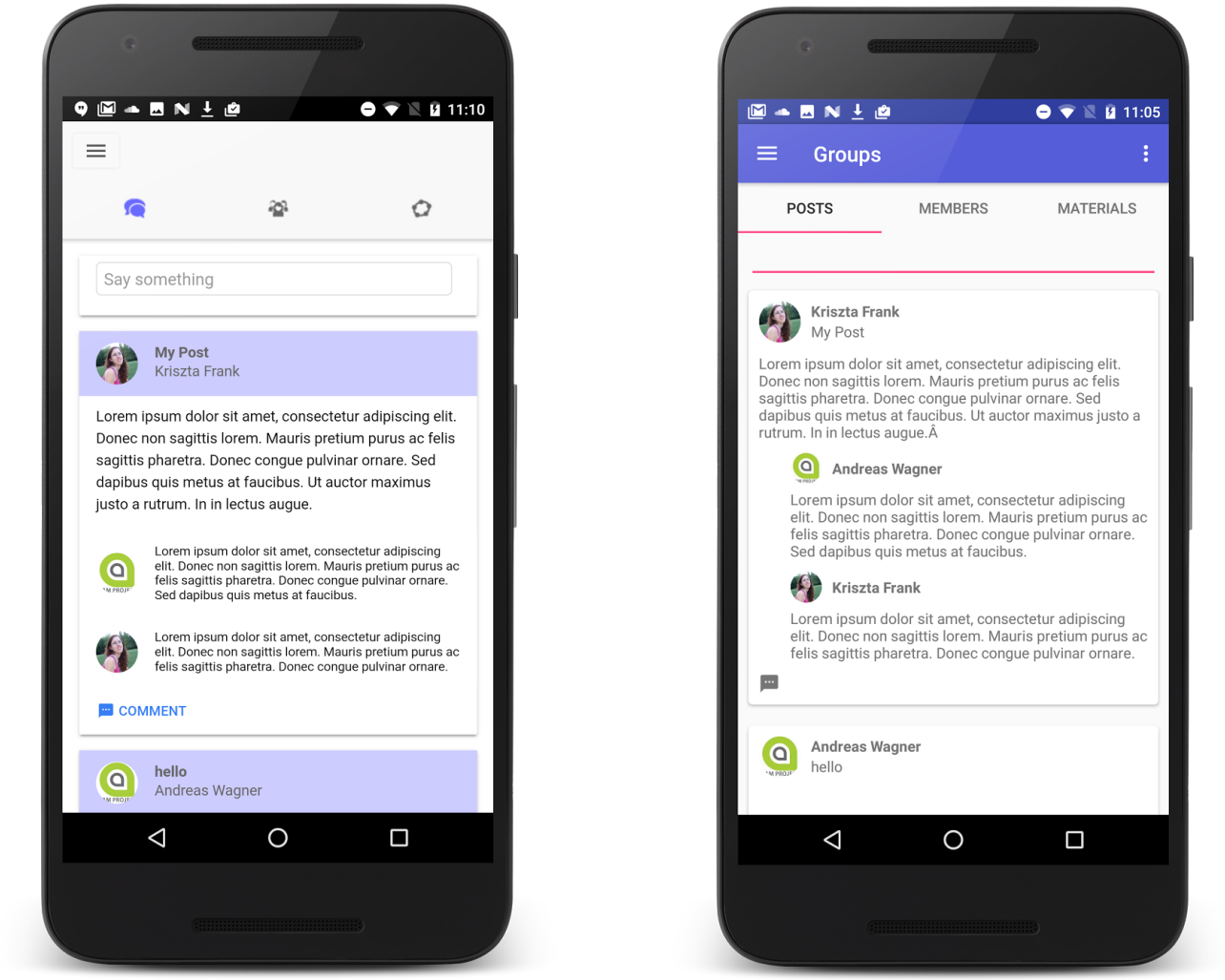

Figure 12. Posts tab of the hybrid (right) and of the native (left) application

The next tab contains the list of all the group members. (Figure 13.) Here the user is able to start direct messaging with any member by clicking on their name.

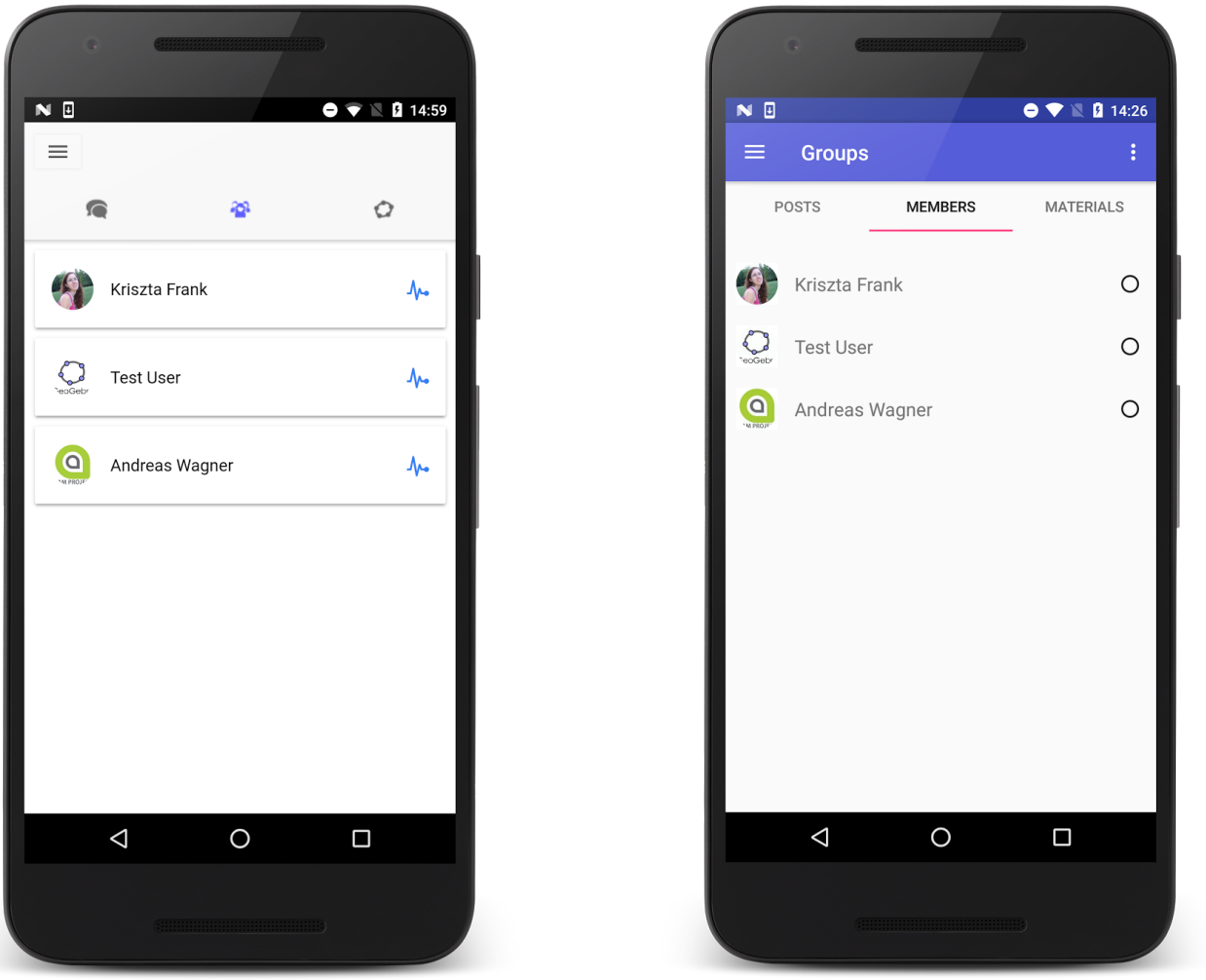

Figure 13. Members tab of the the hybrid (left) and of the native (right) application

In the direct messaging view the previous messages are shown between the users (Figure 14). On the bottom of this view there is an input field, where the user can type the message, then with a click on the button next to send the text to the receiver.

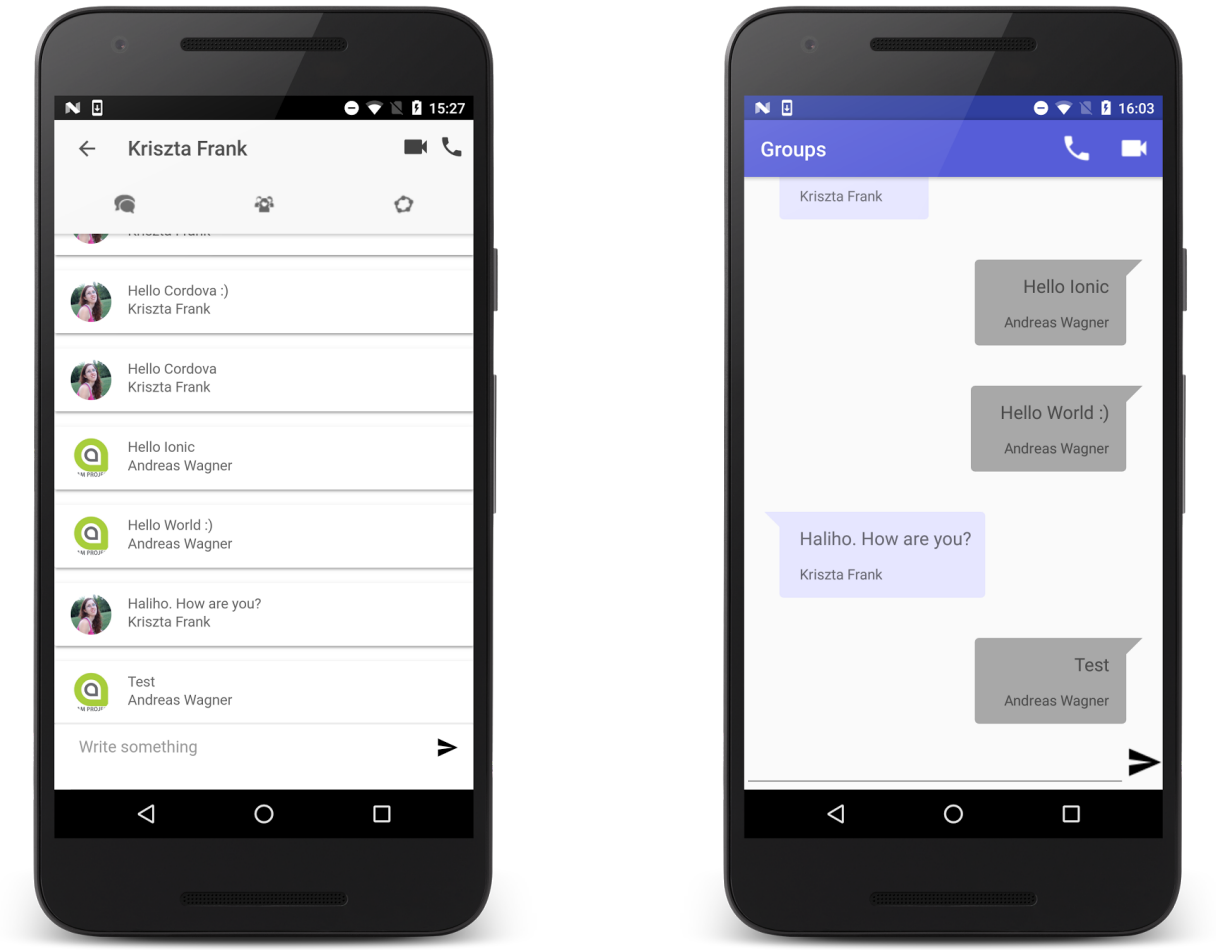

Figure 14. Direct messaging view of the hybrid (left) and of the native (right) app

In the toolbar there are two buttons for audio and video calling. When the user clicks on the video icon, a calling view will appear, and as soon as the other party accepted

the calling request, on both sides the remote video becomes visible on a large surface and a small one with the local video stream. (Figure 15.)

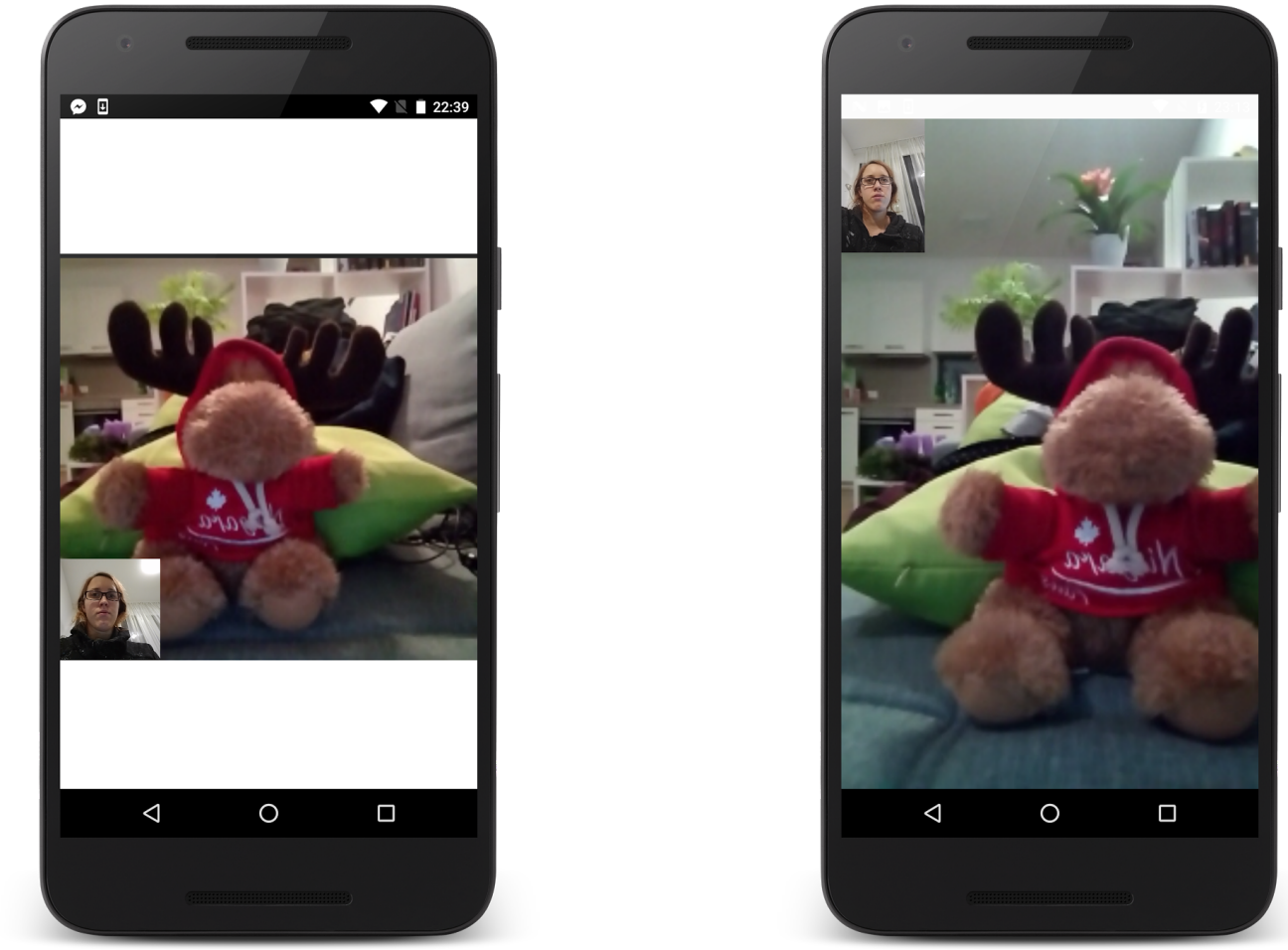

Figure 15. Video call on hybrid (left) and on the native Android (right) application

The last tab in the application is the *Materials* tab. Here the GeoGebra materials are listed which were shared in the group as it can be seen on Figure 16. By clicking on one of

them, the material will be downloaded, and then the application chooser pops up, where it can be selected which application should open it, such as 3D Grapher or the Graphing Calculator.

