Haaga-Helia
ammattikorkeakoulu Oy

Kast – Watch Together Android application

Emil Kaidesoja

Bachelor's Thesis

Information Technology

2020

Summary

Author Emil Kaidesoja	
Degree program Information technology	
Title of the thesis Kast – Watch Together Android application	**Number of pages and appendices** 61 + 2

Kast is a live hangout platform which empowers users to share content and connect over the internet in real time. Kast allows communication with text and voice chat and sharing content with video capture technologies. The topic of this thesis is to develop the Kast Android application. This provides new and existing users a native Kast experience on their Android devices.

The need for applications such as Kast has never been higher. With the current state of the world, spending time with friends and family, watching movies, and organizing business meetings have moved over to the internet. Our mission at Kast is to give users the ability to do all these activities in a single platform.

In this thesis I go through the Kast ecosystem and features with details about the Android application scope and functionalities. I also explain what development tools and technologies I have used to complete the project.

I explore the Android environment and the Android framework by discussing the software stack, the programming language Kotlin and modern robust user interface elements used in the application.

An important aspect of this thesis is the audio and video sharing capabilities completed with Googles WebRTC technology. I explain how the WebRTC fundamentals work in theory and what protocols and technologies it consists of.

In the latter part of this thesis I cover details about the development of Kast Android. I dive deeper into the Party Activity, which is where all the WebRTC functionality takes place, and discuss the WebRTC Android specifics with information on how the features are implemented. I will also go through issues encountered during the development of the project. These sections can also work as a guide for a developer looking to implement WebRTC in their Android client.

Finally, I summarize the results of the project and evaluate which objectives were met. I also discuss my own learning experiences and envisage the future of the application.

Key words
Kast, Android, Android development, Kotlin, WebRTC, AndroidX

Table of contents

1 Introduction

The use of virtual socializing apps has been on the rise for quite some time and the COVID-19 pandemic has skyrocketed the market with millions of new users. For example, Microsoft Teams has experienced a 70 percent user growth in April this year, now totaling at 75 million daily active users (Warren 2020). We at Kast have experienced extraordinary growth during this year as well.
Most of the popular applications used today mainly focus on one specific use case. Work meetings tend to be organized using apps such as Zoom or Google Hangouts, for meeting new people or voice chatting Discord is popular, and hosting watch parties Netflix Party could be a good choice (Andrews 2020). The topic of this thesis is to develop the Android client of a social hangout platform called Kast, which can handle all the prementioned activities.

1.1 What is Kast?

Kast is a live video sharing application where users can share experiences in real time. Our mission is to empower groups of people to spend time together virtually, in any way they want. This can be in a shape of a watch party, video games, virtual classes, team meetings or just hanging out (Kast 2020). Kast was founded in 2015 as Evasyst and rebranded to Kast in the summer of 2019. I joined the Kast team in the spring of 2019 and have been working as a developer since. We are a global team of more than 20 individuals from different corners of the world. Our offices are in San Diego, California and in Helsinki, Finland.

Kast has multiple frontend clients which create the ‘virtual living room’ -like ecosystem. A PC or Mac user can use the downloadable desktop app or Kast web using a chromium-based browser. For phones and tablets the iOS app is available and of course, as the result of this thesis Kast is now also available for Android devices. This variety gives the user the ability to participate from their device of choice and not miss out on anything that happens in their parties.
Parties is where the action happens in Kast. They act like virtual rooms for the users to hop into and start streaming, or as we like to call it, Kasting. Parties can have up to 100 users watching with the ability to customize their view by toggling streams on or off, and full screening individual video streams. The layout limits the number of viewable streams to four, not counting the users own outgoing stream. The Kaster limit is capped at 20 people streaming simultaneously. The type of stream is configurable by the user. This means that they can stream from different sources based on their current client. For example, on

desktop and web, users have screensharing possibilities from individual application windows and more, and from mobile devices users can share their camera video and microphone audio. Both audio and video sources can also be streamed individually. On top of this all parties also have a party chat for text messaging and animated reactions.
Users have specific roles in each party. These range from a basic party member, to a Kaster, with streaming rights, to a party moderator, with abilities to promote and demote Kasters, and the owner of the party with all the rights. Parties can be specified private or public, these mean free to join by anyone or invite only. 90 percent of the action in Kast happens in private parties inside micro communities.
Kast parties are the core functionality of our platform. Kast Android has all these prementioned functionalities implemented but, this thesis will focus on a couple of things more specifically. These are the Android development framework, which gives us modern UI elements to build the Kast Party experience with, and the media streaming enabled by Googles WebRTC technology.

1.2 Goals and objectives

The goal of this thesis is to develop a modern Android application with a robust user interface and media streaming functionalities with WebRTC. This goal divides the project into two different subsections. Firstly, we will look at how we can build the best possible Android user interface with modern elements provided to us by the Android framework. This also includes the choosing of a programming language which promotes code readability and application maintainability without sacrificing efficiency or runtime.
Secondly, we will do a deep dive into WebRTC. Here we discuss the basic principles and concepts of WebRTC in general. After that we look at how we can achieve our goal of streaming media through an Android device with examples on how WebRTC is implemented in Kast Android. We will also discuss corner use cases and identify practices with which we can achieve the best functioning product.
An extra subgoal for the thesis is to provide useful guidance and information to any developer interested in WebRTC in general or looking to implement it in their own Android application.

Kast Android is the first application this size where I am the sole developer, so it will bring me new sets of challenges to solve. This will deepen my knowledge of Android development and WebRTC. Even though I am the developer for this project we do have a team behind the product with designers and other product team members, therefore the result is also a team effort.

At the end of the thesis I will reflect on the goals met and my personal learning experience, with discussion on how the application will be developed going forward.

1.3 Tools

For the project to be completed successfully we require the right tools and libraries. In this section I detail the reasoning which lead me to use the tools and libraries mentioned and in the upcoming sections I will specify more on what make these beneficial.
With the goal of a modern application in mind, we must identify which tools can assist us in reaching this goal. The requirements that a tool or library should have are:

- Continuous support and maintaining from the developer.
- The library needs to perform a task that is difficult enough or too time-consuming to implement ourselves.
- Flexibility in implementation, we need to have the ability to configure things specific to our use case.
- A good readability and writability to efficiency ratio. The less efficient tool the easier it must be to implement, and vice versa, if the tool is very efficient, we are willing to sacrifice other aspects for it.

First things first we need an IDE, which stands for Integrated Development Environment, for this I chose Android Studio, which has various benefits for Android development, as the name would imply.
For the programming language on Android we can choose between Java or Kotlin. I chose Kotlin since it is a more modern and flexible programming language than Java. In Sections 2.1 and 2.2 I go into more detail on what makes Kotlin great.
A modern application also needs a modern UI with robust elements. The Android framework comes with a support library which has basic elements to build an interface with. But this does not meet most developers' requirements, including ours, and therefore the support library has been deprecated. For the UI elements, I will be utilizing the AndroidX library which has been developed to replace the now deprecated support library. AndroidX has various elements that meet our needs in things such as efficiency and flexibility. Section 2.3 includes more specifics on the AndroidX library and a couple of more interesting elements used in the project.
With the streaming of the media in the Kast parties we will use the Google WebRTC technology. WebRTC is what the core of all our applications rely on, so this decision was not mine to make. WebRTC solves a platitude of issues in the connectivity of different users and getting media streams from the devices. More on WebRTC's background and theory, in Section 3 and Kast Android implementations in Section 4.

2 Android framework

Android is a Linux based operating system running on billions of different devices. It is made to be open source and owned by Google (Android 2020). On top of being open source, the development environment for Android is very advanced. Android Studio comes with all the necessary tools for the developer to get started. This is only logical since it is also the official IDE of choice for native Android development (Android Developers 2020). Here are a couple of my favorite features from Android Studio (Android Developers 2020).

- Intelligent code completion support for native Android languages and C++
- Gradle build configuration system for different build variants, for example, configuring builds which use different backends.
- Application real-time profilers to monitor app statistics e.g. networking and heat generation.
- AVD, Android Virtual Device, manager for downloading and managing Android emulators to run your code in.

These are just a fraction of what Android Studio offers, but I have used these during the project extensively.

An Android application consists of multiple components working together to create an experience. In Figure 1 you can see a simplified representation of the Android software stack from the Android developer documentation with the different sections of the platform from lower levels to higher abstraction. I have included the most relevant information and systems from each section to this project.

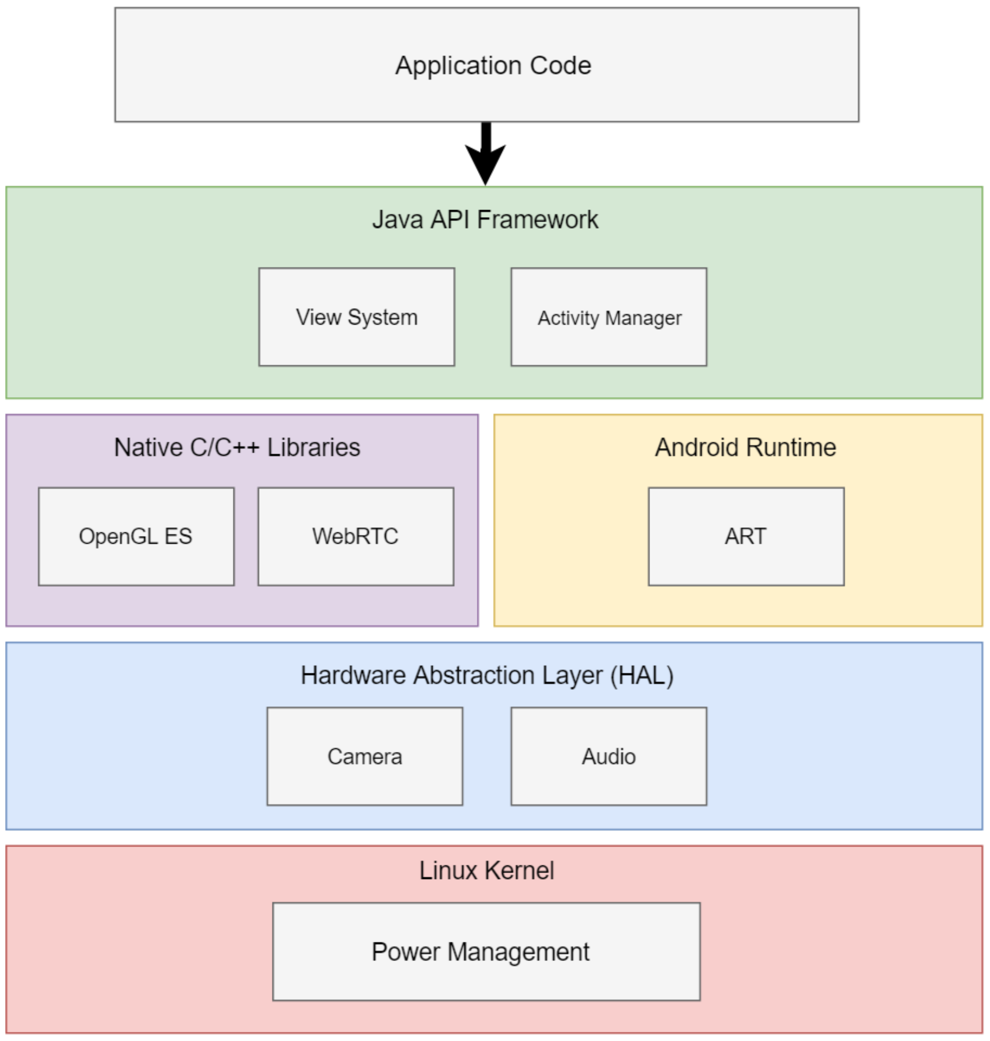


Figure 1. Simplified version of the Android software stack (in accordance with Android Developers)

In the upper part of the figure, outside the visualized software stack is our application code. This consists of everything we write such as the Kotlin and XML code. The application code communicates with the Java API framework, to make work happen in the application. The Java API section contains the view system and activity manager among other things. The view system provides us with all the elements we can use to build the User Interface, UI in short, and the activity manager handles activity specific features such as their lifecycle. With these combined we can create activities which are one of the core things on Android development.

An activity is a view of the application that has one window with multiple layout elements building the UI and the application code handling the functions. Worth to note here is that even though I say that the Java API section handles activity lifecycle and we program and design our Activities, the code that we write is not part of the Java API framework but rather the API provides us tools that we use to write the code and design the layouts.

The beauty of activities come from the fact that they can be started and finished at will. This is useful if the user needs to be redirected to a specific activity from different parts of the application (Android Developers 2020). Activities are started through Intents, which are configurable objects containing information about what to do next. This can include information such as what activity to start and what data to pass on to it. This is the usual use case, but they can also be used for communication with other applications and services on the Android device. For example, accessing the devices gallery, hardware camera, and sending emails (Android Developers 2020).

The activity UI is generally built with XML, Extensible Markup Language, but it can also be manipulated programmatically through the application code with animations and color changes for example (Android Developers 2020). More on the UI elements used in this project in Section 2.3.

From Figure 1 we can also see where our native libraries reside. I included WebRTC into the Figure as well since it helps us visualize the code structure. Android devices do not have the WebRTC library out-of-the-box, so we will add it into the project manually. We also talk about OpenGL ES when we need high performance graphics rendering, in Section 4.

Android Runtime is where our application eventually runs. When the application is built, installed on a device, and started, ART creates a separate process where it translates the program to native instructions for the hardware and systems (Android Developers 2020).

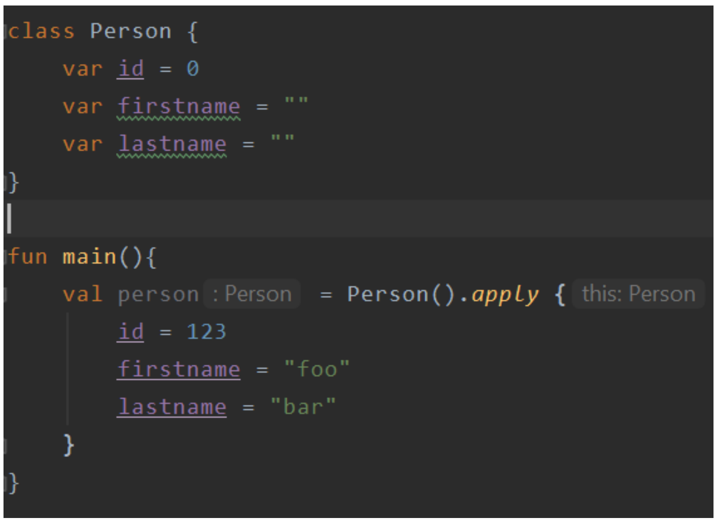
2.1 Kotlin

Previously I described my requirements for choosing tools for this project. In this section I explore Kotlin in more detail and the more specific reasons why I chose it as the programming language for this project.

Android applications are written in Java or Kotlin. Both languages compile to Java bytecode and ultimately run on the ART (Markovic 2020). At the end both programming languages are compiling and running relatively the same, Kotlin being a little slower, so the major differences come from writing the code. Compared to Java, Kotlin is a more flexible language with multiple benefits and fixes for problems Java developers face. Here are couple of examples on what Kotlin can do and what lead me to choose it over Java.

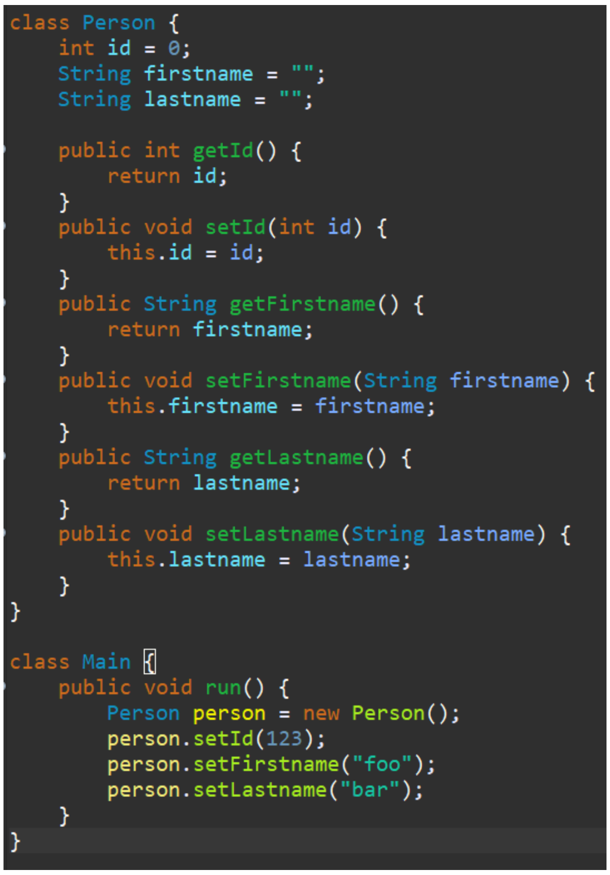
- A significant amount of less boilerplate code with lambdas and removal of key words
- Extension functions
- Variable null safety and smart casting
- Kotlin coroutines

Kotlin allows the removal of a lot of unnecessary boilerplate code to get things done faster and help with code readability. For example, lambda expressions, which are unnamed functions for single use or passing on to a higher-order function. These paired with the removal on unnecessary keywords, such as ‘new’ when constructing objects add up to remove dozens of lines of code from the codebase (Kotlin 2020). In Figures 2 and 3 we can see a simple example of instantiating and setting attributes to a Person class. In Java we use basic setters and in Kotlin we use a scope function ‘apply’, which is a form of lambda. Note the significant amount of less lines of code in Kotlin.



```kotlin
class Person {
    var id = 0
    var firstname = ""
    var lastname = ""
}

fun main(){
    val person : Person  = Person().apply { this: Person
        id = 123
        firstname = "foo"
        lastname = "bar"
    }
}
```

Figure 2. Constructing and setting values to a Person class in Kotlin

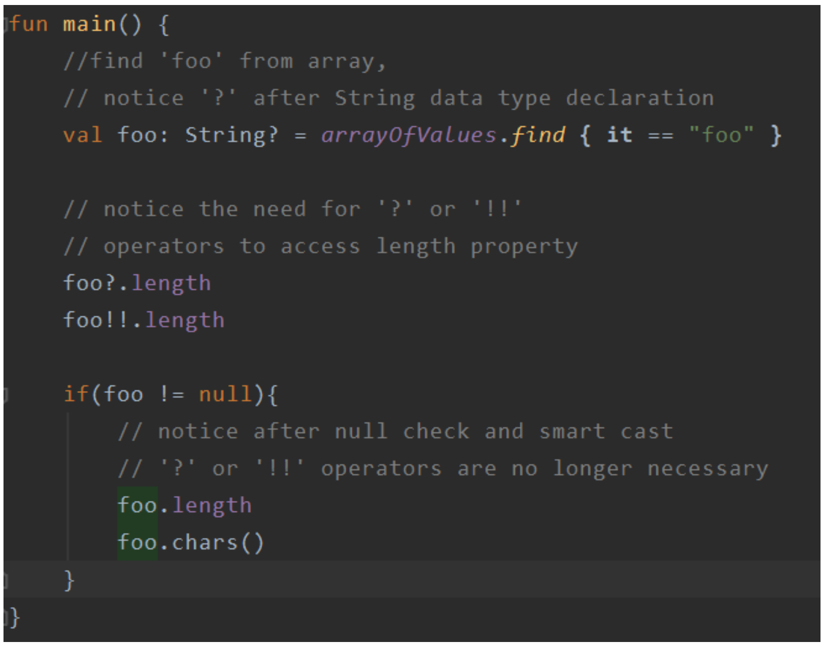


```java
class Person {
    int id = 0;
    String firstname = "";
    String lastname = "";

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getFirstname() {
        return firstname;
    }
    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }
    public String getLastname() {
        return lastname;
    }
    public void setLastname(String lastname) {
        this.lastname = lastname;
    }
}

class Main {
    public void run() {
        Person person = new Person();
        person.setId(123);
        person.setFirstname("foo");
        person.setLastname("bar");
    }
}
```

Figure 3. Constructing and setting values to a Person class in Java

Kotlin also gets rid of primitive data types and handles all variables as objects with keywords as var for a mutable variable and val for a constant. Not having access to primitive datatypes could be a hindrance, but because of this, Kotlin can have the ability to write extension functions to all classes. Extensions allow the developer to write functions for classes or data structures without the need to create a new class and inherit from the original one. This is useful when doing a similar manipulation multiple times, or if working with a third-party library which source code you cannot access (Kotlin 2020).

