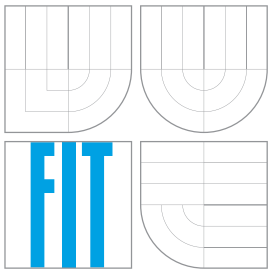
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ANDROID IP KAMERA

ANDROID IP CAMERA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAN CHVÁLA

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Ing. ADAM HEROUT, Ph.D.

BRNO 2015

Abstrakt

Cílem této práce je navrhnout systém, který by umožnil odesílání dat z videokamery mobilního zařízení a jejich zobrazení v reálném čase prostřednictvím webového prohlížeče. Součástí práce je popis použitých technologií a také popis cílové implementační platformy Android. K řešení získání a přenosu multimediálních dat byla využita technologie Web Real Time Communications (WebRTC), která je nativně podporovaná novými prohlížeči a komponentou WebView (Android verze 5.0 a výše). Zasílání push notifikací ze strany serveru na mobilní zařízení pro spuštění streamu je řešeno pomocí Google Cloud Messaging technologie. Výsledný systém umožňuje uživateli pomocí webového prohlížeče spustit aplikaci na mobilním telefonu a tím zahájit přenos multimediálních dat. Ten je možné parametrizovat a zabezpečit pomocí hesla. Přínosem práce je seznámení s technologií WebRTC a demonstrace jejího snadného využití implementací IP kamery na platformě Android.

Abstract

The goal of this thesis is to design a system which would allow video data streaming from a mobile device and real time playback using a standard web browser. The technological background and the implementation platform are both part of this thesis. Web Real Time Communications (WebRTC) technology was used for acquiring multimedia data on mobile device. This technology is natively supported in the latest major web browsers and in WebView component (Android version 5.0 and above). Sending push notifications from a server to a mobile device to start the streaming is done with Google Cloud Messaging technology. The resultant system allows a user to start the application on mobile device with easy web browser access. This starts the multimedia stream from device, which can be parametrized and secured by password. The benefit of this thesis is the overview of WebRTC technology and its demonstration. The IP camera implementation shows how easy it is to use the WebRTC in real applications.

Klíčová slova

Real Time Communications, WebRTC, Android, Google Cloud Messaging, IP kamera

Keywords

Real Time Communications, WebRTC, Android, Google Cloud Messaging, IP camera

Citace

Jan Chvála: Android IP camera, diplomová práce, Brno, FIT VUT v Brně, 2015

Android IP camera

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doc. Ing. Adama Herouta Ph.D. a že jsem uvedl všechny literární prameny, ze kterých jsem čerpal.

.....
Jan Chvála
June 1, 2015

Poděkování

Tímto bych rád poděkoval mému vedoucímu diplomové práce panu doc. Ing. Adamu Heroutovi, Ph.D., za pomoc při výběru a formování tématu, odborné konzultace a potřebnou motivaci.

Contents

Chapter 1

Introduction

Last twenty years of technological innovation and development has brought us Internet Protocol (IP) cameras which are part of our everyday life. As a result of hardware cheapening the application of IP cameras is not restricted only for the enterprise usage but small companies, shops and even houses are protected with them more and more often. Anyone can buy expensive IP cameras or cheaper variations but there is no easy–to–use solution if you just want to test an IP camera without spending any money.

Software is also evolving very fast. The Web Real Time Communications (WebRTC) technology is being developed in recent years. The first technology of its kind for enabling peer–to–peer connection between two endpoints in the internet. It is still not yet completely standardized in the time of writing this thesis. But together with technological draft the reference implementation is developed as open–source project and major web browsers like Chrome, Firefox or Opera claim to support WebRTC or at least the main parts of it.

Mobile devices have the hardware necessities to be used as temporary IP cameras. The wireless connectivity is present and integrated cameras have sufficient resolution even for low cost devices. WebView (Android's native component with web browser engine) supports WebRTC technology for Android version 5.0 and above which makes it the right candidate for the mobile part of the resultant system.

This thesis will focus on exploration of this new technology for creating simple system for video streaming from Android device. The application will allow user to start the stream remotely by using Google Cloud Messaging (GCM) and view the stream on a web page.

The beginning of the thesis (chapter 2) focuses on WebRTC which is the main implementation pillar for the resultant system. Then the Android Operating System and its relevant parts are described in chapter 3. Chapter 4 covers information about GCM technology used for sending messages from server to mobile application client. Design of the resultant system and its implementation are described in chapter 5. Testing and measurements are in chapter 6. The very last chapter 7 summarize the results of this master thesis.