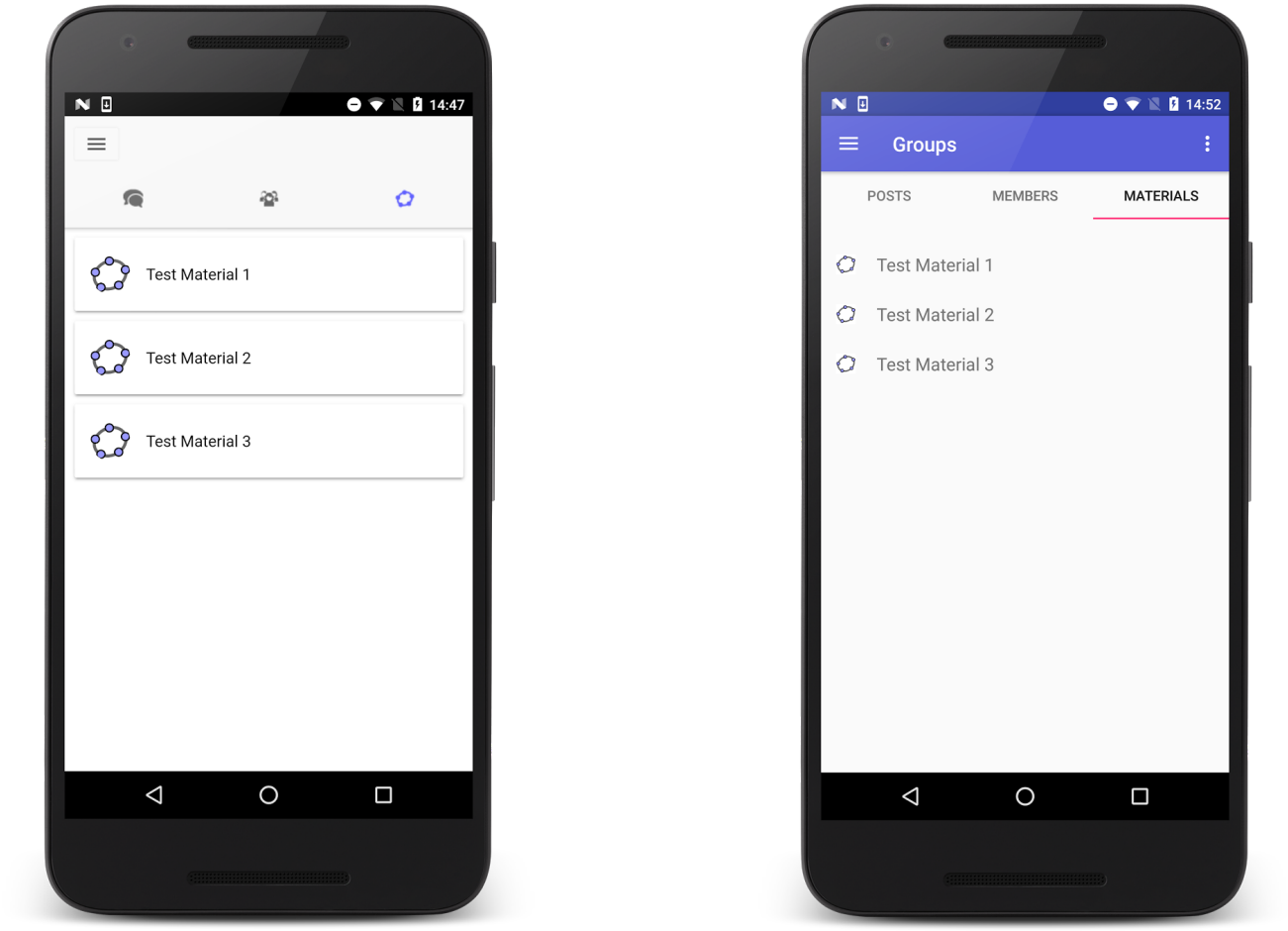

Figure 16. Materials tab of the hybrid (left) and of the native Android (right) application

5. Real-time communication

As mentioned in section 3.4.3., this chapter will show in detail the real-time communication mechanism which was used in the prototypes. Section 5.1. and 5.2 will give detailed overview about the architecture of WebRTC and its working mechanism. Then in section 5.3 and 5.4 different signaling methods and detailed description about the information exchange between the peers will be shown which is a crucial part of WebRTC. Afterwards the experimental results according to the different methods which were used in the prototypes (5.5.) will be presented. The conclusion of the usage of WebRTC combined with different technologies will be summarized in section 5.6.

5.1. Overview

WebRTC (section 3.4.3.) was released by Google in 2011 as an open-source project for real-time communication between browsers [23]. Since that time it became very popular. WebRTC API is currently standardized by the W3C and the WebRTC protocols by IETF. Based on the prognostication of Disruptive Analysis [32] which was made in 2015 the number of WebRTC supporting devices will grow exponentially as it shown in Figure 17.

The same tendency can be noticed by investigating which companies are using WebRTC based solutions in their communication applications. Most important are maybe Google Hangouts and Facebook Messenger but also other ones use it either in browser or in an app such as Appear.in, Slack and Hipchat.

The popularity of WebRTC has several reasons.. First of all it can be used for audio and video calling, data transfer and conferencing between browsers without any additional plugins. At the moment a big percentage of browsers support it, and this will still grow in the future as it will be probably also soon integrated in Safari. (At the moment the

WebRTC feature is still “In development”). The current state can be checked here [33]. Besides the browsers WebRTC also supports Android and iOS. This simplifies the development process because one just needs a WebRTC enabled browser, or app which is independent from the operation system.

Another important point is, that it is free to use. This two things significantly reduce the costs of any real-time communication application development, because there are no additional cost for using a special plugin or maintain a customized solution.

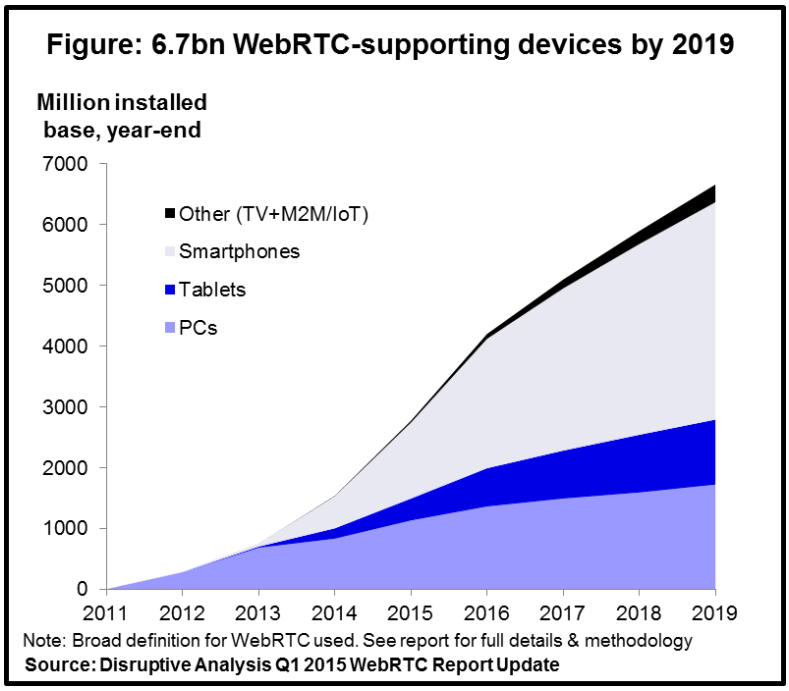

Figure 17. WebRTC supported devices by 2019 [32]

5.2. Core Architecture

The architecture is structured in multiple levels. On the top the WebRTC web API itself can be found which is being standardized by W3C. This API can be used by the developer directly. Below is the session management and the abstract signaling layer, which relies on libjingle [34]. Libjingle is an open-source project which is responsible for

connecting the peers. The last layer contains a collection of engines and codecs for audio and video calling and a set of components for the transmission for example: STUN,TURN and ICE which will be discussed in more detail in the next section. Figure 18 summarize the structure which was described in this paragraph..

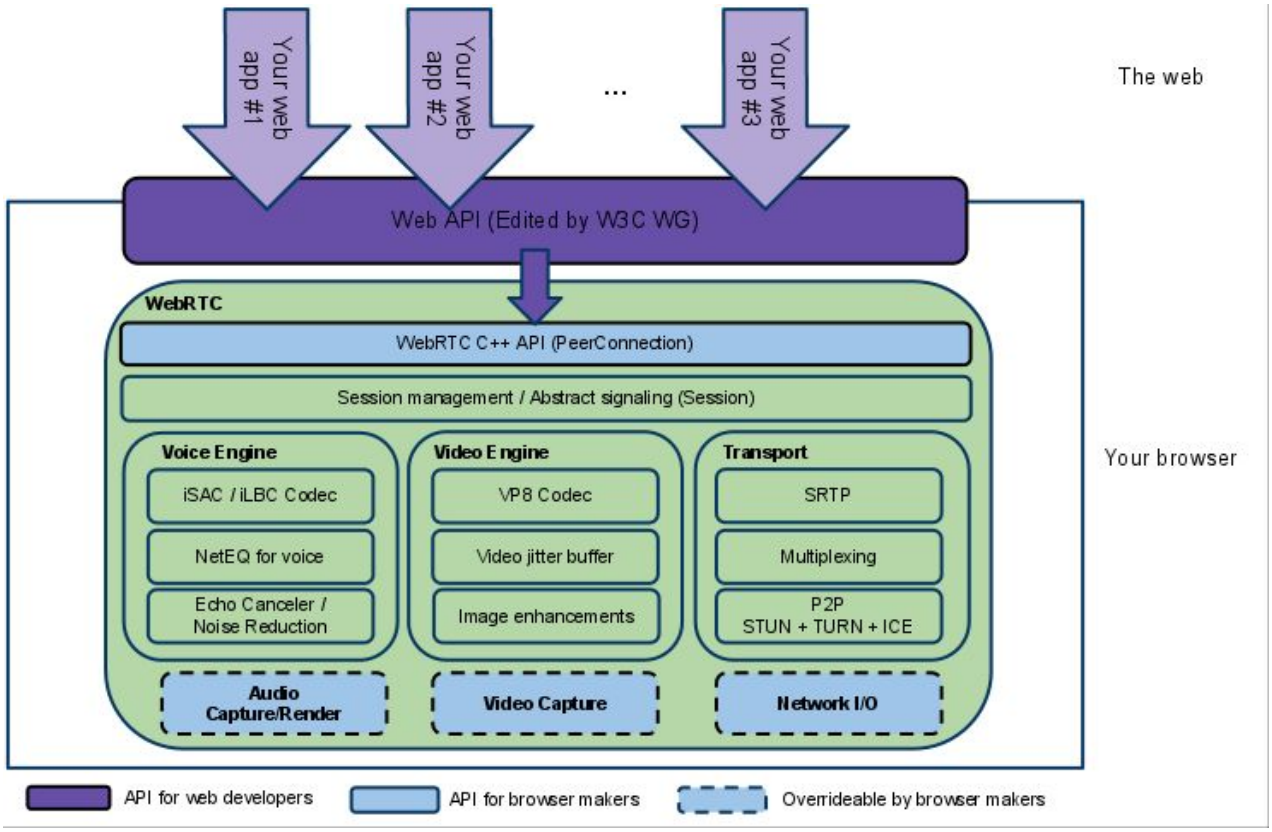

Figure 18. WebRTC architecture [35]

5.3. Signaling

As already explained in Chapter 3, signaling is a mechanism where the peers exchange information to be able to create a peer-to-peer connection for the audio/video call. One of those information is network data. The client who wants to send an offer has to provide its public-facing address and the port. The problem is that if this client is behind a NAT (Network Address Translation) then it doesn't have information about the external IP. Therefore the client needs to use a STUN (Session Traversal Utilities for NAT) server. The

purpose of a STUN server is very simple, it just checks from where the request came, and it sends back the client's external network informations like the IP and the port. Afterwards this information can be sent to the target client via any signaling mechanism as it is shown in Figure 19.