Another important feature in Kotlin is null safety and smart casting. Null safety is self-explanatory, objects or classes cannot be null unless explicitly declared that way. For example, a nullable String would be declared with a '?' at the end of the data type declaration, otherwise passing or declaring the variable as null causes a compilation error. Accessing attributes from nullable variables can be done using the '?' or '!!' operators. Both check if the value is null before accessing its properties. The difference is that '?' stops the operation if the values is null and '!!' throws a null exception.
Casting means converting an object into a different data type. This can cause exceptions if done without checking the initial data. For example, casting a String 'Hello World' into a numeric value does not make any sense. Smart casting is the compiler automatically casting the values to the right data types based on the actions or checks done previously in the code. This means that nullable data can also be smart cast into non-nullable data if the variable is not a complex expression. A complex expression is a variable that can be accessed in multiple places in the code, so after checking if it is null it could be set back to null from somewhere else, therefore we cannot be sure of its non-nullability. If a variable is local, it can be smart cast into a non-nullable value. Therefore, after checking that the data is not null the following code can handle it as a non-nullable data structure (Kotlin 2020).
In Figure 4 is an example of trying to find the value 'foo' from an array of strings with a 'find' scope function. If the array does not contain the value, the function returns null, so we must access its attributes with the null safety operators. After we do a null check the variable is smart cast to a regular String object, and we can be assured that it is not null.



```
fun main() {
    //find 'foo' from array,
    // notice '?' after String data type declaration
    val foo: String? = arrayOfValues.find { it == "foo" }

    // notice the need for '?' or '!!'
    // operators to access length property
    foo?.length
    foo!!.length

    if(foo != null){
        // notice after null check and smart cast
        // '?' or '!!' operators are no longer necessary
        foo.length
        foo.chars()
    }
}
```

Figure 4. Example of null safety and smart casting

2.2 Android multithreading with Kotlin coroutines

When a program starts, it creates a dedicated process for the application to do its work on. This process can have from one to multiple threads where the actual code executes (Bauer 2017). This also applies to Android applications. Android starts an app with only one thread called the ‘Main’ thread. This is the thread responsible for building the UI and interacting with the user through that UI, therefore it is also referred to as the UI thread (Android Developers 2020). In this section we will explore Android threading, which is another core aspect of modern Android development, and how we can leverage a Kotlin feature, called coroutines, with threads.

There are issues with a single threaded application. It can only execute one block of code at a time. Subsequent code will be added to the back stack, which is essentially a queue of code and tasks waiting to be executed. When code is running on the main thread, the main thread is ‘blocked’, and while the code executes the application will not respond to other actions from the user until the thread is idle again. Blocking the main thread is considered bad practice and something you should never do (Android Developers 2020). With a simple app you should be fine with just a single thread, but when the application grows the need for multiple threads and asynchronous execution grows as well.

Android provides worker and background threads to use for time consuming tasks and keep the application responsive. Worker threads should be used for tasks that need time to complete, such as high computational tasks, and background threads should be used for API calling. An app can only have one main thread, but multiple worker and background threads. One thing to note is that worker or background threads cannot access or manipulate the app UI. Trying to do so will cause the application to crash. Usually after a worker thread has completed its task, it is common to need to update the layout or inform the user that the task has been completed. This can be done by forcing the UI task to be run on the main thread. This adds the action to the main thread execution queue, which then runs in order. If the application is developed with proper multithreading the main thread should execute the code virtually instantly (Android Developers 2020).

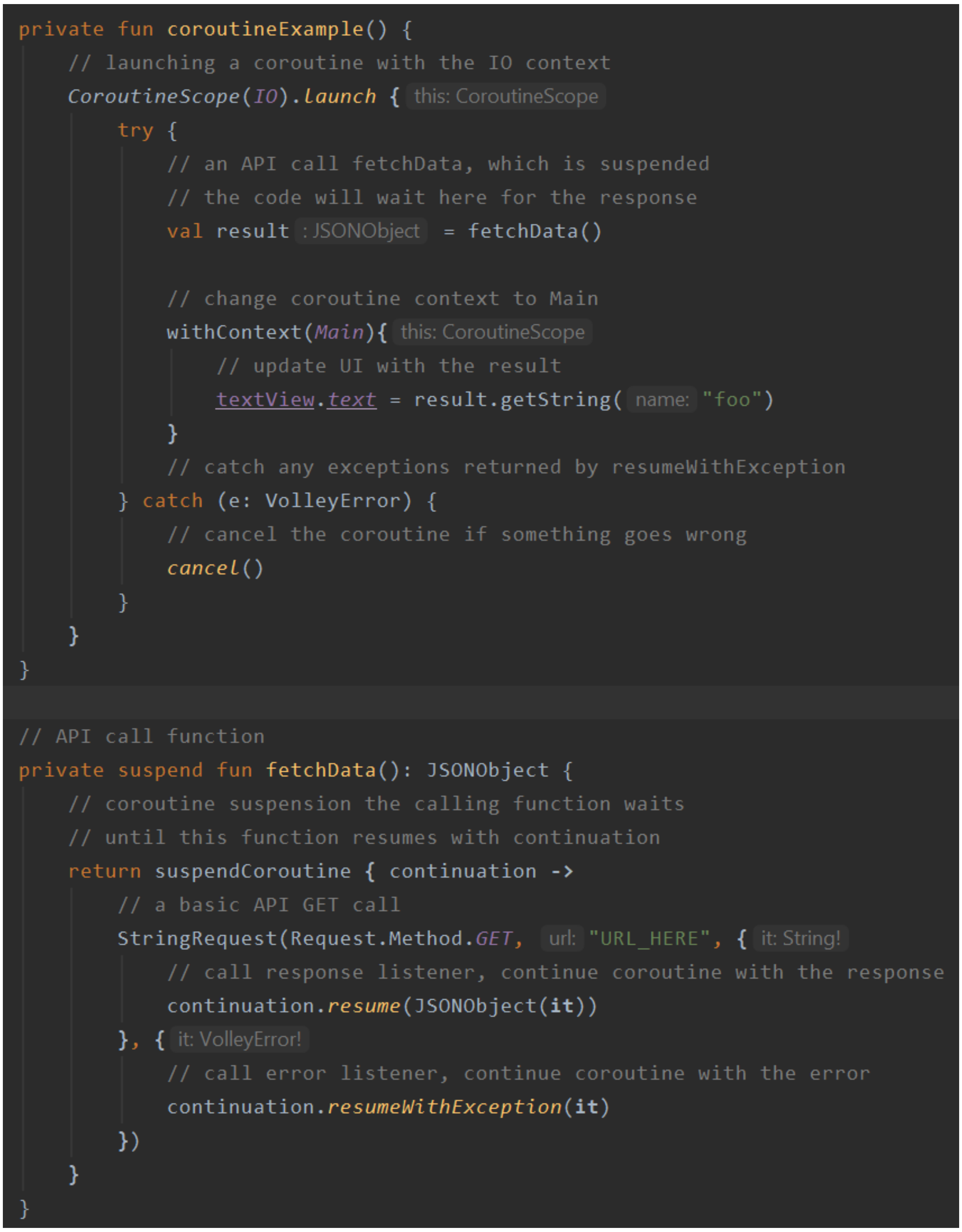
Understanding thread manipulation on Android can be a big game changer, but combining it with Kotlin coroutines, can be even more powerful. Coroutines allow the developer to write more asynchronously running code. This means that multiple tasks and functions can be running at the same time. A coroutine can be configured with a context which specifies what thread it should be run on. For example, ‘default’ runs on a worker thread. You can also create a new thread for a specific coroutine (Kotlin 2020). Table 1 shows the corresponding thread type to coroutine contexts.

Table 1. Corresponding Coroutine context to thread type

Coroutine context	Thread type
Main	Main Thread
Default	Worker Thread
IO	Background Thread

As Android threads, coroutines can also change their context and therefore the thread they run on. This way we can update the layout after a task has been completed (Kotlin 2020).
Coroutines have similar features as Android threads, but they are not the same thing. Coroutines are more lightweight than threads and a single thread can have multiple coroutines executing. This is possible with the ability to suspend and continue coroutines. Suspending means telling the coroutine to wait for a specific action to complete before continuing again. While another coroutine is suspended and waiting, the parent thread can keep on executing tasks and launching new coroutines. This is useful especially with API calling. We can write API calls with suspend functions and have the code wait suspended for the response while the parent thread launches other coroutines with new API calls. This way we can run code more asynchronously and release the UI for the user as soon as we have the critical data necessary.
Suspended coroutines can also be manually cancelled if something goes wrong or the result is not what was expected. This removes the need for various checking of the data along the way and leads to safer code (Android Developers 2020).

In Figure 5 we can see an example of starting a Kotlin Coroutine with the ‘IO’ context, doing a suspended API call, and updating the UI with the result.



```
private fun coroutineExample() {
    // launching a coroutine with the IO context
    CoroutineScope(IO).launch { this: CoroutineScope
        try {
            // an API call fetchData, which is suspended
            // the code will wait here for the response
            val result : JSONObject  = fetchData()

            // change coroutine context to Main
            withContext(Main){ this: CoroutineScope
                // update UI with the result
                textView.text = result.getString( name: "foo")
            }
            // catch any exceptions returned by resumeWithException
        } catch (e: VolleyError) {
            // cancel the coroutine if something goes wrong
            cancel()
        }
    }
}

// API call function
private suspend fun fetchData(): JSONObject {
    // coroutine suspension the calling function waits
    // until this function resumes with continuation
    return suspendCoroutine { continuation ->
        // a basic API GET call
        StringRequest(Request.Method.GET,  url: "URL_HERE", { it: String!
            // call response listener, continue coroutine with the response
            continuation.resume(JSONObject(it))
        }, { it: VolleyError!
            // call error listener, continue coroutine with the error
            continuation.resumeWithException(it)
        })
    }
}
```

Figure 5. An example of a Kotlin coroutine API call

With so many options for thread manipulation and creation, you might find yourself in a situation with a lot of threads running at the same time. This might cause performance issues on some devices, or unexpected execution orders. Newer devices with better CPUs can handle more asynchronous execution, but issues might rise on older devices.
Possibilities exist to assure proper performance for as many users as possible. For example, you should prioritize your threads, especially on activity startup. The creation of an activity usually queues the most functions for execution. Ranking the functions in priority can assure that the UI is accessible as soon as possible, and it also limits the various order the functions finish. You should also cancel coroutines if something invalid is detected to free the memory and resources for other tasks.

2.3 AndroidX UI elements

AndroidX is a support library that consists of various libraries which provide better tools for application development and backward compatibility (Android Developers 2020). Kast Android uses multiple UI elements provided by the AndroidX library; I will go through some of the more advanced and interesting ones.

2.3.1 Constraint Layout

When designing an activity’s layout, the parent container determines mostly how the child elements will behave. The parent container could be chosen from different elements, but AndroidX gives a dedicated option designed to handle container layouts efficiently. This is the ConstraintLayout, and as the name implies, it allows the positioning of elements based on constraints to other sibling elements in the layout. Each element needs at least a horizontal and a vertical constraint. With constraints you can build a flexible UI that works well in screens with different sizes and shapes. With margins, element sizing, constraints and positioning bias the layout can be configured so that the elements are always properly positioned and visible. Constraint positioning bias defines how much the element should move between constraints in the same orientation. For example, two horizontal constraints position the element in the middle of the container, with constraint bias you can adjust this positioning further to each side (Android Developers 2020).
Configuring a ConstraintLayout is simple, you can create and remove constraints by either dragging and connecting with Android Studios Layout Editor’s visual tools or write them with pure XML (Android Developers 2020).

An important tool that goes with ConstraintLayout is ConstraintSet, which is an object containing information about the layout’s constraints. With a ConstraintSet you can modify constraints in a ConstraintLayout. This is helpful when doing minor element constraint changes based on the user’s actions, for example, extending a view on click, or updating the whole layout. If you create a separate XML layout with the same element id’s but different constraints, you can clone those constraints into a set and apply it to the initial layout (Android Developers 2020). This is useful for bigger layout changes, such as orientation changes, where the UI might need drastic updates. To make these layout changes look better, you can combine the Constraints set with a TransitionManager. The TransitionManager needs a transition type and the root layout it applies it to. Then all the next layout changes will be animated with the specified type (Android Developers 2020). Even simple layout transitions make a huge difference for the user experience and the overall feel of your application. For a modern application this is a must have and TransitionManager gives great functionality out-of-the-box.

2.3.2 Recycler View

Another AndroidX essential is the RecyclerView, which is an updated version of its predecessor the ListView. Most applications display some sort of scrollable list of data, this is what we need the RecyclerView for. Adding the view to the layout is just an XML element, but population it with data is more advanced. The result is a combination of different components working together. Here are the things we require.

- One or more XML layouts for items
- A layout manager
- A RecyclerView adapter

The layouts we populate with the items which will be displayed in the way specified by our layout manager. The layout manager can be, for example, a LinearLayoutManager for displaying items in a simple linear list, or it could be a GridLayoutManager for a grid list with several items next to each other.
Lastly, we need an adapter, which oversees populating the list and handling interactions with the user. Reference the adapter code example in Appendix 1 when needed. The appendix has the implementation of an adapter with two different layouts for items and a click listener.

The adapter class needs the list data which to display, and one or more specified view holders. The adapter inflates the view holder, which is a skeleton of one of the XML layouts. With multiple we need to specify which one to inflate based on the data in the current position of the list. See getItemViewType in Appendix 1. To populate the item with the actual data in the list the adapter runs onBindViewHolder. Here we insert the data in its proper positions programmatically and it will be displayed in the RecyclerView of the activity (Android Developers 2020).

It is common to have the need to click specific items in the list. The proper way to do this is to create an interface with the functions you need in the activity. The interface is implemented by the activity in which the recycler resides. The adapter takes the click listener as its parameter and passes it to each view holder where the click listener is set to the element. Then by overriding the interface functions in the activity those functions are called when the click listener triggers. (Senior 2018).

The major updates to RecyclerView compared to ListView are performance related. The adapter is optimized with the handling of the items. When the adapter is created it only inflates the view holders currently visible in the layout. So, if you have a list of a hundred items but you only show five, the adapter runs onBindViewHolder only five times. You can set the start position of the recycler to be anywhere on the list, and it will still only populate the visible items. While scrolling, it populates the views just off the screen and attaches them to the recycler, this makes it seem like a list of items. On scroll some of the items go off screen. These the adapter clears and prepares for re-use, hence the name recycler. Now the items coming to view use the previously recycled view holders. This gives a massive memory and performance boost. With a list of hundred items you might only ever have ten view holders, based on the number of items visible at the same time (Android Developers 2020).
With the recycler populated and the adapter initialized with click listeners and actions the data might change a lot. Items could be removed, added or the dataset might completely change. The adapter handles changes very well and updating the displayed data is just as easy as updating the data list variable and calling an adapter method based on the action taken. Here are a couple of useful adapter functions, note that there are more available.

Table 2. Recycler Adapter update functions and intended use cases

Adapter function	Intended use
notifyItemInserted	Notifies the adapter that an item has been inserted in the given index.
notifyItemRangeInserted	Notifies the adapter that items have been inserted from the given start index forward with the amount specified.
notifyItemRemoved	Notifies the adapter that an item has been removed in the given index.
notifyDataSetChanged	Notifies the adapter that the whole dataset has changed.

As the names in Table 2 imply these functions tell the adapter what has happened to the dataset and it will do the necessary actions. This usually means re-rendering the items in the specified positions (Android Developers 2020). Managing a recycler is a straightforward process once setup properly.
There is, however, a caveat that the developer needs to remember with recycling. When a holder is recycled its modifications done while binding the data to it will not be reset. This needs to be done manually in either onViewDetachedFromWindow or in onBindViewHolder when binding the next item to the holder. For example, given a list of a hundred items where some need a background color of red and the others a blue one. You might think that designing the default XML background blue then checking the need for a red one and setting it in the adapter is sufficient. With continuous scrolling one by one all the items would receive a red background since the overriding of a background is never reset to the default. With a simple example, as in Appendix 1, this can seem obvious, but it is something to remember when developing with multiple view holders and data types and experiencing weird behavior.

Learning the recycler adapter ins and outs can seem time consuming, but it is essential since a RecyclerView is not the only element it can be used in. AndroidX introduces the ViewPager2, which allows sliding between elements horizontally or vertically. This gives a slideshow like effect and is a common design choice in applications today. ViewPager2 has built in gesture detectors out-of-the-box to handle the sliding and swiping, which makes it popular and easy to implement (Android Developers 2020). The difference between ViewPager2 to its predecessor the ViewPager is the ability to inflate views with the recycler adapter. The process and the methods are the same as with the RecyclerView so learning multiple adapters is not necessary (Birch 2019). This also means that you might not need to implement a separate adapter at all. The adapter for your list might already work for the ViewPager2.

3 WebRTC

WebRTC, which stands for Web Real-Time Communication, is an open source library developed by Google. The technology allows peer-to-peer communications between web browsers and mobile devices. The WebRTC library consists of different technologies and network protocols to standardize how users can communicate through the web. Peers can exchange different media from audio and video to files and raw JSON data. The library gives the developer simple but powerful tools to implement their own WebRTC based solutions from basic webcam conferencing apps running on browsers to more advanced multiplatform media sharing applications. (WebRTC 2020). WebRTC was established in 2011 but version 1.0 was announced stable only in 2017 (Blume 2018). Because of this, the library can still be considered in its early stages and therefore updates, and changes are frequent. In this section I will go through the WebRTC fundamentals needed to get started implementing your own solution. Later in Section 4 I dive deeper into Android WebRTC specifics used in this project.

Getting video and audio through the internet is hardly a new thing but sharing these in real-time is a different matter. Making RTC a possibility is a combination of multiple technologies working together. Issues like dropping connections, data loss and NAT traversal would all require different dedicated software to be able to share basic media. WebRTC solves these issues, among multiple others, and comes with prebuilt access to the required devices and technologies such as camera, microphone, video and audio encoding and decoding, transportation layers and session management. On top of that the API comes built in with all modern web browsers, and a pre-compiled library can be added into your Android or iOS project dependencies (Ristic 2015, 1-7).

3.1 Media streams

There are multiple gears that work together on creating connections and getting the data flowing between users. The first aspect of WebRTC to explore is getting the users media into a media stream object, which eventually can be sent through a connection. Getting the media streams is dependent on the platform you are working on, we take a closer look at how we do this on Android in Section 4.2.6, but there are similarities and general rules. All platforms have classes representing the media objects. The methods of getting these from the device varies but the classes have similar behavior and functions for similar actions. These are shown in Table 3 with their intended use.

Table 3. WebRTC media classes with explained use case

Class name	Intended use
MediaStream	Represents a media stream between users, which contains one or more media tracks.
AudioTrack & VideoTrack	A media track added to the MediaStream object, which holds the actual media data.
MediaConstraints	Specific rules for a media stream to follow.

The MediaStream is the object which contains all the VideoTrack and AudioTrack objects, each for their respective media type. The media tracks contain the actual media that is being shared, so these are where we add our own media and receive other users' media from. The media stream object can have one to multiple different media tracks from different media sources. (MDN Web Docs 2020).
Audio and video tracks inside a MediaStream can be configured more specifically with MediaConstraints. As is evident from Table 3, these give the stream specific rules which it follows to get the desired result output. The rules are key value pairs representing different attributes which can be specified for audio or video. Rules can be mandatory or optional, which specify the constraint priority. For example, it can be paramount to force a specific aspect ration for the video, but the actual height and width of the stream is not as important (Ristic 2015, 15-21).