Chapter 2

Web Real Time Communications

This chapter focuses on the Web Real Time Communications (WebRTC) technology which is the main pillar of the resultant system. WebRTC is a group of open standards which are being developed by World Wide Web Consorcium (W3C) and Internet Engineering Task Force (IETF). W3C is concentrating on JavaScript Application Programming Interface (API) which serves as a bridge between web application and Real Time Communications (RTC) function (see 2.1.1). IETF is developing protocols used by RTC functions to communicate with each other. All the specifications are still actively developed in the time of writing this thesis but pre–standard implementation is already available as open–source project under the WebRTC name.

This technology allows to acquire local media data stream through simple API, connect to another endpoint in the internet and stream the multimedia data to it.

Knowledge base for most of the information is called „The WebRTC book“ in it's third edition [8].

2.1 Components

The WebRTC technology is composed of a couple of components. This thesis focuses on web applications which are the main category where this technology should be used. Figure 2.1 shows how the WebRTC is placed in the web browser and WebRTC architecture itself. The important parts are Web API, which is the one we use in applications, RTC function, which handles peer–to–peer connection, and media engines for capturing user media.

2.1.1 Real Time Communications function

The most important part of WebRTC is RTC function module. This module is responsible for communication between other RTC functions using on–the–wire protocols[1] and for communication with operating system. Each RTC function is considered to be an endpoint in the World Wide Web (WWW) and in this thesis it will be also referred to as peer.

Topologies

The connection between peers can be established basically as two different topologies Triangle and Trapezoid. Triangle means that two peers are using the same web application

[1] Such as Transmission Control Protocol (TCP) or User Datagram Protocol (UDP).

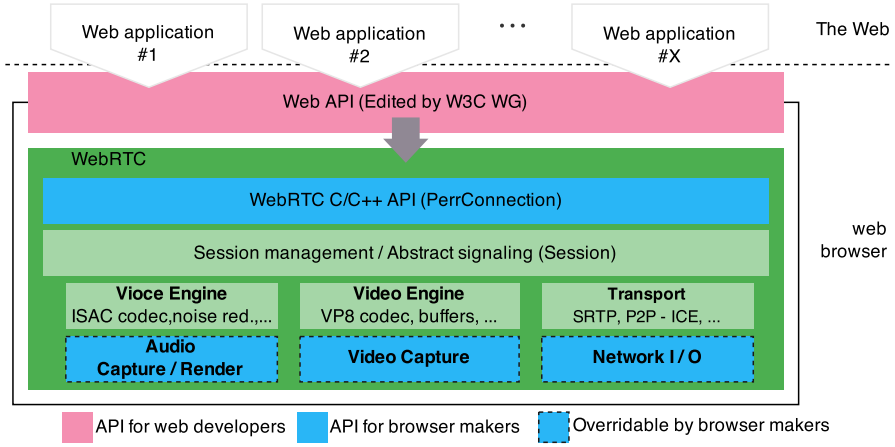


Figure 2.1: WebRTC architecture and it's components.

on the same server[2] while Trapezoid on the other hand is when each peer is using different web application (Figure 2.2) on a different server.

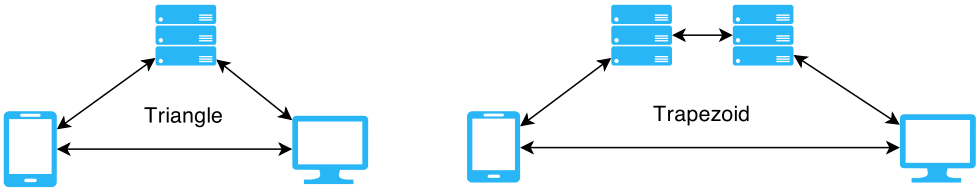


Figure 2.2: WebRTC peer connection topology topology.

2.1.2 Signalling servers

Signalling is not a part of WebRTC effort for standardization but it is intensively used for establishing peer–to–peer connection between two peers. It is described in section 2.4.

WebRTC uses technology to overcome common problems when communicating in WWW on top of signalling process. Usually most of the peers are hidden behind Network Address Translators (NAT)[3]. In this case direct communication between peers behind different NATs is not possible.