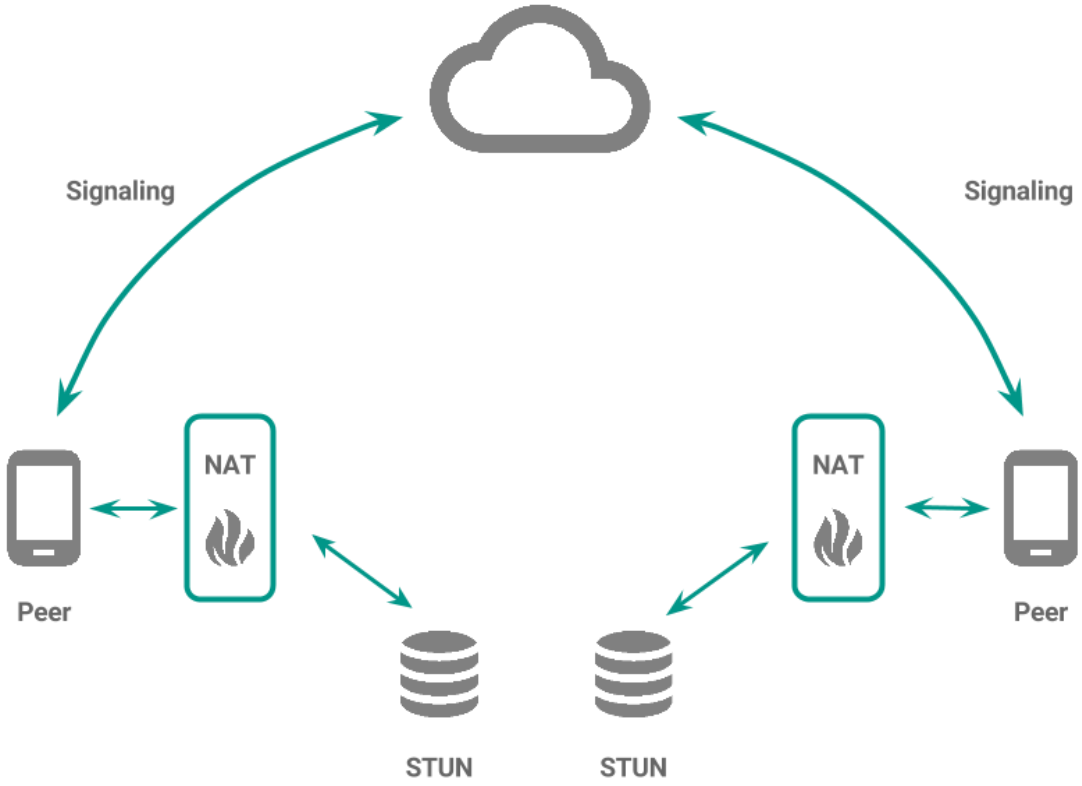

Figure 19. Signaling mechanism

As it was written above the network information is just one part of the data which has to be sent to the other party. To establish a P2P connection further information is needed, such as [34]:

- Types of media (audio/video)
- Network transports
- Used codecs and codecs settings
- Bandwidth information
- Session control messages

After the client has all the data, it just has to exchange this with the other client which is called signaling. However the signaling mechanism is a very important part of the workflow of WebRTC, it is not specified in the API and in the WebRTC standard neither. This means the developer have to decide which kind of signaling mechanism should be used and implement the transmission channel. This gives a lot of flexibility into the developer's hand but also responsibility to choose the best one. Of course, it's not always so easy to choose a signaling method because in a lot of cases WebRTC is just integrated in an existing project, where it is the easiest to use an already implemented transmitting protocol. But it can be still useful to check different ways of signaling mechanism, that's why the author implemented the WebRTC with two different signaling methods to be able to compare them based on some properties, like latency and reliability.

5.3.1. Sockets

For the first signaling method Socket.IO was used, which was introduced in section 3.4.2. The workflow for starting a video call is the following:

- When Alice starts the app, she is connecting to the server by knowing its address and port
- After the handshake protocol a bidirectional connection is established between them
- As a result Alice is successfully connected, her (email,sessionID) pair will be saved
- Alice selects a group and join by clicking on it, which send an *"all_group_members"* request to the server, which returns with a JSON containing the name and email address (which has to be unique) for every group member
- Alice saves this list of users
- When Alice would like to start a video call with Bob, she clicks on the name, and emits a socket message to the server with Bob's email and with other additional information, for example:

```
this.socket.emit("sendInitMessage", callingToEmail, addInfo);
```

- When the server received this "*sendInitMessage*" event, it sends a message to Bob, as his sessionID is saved with his email. The forwarded message will be almost the same, except the "*callingToEmail*" parameter will be changed to Alice email, which the server can get through her sessionID.

- When Bob received this message, he is doing the same like Alice did, so they both shared the necessary information with each other to start a P2P connection.

5.3.2. FCM

For another implementation of the signaling FCM (Firebase Cloud Messaging - section 3.4.5.) was used. The idea behind it was to examinate if there is any noticable difference compared to the Socket.IO implemetation. As it was already described, FCM uses push messages but the device needs to maintain only one connection to the FCM cloud to get those messages for every app. This is certainly more effective for example from the battery life point of view, and also simplifies the developer's job, because the system takes care of the connection itself. From an other side, it is interesting to examine if the functionalities of the app will work the same, or some latency can be noticed. Below the workflow with FCM as signaling method will be described:

- When Alice first time uses the app, the *getToken()* method it called, which takes care of the generation and registration of the token, and it returns with an unique id
- Alice stores this id, then send it to the server which also saves it

Afterwards the logic of joining a room and store the members list is the same as descibed above in the Socket.IO workflow.

When Alice wants to call Bob, and click on his name the signaling is done as the following:

- Alice sends an HTTP request to the server with Bob's email address and the other additional informations for example SDP (Session Description Protocol), which will be described later.

- When the server receives Alice's message, it will first search for Bob's token id stored in the database, then build the FCM message, add the receiver's token id and the information got from Alice, and send it through FCM. The structure of the message is the following:

```
var message = {
  to: toRegid,
  collapse_key: 'call',
  data: {
    callinto: fromEmail,
    content: messageFCM
  }
};
```

- When the message arrived to the FCM server it will be sent to the corresponding device based on the token id
- When the messages arrived to the device the FirebaseMessagingService's *onMessageReceived()* method will be called, and then the event can be handled.
- Finally Bob will do the same like described above. As result of the successful signaling process they can start the P2P connection

5.4. Information exchange

After a signaling method is implemented, the next step is to exchange information such as: bandwidth information and metadata which was also mentioned in the previous section. For describing this set of information, WebRTC is using SDP (Session Description Protocol) and offer-answer model. WebRTC also uses the RTCPeerConnection API which is responsible for the whole peer connection, which will be described later and also to handle this offer-answer model of the SDP, which is working like this: [34]

- First Alice creates the offer (*createOffer()*), and set it as her local description (*setLocalDescription()*), and sends the session offer to Bob (via the implemented signaling method)

- When Bob receives Alice's offer, he set it as his remote description (*setRemoteDescription()*), then create an answer (*createAnswer()*) SDP description, set is as his local one and send it back to Alice
- Finally when Alice receives Bob's session answer, she sets it as her remote description

Figure 20 is summarizing the workflow described above.

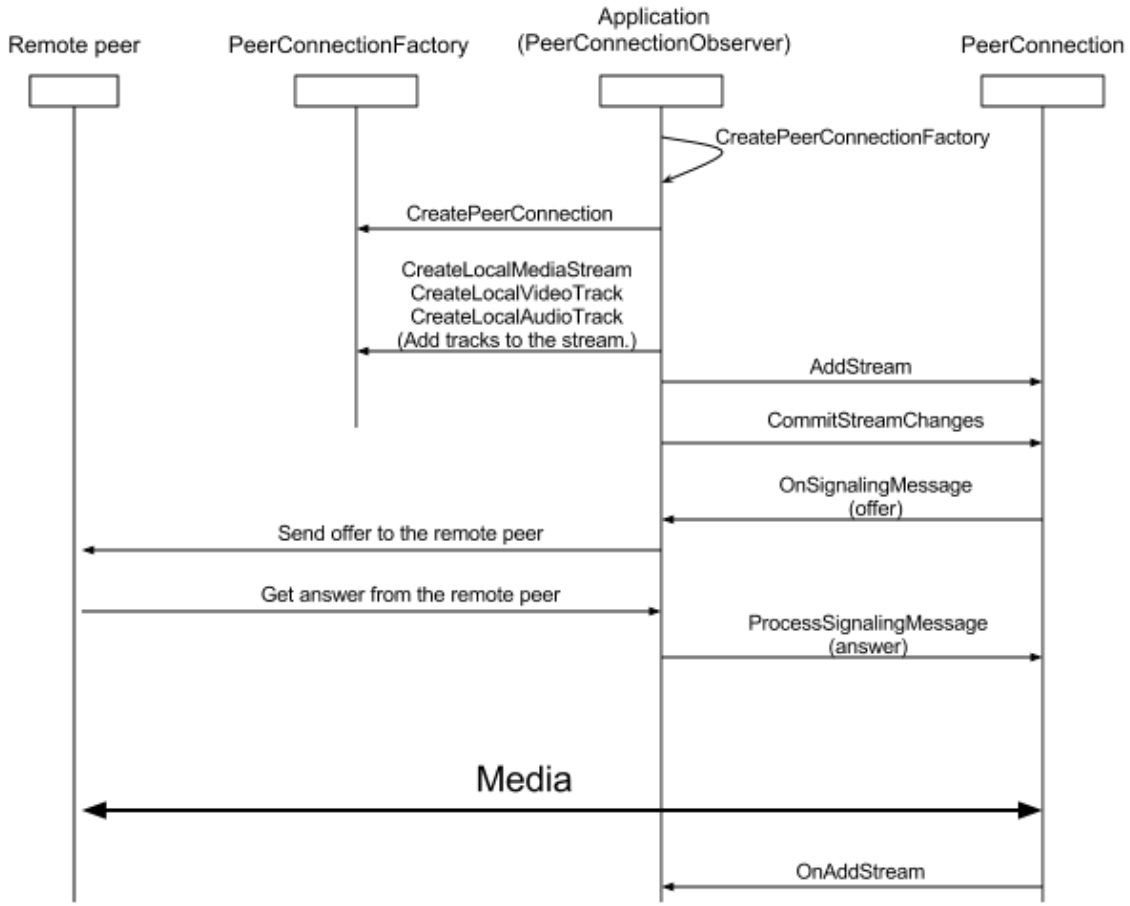

Figure 20. Working flow of setting up a call [36]

Besides the SDP, if a peer wants to connect to another peer, it also needs to share certain network information. For this the WebRTC using ICE (Interactive Connectivity Establishment) framework. The ICE's job is to find the best way to connect the two parties. When the peer is behind NAT then the ICE using a STUN server, and when also this fails, it using a TURN server. (Traversal Using Relays around NAT). TURN servers are the fallback option when the direct peer-to peer connection is not possible, then the communication is going through this server. It is important to note, that in contrast with the STUN servers, TURN servers use a lot of bandwidth, which can be expensive.

When a PeerConnection object is created, a list of ICE servers has to provided which contain STUN and TURN servers. Adding an ICE server is easy, because there are publicly available ones, for example from Google . One can find also a list of free and public TURN servers, but they are not always available, so their service is not reliable which would be the most important. That's why for the prototypes a TURN server from Xirsys[37] was used. Xirsys is providing STUN and TURN server hosting for WebRTC applications.

Figure 21 shows the whole workflow of the WebRTC which was discussed in the previous sections in detail.

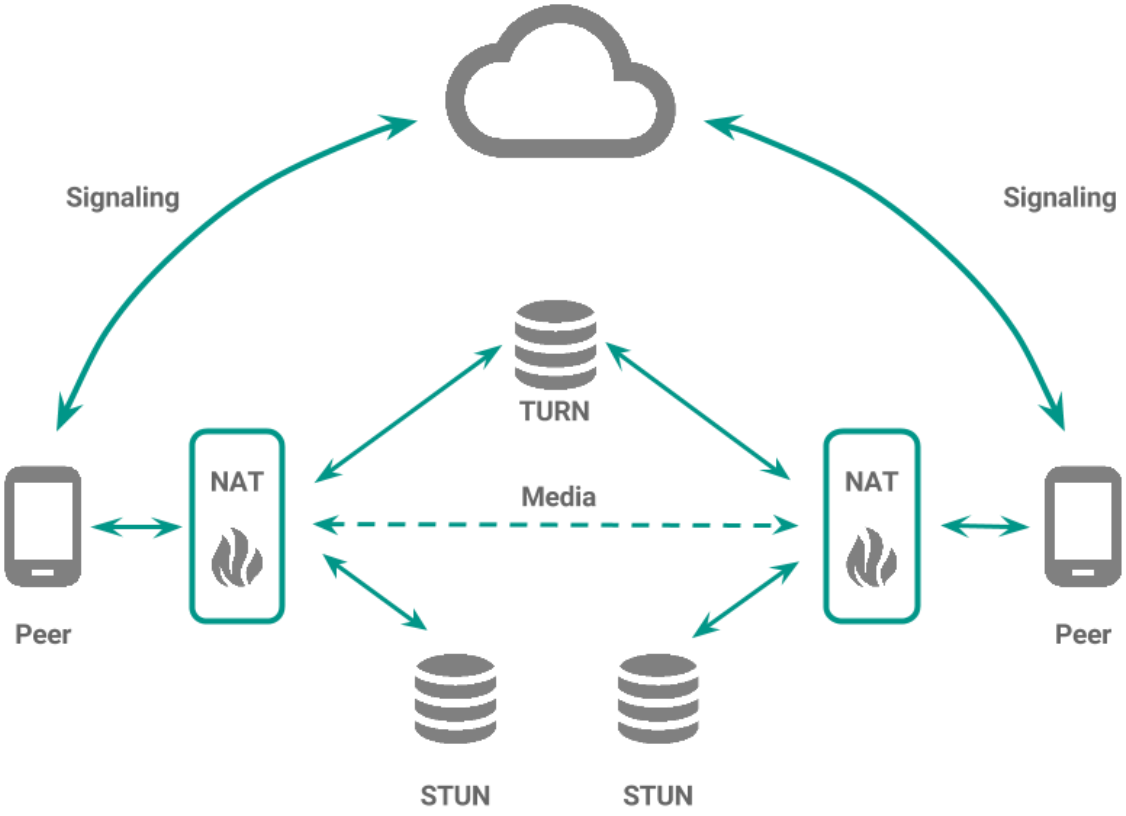

Figure 21. WebRTC workflow

5.5. Experimental results

5.5.1. Integration of WebRTC

For the implementation of WebRTC in the prototypes two different approaches were chosen. For the hybrid application the PhoneRTC Cordova plugin was used which allows to add WebRTC audio and video chat to the app. It provides a Java and Swift implementation for Android and iOS, which can be used in the project by adding the plugin:

```
cordova plugin add https://github.com/alongubkin/phonertc.git
```

For the Android native app the Libjingle artifact from Pristine [38] was used which provides the compiled WebRTC source code. After including it to the project's gradle file the org.webrtc library can be used:

```
compile 'io.pristine:libjingle:11139@aar'
```

However in the hybrid project a "ready to use" plugin was used for comparison with Android where the WebRTC communication had to be implemented from scratch, none of the implementations was straightforward to use. In case of the hybrid application including the plugin is very easy and the creator of the PhoneRTC also provides a step-by-step tutorial for the usage which simplifies the development process. The problem is that this is a third party plugin and there isn't a whole developer team behind it, that's why it can be buggy and less reliable. During the implementation of the hybrid prototype also several errors occurred. In such a situation the main problem is that the developer can not know for sure if there is an error in the application source code or in the plugin itself. In case of the hybrid project it took a lot of debugging, to discover that there was an error in the implementation of the plugin for Android. Afterwards to find the exact problem and fix it wasn't too difficult because the author has knowledge about Android development. From the other side one can choose to implement a hybrid app, because no Android or IOS specific knowledge is needed. For these developers debugging the problem in a device's native source code can be difficult.