3.2 Peer Connection and signaling

Now let us look at how to create a successful connection to another application and send and receive streams through the web. To achieve this, we need to establish a peer connection. A peer connection is how WebRTC handles communication between two different devices using the peer-to-peer protocol (WebRTC 2020). In a peer-to-peer, P2P, network end users, or peers, are connected to each other directly without the shared data going through a dedicated server. By default, all peers act as servers and clients with the same privileges (Rouse 2019).
WebRTC peer connections use multiple protocols to establish the connection and send or receive data packets. Let us first look at the transfer protocols. When the connection is established the media flows using User Datagram Protocol, UDP in short, which prioritizes speed over reliability. In an RTC application we need to send and receive a lot of data in a short period of time, and therefore we require a relatively fast connection and we should not waste resources on unnecessary actions. UDP is a great option for this since it allows

the loss of sent data packets and does not check that every frame has been received. Some data packets might get lost on the way or they might arrive in a different order, UDP does not care about this, since it is more important to send frames continuously. The end user might experience less frames that originally sent, but usually this is hardly noticed since our brain fills in the gaps (Ristic 2015, 31-33). For Kast it is also more important that the stream is in real time rather than with the best possible frame rate, or quality.
Even though UDP works nicely, it is unreliable, and we cannot use it for everything. UDPs counterpart TCP, which stands for Transmission Control Protocol, is needed in WebRTC as well. TCP does all the things and steps that UDP skips but sacrifices speed in doing so. Every sent packet must be acknowledged by the recipient or the sending process will be halted, and the missed packets are sent again. Because of high reliability requirements, TCP is the standard for most web communications that are not concerned about speed (Ristic 2015, 32). In WebRTC a peer connection needs to be established before the media can be exchanged. This connection establishment is called signaling and negotiation, and this is where we use TCP.
The process has multiple steps, but generally you can divide it into two sections. First, we need to find the other device over the network and then we need to figure out how to exchange media. This process also requires a signaling server, which I cover in Section 3.3. As of now just assume we have a working server which gives us connection candidates and relays our messages forward. So, the server handles the first section of finding other users over the internet. This leaves us the second part of figuring out how to exchange media. Figure 6 shows the signaling process of creating a peer connection with another user.

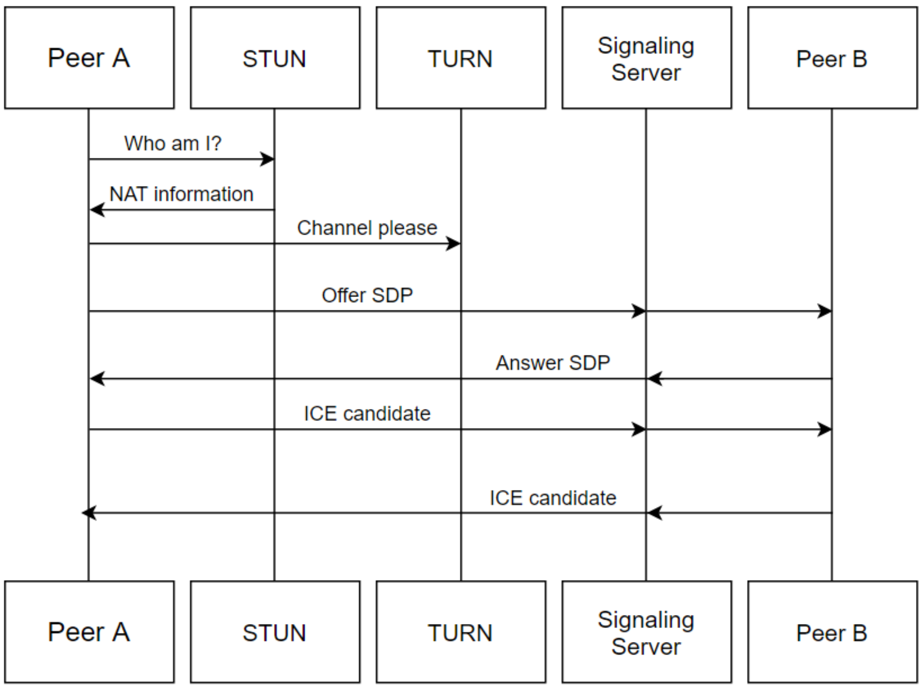


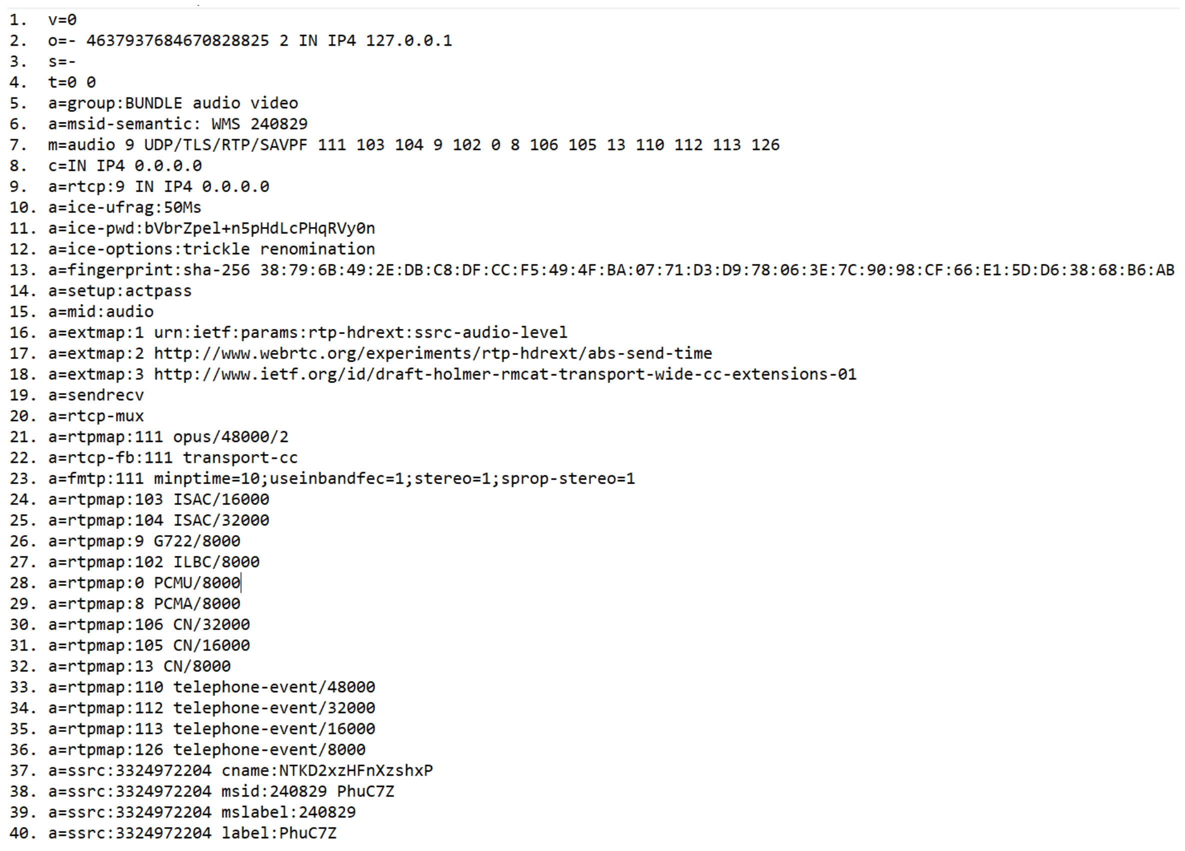
Figure 6. Peer connection establishment flow (in accordance with MDN Web Docs)

In Figure 6 we can see the general flow of establishing a peer connection. In the coming sections we will cover all these steps therefore reference to Figure 6 when we explore these phases. After all these steps have completed, we have successfully established a peer connection.

3.2.1 Session Description Protocol

A signaling session contains two roles, the sender, and the recipient. The process has similar steps but in the different order. We will start the signaling from the sender's viewpoint.
After we connect to our server, it gives us potential connection candidates. Therefore, we know who the other user is, and we have a way to contact them. But we still need to exchange more information about each other's device and about the session we are going to establish.
Firstly, we need to generate an SDP. SDP stands for Session Description Protocol, and as the name implies it contains the information needed to establish the connection and successfully exchange media. We initiate the peer connection signaling process by creating an offer, which generates an SDP. The SDP is a key-value pair raw string file with values separated with line brakes. It contains information for things such as, what media types is the session going to have, what codecs are available on the device, IP addresses, fingerprints and more (Ristic 2015, 37-39).

The goal of this thesis is not to examine what each SDP line means, nor is that completely necessary since the point of WebRTC is to make things easier for the developer. That said, when developing with WebRTC you will encounter SDPs and probably will need to investigate them, so it is important to understand the structure and basics. Here is an example of an SDP taken from Kast Android when initiating a peer connection with a camera video track and a microphone audio track. I have divided it to Figure 7 for the header and the audio section and to Figure 8 for the video section, I have also numbered the rows for better referencing.



```
1.  v=0
2.  o=- 4637937684670828825 2 IN IP4 127.0.0.1
3.  s=-
4.  t=0 0
5.  a=group:BUNDLE audio video
6.  a=msid-semantic: WMS 240829
7.  m=audio 9 UDP/TLS/RTP/SAVPF 111 103 104 9 102 0 8 106 105 13 110 112 113 126
8.  c=IN IP4 0.0.0.0
9.  a=rtcp:9 IN IP4 0.0.0.0
10. a=ice-ufrag:50Ms
11. a=ice-pwd:bVbrZpel+n5pHdLcPHqRVy0n
12. a=ice-options:trickle renomination
13. a=fingerprint:sha-256 38:79:6B:49:2E:DB:C8:DF:CC:F5:49:4F:BA:07:71:D3:D9:78:06:3E:7C:90:98:CF:66:E1:5D:D6:38:68:B6:AB
14. a=setup:actpass
15. a=mid:audio
16. a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
17. a=extmap:2 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time
18. a=extmap:3 http://www.ietf.org/id/draft-holmer-rmcat-transport-wide-cc-extensions-01
19. a=sendrecv
20. a=rtcp-mux
21. a=rtpmap:111 opus/48000/2
22. a=rtcp-fb:111 transport-cc
23. a=fmtp:111 minptime=10;useinbandfec=1;stereo=1;sprop-stereo=1
24. a=rtpmap:103 ISAC/16000
25. a=rtpmap:104 ISAC/32000
26. a=rtpmap:9 G722/8000
27. a=rtpmap:102 ILBC/8000
28. a=rtpmap:0 PCMU/8000
29. a=rtpmap:8 PCMA/8000
30. a=rtpmap:106 CN/32000
31. a=rtpmap:105 CN/16000
32. a=rtpmap:13 CN/8000
33. a=rtpmap:110 telephone-event/48000
34. a=rtpmap:112 telephone-event/32000
35. a=rtpmap:113 telephone-event/16000
36. a=rtpmap:126 telephone-event/8000
37. a=ssrc:3324972204 cname:NTKD2xzHFnXzshxP
38. a=ssrc:3324972204 msid:240829 PhuC7Z
39. a=ssrc:3324972204 mslabel:240829
40. a=ssrc:3324972204 label:PhuC7Z
```

Figure 7. The header and audio section of an SDP

Lines from one to six give some general information about the incoming session. In line two we have a unique ID for the session as the Long. The last three sections on line two are the network and the IP address in IPv4 format. Line five describes the media bundle we are sending; in this case the stream has both audio and video tracks.
Line seven contains media information and describes the media in question, which is audio in Figure 7. The capitalized letters are the protocols used, of which you might recognize UDP from earlier. The numbers correspond to different media format descriptors supported by the device. You can find the corresponding descriptor for the number below in the lines 21 to 36, starting with ‘a=rtpmap:<number>’ (Webrtchacks 2016). For example, specific information for 111 can be found on line 21.

Lines from 10 to 14 help us to negotiate security information with our peer. Later in the signaling process when exchanging ICE candidates covered in Section 3.2.2, both peers are going to use the credentials from ‘ice-ufrag’ and ‘ice-pwd’ to avoid connection attempts from unauthorized endpoints. The fingerprint, which is a certificate hashed with the sha-256 algorithm, is a WebRTC requirement for DTLS-SRTP key agreement (Webrtchacks 2016). DTLS-STRP is a key exchange mechanism which ensures us that the information during signaling is coming from the correct peer. If the fingerprints do not match, the message will be discarded, and the session is rejected (WebRTC Glossary 2020).
In line 15 we specify the media for the coming lines, so all the information until the next media in line 41, will be for audio.

From line 21 we get to the audio codec lines referred earlier in line 7. The interesting lines here are 21 and 23. Line 21 describes the audio codec mostly used, which in this case is Opus. Since we are capturing audio and video in a relatively high quality, we need to break it down to smaller bits to be able to send it over the internet efficiently. This is where we use codecs. These enable the recipient to then decode the received information and experience the media almost at the same quality as captured (Ristic 2015, 3-4). We use Opus as our audio codec, which is a great choice for audio over the internet with the ability to support various bitrates and instances (Opus 2017).
Going back to the SDP line 21. The format of the line is <codec>/<sample rate in Hz>/<number of channels>. As we can see Opus has two audio channels which means we have support for stereo audio out-of-the-box. The default output is mono, but we can enable stereo audio with a little trick called SDP munging.

SDP munging means manually modifying the SDP file before applying it to the connection With SDP munging we can remove unwanted codecs or prioritize some over others, or another common use case for munging, such as ours, is enabling stereo audio playback (Webrtchacks 2020). For a successful munge we need to find the audio payload number which corresponds to the Opus codec from line seven, 111 in this case. Then we need to find the ‘a=fmtp’ line for the same payload number, this is line 23 in our SDP. Now we insert the ‘stereo’ and ‘sprop-stereo’ information and enable them both by setting them to 1 (Google Git 2017). Line 23 shows the result of our stereo SDP munge. Doing this munge enables stereo audio playback once we are connected.
SDP munging is an interesting standard, since it is widely used for different SDP configurations by WebRTC developers, but officially it is not advised. The goal of WebRTC is to have the API handle all the SDP generation and modifying for more reliable and safer applications, but as of right now, the only way to enable something like stereo audio is

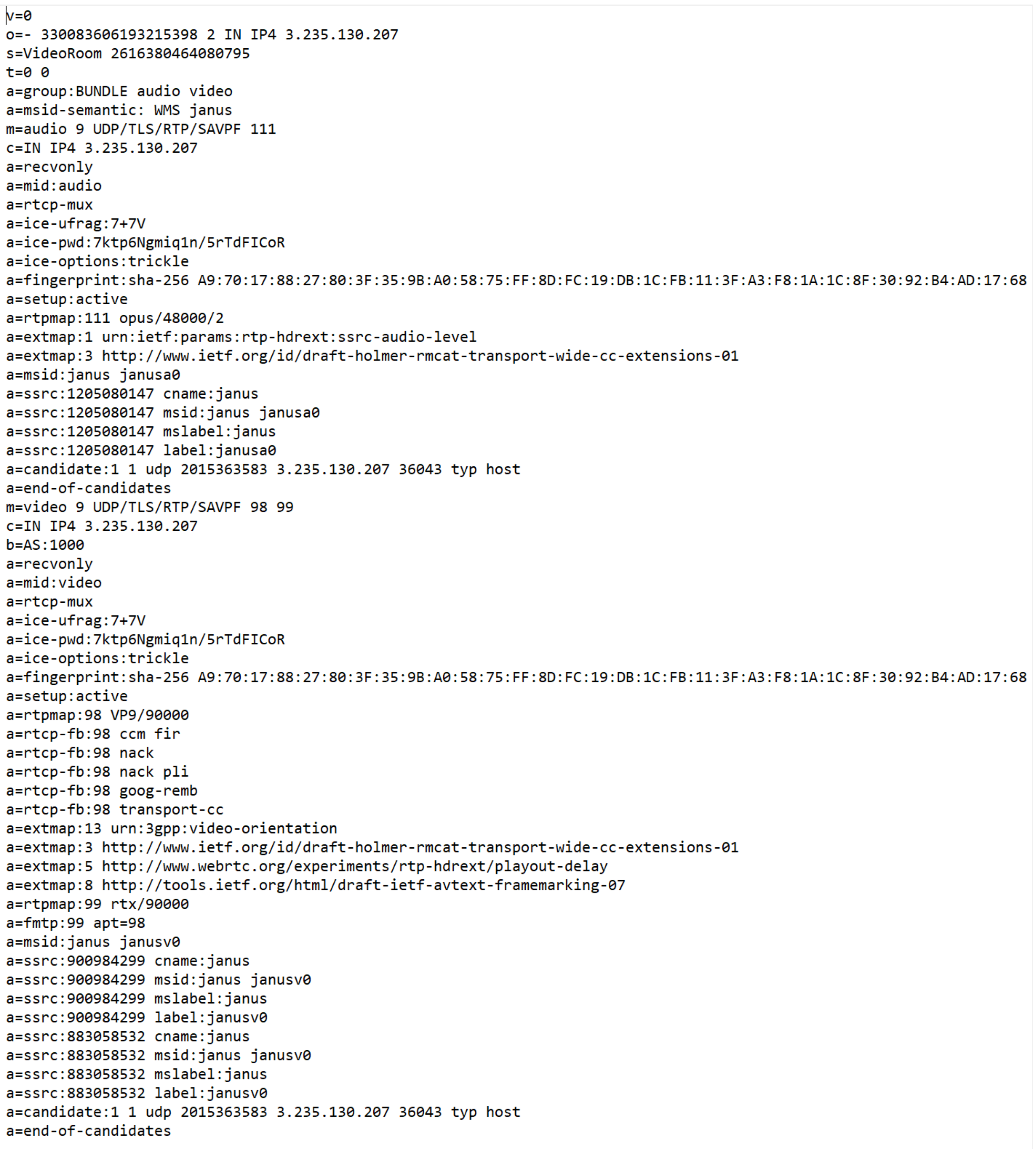
through munging (Webrtchacks 2020). This is a great example that shows how WebRTC is still in its early stages and some features might require deep digging to get working.

```
41. m=video 9 UDP/TLS/RTP/SAVPF 96 97 98 99 100 101 127 123 125 122 124
42. c=IN IP4 0.0.0.0
43. a=rtcp:9 IN IP4 0.0.0.0
44. a=ice-ufrag:50Ms
45. a=ice-pwd:bVbrZpel+n5pHdLcPHqRVy0n
46. a=ice-options:trickle renomination
47. a=fingerprint:sha-256 38:79:6B:49:2E:DB:C8:DF:CC:F5:49:4F:BA:07:71:D3:D9:78:06:3E:7C:90:98:CF:66:E1:5D:D6:38:68:B6:AB
48. a=setup:actpass
49. a=mid:video
50. a=extmap:14 urn:ietf:params:rtp-hdrext:toffset
51. a=extmap:2 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time
52. a=extmap:13 urn:3gpp:video-orientation
53. a=extmap:3 http://www.ietf.org/id/draft-holmer-rmcat-transport-wide-cc-extensions-01
54. a=extmap:5 http://www.webrtc.org/experiments/rtp-hdrext/playout-delay
55. a=extmap:6 http://www.webrtc.org/experiments/rtp-hdrext/video-content-type
56. a=extmap:7 http://www.webrtc.org/experiments/rtp-hdrext/video-timing
57. a=extmap:8 http://tools.ietf.org/html/draft-ietf-avtext-framemarking-07
58. a=extmap:9 http://www.webrtc.org/experiments/rtp-hdrext/color-space
59. a=sendrecv
60. a=rtcp-mux
61. a=rtcp-rsize
62. a=rtpmap:96 VP8/90000
63. a=rtcp-fb:96 goog-remb
64. a=rtcp-fb:96 transport-cc
65. a=rtcp-fb:96 ccm fir
66. a=rtcp-fb:96 nack
67. a=rtcp-fb:96 nack pli
68. a=rtpmap:97 rtx/90000
69. a=fmtp:97 apt=96
70. a=rtpmap:98 VP9/90000
71. a=rtcp-fb:98 goog-remb
72. a=rtcp-fb:98 transport-cc
73. a=rtcp-fb:98 ccm fir
74. a=rtcp-fb:98 nack
75. a=rtcp-fb:98 nack pli
76. a=rtpmap:99 rtx/90000
77. a=fmtp:99 apt=98
78. a=rtpmap:100 H264/90000
79. a=rtcp-fb:100 goog-remb
80. a=rtcp-fb:100 transport-cc
81. a=rtcp-fb:100 ccm fir
82. a=rtcp-fb:100 nack
83. a=rtcp-fb:100 nack pli
84. a=fmtp:100 level-asymmetry-allowed=1;packetization-mode=1;profile-level-id=640c1f
85. a=rtpmap:101 rtx/90000
86. a=fmtp:101 apt=100
87. a=rtpmap:127 H264/90000
88. a=rtcp-fb:127 goog-remb
89. a=rtcp-fb:127 transport-cc
90. a=rtcp-fb:127 ccm fir
91. a=rtcp-fb:127 nack
92. a=rtcp-fb:127 nack pli
93. a=fmtp:127 level-asymmetry-allowed=1;packetization-mode=1;profile-level-id=42e01f
94. a=rtpmap:123 rtx/90000
95. a=fmtp:123 apt=127
96. a=rtpmap:125 red/90000
97. a=rtpmap:122 rtx/90000
98. a=fmtp:122 apt=125
99. a=rtpmap:124 ulpfec/90000
100. a=ssrc-group:FID 4000235697 943454927
101. a=ssrc:4000235697 cname:NTKD2xzHFnXzshxP
102. a=ssrc:4000235697 msid:240829 fexRmD
103. a=ssrc:4000235697 mslabel:240829
104. a=ssrc:4000235697 label:fexRmD
105. a=ssrc:943454927 cname:NTKD2xzHFnXzshxP
106. a=ssrc:943454927 msid:240829 fexRmD
107. a=ssrc:943454927 mslabel:240829
108. a=ssrc:943454927 label:fexRmD
```