Interactive Connectivity Establishment (ICE) helps to deal with NATs and WebRTC and provides the ability to set up signalling so it can use ICE for NAT traversal. ICE uses Session Traversal Utilities for NAT (STUN) to gather all candidate addresses from both peers and systematically tries all possible pairs [9] to establish peer–to–peer connection. Traversal Using Relays around NAT (TURN) servers have to be used if at least one peer is behind symmetric NAT. There are public STUN and TURN servers available and we can always deploy our own servers[1].

[2]Triangle topology is what we will use in the resultant system.

[3]Networking devices for translating local IP addresses to the one used as an endpoint to another network. For more information see [11]

[1]Example of open–source STUN and TURN server implementation:
https://code.google.com/p/rfc5766-turn-server/

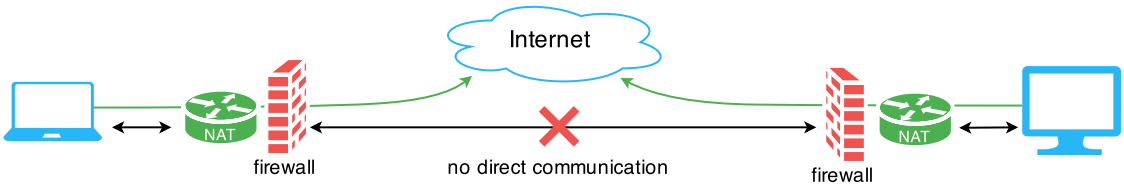


Figure 2.3: Direct peer–to–peer communication is not possible in real world.

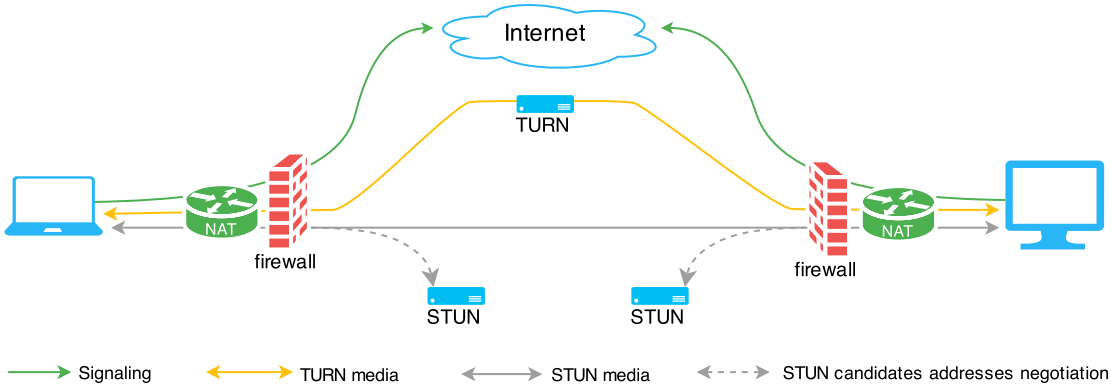


Figure 2.4: Direct peer–to–peer communication using STUN and TURN servers.

2.2 Local media streams and tracks

WebRTC includes the standardization of how the media is being modelled. This section describes Tracks and Streams as the main entities.

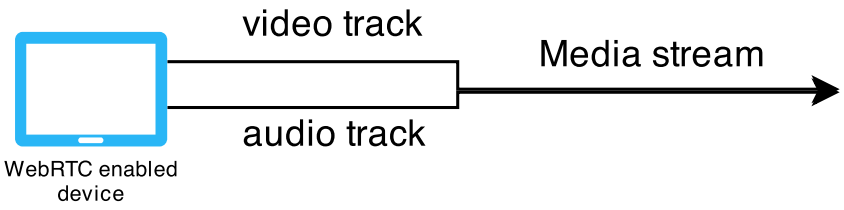


Figure 2.5: WebRTC tracks combined to a media stream.

2.2.1 Media tracks

Media of single type returned from single device is called source. This source can be simple as mono audio or complex as multi–channel surround audio but still a single track. In WebRTC this track is represented as object called *MediaStreamTrack*. It is intended for it to be transferred as a single unit over Peer Connection using Real–time Transport Protocol (RTP) payload. The object encapsulates the source so that the developer cannot manipulate the source directly but rather use the object.

The track contains `muted` and `enabled` boolean attributes which may be manipulated by a user or programmatically. It should allow a user to mute track to temporarily show black video or silent audio. Unlike muted, the disabled tracks are not transmitting any data at all.