In the native Android project the implementation of the WebRTC audio and video communication was not that straightforward neither. The problem in this case was the complexity of the logic and the lack of documentation which was found in the Internet. The developer really has to understand the whole WebRTC workflow to be able to implement it. The most challenging in this case is to implement an own *SDPObserver* and *PeerConnectionObserver* which are responsible for the signaling, for the P2P connection and for the ICE candidates.

As a general complication for both implementations which is still important to mention is the debugging. When the WebRTC is not working properly and a problem has

to be found it is necessary to debug both device at the same time, because one can not know on which side the error occurs. To handle multiple devices for example in the Android Device Monitor needs a lot of resources of the computer, time and unfortunately often happens that connection between the laptop and the device breaks.

5.5.2. Comparison of signaling mechanisms

Signaling mechanism is a very important part of the WebRTC communication as it ensures that the required information are exchanged between the parties to be able to establish a peer to peer connection. According to the nature of signaling, the most important key points of efficiency are the latency and the reliability. There can be other attributes to compare as well but these are the major ones, because in case of calling someone the most important that the signaling message arrives to the receiver and that it arrives as fast as possible. In this section the author will compare the two different signaling methods (FCM, Socket.IO) to investigate which is the better based on these two properties.

5.5.2.1. Latency

For testing the latency of sending the signaling messages two devices were used also a middle and a high category one. During the tests the devices were connected to the same WiFi, and every other application was killed on the phones to reduce incorrect results. The test devices are listed below:

- Nexus 5X - Android 7
- Umi Touch - Android 6

The test results were created by programmatically measuring the time difference between sending and receiving the signaling message. This time represents how long it

took until calling offer reaches the target device. During the tests the latency was investigated in both directions. The results were analyzed per pairs and also all together.

Figure 22 shows the latency between the Nexus 5X and the Umi Touch device in a given time period for both types of signaling methods. It means the tests were done in the same time interval to provide the most similar circumstances. It can be seen on the graphic that both signaling methods gave similar results. The difference between the median of the values is only few hundred milliseconds and even the highest value of the FCM signaling is lower than 0.7 second. However Socket.IO provided better results in this case, both methods have low latency values as a few hundred miliseconds is not noticable for the user. The graphic also gives information about that signaling with socket.io performs nearly the same on both device in both direction.

As the latency can depend on various things, for example state of the current internet connection, on the remote servers or on the mobile device itself, a second test was done on another day. The purpose of this seconds test was to create the same circumstances as before and provide control data next to the original dataset.

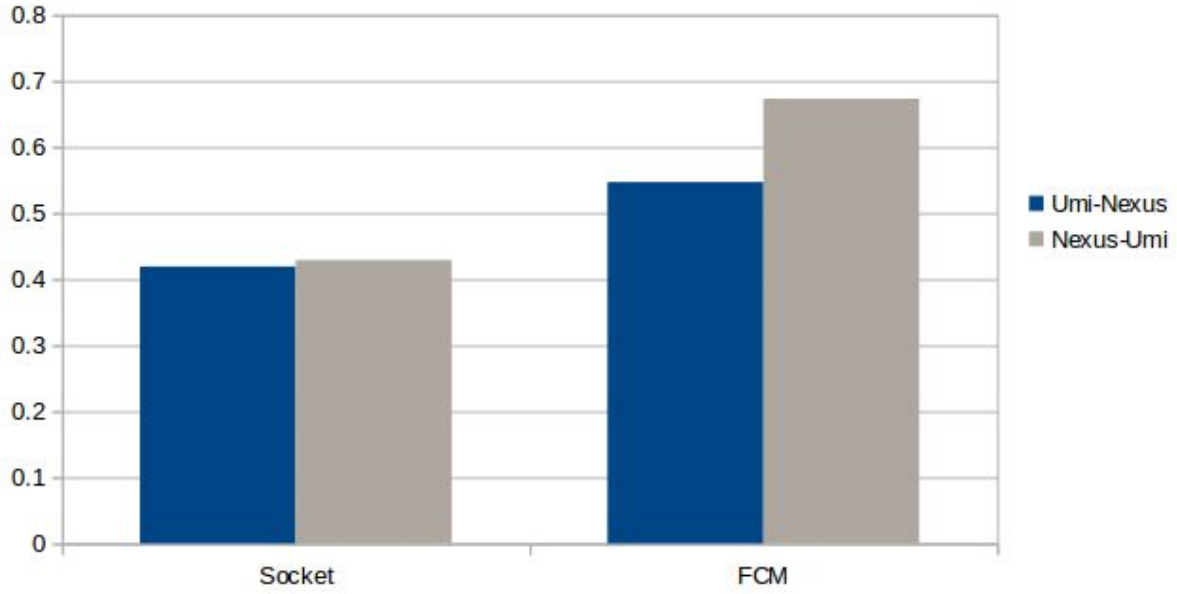

Figure 22. Signaling latency between Umi Touch and Nexus 5X in one time period

Figure 23 and Figure 24 shows both the original dataset and the control dataset together separately for both signaling methods. On both graphics the highest and the lowest values are highlighted with grey color and the bold horizontal line gives information about where is the mean of the data. First it can be already seen that the mean value is lower for Socket.IO signaling, and this value is matching with the original dataset. This means the measurements during the control test gave very similar results like in the first case, which implicates that this signaling can provide a standard performance. By looking at Figure 24 we can confirm that these results are also matching with the first measurements. The mean value is still around 0.6 which is slightly more than for the other signaling method. However it is also clear from the diagrams that the FCM values show much higher variance. Both the highest and the lowest deviance with FCM is twice as high as for Socket.IO. The reason of this can be that handling the FCM connection can be device specific and the whole latency of the FCM is heavily relying on the corresponding FCM server which used to forward the message.

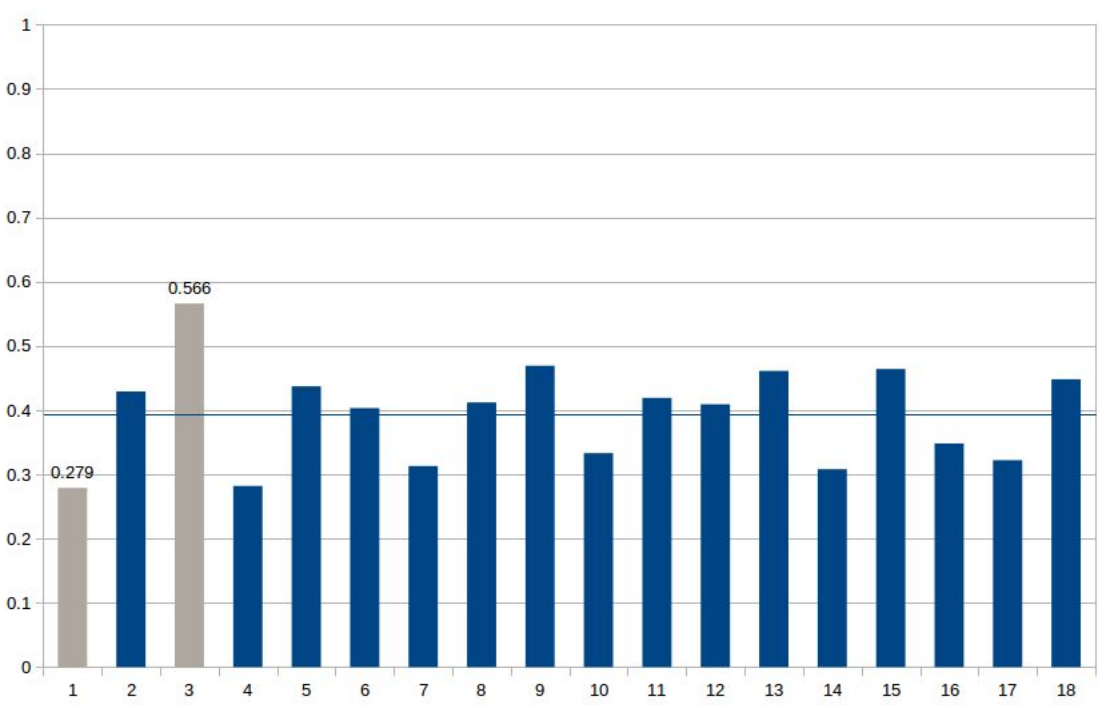

Figure 23. Signaling latency with Socket.IO in various time periods

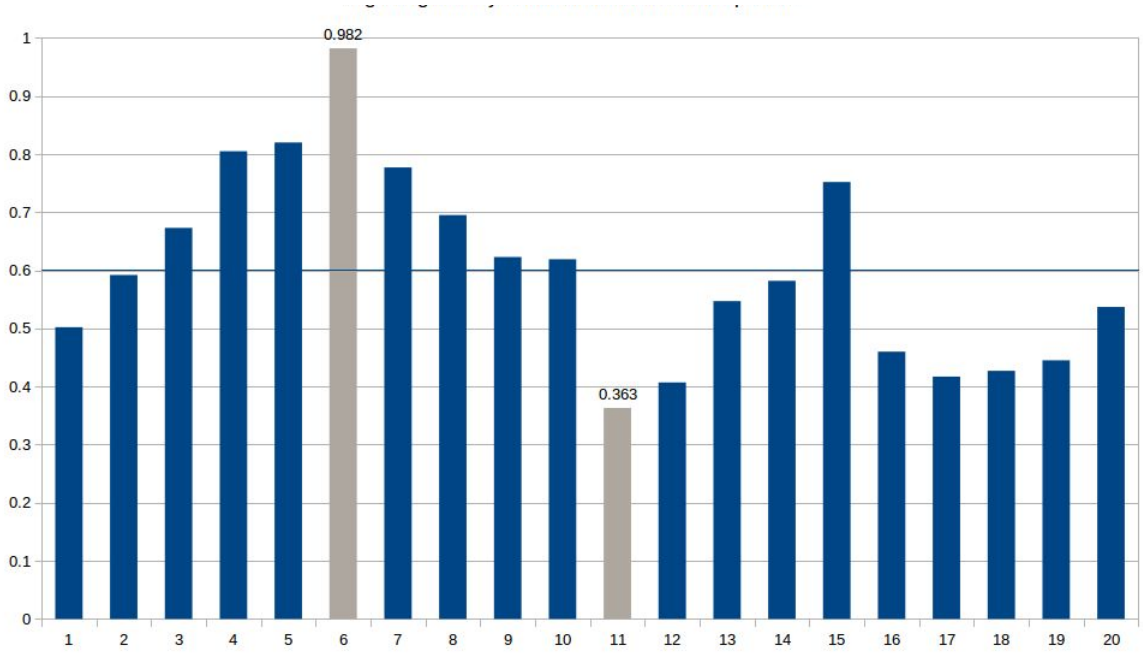

Figure 24. Signaling latency with FCM in various time periods

5.5.2.2. Reliability

In this section it will be described which signaling method is more reliable depending of the different states off the application. To compare them the signaling was tested in different circumstances, which are the following:

- Application is in the foreground
- Application is in the background
- Application is closed and swiped away from the list
- Application is force closed

It is important to check how the app reacts in these situations, because the user doesn't want to receive calls just when the app is open and active, but also when it is not in use, maybe even cleared from the open application list. After checking every scenario the result was that both signaling methods behave the same. They work well in the first three cases, but afterwards if the application is force closed from the settings menu by the user,

receiving calls is not possible anymore. However they provided the same results, they are not necessary working always the same. For Socket.IO signaling a service is used, which is set to run in the foregrund. This means the Android system gives higher priority to it and it won't be killed immediately when there device is is low of memory, although from the official Android documentation it turns out that there is no guarantee, that it will be never killed.

5.6. Conclusion

In this chapter the workflow of WebRTC was described and possible ways of signaling methods. As it can be seen from the results the WebRTC implementation was achievable in both prototypes, although through different approaches. Despite that, in the native Android and also in the hybrid application the video and audio conversation works in the same way. For a user no difference can be recognized regarding on the used signaling method and on the WebRTC itself.

In this thesis the testing was done on Android devices with both approaches, but all the previously described real-time communication technologies work on iOS as well [39] [40]. This means the different mobile platforms and approaches does not cause any limitation from WebRTC and signaling process point of view.

According to signaling section 5.5 sums up the results and shows that both methods have low latency value and both are very reliable. However to optimize this factors probably the best way of implementation would be a combination of these technologies. During the tests FCM was used only for signaling, but it could be used also to notify the user about events. One possible solution could be to use Socket.IO for emitting messages such as posts, direct messages (like in the current implementation) and also for the signaling. But as soon as the socket connection is broken between the server and the client, FCM could be used for signaling and to inform the user about incoming messages or new posts.

With this design we could get the best of both methods because according to the test results Socket.IO provided faster transmission, but FCM is more reliable as the connection is managed by the system and not the app directly. In this way even if the socket connection would be closed for any reason, the user would be still notified about the events through FCM. Although FCM is designed to send short messages up to 4Kb, which means it is not appropriate way to actually emitting the posts and direct messages because they can be longer, but with this it is possible to transmit the signaling message for calls or just sending a short push notification to the device about new events.

To sum it up WebRTC was working with all of the tested signaling approaches. That's why maybe the presented combined method should be used as it gives some additional benefits to the users.