Figure 8. The video section of an SDP

Now looking at the video side of our SDP, we can already see a lot of similarities to the audio but describing video this time. Let us take a closer look at the video codecs at our disposal. In this device we have VP8, VP9 and H264 as our options. Which can be found in lines 62, 70, 78 and 87. Here we can see the similar ‘a=rtpmap’ structure followed by

the playload number and then the codec. The codec format is <name of the codec>/<clock rate>. VP8, VP standing for Video Processor, is a WebRTC standard video codec owned by Google. VP9 is an updated version of VP8. Both are open source and valid choices for video encoding, VP9 beating VP8 in performance and quality since it is newer. H264, is a codec specification by ITU, The International Telecommunication Union, it is also known as AVC (H.264), where AVC stands for an Advanced Video Coding (MDN Web Docs 2020). If you looked closely you might have noticed we have two H264 codecs available on our device, on lines 78 and 87. Notice the corresponding a=fmtp line and that at the end of it we have different profile-level-ids specified. This means we have two H264 profiles available on our device. Simply put, 42e01f is the main baseline of H264 and 640c1f is one of the higher quality profiles in H264 (Google Git 2017).
From these codecs we negotiate which we are going to use for our coming session. In this session we use Opus for audio and VP9 for video. The negotiated SDP removes all the other codecs and leaves the ones we use. The final SDP is shown in Figure 9.



```
v=0
o=- 330083606193215398 2 IN IP4 3.235.130.207
s=VideoRoom 2616380464080795
t=0 0
a=group:BUNDLE audio video
a=msid-semantic: WMS janus
m=audio 9 UDP/TLS/RTP/SAVPF 111
c=IN IP4 3.235.130.207
a=recvonly
a=mid:audio
a=rtcp-mux
a=ice-ufrag:7+7V
a=ice-pwd:7ktp6Ngmiq1n/5rTdFICoR
a=ice-options:trickle
a=fingerprint:sha-256 A9:70:17:88:27:80:3F:35:9B:A0:58:75:FF:8D:FC:19:DB:1C:FB:11:3F:A3:F8:1A:1C:8F:30:92:B4:AD:17:68
a=setup:active
a=rtpmap:111 opus/48000/2
a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
a=extmap:3 http://www.ietf.org/id/draft-holmer-rmcat-transport-wide-cc-extensions-01
a=msid:janus janusa0
a=ssrc:1205080147 cname:janus
a=ssrc:1205080147 msid:janus janusa0
a=ssrc:1205080147 mslabel:janus
a=ssrc:1205080147 label:janusa0
a=candidate:1 1 udp 2015363583 3.235.130.207 36043 typ host
a=end-of-candidates
m=video 9 UDP/TLS/RTP/SAVPF 98 99
c=IN IP4 3.235.130.207
b=AS:1000
a=recvonly
a=mid:video
a=rtcp-mux
a=ice-ufrag:7+7V
a=ice-pwd:7ktp6Ngmiq1n/5rTdFICoR
a=ice-options:trickle
a=fingerprint:sha-256 A9:70:17:88:27:80:3F:35:9B:A0:58:75:FF:8D:FC:19:DB:1C:FB:11:3F:A3:F8:1A:1C:8F:30:92:B4:AD:17:68
a=setup:active
a=rtpmap:98 VP9/90000
a=rtcp-fb:98 ccm fir
a=rtcp-fb:98 nack
a=rtcp-fb:98 nack pli
a=rtcp-fb:98 goog-remb
a=rtcp-fb:98 transport-cc
a=extmap:13 urn:3gpp:video-orientation
a=extmap:3 http://www.ietf.org/id/draft-holmer-rmcat-transport-wide-cc-extensions-01
a=extmap:5 http://www.webrtc.org/experiments/rtp-hdrext/playout-delay
a=extmap:8 http://tools.ietf.org/html/draft-ietf-avtext-framemarking-07
a=rtpmap:99 rtx/90000
a=fmtp:99 apt=98
a=msid:janus janusv0
a=ssrc:900984299 cname:janus
a=ssrc:900984299 msid:janus janusv0
a=ssrc:900984299 mslabel:janus
a=ssrc:900984299 label:janusv0
a=ssrc:883058532 cname:janus
a=ssrc:883058532 msid:janus janusv0
a=ssrc:883058532 mslabel:janus
a=ssrc:883058532 label:janusv0
a=candidate:1 1 udp 2015363583 3.235.130.207 36043 typ host
a=end-of-candidates
```

Figure 9. The final SDP after negotiation

Before we reach the final SDP, we still have couple of steps. The goal is to exchange offers and answers and reach a state where both peers have their own SDP as their local description and the peers SDP as their remote description. Reference the offer and answer exchange shown previously in Figure 6.

We created the offer to share media and generated the initial SDP, and now we need to set our SDP as our connection's local description. If we need to do munging on our SDP, it needs to be modified before setting it to our local description. After that we send the SDP to our peer through the server and wait for an answer. On the receiver's side once our offer reaches them, they set our SDP as their connection's remote description. Now they create an answer for our offer. Creating an answer also generates an SDP. They set their own generated SDP as their local description and send it to us through the server. We receive the answer and set their SDP as our remote description (WebRTC 2020).

This offer and answer model with setting local and remote descriptions is the core of creating a peer connection. The roles just change based on who is the sender and who the recipient. Now both peers have their own SDP as their local description and the peers SDP as the remote description.

3.2.2 ICE and NAT Traversal

Now with the connection's descriptions set, both peers know the necessary information about each other's end devices, but the connection is not established yet, and the media is not yet flowing. We still need to exchange ICE candidates. Reference the exchange of ICE candidates shown in Figure 6.
ICE stands for Interactive Connectivity Establishment, and the candidates hold information about how to reach us through our network. Because networks are complex, with firewalls, public and private IPs, and different access layers, we need to specify how the other peer can find us, so the data can reach us. Hence, we need to exchange ICE candidates. But we might not know the topology of our network, or maybe the network restrictions are too high for pure peer connections. Therefore, we need STUN and TURN servers for NAT traversal and help us tell our peers how to reach us.
A STUN server stands for Session Traversal Utilities for NAT, which stands for Network Address Translation. STUN helps us in situations where we do not know everything about our current network. We might know our own IP, but chances are we are behind a router and hopefully a firewall as well, so giving that information is not enough. STUN is designed to identify users public IPs and port numbers that NAT has assigned. We can get this information by just calling the server with a STUN protocol and it returns us what we need. Then we can send that information to our peer and the media can reach us. You can setup your own STUN server, but companies like Google offer widely established STUN servers for free use, so setting up your own might not be necessary (Ristic 2015, 40-41).
The situation is trickier when inside a restrictive network that forbids STUN server calls or peer connections. We can still establish connections with WebRTC from these networks, but to do this we need a TURN server. TURN stands for Traversal Using Relays around NAT, and as the name implies, we use the TURN server as a relay to connect to the other peer. The peer connection is established to the TURN server and the server relays all the

packets to the actual end device. TURN servers are typically used as a last resort because of their high costs. Fortunately, TURN servers are also rarely even needed and not having one will still allow most of the users to connect.
The process of getting ICE candidates from STUN and possible TURN servers can seem intimidating, but fortunately WebRTC and more specifically ICE handles most of this for us. We just need to indicate what servers we are using, and all the network querying is handled for us. ICE fetches possible addresses available from the STUN server, it sets up a TURN route as a backup, and returns the ICE candidate to us. We need to send it to the other peer for testing. This process happens multiple times until ICE knows enough about both networks and can find the best connection candidate and backup routes for both peers. Once that is ready, the ICE gathering state is complete, and we can establish a connection and the data can start flowing (Ristic 2015, 41-42).

3.3 Server

Thus far we have not been covering the server steps in signaling. A server is an important part of a WebRTC application but also the most flexible. Because there are multiple ways to implement a signaling server, and this also falls a bit out of the scope of this thesis, I will not go into specifics on how to build a server. Instead, in this section I will focus on how we can connect to a server from a client and what basic functionality your server should implement.

For a server to be able to send messages and events to the target users, it needs to have proper user identification. It is essential that the ids are unique, and that the server is aware which clients are currently connected and how to identify between them. A way to do this could be to require information about the user in a login message once they connect. The server can store the information in memory and recall it when needed. As with the login message we should also be able to differentiate between other message types. Types from the previous section, such as offer, answer and ICE candidate should be handled accordingly. These messages should also indicate the recipient if we are working with a session of more than two end users. To keep track on connected users, a logout message would be useful to keep the information up to date, or alternatively we could implement logic to detect disconnections (Ristic 2015, 59 -72).
To connect to a signaling server we require the WebSocket technology. A WebSocket is a connection between a server and a client that stays open until manually closed or if something goes wrong. Web sockets enable easy and fast messaging both ways over TCP. To

connect to a web socket, we need its URL with a ws or wss scheme. These are the insecure and secure web socket connection protocols equivalent to http and https. Preferring wss over ws is advised for its SSL certificate (MDN Web Docs 2020).

As I mentioned earlier, the look of your web socket implementation is very dependent on your server, but with these concepts you should have an idea what needs to be implemented. The same can be said about the WebRTC implementation, although that has more strict rules that need to be followed. In general, when developing with WebRTC you will not run out of things to research or try out.
The concepts explored in the previous sections are paramount when developing with WebRTC. In the next section we will take a closer look at how we use these concepts in the Kast Android application to help us achieve our goal of streaming audio and video over the internet through a peer connection.

4 Development

In this section I will go into detail about the Kast Android development. I will discuss the Kast Party Activity features and functions and later issues encountered in the project. In the Party Activity Section I will explain how the activity is structured and how various features are implemented in practice. These sections give insight on how Kast Android uses the previously mentioned AndroidX and WebRTC features and concepts with some corner use cases and advanced implementations. The Issues Section contains two examples from the project with the features and or fixes they eventually resulted in.

4.1 Party Activity

In this section we will take a closer look at the Kast Android party view. This is the core use case of the application and where all the WebRTC functionality takes place. We will look at the WebRTC implementations with some extra use cases, and the UI layouts and elements with both screen orientations.

First, we must navigate to the party in the application. Opening Kast Android, the user gets greeted with the login page, or most of the time we can redirect them straight to the discovery page if we they have logged in previously.

The discovery view is the apps main hub page. It is the lowest level of our activity back-stack, from where we redirect to all subsequent activities. In Figure 10 we can see a screenshot of the discovery activity.

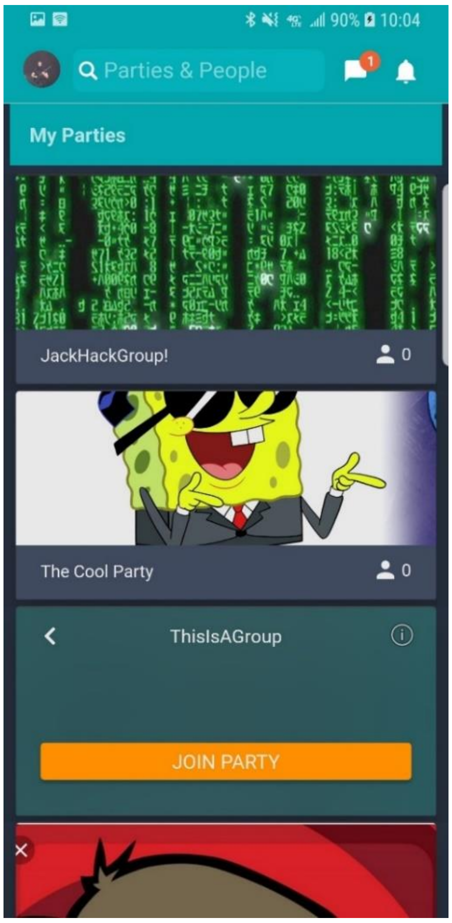


Figure 10. Kast Android discovery view

In the top of the image we can see a toolbar with routes to various activities and views such as account, search, social and notifications. The main content, which are the Parties, is a list of the user's own Parties and later other active public Parties. The list is implemented using a recycler view element with multiple layout view holders. Each Party is populated in a card which can be flipped on click to reveal the other side with various options. In the figure the third card shows the flipped layout. From these buttons the user can navigate either to party details from the info icon on the upper right hand corner of the card, from there the user can navigate to the Party, or the user can join the Party directly from the card by pressing the join party button.

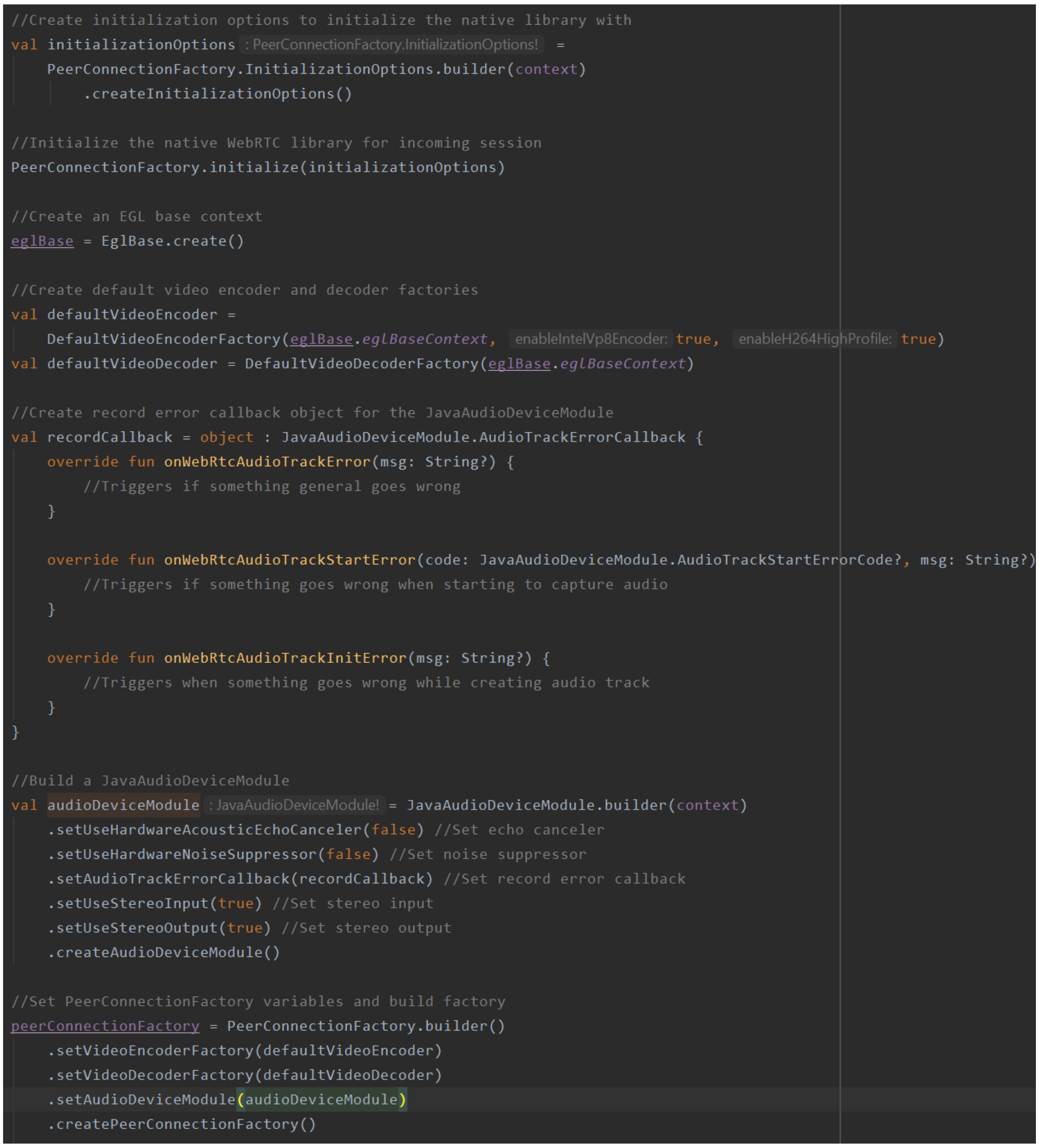
4.1.1 Party setup

When the Party Activity is created, we do some setup for the incoming session. We launch various coroutines to fetch necessary data for the Party in question. We do this without blocking the UI, instead we disable specific elements such as buttons, related to the functions we are fetching data for. Then once we have the information available, we re-enable the elements and actions related to them. For example, before we are connected to a signaling server buttons related to streaming are disabled, and before we have the party chat connected and displayed a user cannot attempt to send messages. Doing it this way we do not have to check if all necessary information is present before making the UI accessible, but rather each individual element is enabled once it can be used.

The Party startup also has WebRTC setup requirements. This includes initializing a PeerConnectionFactory and creating an EGL context. The PeerConnectionFactory is used to create all our peer connections with and EGL context is an interface from the WebRTC API library that converses with Androids OpenGL ES native API (Google Git 2015).
OpenGL ES is a powerful graphics drawing and rendering library, specified for embedded devices. OpenGL ES is widely used in applications which require heavy graphics rendering (Android Developers 2020). In Kast Android, we benefit from the high-performance rendering, since we can have multiple high-quality streams coming in simultaneously. Table 4 shows the WebRTC classes we utilize in the setup, and Figure 11 show our configuration for the PeerConnectionFactory.