There is also `readyState` attribute which is set by WebRTC implementation internally. It is treated as follows:

- **new** – Created track which is not connected to media yet.
- **live** – Track which is ready to be streamed.
- **ended** – Source is not providing data any more and it is not possible that it will provide any data in the future again.

These attributes are independent, so the track may be *live*, *enabled* and *muted*.

2.2.2 Media Streams

Media tracks can be bundled together in *MediaStream* object. This object can be obtained by requesting local media, by duplicating the existing *MediaStream* or by receiving streams from Peer connections. It contains a collection of tracks which can be manipulated with `addTrack()` and `removeTrack()` methods.

Mixing tracks from multiple sources is allowed and thus one stream can contain media tracks e.g. from two microphones and a video camera. In current implementation all the tracks are synchronized but it is being discussed in WebRTC draft [4] to allow disabling of the synchronization to avoid delays.

The *MediaStream* has attribute `active` which is set to true if at least one its track is not ended. Otherwise it is false and indicates that it will no longer provide any data.

2.3 Using Web Real Time Communications

When using WebRTC four main actions have to be taken in order to successfully create WebRTC session (Figure 2.6):

1. Get local media.
2. Establish P2P connection.
3. Add media and data channels to connection.
4. Exchange session description with other peer.

2.3.1 Getting local media

WebRTC API provides `getUserMedia()` function which was created to simplify the process of acquiring single local media stream as *MediaStream* object which can be combined together with *MediaStream* API.

The `getUserMedia()` function takes a JSON object as a parameter which represents settings and constraints for the required media. There are `audio` and `video` properties

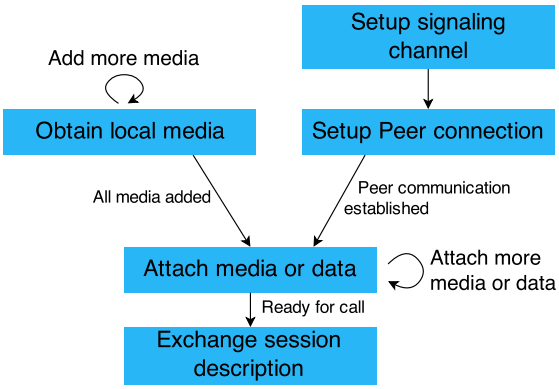


Figure 2.6: Setting up and exchanging session description.

representing each media type which can be set to Boolean value or an object. Boolean indicates whether the media type is required or not, while object represents a set of `mandatory` and `optional` constraints. Currently supported video constraints are `width`, `height`, `framteRate`, `aspectRation` and `facingMode`. Audio constraints are `volume`, `sampleRate`, `sampleSize` and `echoCancelation`.The `successCallback` is invoked when all constraints are fulfilled and `errorCallback` if they are not.

Example video object with constraints:

```
{
   mandatory: {
     width: { max: 640 }
   },
   optional: {
       facingMode: 'user',
       width: { min: 320 }
   }
}
```

For security reasons the applications should indicate that the local media is being accessed by asking the user for permission.

2.3.2 Peer connection

Direct connection between two endpoints (peer) in World Wide Web is handled with RTCPeerConnection API. This allows peers to be connected without the need of any server once the connection is established. When joining a conference[1] the Peer Connection has to be created between every two peers.

2.3.3 Exchanging media

Peer connection allows multiple media streams to be attached. Renegotiation of how the media are going to be represented is needed whenever the media changes. The representation is managed by RTCSessionDescription API which currently supports only Session

[1]Conference is connection between more than two peers.

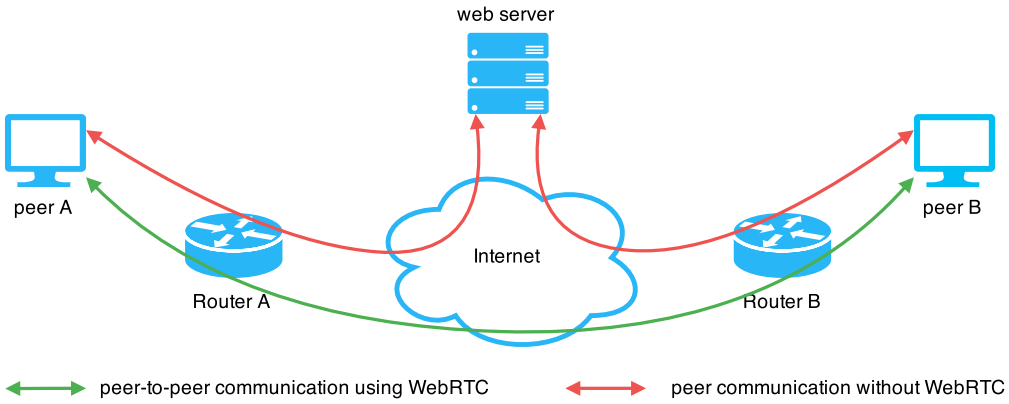


Figure 2.7: Peer–to–peer connection and media transfer.