6. Comparison of hybrid and native platform

This chapter will present essential differences between the hybrid and the native implementation from different aspects. In section 6.1 the author will give a short overview about the goals and how the comparison was done. Then the differences will examined from various points of views such as development point of view (6.2.), feature point of view (6.3.) and user experiments point of view (6.3.) Afterwards the experimental results will be shown through certain performance analysis (6.4.). Finally section 6.5. will summarize the results and give a conclusion.

6.1. Goals

As it was earlier already written, the main goal of the thesis is to introduce and use different technologies for building a mobile application with social features. As result we should be able to distinguish which way would be the best to implement such an application. . As the app provides different ways of communication mechanism, the one major topic was to examine different implementations of WebRTC and compare their usability which is summarized in Chapter 5. The other important point is to analyze the different approaches of mobile development like hybrid vs. native.

The goal of the two different implementations of the application was to provide experimental results about the differences and difficulties. The main focus of this comparison is to give information from technical and development aspect, but it will also answer questions like how long does it take to start the app or how long are the response times of the UI elements. Thróugh these tests one can get a more detailed picture also from the UX view, which is a crucial point in the application development.

6.2. Overview

Section 3.2 gave already an overview about what is the hybrid and the native mobile development about and which are the major differences between them. One of these is the process of the implementation which will be described in section 6.3. from development point of view and also from the business point of view. It is important to be aware of the advantages and disadvantages of these two approaches to decide which one should be choosed.

As it was already written in this thesis, a hybrid application has one code base and is using web technologies for example HTML, CSS and some scripting language like JavaScript or TypeScript. These technologies are well known by web developers already, which implicates that it's much easier for a company to develop a mobile application for every mobile platform, because they don't need to hire mobile developers with specific knowledge about Android or iOS development. Although it sounds like a big benefit, hybrid apps are not always the best choice for a product. Considering the known limitations of an hybrid app, it can be determined in which field they can be useful and in which not.

Figure 25 summarize the distribution of hybrid applications in different fields and areas. This table was made in 2015 by Ivano Malavolta, Stefano Ruberto and Tommaso Soru [14], who did a research by examining the 500 top rated applications on Google Play for each category. Already a first look at the results shows that hybrid applications are certainly lower represented in the top 500 app compared to native Android ones. It can have several reasons, for example that hybrid mobile development is still not so widely used by developers or because they are providing lower user experience so they get lower ratings.

To investigate the second reason in more detail, the table give us more information. It clearly shows that the percentage of hybrid applications are the biggest in such areas like: finance, medical,travel, fitness and social. In contrast with those the lowest values are in the personalization, tools and game categories. These results are also correlating to the known

advantages and disadvantages of the hybrid apps, for example an app for personalization needs closer interaction with the operating system, which can be easier done with a native application.

	Total	Native	Hybrid (%)	Apache Cordova	Appcelerator Titanium	PhoneGap
Finance	463	410	53 (11.45)	23	29	1
Medical	490	445	45 (9.18)	24	11	2
Transportation	438	404	34 (7.76)	18	11	2
Travel & Local	484	450	34 (7.02)	23	5	4
Health & Fitness	352	331	21 (5.97)	15	6	
Libraries & Demo	418	410	8 (5.97)	6	1	
Business	488	459	29 (5.94)	17	3	3
Lifestyle	497	468	29 (5.84)	12	11	5
Social	491	465	26 (5.30)	16	3	5
Sports	497	473	24 (4.83)	15	6	2
Shopping	487	464	23 (4.72)	12	6	3
Education	493	473	20 (4.06)	14	1	4
Book & References	472	457	15 (3.18)	12	2	
Communication	487	471	16 (3.29)	13		1
Entertainment	487	472	15 (3.08)	10	2	3
News & Magazines	491	478	13 (2.65)	2	11	
Comics	465	456	9 (1.94)	7		
Weather	495	488	7 (1.41)	4	3	
Media & Video	483	477	6 (1.24)	4	1	1
Productivity	495	489	6 (1.21)	3	1	1
Photography	494	489	5 (1.01)	4		
Music & Audio	477	473	4 (0.84)	1	3	
Tools	489	486	3 (0.61)	3		
Game	492	491	1 (0.20)	1		
Personalization	493	492	1 (0.20)	1		
ALL	11,917	11,470	445 (3.73)	258	116	37

Figure 25. Hybrid mobile apps and frameworks distribution in the Google Play Store [14]

The highest represented categories, mentioned above, are so-called data-intensive applications [14], which main functionalities are to deliver content, organise information and interact with them. In contrast with these, on the other end there are apps which require access to the operating system's native features and settings like the applications in the tools and personalization category. The other one of the least represented category is the game, which is also not a big surprise, because they are usually need high performance and rich graphics, which an hybrid app can't provide as good as a native one.

These consequences are also confirmed with the results of the previously mentioned research which also gives information about the ratings distribution over the different categories compared to the natives. [14] In the categories where the distribution of the hybrid apps are higher, these apps got nearly the same ratings like the native ones, but in the categories Tools, Games and Personalization the rating of the native apps are significantly higher.

In conclusion it looks like there are areas where hybrid applications can certainly provide the same user experience than native applications. As the prototypes which were implemented for this thesis focus also on social features, it is relevant and reasonable to do a comparison between them. In the next sections the experimental results will be shown from different perspectives.

6.3. Development based comparison

6.3.1. Platform support

Platform support is an important question in mobile development, because it is a key factor. It depends on the chosen approach if it is possible to reach more platforms with one code base and if yes, which ones. In case of native apps, the answer is easy, the native code will mainly run just on that platform for which it was written for. According to hybrid applications the supported platforms are similar. Cordova supports all the major operating systems for smartphones which are shown in Figure 26, just as the different hybrid mobile UI frameworks. However there can be differences in support of other operating systems, which has lower present, for example Amazon Fire OS.

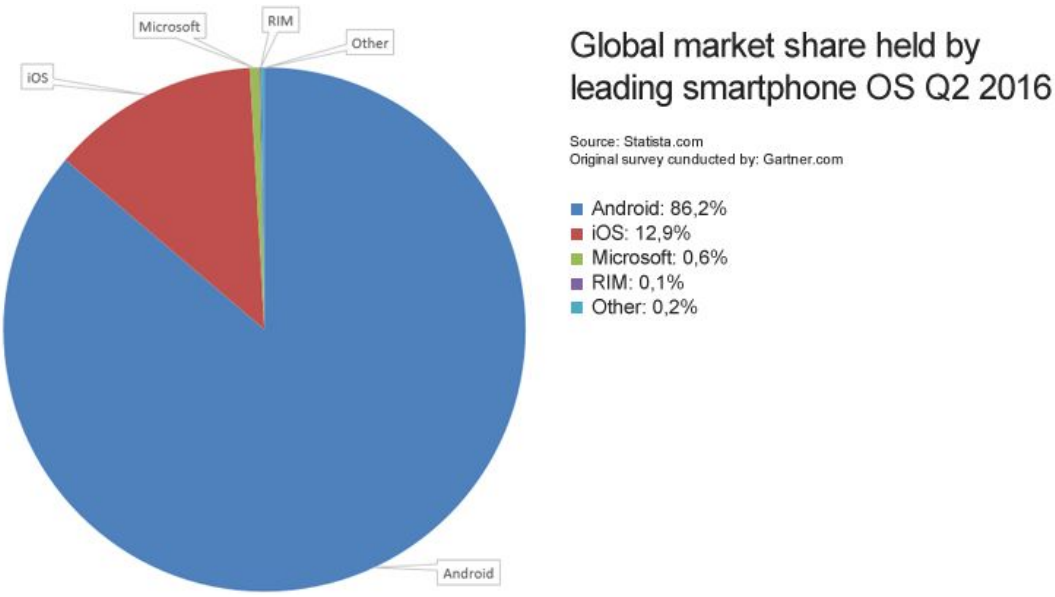

Figure 26. Mobile OS market share- Q2 2016 [41]

6.3.2. Environment and structure

The implementation of the prototypes was done on Ubuntu 14.04 using Dell XPS 13 Developer Edition laptop with i7 CPU and 8G RAM. In Chapter 4 the requirements were already written for both prototypes.

The native Android application was implemented in Android Studio. For the hybrid application several editors were used. As the application using TypeScript, first WebStorm was used, which is one of the best editors for TypeScript files. Unfortunately the IDE was frozen a lot of times, and the response time of the UI also got worse with longer time of usage. Afterwards Eclipse Mars was tried out, with a TypeScript plugin, which was first working well, but afterwards it was just not able to load the source files. Creating a new workspace solved the problem, but it occurred days later again. In the end the editor was switched to Kate, which is a simple lightweight text editor. The disadvantage of it is that it's not offering code completion, but at least it's providing the basic features such as: syntax highlighting, project structure and the embedded console made it easier to run the application from command line.

Just as the development environment also the project structure is very different. Figure 27 shows the structure of both apps. The right figures shows the traditional Android project structure. The source code can be found in the *java* folder, and all the resources such as images, drawables and layout XML files are in the *res* folder. It is important to mention the Manifest file which is always in the root and it provides the essential information for the app. In contrast to this in the hybrid project the index.html represent the root, which includes JS and CSS files. The main code for the app is lying under the app folder, where every UI component has its own folder and contains 3 files : HTML file for the UI, an SCSS file for custom style and the TypeScript files which implements the logic. The different project specific settings can be configured in the config.xml and in the package.json file. Finally the built applications can be found under the platforms folder for each of them.

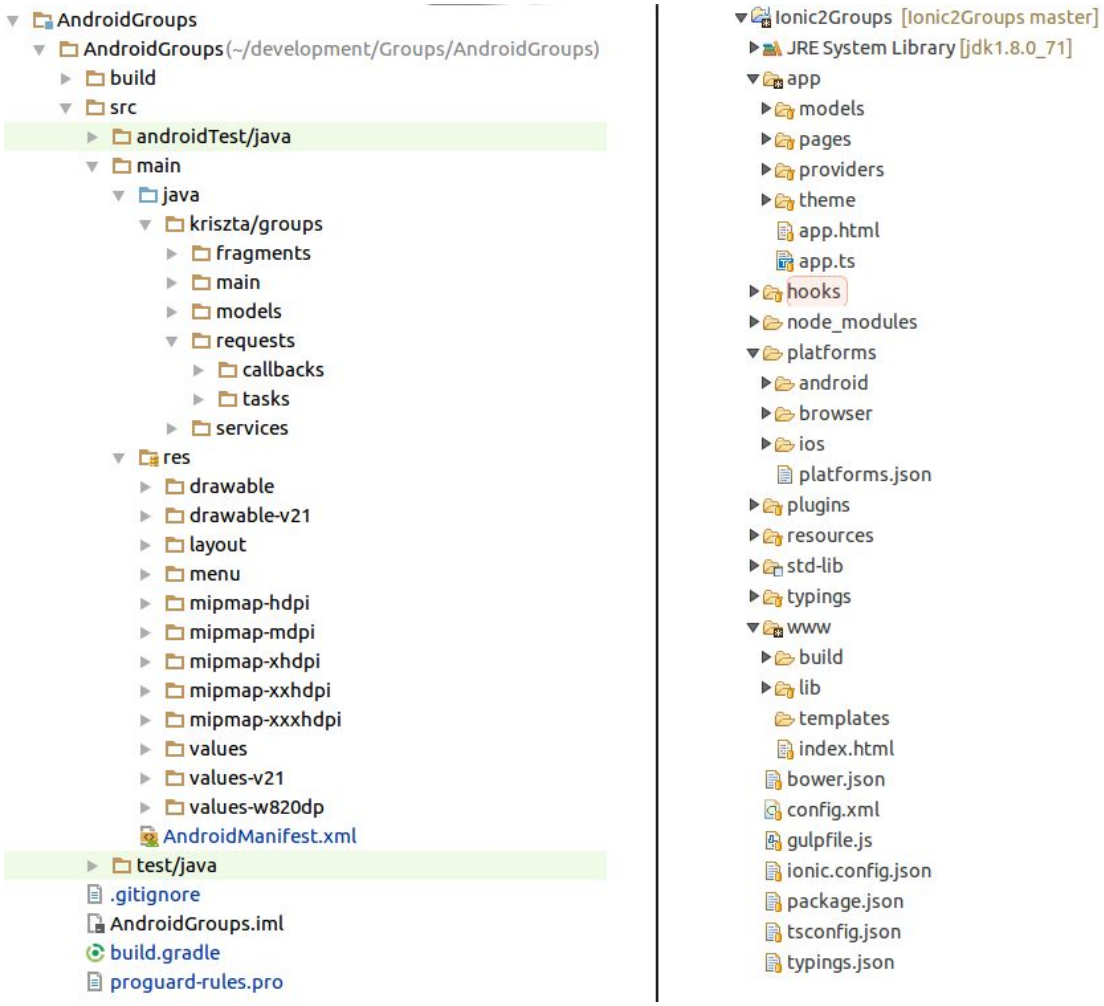

Figure 27. Project structure of the native (left) and the hybrid (right) application

6.3.3. Maintainability

Maintainability is an important and also difficult question. Henning Heitkötter, Sebastian Hanschke and Tim A. Majchrzak define it in their research [42] as: "The lines of code (LOC) indicator is employed to evaluate maintainability [32, p. 53f.]. The choice of this indicator is based on the assumption that an application is easier to support when it has less LOC, because e.g. training of new developers will be shorter, source code is easier to read etc."

However the lines of code is definitely relevant, but there are other code quality metrics which has to be considered and examine them together to calculate the maintainability. Traditionally three metric values are used for the maintainability index, which are the following: cyclomatic complexity, Halstead volume and LOC. The cyclomatic complexity indicates the complexity of the code by measuring the "number of linearly independent paths through a program's code" [43]. The Halstead volume is calculated statically from the source code to " identify measurable properties of software, and the relations between them" [44].

For the maintainability index multiple formulas exist with small differences. For calculating this index for the prototypes the same formula was used which is used by Visual Studio, shown on Figure 28. In contrast with the original one, the result is scaled which means the index value is between 0 and 100. Visual Studio is using multiple ranges to determine the maintainability of the project, which are listed below:

- 0-9 = Red (low maintainability)
- 10-19 = Yellow (moderate maintainability)
- 20-100 = Green (good maintainability)

```
Maintainability Index = MAX(0,(171 - 5.2 * ln(Halstead
Volume) - 0.23 * (Cyclomatic Complexity) - 16.2 * ln(Lines
of Code))*100 / 171)
```

Figure 28. Formula used for Maintainability Index [45]

Lines of code, and the maintainability are not necessarily directly proportional, which was confirmed by the code metrics result of the prototypes. The analysis was done by checking the required metric values per method and calculated the maintainability index for each, then to determine the result for the whole project, the average values were used.