Table 4. WebRTC Java API classes used in the Party setup

Class name	Intended use
PeerConnectionFactory	Initialize peer connections, media streams and tracks with
PeerConnectionFactoryBuilder	Build and configure the factory
JavaAudioDeviceModule	Configure audio attributes and our audio source
EglBase	An interface which converses with the Android native OpenGL ES rendering library



```
//Create initialization options to initialize the native library with
val initializationOptions : PeerConnectionFactory.InitializationOptions! =
    PeerConnectionFactory.InitializationOptions.builder(context)
        .createInitializationOptions()

//Initialize the native WebRTC library for incoming session
PeerConnectionFactory.initialize(initializationOptions)

//Create an EGL base context
eglBase = EglBase.create()

//Create default video encoder and decoder factories
val defaultVideoEncoder =
    DefaultVideoEncoderFactory(eglBase.eglBaseContext,  enableIntelVp8Encoder: true,  enableH264HighProfile: true)
val defaultVideoDecoder = DefaultVideoDecoderFactory(eglBase.eglBaseContext)

//Create record error callback object for the JavaAudioDeviceModule
val recordCallback = object : JavaAudioDeviceModule.AudioTrackErrorCallback {
    override fun onWebRtcAudioTrackError(msg: String?) {
        //Triggers if something general goes wrong
    }

    override fun onWebRtcAudioTrackStartError(code: JavaAudioDeviceModule.AudioTrackStartErrorCode?, msg: String?)
        //Triggers if something goes wrong when starting to capture audio
    }

    override fun onWebRtcAudioTrackInitError(msg: String?) {
        //Triggers when something goes wrong while creating audio track
    }
}

//Build a JavaAudioDeviceModule
val audioDeviceModule : JavaAudioDeviceModule! = JavaAudioDeviceModule.builder(context)
    .setUseHardwareAcousticEchoCanceler(false) //Set echo canceler
    .setUseHardwareNoiseSuppressor(false) //Set noise suppressor
    .setAudioTrackErrorCallback(recordCallback) //Set record error callback
    .setUseStereoInput(true) //Set stereo input
    .setUseStereoOutput(true) //Set stereo output
    .createAudioDeviceModule()

//Set PeerConnectionFactory variables and build factory
peerConnectionFactory = PeerConnectionFactory.builder()
    .setVideoEncoderFactory(defaultVideoEncoder)
    .setVideoDecoderFactory(defaultVideoDecoder)
    .setAudioDeviceModule(audioDeviceModule)
    .createPeerConnectionFactory()
```

Figure 11. Configuration of a PeerConnectionFactory and a JavaAudioDeviceModule

Starting from the top of Figure 11, to create a PeerConnectionFactory, which is responsible for creating all our peer connections, our own media stream and media tracks, there are couple of steps to take and options to consider. Initializing the factory is the very first thing we do since it loads the native WebRTC library and initializes its global variables. When initializing the factory, we also create the shared EGL base context for it. This is done by calling create on the EglBase interface and it will figure out which OpenGL version is available for that device and create a context for us.
There are couple of things we need to be aware of regarding the EGL context when managing our session. All our peer connections and surfaces, where we attach the videos to, need to have the same shared EGL context to work. Streams will not attach to surfaces if

the connection factory does not have the same shared context as the surface. The EGL context also need to be released manually from every surface and creating multiple ones can cause resource leaks due to surface release failures and ultimately cause native crashes (Google Git 2015). Therefore, we keep the same EGL context for the duration of the Party Activity and dispose of it only when the activity gets destroyed.

Now we can start setting options for the PeerConnectionFactory with its builder object. The essential configurations we need are the video encoding and decoding factories and the JavaAudioDeviceModule with its variables, shown in Figure 11. For the video encoding and decoding we use DefaultVideoEncoderFactory and DefaultVideoDecoderFactory. These classes handle the video encoding before sending and the decoding for incoming streams. In Kast we use VP9 as our video codec, which the default factories have at their disposal. We configure our audio source for our outgoing stream when creating the PeerConnectionFactory. This is done with the JavaAudioDeviceModule, which when created, we pass to the peer connection factory builder, so that all the audio tracks created follow the audio module configuration.
The JavaAudioDeviceModule acts as the system audio manager for a WebRTC session. To get started configuring we also create a builder object where we set our arguments. The default audio source is VOICE_COMMUNICATION from Androids audio source attributes, which is the devices microphone. This is what we want so we do not need to change the audio source in the module. We can also attach callbacks to the audio module. For example, in Figure 11 we attach an error callback which triggers if something goes wrong when starting the record or if the configuration is invalid.
The audio module controls stereo output and input. For Android we need to munge the SDP, as was covered in Section 3.2.1, and set the audio module stereo values to true for stereo audio to be played. Otherwise either the audio module overrides the SDP with its default mono audio, or if we do not munge the SDP the peer’s audio will not be routed to stereo.
We also specify some audio quality attributes like hardware echo cancellation and noise suppressor. We leave these as false, since the Parties can have continuous audio steaming, such as music, and these features would cancel out the outgoing audio when the incoming audio is played through the speakers.
After we are done with the audio module configurations, we call create on the builder and create our audio module. Now we can also take all our attributes for the PeerConnection-Factory and set them to its builder class and create the factory to start connecting.

4.1.2 Web Socket

After the initial setup is complete, we connect to our signaling server using the OkHttp library. OkHttp is an Android networking library which can be used for basic http requests, image loading and web socket connections. To open a web socket, we build a request from our endpoint with the request builder (Square 2019). Now we create a web socket with the newWebSocket function which needs a WebSocketListener that listens to events and the state of the connection. The listener has five functions shown in Table 5.

Table 5. WebSocketListener functions and uses

Function name	Intended use
onOpen	Triggers when the connection is successfully opened
onMessage	Triggers when a message is received
onFailure	Triggers when the connection fails unexpectedly
onClosing	Triggers when the connection begins to close
onClosed	Triggers when the connection has been closed

When connecting, onOpen is the first one to trigger if the connection is successfully established. Now we can start sending and receiving messages, the first thing we do is send a login message to inform who connected. To send messages we use the web sockets send function. For the payloads we use JSON objects which we can convert to a string and send to the server. OnMessage is the most used listener function, since it is where all the messages sent by the server arrive. We differentiate between the message type and handle them accordingly. For example, we have events for new incoming streams and responses for our sent messages.
OnClosing and onClosed functions from the web socket listener are self-explanatory, they trigger when the connection is manually closed. Compare it to onFailure which triggers if something goes wrong. OnFailure acts like a catch clause in a simple try-catch block, catching errors that happen in the socket connection and adding safety handling to abrupt disconnections (Square 2019). The function has the error cause as its parameter which we use to figure out what happened, by sending it to our error analytics.

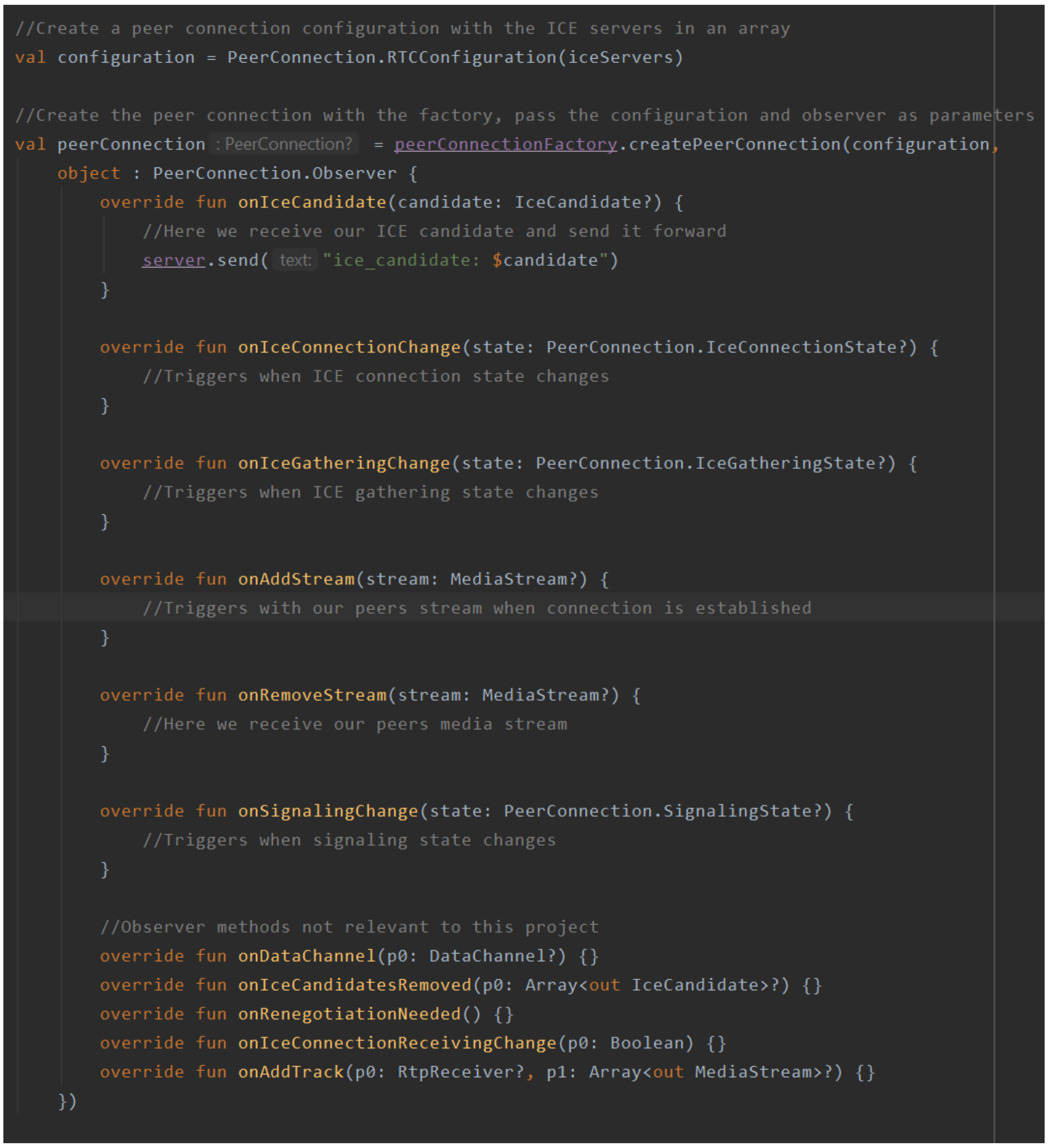
After we login successfully we also receive an event which contains information about the current state of the party. This means all the currently joined users with differentiation between who are spectating, and who streaming. We launch coroutines with the IO context to fetch necessary information for each user connected to the party. Later, when we establish peer connections with the other users, we attach those classes into a larger participant class with other user information. This helps us handle events and actions for each user without needing to iterate over various arrays of classes.

4.1.3 Peer connections

Once connected to the server we start listening to events. When we get an event that another user has started to publish a stream, we start the connection process. But before we send anything, we have a couple of extra checks. We need to check what medium the sender is broadcasting and how many video streams are we already connected to. We limit the video amount to four incoming and one outgoing stream therefore we need to check the current activity state. If the other user is attempting to share video and we have room for it, we instantiate a peer connection without anything specific. We do this by sending a request to connect to the server. Otherwise we would modify our answer to receive media based on the current state of our session. This means that if our maximum video amount is reached, we modify our media constraints to not receive video. Table 6 shows the classes used and Figure 12 and 13 show the logic for establishing a peer connection on Android.

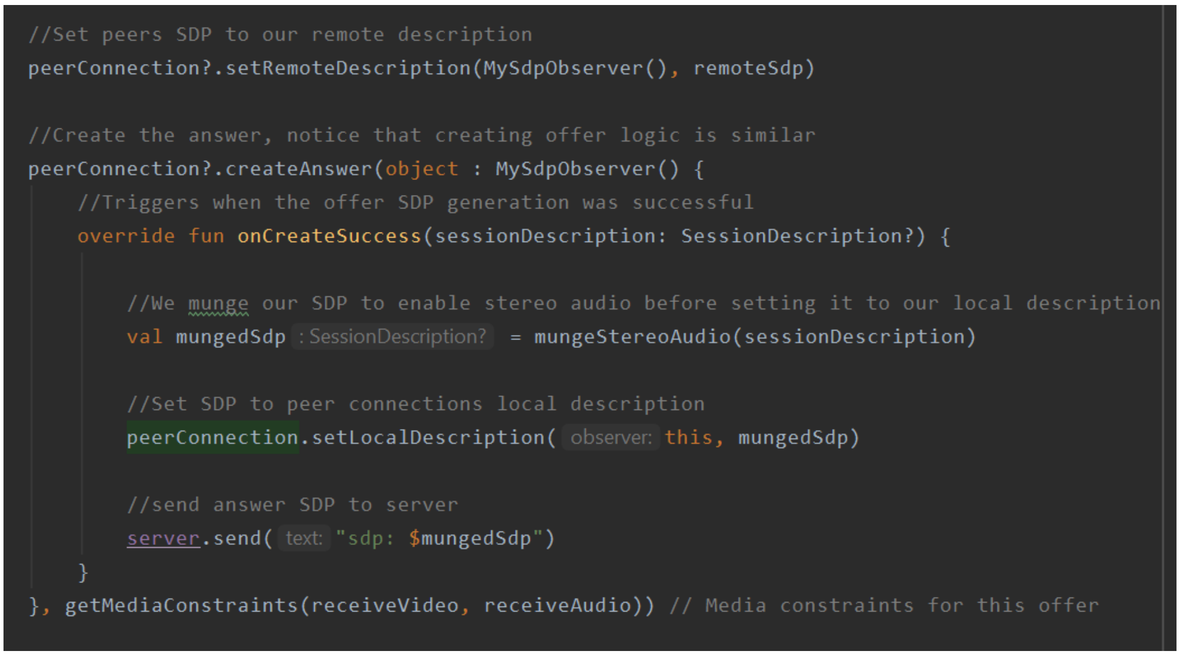
Table 6. WebRTC Java API classes used to establish a peer connection

Class name	Intended use
PeerConnection.RTCConfiguration	Used to configure specifics of the peer connections, such as TURN and STUN servers
PeerConnection	Class that represents the connection and handles events
PeerConnection.Observer	Observes the connection state. Has various listener functions shown in more detail in Table 7
IceCandidate	Represents an ICE candidate with information for the peer on how to reach us
SdpObserver	Monitors the SDP generation and setting of the descriptions



```kotlin
//Create a peer connection configuration with the ICE servers in an array
val configuration = PeerConnection.RTCConfiguration(iceServers)

//Create the peer connection with the factory, pass the configuration and observer as parameters
val peerConnection : PeerConnection?  = peerConnectionFactory.createPeerConnection(configuration,
    object : PeerConnection.Observer {
        override fun onIceCandidate(candidate: IceCandidate?) {
            //Here we receive our ICE candidate and send it forward
            server.send( text: "ice_candidate: $candidate")
        }

        override fun onIceConnectionChange(state: PeerConnection.IceConnectionState?) {
            //Triggers when ICE connection state changes
        }

        override fun onIceGatheringChange(state: PeerConnection.IceGatheringState?) {
            //Triggers when ICE gathering state changes
        }

        override fun onAddStream(stream: MediaStream?) {
            //Triggers with our peers stream when connection is established
        }

        override fun onRemoveStream(stream: MediaStream?) {
            //Here we receive our peers media stream
        }

        override fun onSignalingChange(state: PeerConnection.SignalingState?) {
            //Triggers when signaling state changes
        }

        //Observer methods not relevant to this project
        override fun onDataChannel(p0: DataChannel?) {}
        override fun onIceCandidatesRemoved(p0: Array<out IceCandidate>?) {}
        override fun onRenegotiationNeeded() {}
        override fun onIceConnectionReceivingChange(p0: Boolean) {}
        override fun onAddTrack(p0: RtpReceiver?, p1: Array<out MediaStream>?) {}
    })
```

Figure 12. Creating a peer connection with an observer



```
//Set peers SDP to our remote description
peerConnection?.setRemoteDescription(MySdpObserver(), remoteSdp)

//Create the answer, notice that creating offer logic is similar
peerConnection?.createAnswer(object : MySdpObserver() {
    //Triggers when the offer SDP generation was successful
    override fun onCreateSuccess(sessionDescription: SessionDescription?) {

        //We munge our SDP to enable stereo audio before setting it to our local description
        val mungedSdp :SessionDescription?  = mungeStereoAudio(sessionDescription)

        //Set SDP to peer connections local description
        peerConnection.setLocalDescription( observer: this, mungedSdp)

        //send answer SDP to server
        server.send( text: "sdp: $mungedSdp")
    }
}, getMediaConstraints(receiveVideo, receiveAudio)) // Media constraints for this offer
```

Figure 13. Setting our remote description and creating an answer

From the top of Figure 12, we create the PeerConnection object using the factory initialized when setting up the Party. For this we need a PeerConnection Configuration and an observer for the connection. The configuration needs ICE servers in its constructor. These are our STUN and TURN servers in an array.

The peer connection observer is paramount, since it has multiple functions which monitor the state of the connection during signaling and is also where we receive our peer’s media stream. From Figure 12 we can see all the functions the observer has, but all are not necessary for our goal of streaming video and audio media through the connection.

The observer’s functions which are relevant to this project are shown in Table 7.

Table 7. Peer connection observer functions used in this project

Function name	Intended use
onIceCandidate	Gives a new ICE candidate when it is found
onIceConnectionChange	Triggers when the ICE connection state changes during signaling
onIceGatheringChange	Triggers when changes occur in the ICE gathering state
onAddStream	Gives the peers stream when the connection is established
onRemoveStream	Triggers when the peer’s stream is removed
onSignalingChange	Triggers when a change occurs in the overall signaling

We get a response for our request to connect from the server with our peers SDP. Now we create the peer connection object and set their SDP as our remote description, these can be seen in Figure 12 and 13. We also create our answer by calling createAnswer on the peer connection object. This needs a SdpObserver, which is an interface for monitoring the SDP creation and description setting. We monitor the generation by overriding onCreateSuccess, which gives us the SDP, and then setting the SDP as our local description as seen in Figure 13. Before that we do our SDP munge, so we achieve stereo audio playback.
The other parameter is our media constraints. For these we use the MediaConstraints class which has two lists of key value pairs, mandatory and optional constraints (Google Git 2013). On Android various constraints are done by the PeerConnectionFactory but we do have to specify which media we want to receive from the peer. Here we set the video parameter to false if our incoming streams are already full. Otherwise we set receive audio and video to true.
Creating an answer triggers the onSignalingStateChange function, which we use to monitor the current state of our signaling. For example, the state now is HAVE_REMOTE_OFFER since we set the peers offer SDP as our remote description.
When both peers have set their local and remote descriptions, in other words when the offer and answer have been exchanged, the signaling state changes to STABLE and ICE gathering starts. This triggers onIceGatheringChange with the state GATHERING. Once ICE finds a candidate onIceCandidate triggers with the new candidate which we send to the peer through our signaling server, this can be seen in Figure 12. OnIceCandidate runs multiple times until enough information about the network is gathered. After we have enough information the ICE gathering, and connection states change to COMPLETE and CONNECTED.

Now we receive an event from our server which has the status of the stream. If the signaling was successful, the status will be “up” and the onAddStream function triggers with the peers MediaStream object. The media stream contains the peer’s media tracks, which contain the data. The audio is automatically routed to our device’s speakers and the video track we can attach to a surface to be viewed.