Description Protocol (SDP) for session description format[1]. This description may also be adjusted manually but it is expected not to be touched in most cases.

Successful media session exchange triggers the ICE hole punching process for NAT traversal using STUN servers followed immediately with key negotiation for securing the media session transport. JavaScript API also provides the possibility to add TURN servers to rely on media when symmetric NATs are used.

2.3.4 Closing connection

Closing session may be done manually or caused by connectivity loss. When the connection is interrupted, the ICE will try to restore it automatically. The hole punching will be initiated again and if that also fails, the session and all its permissions to access the media are invalidated[2].

Each peer should close RTCPeerConnection with its `close()` function when it is no longer needed. This will stop the connection correctly and no attempts to restart session will be performed.

2.4 Signalling process is important

Signalling process is essential for establishing peer-to–peer connection. It has an important role in WebRTC but in contrast to other parts of WebRTC, it does not need to be standardized. WebRTC can work with multiple signalling protocols so the developer can choose the right one for his purposes. You can see component communication together with signalling servers in figure 2.8.
These four things summarize the purpose of signalling in WebRTC:

1. Media capabilities and settings negotiation.

2. Participants identification and authentication.

[1]WebRTC in its version 1.1 may include object session representation which is actively developed as ObjectRTC (ORTC).

[2]New permissions have to be obtained in case that new session will be stated.

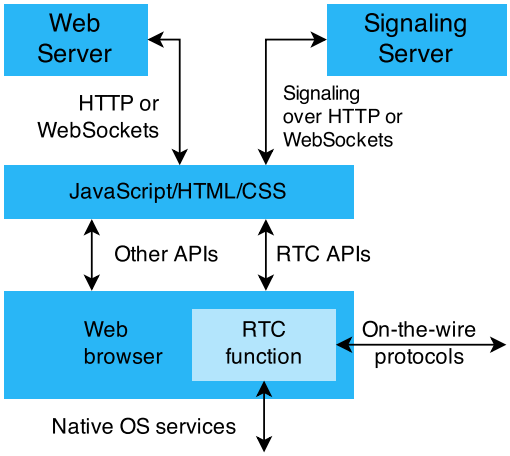


Figure 2.8: WebRTC components and communication.

3. Controlling the media session.

4. Resolution in conflicting session change from both sides at the same time.

The lack of standardization is because it does not have to be standardized at all. The important thing is that a server has to ensure that both peers are using the same signalling which can be easily achieved by serving the same JavaScript code which encapsulates the type of signalling. In comparison with for example Voice over Internet Protocol (VoIP), where there is no possibility for changing the signalling protocol[1], this is definitely a big advantage.

2.4.1 Media capabilities negotiation

The essential function is the negotiation between peers about session description. For these purposes the Session Description Protocol (SDP) is used. The object API should be supported to replace the SDP in WebRTC 1.1 because SDP is hard to parse in JavaScript language. The SDP contains information for Real–time Transport Protocol (RTP) about included media, codecs and its parameters and bandwidth.

Another role of signalling is to exchange information about candidate addresses used for Interactive Connectivity Establishment (ICE) hole punching which make NAT traversal technique possible. This information may be sent together with SDP or outside its scope.

2.4.2 Signalling transport

Signalling in WebRTC relies on bi–directional signalling channel between two peers. This can be achieved by HTTP, WebSockets or the data channels.

When using HTTP transport, the signalling information messages can be transferred using HTTP GET and POST methods or in their responses. Peers can send information to server easily but in order to be able to send information from server we need to use things like AJAX[2] or pooling the GET request which leaves transport connection open.

[1] Both nodes have to use the same protocol for example SIP or Jingle.
[2] Asynchronous JavaScript and XML

By establishing WebSockets connection from peer to server a bi-directional channel is created. Exchanging signalling information is easy from both sides. The WebSockets channel can not be created between two peers directly because of NATs in the way. Although this would seem as a perfect solution, some firewalls and web proxies block WebSockets connections.