However in a big project it can be misleading, because it is possible to have overall a good maintainability index even when there are smaller modules which very low index. Figure 29 shows the distribution of the maintainability values of the methods in the hybrid and the native Android prototype.

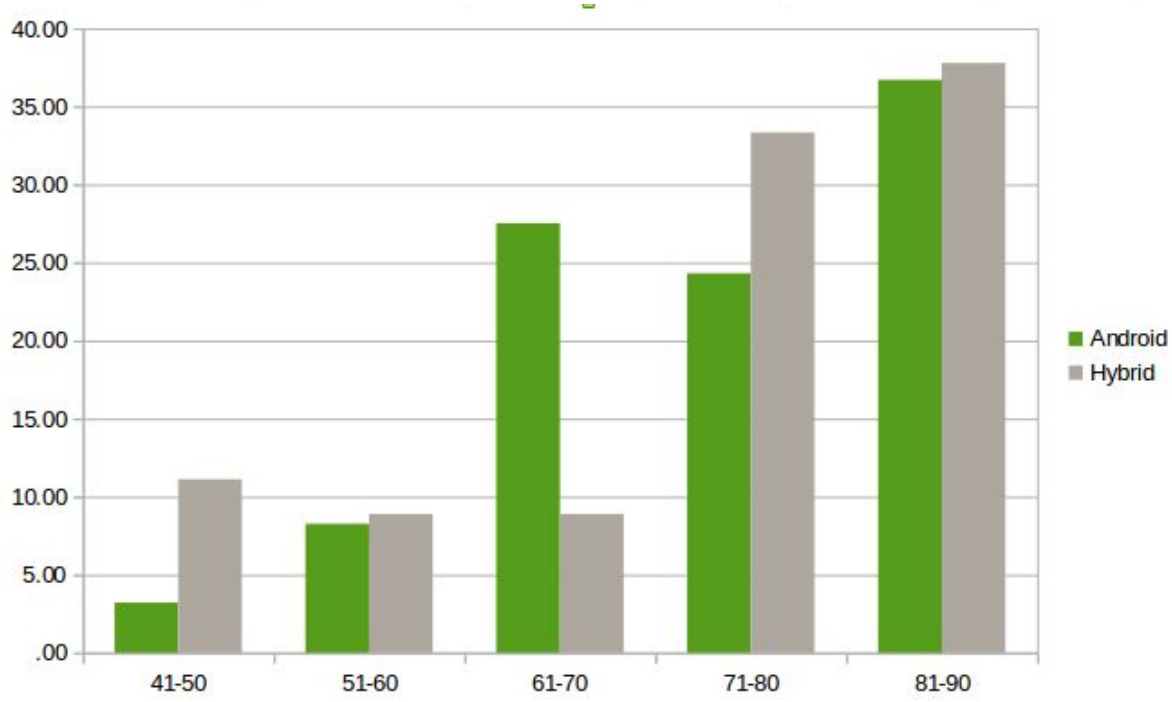

Figure 29. Distribution of the maintainability index by percentage (y) and the index value ranges(x)

The *y* axis shows the percentage of the number of methods with certain indexes. Although the distribution in some intervals are different between the native and the hybrid implementation but the tendency is the same for both, namely that methods have minimum 40 or higher index value and maximum 90. Furthermore it can be also seen on the graphic that methods with lower maintainability have lower occurrence in the entire project.

Figure 30 presents the average maintainability index of the applications and the summary of LOCs of the methods. The results clearly shows that they are not directly proportional as the calculated index for the Android project is 72.8 and for the hybrid is

74.5. There is just a very small difference between the values however the native Android project is three times bigger and has more LOCs.

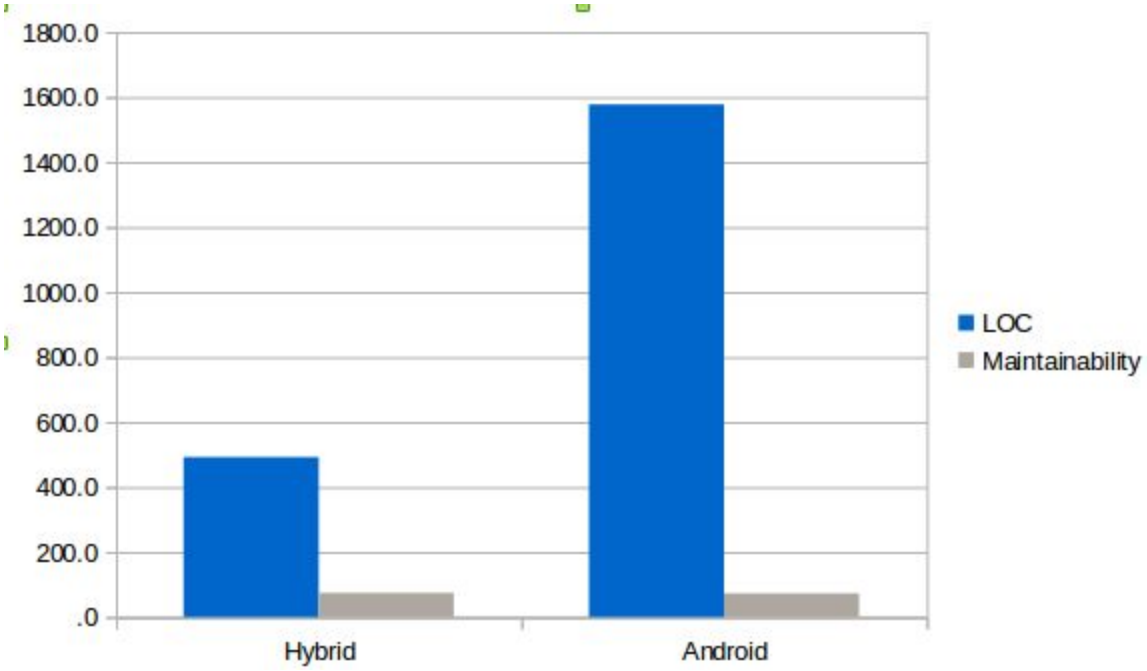

Figure 30. Connection between LOC and MI for the prototypes

6.4. Feature based comparison

In this section will be shown how some of the native Android features can be done in the hybrid application and in how is the native way to do them. The presented features were chosen because they are important for the functionalities of the GeoGebra Group app, such as permission handling, notifications, and the connection between different apps.

6.4.1. Local notifications

First it will be shown how the local notifications are done in the hybrid and in the native prototypes. Such notifications are important, because the application is not always in the foreground or even when it is, only just one view is visible which means for example if the user is reading the posts in the group, then he or she will not see if someone wrote a

direct message. That's why it is important to notify the user, who can then decide to click on the notification and open it, or not.

To create a local notification in the hybrid app a plugin was used, which called: de.appplant.cordova.plugin.local-notification. The usage of this plugin is very straightforward, and built up similar to the native one as it can be seen on Figure 31 and on Figure 32.

```
cordova.plugins.notification.local.schedule({
  id: 1,
  title: "New message from " + msg.userFrom.firstname + " " + msg.userFrom.lastname,
  text: msg.message
});
```

Figure 31. Triggering local notification in hybrid application

```
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this);
mBuilder.setSmallIcon(R.drawable.ic_group_black_24dp);
mBuilder.setContentTitle("New message from "+ fromUserStr);
mBuilder.setContentText(msg.getMessage());

int mNotificationId = 001;
NotificationManager mNotifyMgr =
        (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
mNotifyMgr.notify(mNotificationId, mBuilder.build());
```

Figure 32. Triggering local notification in native Android application

It can be seen that the structure is the same in both case and after the plugin was added to the Cordova project it can be easily used. Figure 33 shows the result, when a notification arrived to the device which was triggered after direct message from another user was received. As the screenshots show there is no difference between them, although the hybrid application using a plugin, but the plugin itself using the same local notifications of the system like the native.

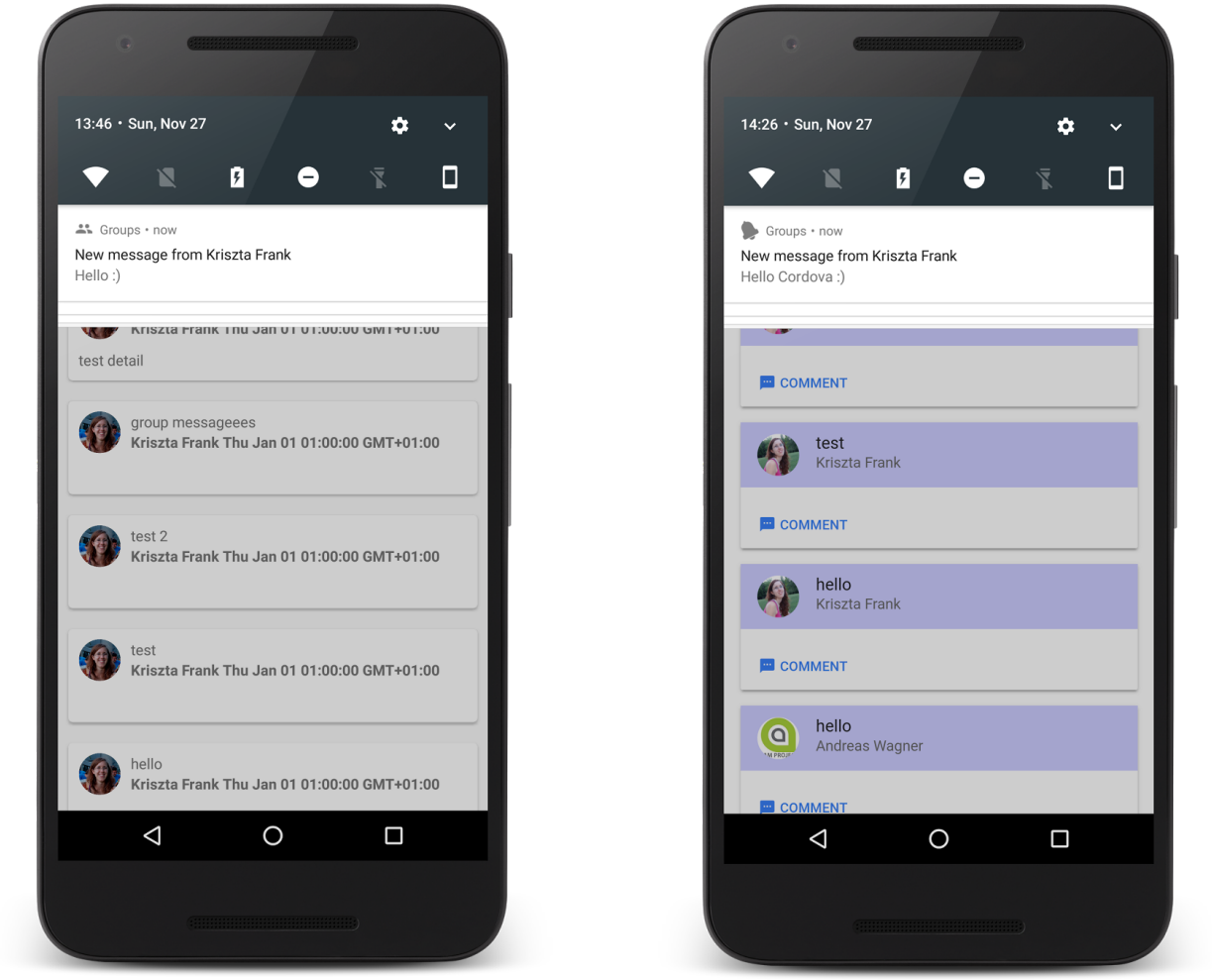

Figure 33. Local notifications with native Android (left) and hybrid app (right)

6.4.2. Permission handling

Permission handling was earlier not so critical from the implementation point of view, because it was enough the define the required permissions in the Manifest file, which were listed afterwards when the user wanted to install the app from Google Play. In this case either the user accepted those and installed it or cancelled the whole installation process. However this changed in Android 6, where the user has the opportunity to

managing the permissions at runtime. That's why it is important to check the state of the corresponding permissions before an action can be done and request them if it is needed. For example in the prototypes it has to be checked if the user gave access to the camera before he or she clicks on the video call button.

Fortunately there is a plugin for Cordova called *cordova.plugins.diagnostic* which can handle the runtime permission requesting. Figure 34 and Figure 35 shows a small code snippet to compare how it is done in the native and the hybrid environment.

```
cordova.plugins.diagnostic.requestCameraAuthorization(function(status){
switch(status){
    case cordova.plugins.diagnostic.permissionStatus.NOT_REQUESTED:
        console.log("Permission not requested");
        break;
    case cordova.plugins.diagnostic.permissionStatus.GRANTED:
        console.log("Permission granted");
        break;
    case cordova.plugins.diagnostic.permissionStatus.DENIED:
        console.log("Permission denied");
        break;
    case cordova.plugins.diagnostic.permissionStatus.DENIED_ALWAYS:
        console.log("Permission permanently denied");
        break;
}
}, function(error){
    console.error(error);
});
```

Figure 34. Request for camera permission in Cordova

```
if (ContextCompat.checkSelfPermission(thisActivity,
     Manifest.permission.CAMERA) != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(thisActivity,
              new String[]{Manifest.permission.CAMERA},
              MY_PERMISSIONS_REQUEST_CAMERA);
    }
}
```

Figure 35. Request for camera permission in Android

As it can be seen the methods to request a permission is very similar and simple in both cases and works the same as shown in Figure 36.