4.1.4 Displaying video tracks

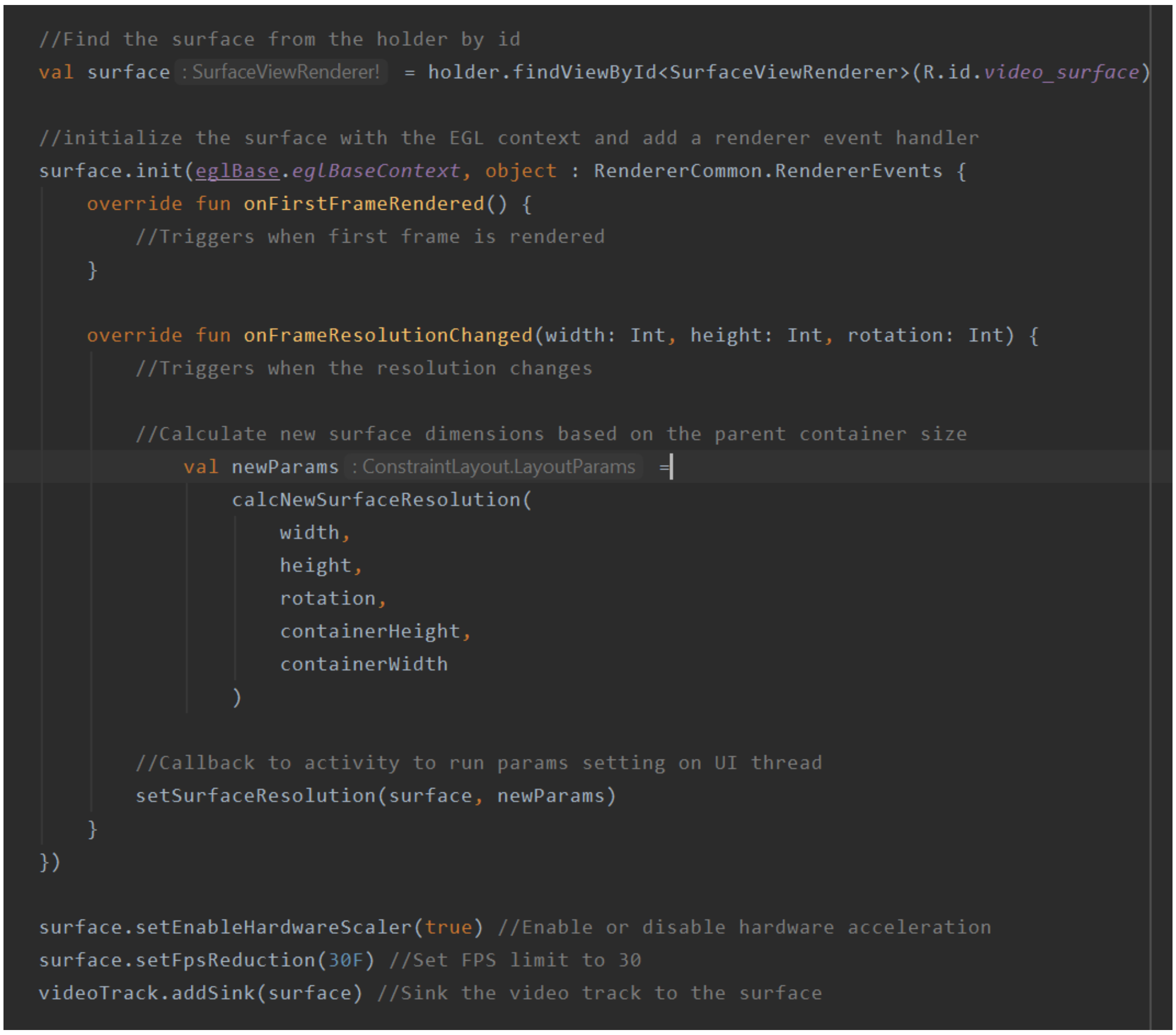
To attach a video track from a media stream to a surface we need a specific class and some configuration. Table 8 shows the WebRTC classes we use to display a video and handle its events.

Table 8. WebRTC Java API classes used to display a video

Class name	Intended use
SurfaceViewRenderer	The surface which displays the video track
RendererCommon.RendererEvents	A surface event listener which listens to changes in the video stream

With all our video tracks we use the SurfaceViewRenderer class as their displaying surface. The SurfaceViewRenderer is a Java WebRTC class which extends the generic SurfaceView class for drawing and rendering frames. An interesting thing to note on the SurfaceViewRenderer, is that its placement on the layout does not follow the typical way. A SurfaceView consists of a holder and the actual surface. The holder's placement determines where the surface is located on the UI, and the surface is inside the holder beneath all the other UI elements. Because of this non-conventional Z order, the holder punches a hole through the other UI elements so the surface can be viewed (Android Developers 2020). To understand more thoroughly why it works this way, we need to examine what happens when we create the EglBase context.
The logical way to think is that the surface for a video track is created just for the area determined by the holder. But this is not the case. EglBase creates an area for the surfaces to attach to that is the size of the whole window. Then with the holder we just determine the dimensions and the location of the video. This is why we pass the EGL shared context to all the SurfaceViewRenderers, so that they will get attached and displayed on the surface created by EglBase behind the UI elements.

In Kast Android, on portrait orientation our incoming video streams are displayed in a ViewPager and a RecyclerView on landscape. These both use the same adapter to populate the items, therefore the attachment logic is the same in onBindViewHolder. Since the ViewPager and RecyclerView use the same adapter, in this section I reference them as one and the same recycler. Figure 14 shows the attachment logic.



```
//Find the surface from the holder by id
val surface : SurfaceViewRenderer!  = holder.findViewById<SurfaceViewRenderer>(R.id.video_surface)

//initialize the surface with the EGL context and add a renderer event handler
surface.init(eglBase.eglBaseContext, object : RendererCommon.RendererEvents {
    override fun onFirstFrameRendered() {
        //Triggers when first frame is rendered
    }

    override fun onFrameResolutionChanged(width: Int, height: Int, rotation: Int) {
        //Triggers when the resolution changes

        //Calculate new surface dimensions based on the parent container size
            val newParams : ConstraintLayout.LayoutParams  =
                calcNewSurfaceResolution(
                    width,
                    height,
                    rotation,
                    containerHeight,
                    containerWidth
                )

        //Callback to activity to run params setting on UI thread
        setSurfaceResolution(surface, newParams)
    }
})

surface.setEnableHardwareScaler(true) //Enable or disable hardware acceleration
surface.setFpsReduction(30F) //Set FPS limit to 30
videoTrack.addSink(surface) //Sink the video track to the surface
```

Figure 14. Attaching a video track to a surface

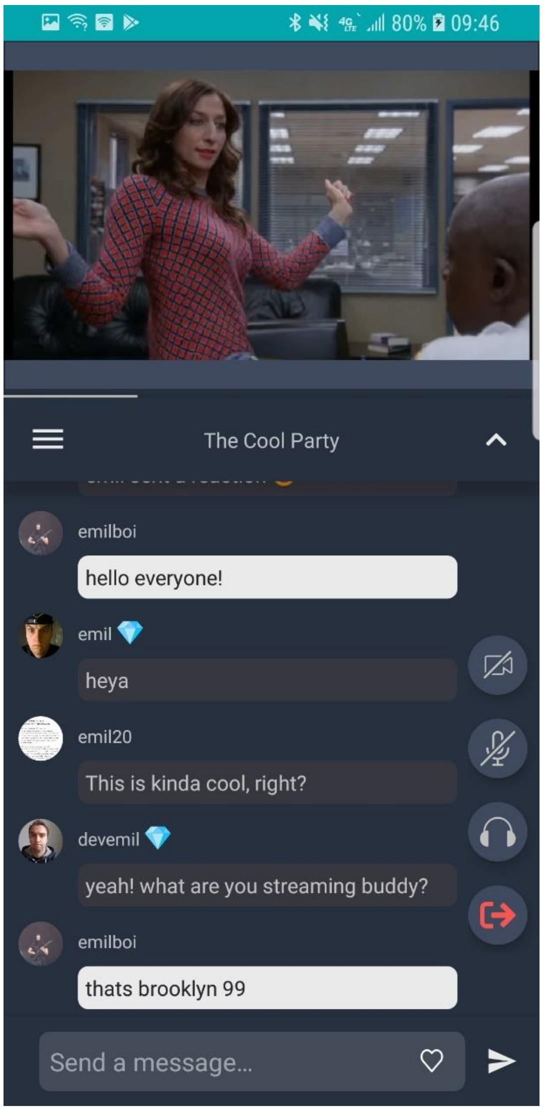
We create the view holders for the recycler in XML which the adapter inflates. Starting from the top of Figure 14, we find the surface from the holder by id and initialize it. This takes the EGL shared context and a renderer event handler as parameters. The need for the EGL context has been covered in previous chapters, but the renderer event handler is a new object. The event handler is a WebRTC interface called RendererEvents from the RendererCommon class. As seen in Figure 14, the interface has two listener methods, onFirstFrameRendered and onFrameResolutionChanged. We ignore the first frame rendered but with the resolution change we receive the dimensions and rotation of the incoming stream. We use these with the parent containers dimensions to calculate the new aspect ratio of the video and update the surfaces size based on the result. In practice we set the new dimension with a callback to the activity. We need to do it this way since UI updates have to run on the UI thread which we have access to in the activity. This helps us assure that we always display the whole incoming stream and react to dimension changes in streams. For example, in situations such as when a streamer turns their device, or an application window is resized.

Once the surface is initialized, we have couple of configuration options available, for example, we enable hardware acceleration. This allocates some heavy UI tasks to the devices GPU to handle. This will generally result in better rendering performance (Android Developers 2020). We also set the fps reduction, or more specifically, cap to a maximum frame render rate. This is set to 30 frames per second so frames over that will be ignored. These configurations are one of the results of an overheating issue, which I cover in Section 4.2.2.
Now with the surface configured, we call addSink on the VideoTrack and pass the surface as the parameter. This adds the surface as a rendering location for the video track and we can see the stream on our screen in the recycler.

When a user stops streaming, we get an event from the server. The peer connection closing runs automatically, and the closing process might be initiated by either user. We still need to do manual handling when a connection is closed. Firstly, we clear the surface resources. For this we call removeSink and release on the surface. Remove sink does the opposite of add sink, and release removes the EGL context from the surface. Release is important since otherwise the surfaces listeners will keep running and waste resources and eventually cause a native crash (Google Git 2015). Now we also notify the recycler adapter on which stream was removed. This is done by finding the streams index in the item list in the adapter and calling notifyItemRemoved with that index. The adapter handles the UI updating by removing the item in the index specified and moving the remaining items to fill that position.

4.1.5 Party UI

We have two different XML layouts for both screen orientations. These are both ConstraintLayouts with different constraint sets which allows us to update the UI with a Transitionmanager and giving a better feel to the activity. Figures 15 and 16 show the UI with a simulated party of five people with four incoming streams. Three are coming from different Kast clients and one is from another Android client.


80% 09:46

The Cool Party

emilboi

hello everyone!

emil

heya

emil20

This is kinda cool, right?

devemil

yeah! what are you streaming buddy?

emilboi

thats brooklyn 99

Send a message...

Figure 15. Party view portrait orientation

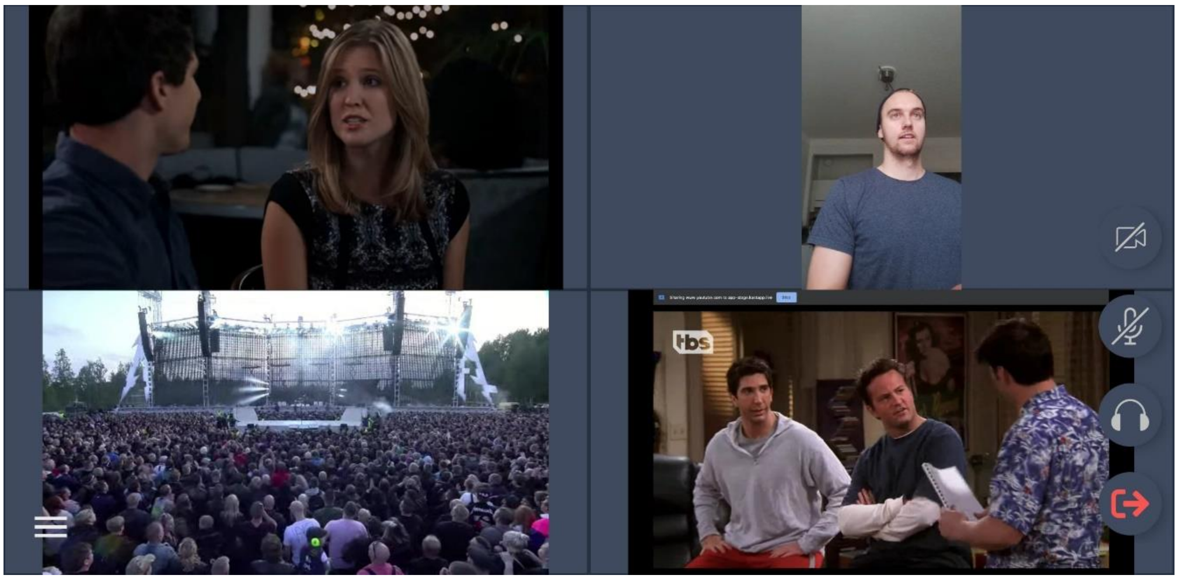


Figure 16. Party view landscape orientation

We update the layouts based on the benefits from the screen orientation. In portrait we have a ViewPager where we populate the streams and a party chat running below that. On landscape orientation we focus on the streams by hiding the chat element and changing the ViewPager to a RecyclerView with a grid layout as its layout manager. In Figure 16, notice the different video stream dimensions. These show the result of the resolution change functions from the previous chapter in practice.

On the video grid, seen in Figure 16, we can expand videos to full screen. We pass a click listener from the Party Activity to our recycler adapter and attach it to the surfaces if we are on landscape orientation. When our click listener triggers, we remove sink on all our video tracks and attach the stream that was clicked to a separate surface which is on top of the video grid and the size of the whole screen. When the full screen surface is clicked, we clear it and re attach all the video tracks to their surfaces. This way we do not waste any resources by rendering streams that are not visible, and the users full screen gesture would also indicate that they are more interested in the clicked video.

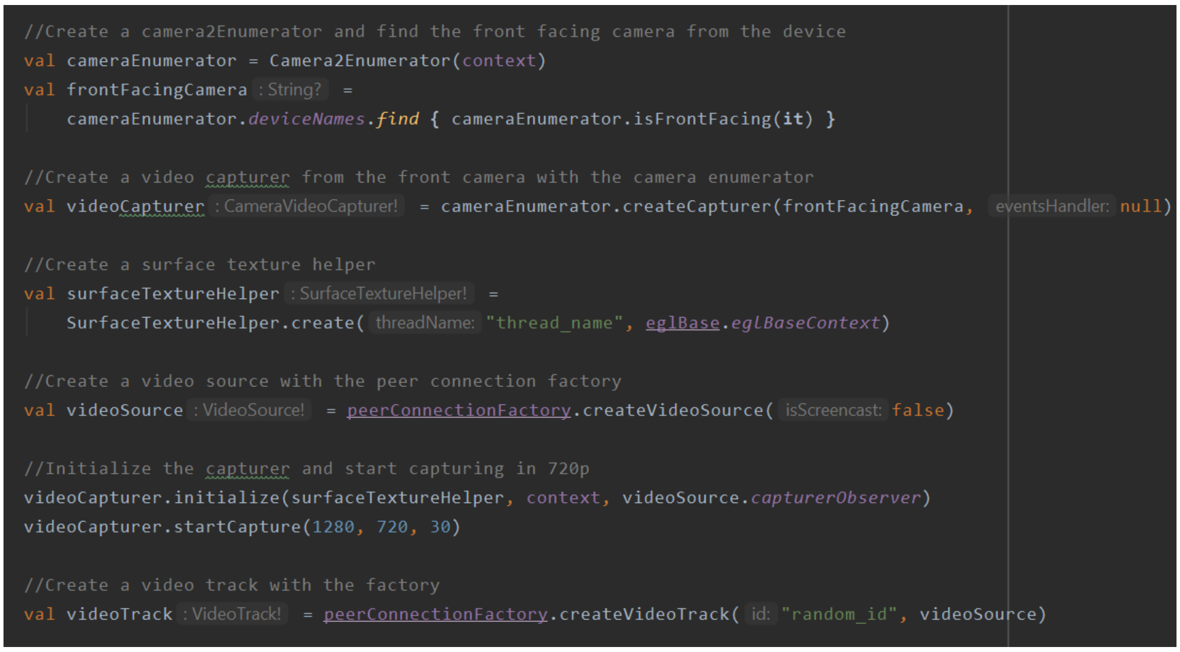
Both layouts have the party broadcast buttons on the right side of the screen. With these the user can control their video and audio tracks, toggle the overall party audio, or leave the Party Activity.

4.1.6 Getting media streams

When the user starts streaming by clicking either the video or audio icon, we check the necessary permissions to access the devices camera and or microphone and then start getting the users media streams. On Android we use the Camera2 API for video, and the microphone for audio. We configured our audio when we created our JavaAudioDeviceModule, in the Party setup, so creating our audio track will follow that configuration. Therefore, let us focus on the video configuration first.
In Table 9 we can see WebRTC classes we use to get the video track from our camera and Figure 17 shows the flow of creating a video track in Kotlin.

Table 9. WebRTC Java API classes used to get the video track and display it

Class name	Intended use
CameraVideoCapturer	Handles the capturing from Camera2
Camera2Enumerator	Enumerate between all the cameras in a device
VideoSource	The video source from where we can create our VideoTrack object
SurfaceTextureHelper	Helper class that resolves each frame to a state a surface can understand and display

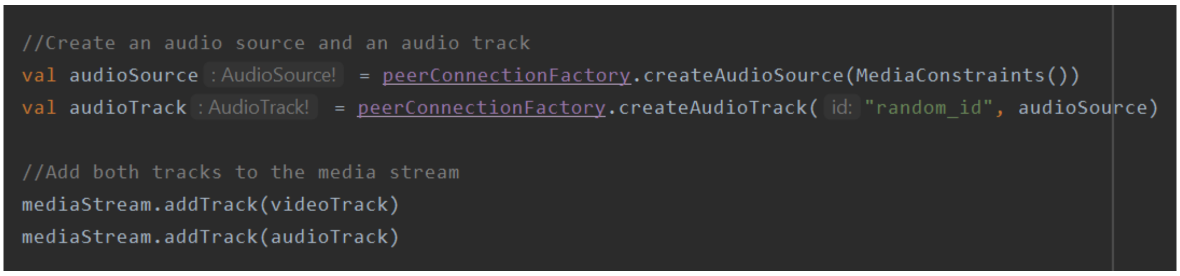


```
//Create a camera2Enumerator and find the front facing camera from the device
val cameraEnumerator = Camera2Enumerator(context)
val frontFacingCamera : String?  =
    cameraEnumerator.deviceNames.find { cameraEnumerator.isFrontFacing(it) }

//Create a video capturer from the front camera with the camera enumerator
val videoCapturer : CameraVideoCapturer!  = cameraEnumerator.createCapturer(frontFacingCamera,  eventsHandler: null)

//Create a surface texture helper
val surfaceTextureHelper : SurfaceTextureHelper!  =
    SurfaceTextureHelper.create( threadName: "thread_name", eglBase.eglBaseContext)

//Create a video source with the peer connection factory
val videoSource : VideoSource!  = peerConnectionFactory.createVideoSource( isScreencast: false)

//Initialize the capturer and start capturing in 720p
videoCapturer.initialize(surfaceTextureHelper, context, videoSource.capturerObserver)
videoCapturer.startCapture(1280, 720, 30)

//Create a video track with the factory
val videoTrack : VideoTrack!  = peerConnectionFactory.createVideoTrack( id: "random_id", videoSource)
```

Figure 17. Create a video track from a front facing camera

In Figure 17 we can see how a video track object is created. Since most devices have multiple cameras in them, we need a Camera2Enumerator with which we can get the available cameras and identify the most suitable. We can check the supported resolutions for each camera device and if it is a back or front facing camera. In Figure 17 we find the front facing camera on the device. Even though we are using Camera2, we do not access it directly, but through the WebRTC API. We use the CameraVideoCapturer class for the actual capturing. This is an interface extending the VideoCapturer class with event handling and helper functions. For example, it handles camera source changes which lets us switch between cameras without needing to configure the video source again and interrupting the data flow in the process. The VideoCapturer class, is a general capture observer which purpose is to communicate with the WebRTC C layer (Google Git 2016).

Once we have the camera, we use the enumerator to create the CameraVideoCapturer object, but so far, no capturing is taking place yet. We still need a SurfaceTextureHelper and a VideoSource to initialize the capturer. A SurfaceTextureHelper acts as a middleman between the capturer and the surface where the stream gets displayed to by resolving every frame to a state the surface understands.
We create the VideoSource with the PeerConnectionFactory with the isScreencast parameter as false since we get the video from the camera. The video sources capture observer and the video capturer need to be attached to each other, which we do when we initialize the video capturer. We call initialize on the capturer and pass the SurfaceTextureHelper, application context and the video source observer as parameters to it. Now the video capturer is ready to capture, so we call startCapture with the capture resolution and the desired frame rate. In Figure 17, we capture in 720p, but these values could be anything supported by the device. Now we have our capture running and the frames are being passed to the video source. From the source we create our video track, again using the peer connection factory.
Getting audio is a bit simpler. Figure 18 shows the creation of an audio track and adding tracks to the media stream.



```
//Create an audio source and an audio track
val audioSource : AudioSource!  = peerConnectionFactory.createAudioSource(MediaConstraints())
val audioTrack : AudioTrack!  = peerConnectionFactory.createAudioTrack( id: "random_id", audioSource)

//Add both tracks to the media stream
mediaStream.addTrack(videoTrack)
mediaStream.addTrack(audioTrack)
```

Figure 18. Creating an audio track and adding both tracks to the media stream

As with the video source, we use the PeerConnectionFactory to create an audio source and from that source we create the audio track. The source takes media constraints, but we leave these empty. Configuring the audio source was done when we created the JavaAudioDeviceModule and passed it to our PeerConnectionFactory, therefore this process is straightforward. Once we have our audio track, we add both our media tracks to our media stream object.
Now we are ready to publish our stream. We do this by creating a peer connection object, as we did when receiving media, and add our media stream object to it. After this we create an offer which we send to the server to be sent to the other party participants. From there the signaling process continues as standard.

We also show a preview of the capturing video to the user. The stream attachment logic is the same for our preview as it was for all incoming streams. A thing to note here is that sinking a video track to a surface does not consume its resources, it just specifies a location where to send the bytes. Then the surface converts the data and renders it on the surface (Google Git 2013). This way we can be assured that even if we show a preview the outgoing stream will be unaffected.

Although, because of how EGL creates the window which it attaches the surfaces to there is a caveat we need to remember. If we attach two streams that overlap each other they start to flicker in between since they are on the same Z layer. Therefore, we need to set setZOrderMediaOverlay to true for our preview surface. This determines that the surface is to be displayed on top of all other surfaces and therefore, removes flickering of any overlapping streams. The flickering would be an issue on landscape orientation with four incoming streams since the preview and an incoming stream would be in the same location.

We also attach a gesture listener to the preview surface, which listens to single and double clicks. A single click reveals the camera switch button and a double click switches straight between the front and back camera. The switching is done by calling switchCamera on the videoCapturer, which happens without interrupting the outgoing stream.

The goal is to give the user a lot of control over what specific media they want to publish; this means they can either share only audio, video, or both. In practice this happens by checking what the user is already streaming and updating the media stream object by adding or removing specific media tracks and then republishing again. Figure 19 shows the layout when a user is publishing both media with the camera preview visible.

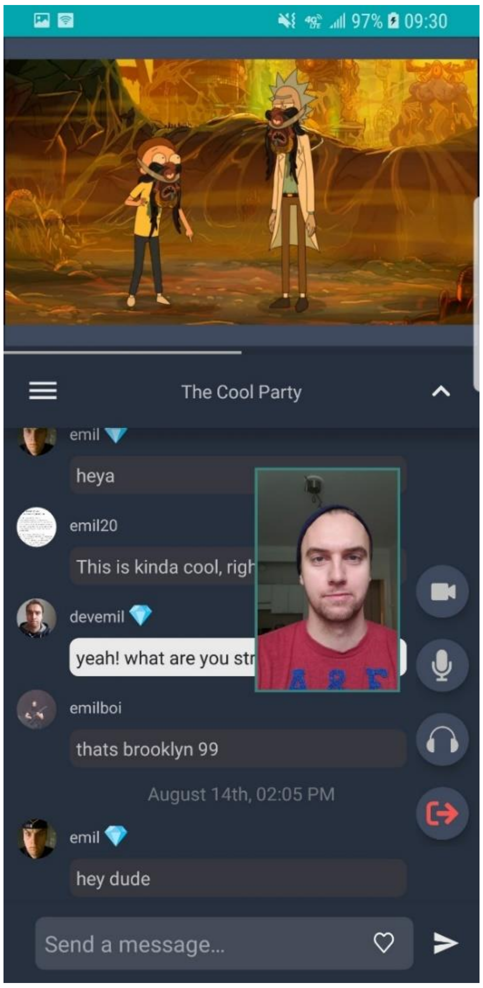


Figure 19. Party Activity UI with the camera preview visible

4.1.7 Toggling incoming streams

On the left side of the screen, on both orientations, we can find a hamburger icon. Clicking this opens the participants drawer flyout where we can see all the users currently in the party. This can be seen in Figure 20.

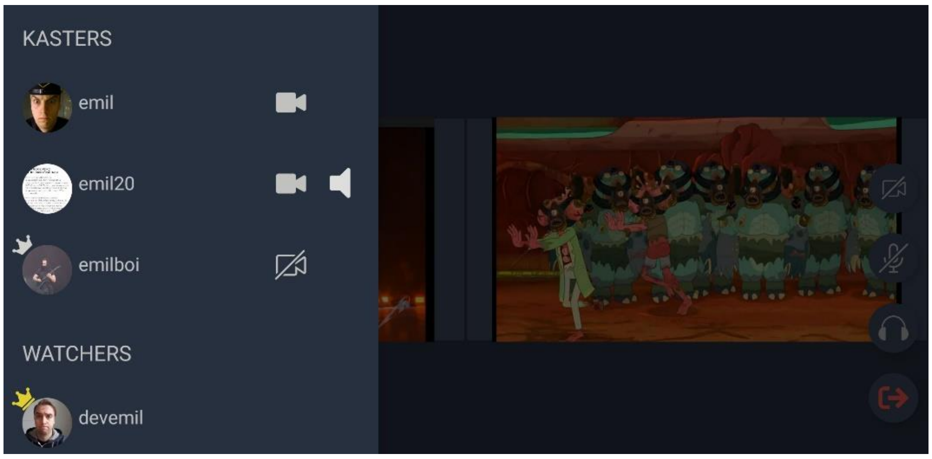


Figure 20. The participants drawer flyout open with one video stream turned off

The participants are added into a simple linear recycler and are divided into ‘Kasters’ and ‘Watchers’. Regarding the ‘Kasters’ we also see icons indicating what media they are currently sharing. We attach click listeners to these icons and by clicking, we can disable or enable specific media tracks on the stream. With these the user can customize their own view to their needs. Toggling individual tracks in a stream is a corner use case and not provided by WebRTC out-of-the-box.

We handle incoming audio and video toggling a bit differently from each other, because of the amount of bandwidth each require. When a Kasters audio icon is clicked we can call setEnabled with true or false on the audio track object. A disabled track means it is not in use so the audio will not be played, but the track is still connected and enabling it again does not require any extra signaling.

For video, we handle toggling more thoroughly. To toggle the video off on a stream, we first must close the actual peer connection and then re-open it by modifying the media constraints to receive only audio. Doing it this way saves a lot of bandwidth and unnecessary network usage. The video track object has the same setEnable function as the audio track, and calling it would have the same effect, but this would not stop the initial sending and the users receiving of the packets. Therefore, the user would still be receiving all the frames and only skipping the rendering part. With a scenario of multiple streams turned off this would result in a lot of wasted resources.

4.1.8 Leaving Party

The bottom most button from the broadcast controls, seen in the figures in the previous sections, starts the party leaving process. We must do various clean up before we can successfully finish the activity and return the user to the discovery view, otherwise the native libraries and tools will remain initialized and cause crashing or hinder the subsequent party setup.

We start by disconnecting all our peer connections and clearing their surface if they are streaming video. We clear the surface with removeSink and release the EGL context from it, as was discussed in previous chapters. On top of this we also dispose the peer’s media stream object to release its resources. To close the connection, we call close on the peer connection object and send a close request message to the server. This is done systematically to all incoming streams and afterwards we notify the recycler adapter of the removed streams to update the UI.

With the incoming streams gone, we also need to stop our own outgoing stream. We stop capture on our video capture object, clear the preview surface and dispose of the media tracks and the media stream.

Now no media is being exchanged anymore, so we can disconnect from the server. This is done by sending a logout message to notify the server on who is disconnecting and calling disconnect on the web socket. We send the logout message that the server will stay on top of who are currently connected to the party session.
When that is successfully done, we dispose of our peer connection factory and our EGLBase context. With these released we are in the same situation when we joined the party, therefore we can finish our activity and return to the discovery view.

4.2 Issues

When working with Kast Android I have experienced various errors and bugs. These have included native crashing and other issues from the application code. But there can also be other types of issues than crashing in applications, for example resource management and user experience issues. Let us take a closer look into two situations from the project and the features and or fixes that they resulted in.

4.2.1 Party audio manager

The first one is our audio manager in the Party Activity. The party audio manager is not a bug fix but rather a feature overriding a WebRTC default. I mentioned the WebRTC JavaAudioDeviceModule class, it redirects all incoming audio to the stream voice call mode, which means the audio is handled as a phone call and controlled with the ‘in-call’ slider. This is not what we wanted, and it also raised questions in the Kast community since we do not portray ourselves as a call over internet application. The goal was to have the media slider in the device to control all the audio in the Party.
To achieve the desired behavior, we needed to override the existing audio manager by implementing our own. The actual overriding is simple, we just need the audio service from the system. To do this we can call getSystemService on the activity context and pass the required service as the parameter, audio service in this case (Android Developers 2020). Now the default audio manager is overridden, but by doing so we lose some basic functionality. For example, simple things such as audio redirection to headphones instead of the speaker or volume adjustment with the volume keys no longer work. These all need to be implemented manually.
Firstly, we need to set the audio mode to normal, which specifies that we want to use all the speaker resources available on the device. Then we also need to register a device callback, which triggers every time a change occurs on the audio devices. This means attaching or removing headphones, Bluetooth devices, among others. The callback gives all the connected devices in an array where we determine if a device is attached from which the user would expect to hear audio from. If so, setting the isSpeakerphoneOn to false

gives us the desired playback by routing the audio to the call speaker, which in turn routes the audio to headphones. In turn when we do not detect any headphones or similar devices, we set the speaker value to true, which then plays the audio from the speakers. Now the audio is being rerouted and the attaching and removing devices is being handled, but we also need to handle volume changes. This we can achieve by overriding onKeyDown in the activity and setting an audio settings observer. OnKeyDown triggers every time a key is pressed on the device while the activity is running. The function gives us the keycode of the pressed key, which we use to determine if it is a volume key, and the direction adjusted in. The audio observer detects if the slider is manually dragged in the audio mixer, then it returns with the new value which we use to adjust the volume. With these methods we can adjust the STREAM_MUSIC audio mode on the audio manager, which is the media slider in the mixer. We also adjust the individual volume of each WebRTC audio track we have based on the current volume in our media slider. These combined enable the overall audio to be controlled with the media slider and therefore completing the feature.

4.2.2 Device overheating

The other issue was general device overheating. This was the biggest issue faced during the project. The usual scenario was this; we have multiple streams coming in and the user is also sharing their own camera. The amount of resources used starts draining the battery and generate heat. With the battery draining the reaction is usually to plug it in to a charger, which of course stops the battery draining but also generates more heat. The device keeps heating and eventually the system starts killing operations. First it stops the charging and ultimately the app itself as well.
The causes are difficult to pinpoint since the behavior is very device dependent. Some devices overheat while others face no issues whatsoever. The device size is a huge factor simply because there is more area for the heat to spread out to. For example, tablets never overheat, nor do bigger Android phones such as Galaxy Notes. The age of the device also seems to affect this. Newer devices coming into the market do not seem to generate nearly the same amount of heat than older ones. I suspect it has something to do with the combination of better CPUs and optimized resource management. So technically this problem is getting better by the new device release. But that is obviously not good enough.
I have spent a lot of time evaluating this issue and concluded that there are two major aspects that require the most computing and therefore generate the most heat. This is the camera capturing and all the rendering done for multiple incoming streams. When both are working hard simultaneously, they generate a lot of heat, which some devices cannot

handle. However, both running on their own tend to work fine. With the camera there is not much we can or want to do. We could lower the capturing resolution, but that is not something we want from a user experience standpoint. On the surfaces we can leverage couple of attributes I mentioned in Section 4.1.4. We can enable hardware acceleration in our surfaces and push rendering to the GPU. We can also limit the rendering fps. For the fps we do not want to limit the incoming streams, because that would again affect the UX and some devices do not even require the limiting. But what we can limit is the user own preview. Even if we limit the preview rendering to 20 fps it does not affect the quality we capture or send. And hopefully the point for the user is not to watch their own face all the time, so we can get away with a little less rendering and save resources.
With these we have managed to lower the amount of heat generated and therefore combat the overheating in some devices. But unfortunately, we are still talking about a very resource intense application- So if you are running Kast Android with a small old phone in a full party, you might still experience overheating.

5 Conclusion

Kast Android has been the most technical and challenging project I have done. The goal was to develop a modern Android application, with WebRTC functionalities for the Kast community to enjoy.
We needed to identify tools and libraries to achieve a great UI which is efficient and responsive. With AndroidX we can use the newest UI elements and leverage their effectiveness in the application. This combined with Kotlin coroutines helps us keep the UI responsive while simultaneously handling tasks in the background. With Kotlin in general we can build various functionalities which are easier to maintain due to the minimal nature of Kotlin code. In the long run this also minimizes the time a developer needs to spend figuring out old code when, for example, refactoring.
With WebRTC, the initial goal was to get connected to a peer and receive their stream. This was because WebRTC can be quite challenging to get started with and Kast Parties have various simultaneously moving parts and corner use cases. After the ability to stream and connect the focus moved to creating the best experience with optimizing and fixing issues, such as the use cases mentioned in the previous section. This process will continue with new WebRTC releases, which hopefully will give the developers more possibilities for their applications.

Overall, I would say that the project was a success with the objectives met. On top of this, the project has also been an extensive learning process. My knowledge about the Android framework has increased significantly during the development. Kotlin has largely become my main go to language, which is great not just for Android development but also due to its proximity to Java, which is one of the most used programming languages in the world. Good skills in either language makes it easy and comfortable to work with the other one.
My skills with WebRTC have also deepened vastly. I had worked with WebRTC on our web client previously, but this project made me really dive deep into its inner workings since I had to build everything from scratch. On top of this I had no experience in working with WebRTC in a native Android environment. The WebRTC Java API was new to me so to achieve our goal I had to really inspect its classes and functionalities to use it to its full potential. Going forward I would not feel anxious about working with WebRTC on any platform.
From a general software development standpoint, I also have learned the importance of writing code that gives the best user experience. User experience is a combination of multiple things, but with code a major aspect that can be affected is the users wait time when information is being fetched or processed. Previously, I might have redeemed a function

ready if it worked properly and overlooked any possibilities in optimizing. After this project with more experience in asynchronous programming, I have learned that keeping the application responsive with task prioritization and the use of background execution are a very high priority when designing and implementing features. In practice on Android the proper use of multithreading and Kotlin coroutines are the key to this kind of programming.

The community has adopted the Android client with over a 100 000 thousand downloads, as can be seen in the additional information section in the Google Play Store. What can also be seen is that the user reviews and responses have been quite polarized with users either completely loving the application or absolutely hating it. I think there are multiple reasons for this. The app of course still has bugs and issues remaining to be squashed, which can hinder the overall user experience. But I think a major reason for this can be observed from the user behavior.
Generally, the user activity follows the same behavior we have experienced with our other clients. The traffic tends to pick up closer to the weekend and usually peaks on Friday or Saturday evenings US time. A standard Kast user uses our applications in a weekly basis, which also applies to Kast Android. The use cases tend to vary, which I believe is because we do not try to guide the users to a certain behavior in any way. That said, there is a specific use case which is on top of the others. This is the act of participating from the Android client in a larger series or movie watch party. The content is shared from our desktop or web client and Android users join the party to watch and communicate by sharing their microphone or camera. This would indicate that Kast Android requires someone running the party from a web or desktop client to be able to fulfill the largest use case.
With this in mind Kast Android seems to be a secondary application in the Kast ecosystem, which is expected due to its young age compared to the desktop or web client. The application still has various parity features missing such as the ability to host a party to the same extent web and desktop can. This can also be directly observed in some of the review feature requests in the Google Play Store. For the development going forward, this is where I would continue.
An important feature for Kast Android would be the screensharing capabilities. With the current state of WebRTC this can be partially done. We can capture the screen of a device through the WebRTC API directly, but not the device audio. To capture device audio, we can use Androids media projection API, but to attach that to a WebRTC audio track would require overriding of the WebRTC native C audio processing. My experience with the C library is limited so to be able to give an estimate or concrete possibilities on this would require more research on my part.
An interesting feature that is coming is our new KastTV. With KastTV users can watch and control specific content from different services, such as our internal movie library, Tubi

and Youtube. This content is served over Http Live Streaming and it gives the parties a virtual TV to control and watch content from, even from Kast Android. This would give Android users more control over parties they ask for.

These features in mind the development of Kast Android will continue with the same team and myself as the developer. With KastTV and possibly later the screensharing capabilities Kast Android will be able to host virtual watch parties and therefore fulfill the most desired use case. These combined with other parity features will help to reach our ultimate goal of developing the application to be on par with the other clients and being able to offer the full Kast experience to the growing amount of Kasters.

Sources

Andrews, A. 2020. The ultimate guide to which socializing app is right for you, from Zoom to Netflix Party. The Washington Post. Readable: https://www.washingtonpost.com/technology/2020/03/26/zoom-skype-houseparty-discord-netflix-party/. Read: 25.5.2020.

Android 2020. What is Android. Readable: https://www.android.com/what-is-android/. Read 29.5.2020.

Android Developers 2020. OpenGL ES. Readable: https://developer.android.com/guide/topics/graphics/opengl. Read 18.6.2020.

Android Developers 2020. Android Studio. Readable: https://developer.android.com/studio. Read 29.5.2020.

Android Developers 2020. Meet Android Studio. Readable: https://developer.android.com/studio/intro. Read 29.5.2020.

Android Developers 2020. Activity. Readable https://developer.android.com/reference/android/app/Activity. Read 1.6.2020.

Android Developers 2020. Allowing Other Apps to Start Your Activity. Readable: https://developer.android.com/training/basics/intents/filters. Read 1.6.2020.

Android Developers 2020. Projects Overview. Readable: https://developer.android.com/studio/projects. Read 1.6.2020.

Android Developers 2020. Layout. Readable: https://developer.android.com/guide/topics/ui/declaring-layout. Read 1.6.2020.

Android Developers 2020. Application Fundamentals. Readable: https://developer.android.com/guide/components/fundamentals. Read 2.6.2020.

Android Developers 2020. Processes and Threads overview. Readable: https://developer.android.com/guide/components/processes-and-threads. Read 3.6.2020.

Android Developers 2020. Better performance through threading. Readable: https://developer.android.com/topic/performance/threads. Read 3.6.2020.

Android Developers 2020. Kotlin coroutines on Android. Readable: https://developer.android.com/kotlin/coroutines. Read 4.6.2020.

Android Developers 2020. AndroidX Overview. Readable: https://developer.android.com/jetpack/androidx. Read 5.6.2020.

Android Developers 2020. Layouts. https://developer.android.com/guide/topics/ui/declaring-layout. Read 5.6.2020.

Android Developers 2020. ConstraintLayout. Readable: https://developer.android.com/reference/androidx/constraintlayout/widget/ConstraintLayout. Read 5.6.2020.

Android Developers 2020. Build a Responsive UI with ConstraintLayout. Readable: https://developer.android.com/training/constraint-layout. Read 9.6.2020.

Android Developers 2020. ConstraintSet. Readable: https://developer.android.com/reference/androidx/constraintlayout/widget/ConstraintSet. Read 9.6.2020.

Android Developers 2020. Animate layout changes using transitions. Readable: https://developer.android.com/training/transitions. Read 9.6.2020.

Android Developers 2020. Create a List with RecyclerView. Readable: https://developer.android.com/guide/topics/ui/layout/recyclerview. Read 9.6.2020.

Android Developers 2020. ViewPager2. Readable: https://developer.android.com/reference/kotlin/androidx/viewpager2/widget/ViewPager2. Read 10.6.2020.

Android Developers 2020. Slide between fragments using ViewPager2. Readable: https://developer.android.com/training/animation/screen-slide-2. Read 10.6.2020.

Android Developers 2020. android.hardware.camera2. Readable https://developer.android.com/reference/android/hardware/camera2/package-summary. Read 16.6.2020.