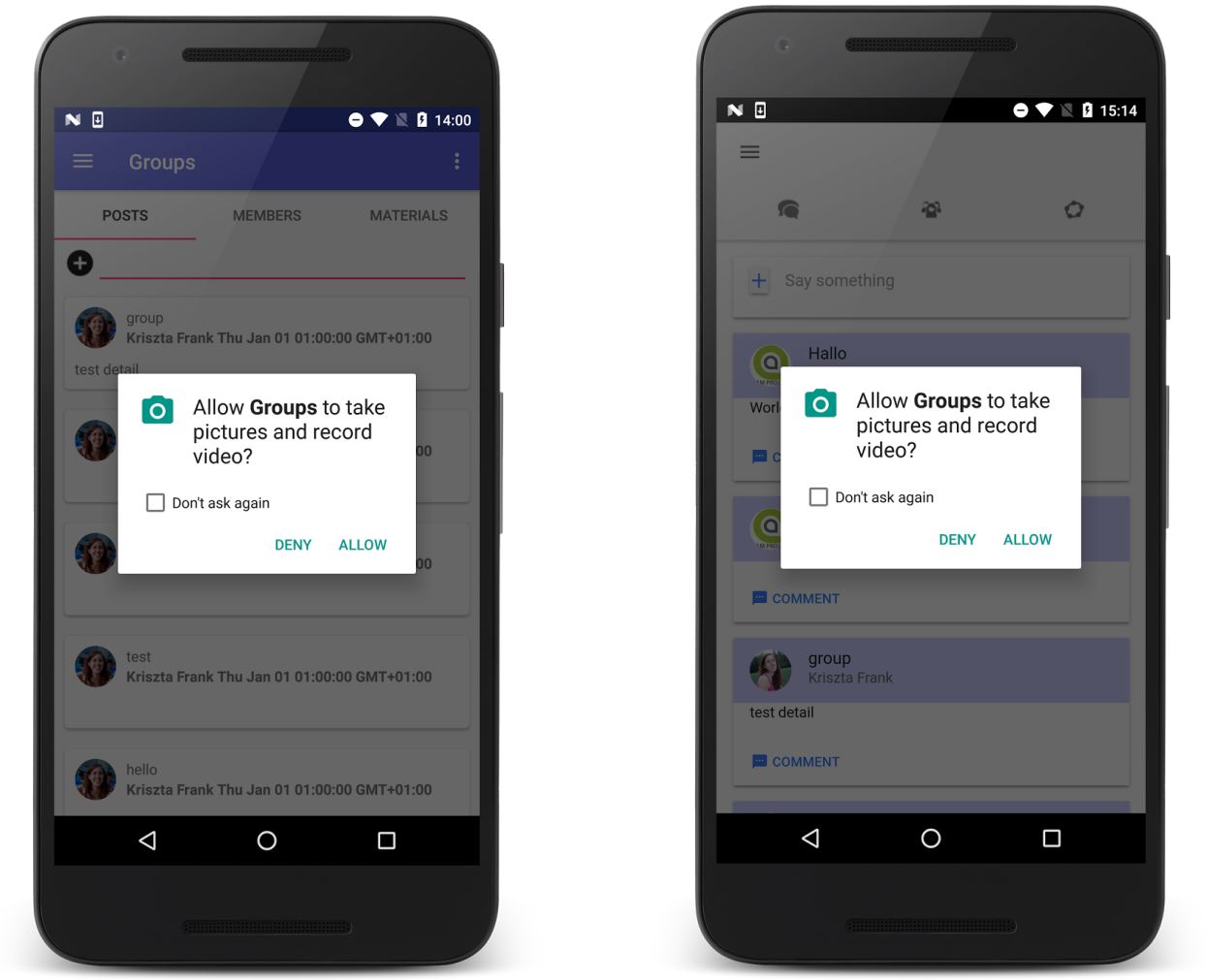

Figure 36. Runtime permission request in native Android (left) and in hybrid app (right)

6.4.3. Connection between apps

The last feature which will be examined is opening an external app from the application. It is interesting, because Android is using a so called Intent for this, which is a platform specific design. Intents are used for performing operations. That implicates that in Android the implementation can be done simply. In the hybrid app it is necessary again to use a plugin. The chosen plugin is called *com.lampa.startapp,* and as it can be seen on Figure 37 and Figure 38, the syntax and the logic is very similar to each other. With the plugin, we have to give the same attributes as key-value pairs, which are passed afterwards to the native logic. It is still worth to mention, that during the usage of this plugin an exception occurred, which could be only solved by analyzing the source code of it. This shows again that sometimes it can be more difficult and problematic to use a plugin, which was already mentioned in Section 5.5.1.

```
var sApp = startApp.set({
  "action":"ACTION_VIEW",
  "category":"CATEGORY_DEFAULT",
  "type":"application/ggb",
  "uri":material.fileurl,
  "flags":["FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET","FLAG_GRANT_READ_URI_PERMISSION"],
}, { /* extras */
}).start();
```

Figure 37. Opening file in external application in the hybrid project

```
Intent i = new Intent();
i.setAction(android.content.Intent.ACTION_VIEW);
i.addFlags(Intent.FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET);
i.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
i.setDataAndType(uri, getActivity().getContentResolver().getType(uri));
startActivity(i);
```

Figure 38. Opening file in external application in the native project

6.5. Perceived performance analysis

6.5.1. Start-up time

For the performance analysis an important indicator is the start-up time, because it gives an initial feeling to the user about the product. If the start-up takes too long then the user has already the impression that this application is slow which has a big impact on the user experience. According to a survey which was done by Compuware [46] 50% of the users will exit the app if the loading time is over 5 seconds.

To analyze the start-up time for the hybrid and also for the native application several tests were done. The loading time was investigated separately depending on how the user started the application. In Android three different cases exist, which are the following: cold start, lukewarm start and warm start. The cold start is when the user didn't open the application before, it is the first time starting the application. This takes the longest time, because everything has to be loaded. The second type is the lukewarm start which means the application was running before, it was closed and then the user opened it again. In this case the application has a state, where several parts were already destroyed, but in the memory probably there are some modules of the app which are still alive, so at the start it is not necessary to load everything from the beginning. Finally the third type is the warm start, which occurs when the user wasn't closing the app, and it is still running in the background, so when the user opens it, the system brings it back to the foreground.

Figure 39 shows the different starting times for the hybrid and the native application distinguishing the different starting methods. The graphic clearly shows that the loading time is shorter for the native Android application. In case of the cold start there is around 300 milliseconds difference, which is still acceptable because the application is starting for the first time. Which is more surprising, that in case of lukewarm start the starting time of the hybrid application is more than twice as long as the native one, and this value is almost the same for the cold start. This means if the user is not just bringing the app from the background to the foreground then there will be already bigger differences in the starting

times. However the starting time of the hybrid application is still under one second even in the worst case, this value is not representing correctly the perceived performance from the user point of view. In case of the hybrid application, the value of the start-up time only means, that the system started the application, but does not mean that the UI is ready to use. It can occur, because after the application started it still has to load the native shell, Cordova and the different plugins. After all these necessary modules are loaded, Cordova's deviceReady() event is fired and the UI elements can be rendered. Based on the tests, most of the time it usually took 3-4 seconds until the Cordova API was ready. Afterwards rendering the UI took again an additional 1-2 seconds. In this case the difference between the hybrid and the native application is significantly bigger, and easily noticed by the user. These results match also with the results of a research made by Florian Rösler, André Nitze, Andreas Schmietendorf [47].

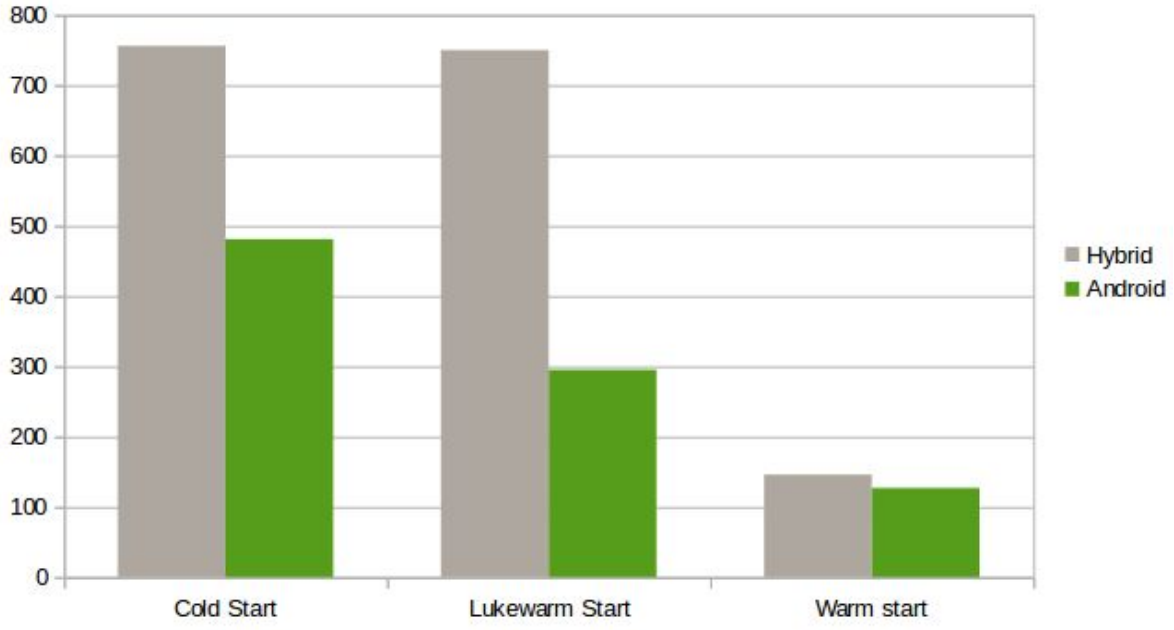

Figure 39. Start-up times for the hybrid and the native application

6.5.2. UI response time

Measuring and evaluating the response time of an application is not always easy, and it can be different from device to device. From the UX point of view the most important is that the user has a feeling of smooth working UI.

Comparing the hybrid and the native application, and their UI elements, there are not significant differences. However in some cases the hybrid prototype has longer response times. An example is the switching between the tabs after the application started. When the user selected a group to join, the *Posts* tab is opened, but when the user clicks on one of the other tabs for the first time, it takes longer to switch there, and opening it compare to the native one. The reason of this behavior is that in the native app the tabs are loaded differently. When the first tab is opened, in the same time the next one is also preloaded to provide smooth swiping. In contrast to the hybrid application the tabs are not preloaded before they would get visible.

As an other difference between the behaviour of the UI elements, it is important to mention the direct messaging view. In both prototypes when the view is opened, then it is automatically scrolling to the bottom, where the newest messages are. However in the hybrid app, when a new message is sent, it is supposed to scroll down, but it only scrolls to the element before the last one. Also in the same view when the *Send* button is clicked, first the keyboard closes because the input field lost the focus then it is necessary to click on the button again for sending. However probably there is a workaround to solve these issues, it is still interesting to see which kind of problems can occur.

6.5.3. Loading lists

A good user experience next to other factors depends on fast loading views, and as the prototypes contain a more views with long lists, it is important to examine which approach provide better user experience for this. In the application there are two cases

when it has to handle long lists: in the group, where the posts and comments are shown and in the direct messaging view, where the previous conversation has to be loaded. Both situations were tested, however the time results will be evaluated relatively to each other, because the data are coming from the server, which means the loading time contains not only the rendering the UI but also the server-client communication.

In case of the post messages for the same group with around 100 elements took in both application around 1.9 seconds to load and fully render the view. In the direct messages view, the loading time was around 1.3 seconds in the hybrid, and 1.5 in the native Android app. The length of the list was 60 elements which was loaded from the server. We can see from the results that in the first case, where the UI itself was also very similar, the rendering time was the same, and also in the second test case the difference was minimal, which can be caused by the simpler UI of the hybrid app for the direct messaging view.

However this values can be, and also should be reduced to a minimum for a smoother user experience. For this there are several ways to do that. A possible solution would be to always load only the last 20-30 messages and posts, and the rest should be just loaded from the server, when the user is scrolling down or up.

6.6. Device performance analysis

This section will present results according to the performance of the applications. There are several factors which has a strong effect on the performance such as memory usage and CPU consumption. These indicators will be examined in the following sections, and the results will give a detailed view about the differences between the hybrid and native implementation from a performance point of view.

6.6.1. Memory usage

As new phones are released, the hardware parameters are also getting better. Although these phones can have already 3-4 giga RAM it doesn't mean that a developer doesn't have to take care of the memory usage of the app. There are a lot of different Android devices on the market from low to high category which provides a big range of hardware. Some phones may just has 1 GB RAM, or in case of some older devices, even lower. That's why it is important to investigate the memory usage of the native and also of the hybrid application. Besides the normal usage when the app is in the foreground, it is relevant to check also how much RAM the application needs when it is in the background. This is important because the Android system will start to kill the apps in the background when there is not enough memory available.

Figure 40 shows the memory consumption of the hybrid and the native Android app in MB. On the left side of the graphics there is the RAM usage right after the start-up and on the left side when the app is in the background. One can see that the hybrid prototype needs nearly twice as much RAM in both case. This result is not so surprising because a hybrid Cordova app needs more resources. As it is written in [46]: “Hosting what is essentially a browser inside of your app is not inexpensive. This can be a significant issue on low-end phones.” In the same article the author also comparing two basic Hello World applications which are using 26 MB (native Android) and 59 MB (Cordova) RAM, which also shows that a Cordova app by default needs more resources than native ones. However there is a limit for the Android system, and application above the limit will be stopped, this threshold depends on the device parameters, and even for phones with lower RAM size is minimum 150 MB.

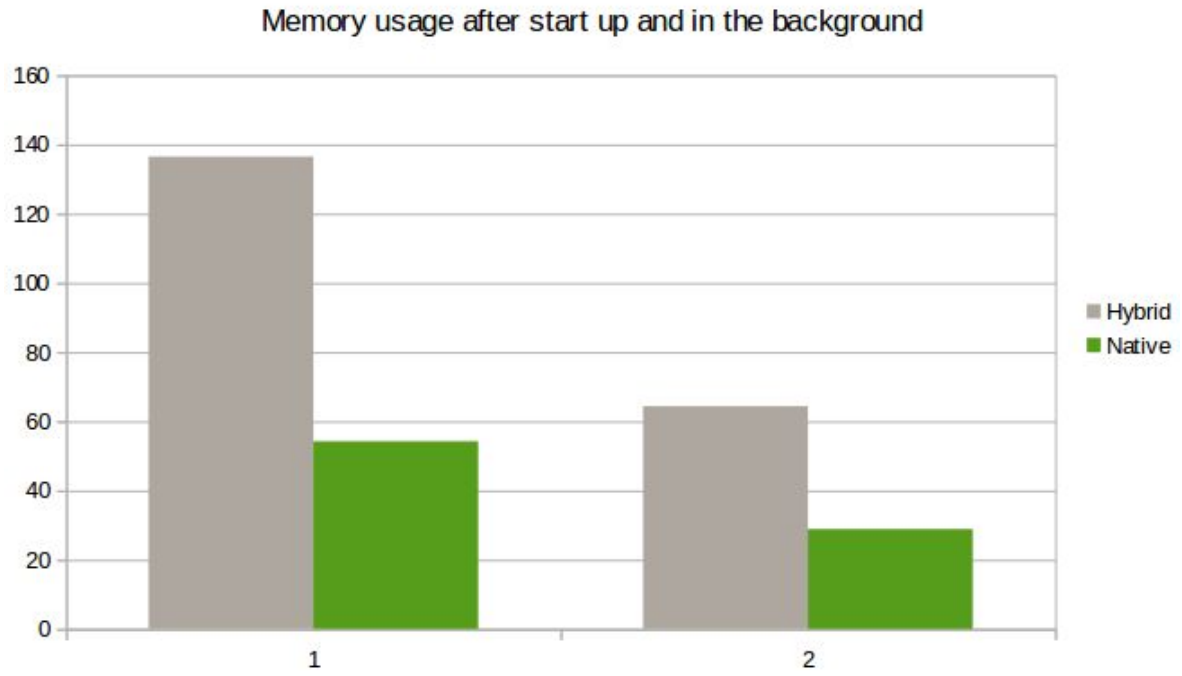

Figure 40. Memory usage after start up and in the background

6.6.2. CPU consumption

CPU usage is also an important factor in the analysis, because it has an impact on battery consumption too. As it is written in [49]: “The effect of the percentage CPU utilization on the rate of current discharge can be accurately represented linearly.” That's why it is interesting to compare the hybrid and the native application, if there is any noticeable difference.

First the CPU usage was examined after the app was opened. In this case only the simple functionalities were used such as posting, commenting, switching groups and write direct messages. During the tests the native Android prototype was using approximately 1-3% of the CPU, the hybrid app used a little bit more, between 2 and 4 percentage. As this values are very low, and does not have really measurable effect on the battery consumption a second test was also done. In this test the CPU usage was measured during a video call.

Figure 41 shows the results, and as it can be seen the CPU consumption is significantly higher than before caused by the video rendering. Although the hybrid application uses a plugin, and the native an own implementation, there is not big difference, because both use the same codecs, which has the highest effect on the CPU.

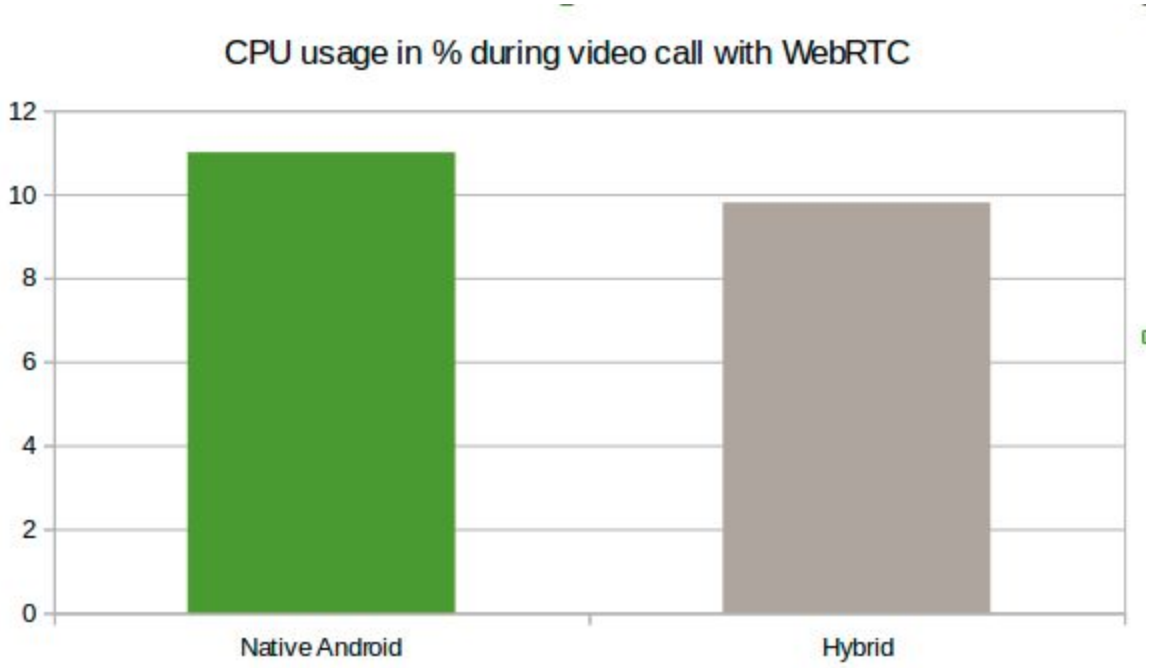

Figure 41. CPU usage in % during video call with WebRTC

6.7. Conclusion

In this chapter it was presented which are the main differences between the hybrid and the native implementation from several points of view also in general, and specifically for this project.

From a developer perspective the implementation of the two prototypes was entirely different. It is difficult to distinguish, which way is easier as both has their advantages and disadvantages. In the hybrid project the UI could be done pretty fast, but it was significantly more difficult to implement a background service for the client-server communication. In contrast with that, in the native application it was very straightforward, however to create a similar user interface took a lot more time and coding.

From UX point of view despite to the foreknowledge, the hybrid application gave good results, as the response time and the speed of loading the views is very close to the native application. However the UI elements of Ionic look good, and native-look like, one can still recognize that it is not a native application. This can be solved by adding plugins which adds smooth native-like animations for better UX, but this can lead to another problem. During the implementation of the hybrid prototype the main issues was caused by plugins. First problem is that sometimes it's difficult to find one, which is fitting to the purpose, supporting the needed platforms and native SDK levels, not outdated and well documented. The second, even bigger problem is that it can be still possible, that the plugin is not working properly, and it takes a lot of time to find the source of the problem. Definitely if the developer has no deep knowledge about the certain platform language. Beside the mentioned problems with the plugins, they make also the start-up time of the application longer, that's why additional plugins should be just used if really needed.

The results from the performance point of view showed that the CPU usage is the same for both application, but the hybrid app needed twice as much memory which can have a negative effect for low-end phones with low RAM.

The overall conclusion is that an hybrid app could provide the same user experience and functionalities as the native Android one. It can be a perfect choice for applications which doesn't need a lot of interaction with the native platform as it can dramatically speed up the development time. Otherwise probably a native implementation would be better even if takes more resources and time to implement for multiple OS-s, but it cause less struggling to use a native feature of the platform.

7. Conclusion

7.1. Results

One of the main purposes of this thesis was to present the native and the hybrid application development and decide based on the results which one is the better approach, although this is a question for what there is not a single answer. In generally it cannot be said that one is better than the other one, because it fully depends on the goal and on the nature of the application. The experimental results of the implementation and the applications showed the strong and the weak side of both types which were summarized in section 6.7. These results provide information, which can lead to the good direction of the implementation. However in case of the GeoGebra Groups prototype it was difficult to determine the better approach. A big part of the application can really benefit from the advantages of the hybrid development such as quick construction of the UI, and simple way to do the REST calls. Despite that, in case of this application probably the native development would be better as it was several times necessary to fix bugs in the applied plugins, which required knowledge about the platform specific language and this is against the main concept of the hybrid development.

The other big topic of the thesis was to explore and compare the real-time communication methods for mobile applications, and to check if they behave different in the hybrid and in the native application. As it turned out during the research, for audio and video conversation WebRTC is currently the most used approach everywhere. According to the design of WebRTC, the signaling method is not part of the protocol, which gave the flexibility to test more alternatives. Both of the examined signaling mechanism ended up with good results, and there was not perceived differences in the functionalities in the two types. Furthermore as it was desibed in section 5.6, to reach the best reliability, a combination of Socket.IO and FCM would be the best approach for this application.

7.2. Future work

As the presented applications were only prototypes there are many further development opportunities. One of the most important would be to continue the development on the native Android application as from the result this approach seemed better. The other further development would be to improve the current functionalities and to extend the application with new the features, which are the following:

- Group conversations
- Sharing files and documents in direct messaging
- Upload files and documents in the group
- Possibility to share GeoGebra materials via an integrated Material chooser
- Add Task/Feedback as in on the GeoGebra Groups website
- Improve the real time communication method with combination of FCM and Socket.IO
- Profile view for every user
- Own public/private posts on the user page
- Possibility to follow other users

8. Bibliography

[1] Business Insider: http://www.businessinsider.de/smartphone-adoption-platform-and-vendor-trends-in-major-mobile-markets-around-world-2015-3?r=US&IR=T

Date: 09.07.2016

[2] Smart Insights: http://www.smartinsights.com/managing-digital-marketing/marketing-innovation/marketing-trends-2016/

Date: 09.07.2016

[3] Smart Insights: http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/

Date: 09.07.2016

[4] Statista: http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

Date: 03.07.2016

[5] Bitport: http://bitport.hu/porog-es-gyorsan-telitodik-az-app-fejlesztok-piaca

Date: 03.07.2016

[6] WebRTC: https://webrtc.org/

Date: 09.07.2016

[7] Socket.IO: http://socket.io/

Date: 09.07.2016

[8] NetmarketShare: https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1

Date: 03.07.2016

[9] GeoGebra Web: https://www.geogebra.org/graphing

Date: 11.10.2016

[10] GeoGebra Groups: https://www.geogebra.org/groups

Date: 10.10.2016

[11] Salma Charkaoui, Zakaria Adraoui, El Habib Benlahmar: Cross-platform mobile development approaches, IEEE, 2014

ISBN: 978-1-4799-5979-2/14/

[12] Aleksandar Karadimce , Dijana Capeska Bogatinoska: Using hybrid mobile applications for adaptive multimedia content delivery, IEEE, 2014

ISBN: 978-953-233-077-9

[13] What is a Hybrid Mobile App?

http://developer.telerik.com/featured/what-is-a-hybrid-mobile-app/

Date: 10.12.2016

[14] Ivano Malavolta , Stefano Ruberto , Tommaso Soru , Valerio Terragni: End Users' Perception of Hybrid Mobile Apps in the Google Play Store, IEEE, 2015

ISBN: 978-1-4673-7284-8

[15] Graph - Comparing five popular frameworks for mobile development in 2016: https://www.graph.uk/blog/mobile-development-frameworks-in-2016

Date: 31.10.2016

[16] Apache Cordova: https://cordova.apache.org/docs/en/latest/guide/overview/index.html

Date: 29.07.2016

[17] John M. Wargo: Apache Cordova API Cookbook, Addison-Wesley Professional., 2014

ISBN: 978-0321994806

[18] Cordova Core Plugins: https://cordova.apache.org/docs/en/latest/guide/support/index.html#core-plugin-apis

Date: 30.07.2016

[19] Ionic Framework: http://ionicframework.com/docs/guide/preface.html

Date: 30.07.2016

[20] iOS Technology Overview: https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html

Date: 01.11.2016

[21] WebSocket https://en.wikipedia.org/wiki/WebSocket

Date: 11.09.2016

[22] Socket.IO Wikipedia https://en.wikipedia.org/wiki/Socket.IO

Date: 02.10.2016

[23] WebRTC Wikipedia: https://en.wikipedia.org/wiki/WebRTC

Date: 16.10.2016

[24] WebRTC W3C Working Draft: https://www.w3.org/TR/webrtc/

Date: 16.10.2016

[25] Real-Time communication with WebRTC https://www.safaribooksonline.com/library/view/real-time-communication-with/9781449371869/ch01.html

Date: 29.11.2016

[26] Browser support scorecard http://iswebrtcreadyyet.com/

Date: 15.10.2016

[27] Node.js https://nodejs.org/en/

Date: 30.11.2016

[28] OpenShift https://www.openshift.com/

Date: 04.08.2016

[29] Ionic 2: http://ionic.io/2

Date: 01.11.2016.

[30] Android Platform Guide https://cordova.apache.org/docs/en/latest/guide/platforms/android/index.html

Date: 20.10.2016

[31] iOS Platform Guide https://cordova.apache.org/docs/en/latest/guide/platforms/ios/index.html

Date: 20.10.2016

[32] Disruptive Analysis: http://disruptivewireless.blogspot.co.at/p/blog-page_30.html

Date: 01.11.2016.

[33] WebKit Feature Status: https://webkit.org/status/

Date: 30.10.2016

[34] Cui Jian, Zhuying Lin: Research and implementation of WebRTC Signaling via WebSocket based for Real-time Multimedia Communication, Atlantis Press, 2016

ISBN: 978-94-6252-156-8

[35] WebRTC Architecture https://webrtc.org/architecture/

Date: 10.10.2016

[36] WebRTC Native API https://webrtc.org/native-code/native-apis/

Date: 01.11.2016

[37] XirSys: https://xirsys.com/

Date: 24.10.2016

[38] Automate WebRTC building https://pristine.io/2015/02/automated-webrtc-building/

Date: 22.10.2016

[39] WebRTC for iOS https://webrtc.org/native-code/ios/

Date: 20.11.2016

[40] Setting Up a Firebase Cloud Messaging Client App on iOS: https://firebase.google.com/docs/cloud-messaging/ios/client

Date: 20.11.2016

[41] Global market share held by leading smartphone OS Q2 2016 http://www.bforsite.com/android-os-reaches-86-market-share-world-wide/

[42] Henning Heitkötter, Sebastian Hanschke, Tim A. Majchrzak: Evaluating Cross-Platform Development Approaches for Mobile Applications, Springer, 2013

ISBN: 978-3-642-36607-9

[43] Cyclomatic complexity https://en.wikipedia.org/wiki/Cyclomatic_complexity

Date: 12.11.2016

[44] Halstead complexity measures: https://en.wikipedia.org/wiki/Halstead_complexity_measures

Date: 12.11.2016

[45] Maintainability Index: https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/

Date: 12.11.2016

[46] What is slowing down your mobile apps?: http://marketinghits.com/blog/infographic-what-is-slowing-down-your-mobile-apps/

Date: 15.11.2016

[47] Florian Rösler, André Nitze, Andreas Schmietendorf: Towards a Mobile Application Performance Benchmark, IARIA, 2014

ISBN: 978-1-61208-361-2

[48] Evaluate the performance costs of a Cordova app https://taco.visualstudio.com/en-us/docs/cost-cordova/#startup

Date: 22.11.2016

[49] Rahul Murmuria, Jeffrey Medsger, Angelos Stavrou, Jeffrey M. Voas: Mobile Application and Device Power Usage Measurements, IEEE, 2012

ISBN: 978-0-7695-4742-8