Android Developers 2020. SurfaceView. Readable: https://developer.android.com/reference/android/view/SurfaceView. Read 19.6.2020.

Android Developers 2020. Hardware Acceleration. Readable: https://developer.android.com/guide/topics/graphics/hardware-accel. Read 19.6.2020.

Android Developers 2020. Detect and diagnose crashes. Readable https://developer.android.com/games/optimize/crash. Read 15.7.2020.

Android Developers 2020. Application. Readable https://developer.android.com/reference/android/app/Application. Read 12.8.2020.

Android Developers 2020. SharedPreferences. Readable: https://developer.android.com/training/data-storage/shared-preferences. Read 12.8.2020.

Android Developers 2020. JNI Tips. Readable: https://developer.android.com/training/articles/perf-jni. Read 20.8.2020.

Android Developers 2020. Context. Readable: https://developer.android.com/reference/android/content/Context. Read 27.8.2020.

Android Developers 2020. Platform Architecture. Readable: https://developer.android.com/guide/platform. Read 28.8.2020.

Android Developers 2020. Fragments, Readable: https://developer.android.com/guide/components/fragments, Read 15.9.2020.

Bauer, R. 2017. What’s the diff: Programs, Processes and Threads. Readable: https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/. Read 3.6.2020.

Birch, J. 2019. Exploring the View Pager2. Readable: https://medium.com/google-developer-experts/exploring-the-view-pager-2-86dbce06ff71. Medium. Read 10.6.2020.

Blume, C. 2018. History of the WebRTC API. Readable: https://www.callstats.io/blog/2018/05/11/history-of-webrtc-infographic. Read 11.6.2020.

Google Git 2016. CameraVideoCapturer. Readable: https://chromium.goog-
lesource.com/exter-
nal/webrtc/+/a8eab866a16dd1177c0c2a5118ec597eacdb225f/webrtc/api/java/an-
droid/org/webrtc/CameraVideoCapturer.java. Read 17.6.2020.

Google Git 2013. VideoCapturer. Readable: https://chromium.googlesource.com/exter-
nal/webrtc/+/b6760f9e4442410f2bcb6090b3b89bf709e2fce2/webrtc/api/an-
droid/java/src/org/webrtc/VideoCapturer.java. Read 17.6.2020.

Google Git 2015. Camera2Enumerator. Readable: https://chromium.goog-
lesource.com/external/webrtc/+/HEAD/sdk/android/api/org/webrtc/Camera2Enumera-
tor.java. Read 17.6.2020.

Google Git 2013. VideoSource. Readable: https://chromium.googlesource.com/exter-
nal/webrtc/+/HEAD/sdk/android/api/org/webrtc/VideoSource.java. Read 17.6.2020.

Google Git 2013. PeerConnectionFactory. Readable: https://chromium.goog-
lesource.com/external/webrtc/+/HEAD/sdk/android/api/org/webrtc/PeerConnectionFac-
tory.java. Read 17.6.2020.

Google Git 2015. SurfaceTextureHelper. Readable: https://chromium.goog-
lesource.com/exter-
nal/webrtc/trunk/talk/+/5eb96fc8640036ecbe789eb0c89fc55da319e2a1/app/webrtc/java/a
ndroid/org/webrtc/SurfaceTextureHelper.java. Read 17.6.2020.

Google Git 2015. EglBase. Readable: https://chromium.googlesource.com/exter-
nal/webrtc/+/HEAD/sdk/android/api/org/webrtc/EglBase.java Read 18.6.2020.

Google Git 2015. SurfaceViewRenderer. Readable: https://chromium.goog-
lesource.com/external/webrtc/+/HEAD/sdk/android/api/org/webrtc/SurfaceViewRen-
derer.java. Read 18.6.2020.

Google Git 2015. RendererCommon. Readable: https://chromium.googlesource.com/ex-
ternal/webrtc/+/172683173dd84a72659ad494962245445eb2a353/webrtc/api/java/an-
droid/org/webrtc/RendererCommon.java. Read 19.6.2020.

Google Git 2013. VideoTrack. Readable: https://chromium.googlesource.com/exter-
nal/webrtc/+/HEAD/sdk/android/api/org/webrtc/VideoTrack.java. Read 23.6.2020.

Google Git 2013. AudioTrack. Readable: https://chromium.googlesource.com/exter-nal/webrtc/+/HEAD/sdk/android/api/org/webrtc/AudioTrack.java. Read 23.6.2020.

Google Git 2017. h264_profile_level_id.cc. Readable: https://chromium.goog-lesource.com/external/webrtc/+/master/media/base/h264_profile_level_id.cc. Read 1.7.2020.

Google Git 2017. DefaultVideoEncoderFactory. Readable: https://chromium.goog-lesource.com/external/webrtc/+/HEAD/sdk/android/api/org/webrtc/DefaultVideoEncoder-Factory.java. Read 2.7.2020.

Google Git 2017. DefaultVideoDecoderFactory. Readable: https://chromium.goog-lesource.com/external/webrtc/+/HEAD/sdk/android/api/org/webrtc/DefaultVideoDecoder-Factory.java. Read 2.7.2020.

Google Git 2018. JavaAudioDeviceModule. Readable: https://chromium.goog-lesource.com/external/webrtc/+/HEAD/sdk/android/api/org/webrtc/audio/JavaAudi-oDeviceModule.java. Read 2.7.2020.

Google Git 2013. PeerConnection. Readable: https://chromium.googlesource.com/exter-nal/webrtc/+/HEAD/sdk/android/api/org/webrtc/PeerConnection.java. Read 3.7.2020.

Google Git 2013. MediaStream. Readable: https://chromium.googlesource.com/exter-nal/webrtc/+/HEAD/sdk/android/api/org/webrtc/MediaStream.java. Read 7.6.2020.

Google Play 2020. Kast – Watch Together (Beta). Readable: https://play.google.com/store/apps/details?id=com.evasyst.Kast&hl=en_US Read.25.9.2020

Kast 2020. About Kast. Readable: https://kast.gg/company.html. Read. 26.5.2020.

Kotlin 2020. Comparison to Java programming language. Readable: https://kotlin-lang.org/docs/reference/comparison-to-java.html. Read 2.6.2020.

Kotlin 2020. Basic Types. Readable: https://kotlinlang.org/docs/reference/basic-types.html. Read 2.6.2020.

Kotlin 2020. Extensions. Readable: https://kotlinlang.org/docs/reference/extensions.html. Read 2.6.2020.

Kotlin 2020. Higher-order functions and Lambdas. Readable: https://kotlin-lang.org/docs/reference/lambdas.html. Read 2.6.2020.

Kotlin 2020. Type Checks and Casts. Readable: https://kotlinlang.org/docs/reference/type-casts.html. Read 3.6.2020.

Kotlin 2020. Null Safety. Readable https://kotlinlang.org/docs/reference/null-safety.html. Read 3.6.2020.

Kotlin 2020. Coroutines for asynchronous programming and more. Readable: https://ko-tlinlang.org/docs/reference/coroutines-overview.html. Read 3.6.2020.

Kotlin 2020. Coroutine Context and Dispatchers. Readable: https://kotlinlang.org/docs/ref-erence/coroutines/coroutine-context-and-dispatchers.html. Read 4.6.2020.

Markovic, B. 2020. Process of compiling Android app with Java/Kotlin code. Readable: https://medium.com/@banmarkovic/process-of-compiling-Android-app-with-java-kotlin-code-27edcfcce616. Medium. Read 2.6.2020.

MDN Web Docs 2020. Media Capture and Streams API. Readable: https://devel-oper.mozilla.org/en-US/docs/Web/API/Media_Streams_API. Read 16.6.2020.

MDN Web Docs 2020. Web video codec guide. Readable: https://devel-oper.mozilla.org/en-US/docs/Web/Media/Formats/Video_codecs. Read 29.6.2020.

MDN Web Docs 2020. RTCPeerConnection.signalingState. Readable: https://devel-oper.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/signalingState. Read 3.7.2020.

MDN Web Docs 2020. RTCPeerConnection.iceGatheringState. Readable: https://devel-oper.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/iceGatheringState. Read 6.7.2020.

MDN Web Docs 2020. RTCPeerConnection.onicecandidate. Readable https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/onicecandidate. Read 6.7.2020.

MDN Web Docs 2020. RTCPeerConnection.iceConnectionState. Readable: https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/iceConnectionState. Read 6.7.2020.

MDN Web Docs 2020. WebSockets. Readable: https://developer.mozilla.org/en-US/docs/Glossary/WebSockets. Read 8.7.2020.

MDN Web Docs 2020. WebRTC Connectivity. Readable: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity. Read 2.9.2020.

Opus 2017. Opus Interactive Audio Codec. Readable: https://opus-codec.org/. Read 26.6.2020.

Ristic, D. 2015. Learning WebRTC. Packt Publishing. Readable: http://3.droppdf.com/files/knA3k/learning-webrtc.pdf.

Rouse, M. 2019. peer-to-peer (P2P). Readable: https://searchnetworking.techtarget.com/definition/peer-to-peer. Read 24.6.2020.

Senior, N. medium.com, 2018. RecyclerView Item Click Listener the Right Way. Readable: https://medium.com/android-gate/recyclerview-item-click-listener-the-right-way-daecc838fbb9. Medium. Read 9.6.2020.

Square 2019. OkHttp. Readable: https://square.github.io/okhttp/. Read 9.7.2020.

Square 2019. OkHttpClient. Readable: https://square.github.io/okhttp/4.x/okhttp/okhttp3/-ok-http-client/. Read 9.7.2020.

Square 2019. newWebSocket. Readable: https://square.github.io/okhttp/4.x/okhttp/okhttp3/-web-socket/-factory/new-web-socket/. Read 9.7.2020.

Square 2019. WebSocketListener. Readable: https://square.github.io/okhttp/4.x/okhttp/okhttp3/-web-socket-listener/. Read 9.7.2020.

Warren, T. 2020. Microsoft Teams jumps 70 percent to 75 million daily active users. The Verge. Readable: https://www.theverge.com/2020/4/29/21241972/microsoft-teams-75-million-daily-active-users-stats. Read 3.8.2020.

WebRTC 2020. Real-time communication for the web. Readable: https://webrtc.org/. Read 11.6.2020.

WebRTC 2020. Getting started with peer connections. Readable: https://webrtc.org/getting-started/peer-connections. Read 24.6.2020.

WebRTC Bugs 2017. Issue 8133: OPUS stereo audio over RTP is muxed to mono Readable: https://bugs.chromium.org/p/webrtc/issues/detail?id=8133#c25. Read 29.6.2020.

WebRTC Glossary 2020. DTLS-SRTP. Readable https://webrtcglossary.com/dtls-srtp/. Read 26.6.2020

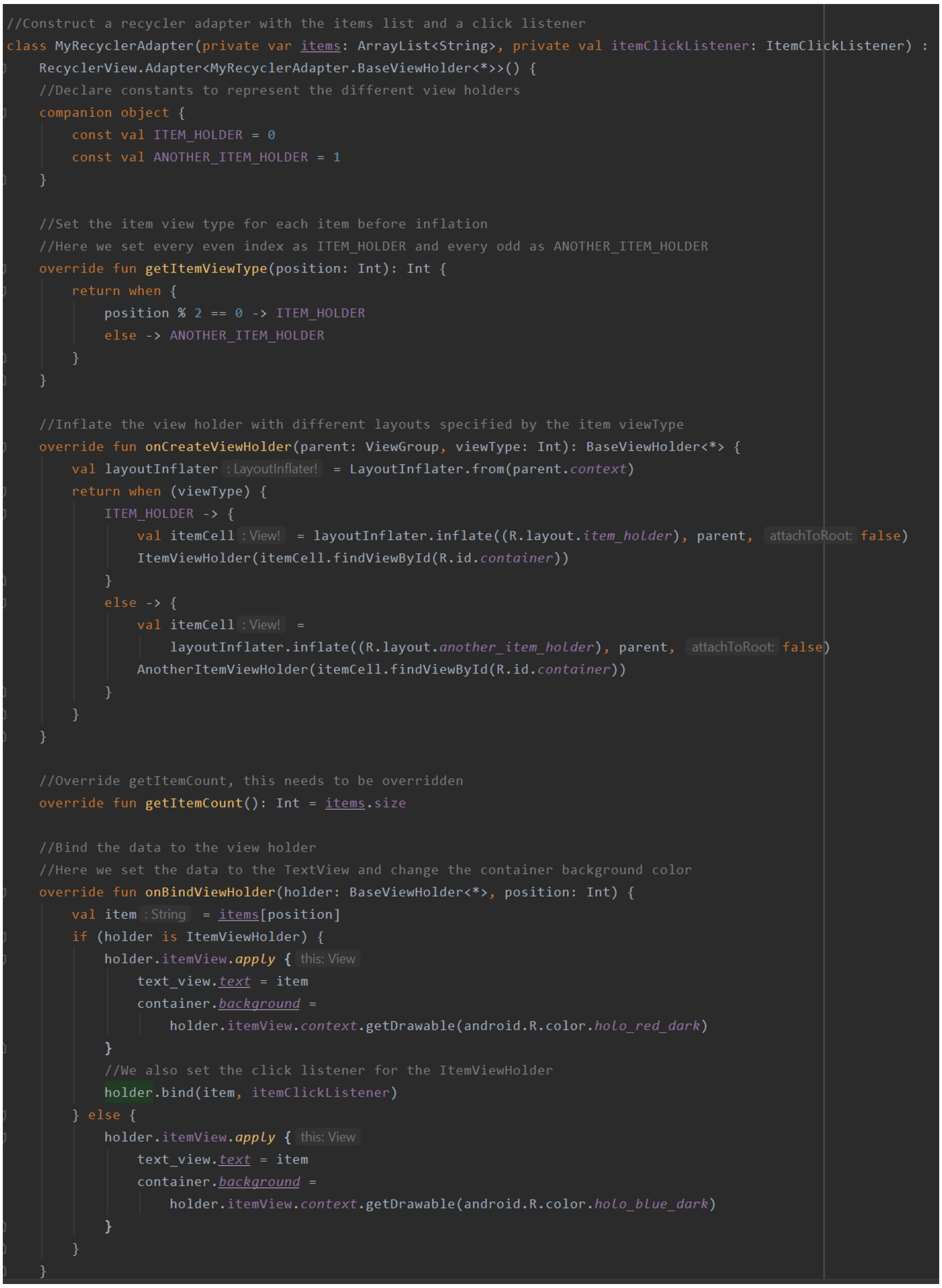
Webrtchacks 2016. Anatomy of a WebRTC SDP. Readable: https://webrtchacks.com/sdp-anatomy/. Read 26.6.2020.

Webrtchacks 2020. Not a Guide To SDP Munging. Readable: https://webrtchacks.com/not-a-guide-to-sdp-munging/. Read 29.6.2020.

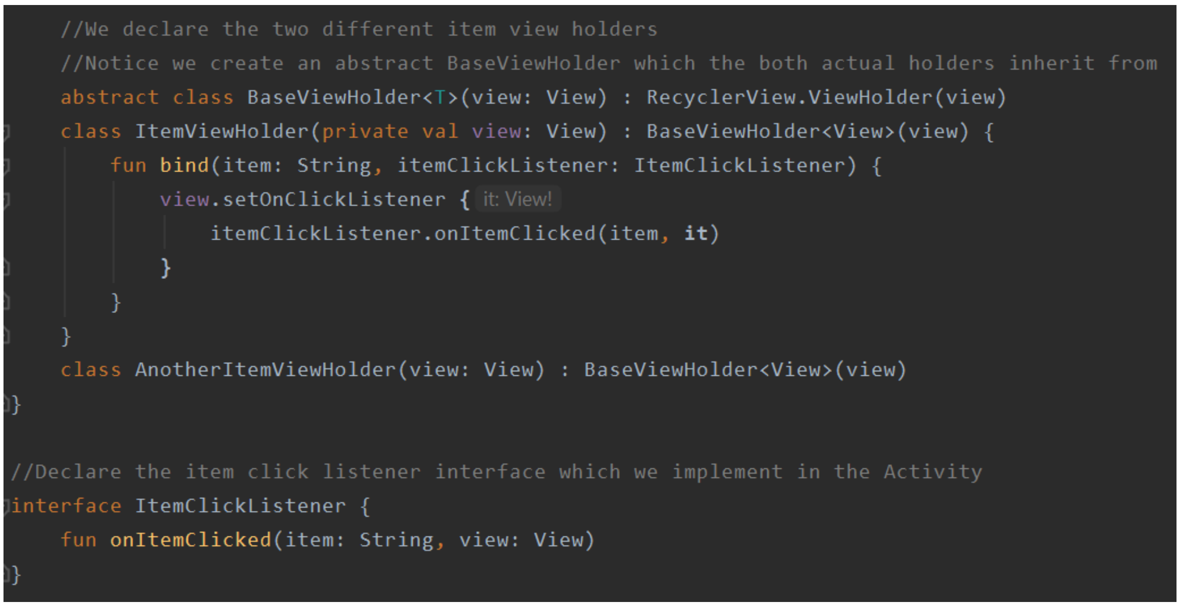
Appendix

Appendix 1. Recycler View adapter example

An example of a RecyclerView adapter with two view holders and a click listener.



```
//Construct a recycler adapter with the items list and a click listener
class MyRecyclerAdapter(private var items: ArrayList<String>, private val itemClickListener: ItemClickListener) :
    RecyclerView.Adapter<MyRecyclerAdapter.BaseViewHolder<*>>() {
    //Declare constants to represent the different view holders
    companion object {
        const val ITEM_HOLDER = 0
        const val ANOTHER_ITEM_HOLDER = 1
    }

    //Set the item view type for each item before inflation
    //Here we set every even index as ITEM_HOLDER and every odd as ANOTHER_ITEM_HOLDER
    override fun getItemViewType(position: Int): Int {
        return when {
            position % 2 == 0 -> ITEM_HOLDER
            else -> ANOTHER_ITEM_HOLDER
        }
    }

    //Inflate the view holder with different layouts specified by the item viewType
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): BaseViewHolder<*> {
        val layoutInflater : LayoutInflater! = LayoutInflater.from(parent.context)
        return when (viewType) {
            ITEM_HOLDER -> {
                val itemCell : View! = layoutInflater.inflate((R.layout.item_holder), parent,  attachToRoot: false)
                ItemViewHolder(itemCell.findViewById(R.id.container))
            }
            else -> {
                val itemCell : View! =
                    layoutInflater.inflate((R.layout.another_item_holder), parent,  attachToRoot: false)
                AnotherItemViewHolder(itemCell.findViewById(R.id.container))
            }
        }
    }

    //Override getItemCount, this needs to be overridden
    override fun getItemCount(): Int = items.size

    //Bind the data to the view holder
    //Here we set the data to the TextView and change the container background color
    override fun onBindViewHolder(holder: BaseViewHolder<*>, position: Int) {
        val item : String = items[position]
        if (holder is ItemViewHolder) {
            holder.itemView.apply { this: View
                text_view.text = item
                container.background =
                    holder.itemView.context.getDrawable(android.R.color.holo_red_dark)
            }
            //We also set the click listener for the ItemViewHolder
            holder.bind(item, itemClickListener)
        } else {
            holder.itemView.apply { this: View
                text_view.text = item
                container.background =
                    holder.itemView.context.getDrawable(android.R.color.holo_blue_dark)
            }
        }
    }
```



```
    //We declare the two different item view holders
    //Notice we create an abstract BaseViewHolder which the both actual holders inherit from
    abstract class BaseViewHolder<T>(view: View) : RecyclerView.ViewHolder(view)
    class ItemViewHolder(private val view: View) : BaseViewHolder<View>(view) {
        fun bind(item: String, itemClickListener: ItemClickListener) {
            view.setOnClickListener { it: View!
                itemClickListener.onItemClicked(item, it)
            }
        }
    }
    class AnotherItemViewHolder(view: View) : BaseViewHolder<View>(view)
}

//Declare the item click listener interface which we implement in the Activity
interface ItemClickListener {
    fun onItemClicked(item: String, view: View)
}
```