Signalling using data channels is a special case. Data channels are fast, reliable connection with low latency between peers. However, in order to establish the data channel, you need to have a separate signalling process. Data channels are not meant to be used for a complete signalling process but rather for signalling audio and video media changes once the connection is established.

2.5 Support and future

Unfortunately, WebRTC is still not fully working in all major browsers today. There are disagreements about the used codecs, so the full standard specifications have not been released yet. It seems that WebRTC is not ready for mass production yet but e.g. Google Hangouts or Facebook Chat now supports WebRTC based video calls in compliant browsers.

2.5.1 Browsers support

Chrome, Firefox, Opera and Bowser already natively support WebRTC pre–standard while Internet Explorer and Safari need external plug–in to work with it.

Microsoft is actively collaborating on Object Real Time Communication (ORTC) API for WebRTC standard which should be a part of WebRTC 1.1 and it should overcome the painful SDP format which is not convenient to work with in JavaScript. They are working on the implementation [3] but we will have to wait if this is going to be included in Internet Explorer or the new Edge browser[1].

The intentions of Apple about WebRTC support in their products are as always kept in secret.

Figure 2.9: WebRTC browser support – May 2015.

2.5.2 Business and mass production

Enterprise video conferencing companies are not so enthusiastic about WebRTC and they are waiting for WebRTC to overcome its flaws[2]. They need quality assurance for paid services and also high security policies which are not completely satisfactory.

[1] Miscrosoft Edge web browser: http://www.browserfordoing.com/en-us/
[2] WebRTC compromises VPN tunnels by leaking user's real IP address [7]

WebRTC is still a draft in its early developmental state but the working progress is fast and it seems to have bright future. Users will benefit from its easy usage and companies from reducing their operational costs. The 2015 could be the year of WebRTC to become a huge player in the field of multimedia streaming technologies.

Chapter 3

Android operating system

This chapter introduces the Android platform and its components which are important for understanding the implementation specifics of the resultant mobile application. Then it describes how to access and work with camera on Android device using the WebRTC technology described in the chapter 2.

Existing applications related to Android and video streaming are at the end of this chapter.

3.1 Android platform

Android is a well known operating system based on Linux kernel. It was created mainly for mobile devices but it has spread into other types of devices such as tablets, TVs, watches, other wearable technology and it can also be found in modern cars. There is not any other operating system on the market which is used in so many various situations.

The beginning of this section contains a brief history of Android operating system, its architecture and components. Development tools as Android Studio[1] and also new build system called Gradle are described at the end of this section.

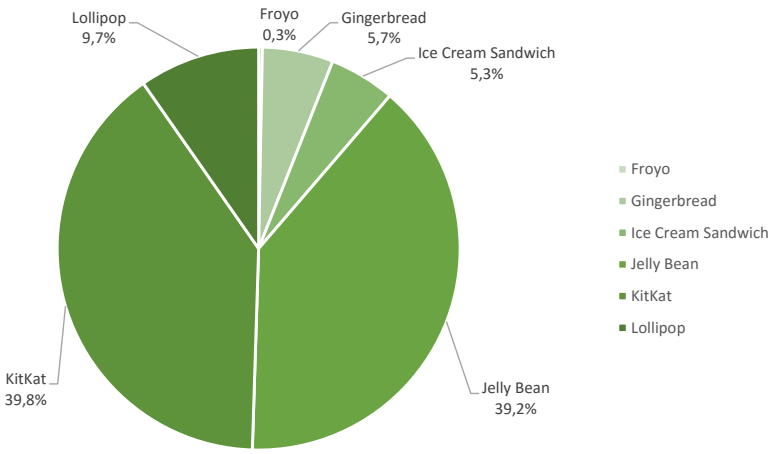


Figure 3.1: Android APIs. See Android Dashboards for recent information [5].

[1] new Integrated Development Environment (IDE) for Android development

3.1.1 History

History of Android starts in the year 2003 when Android, Inc. was founded by Andy Rubin, Rich Miner, Nick Sears and Chris White. At first they wanted to create a simple and powerful operating system for digital cameras but as soon as they realized that the market is not big enough„ they decided to focus more on mobile operating systems. In 2005 they ran out of money and they were taken over by Google, Inc.. There was not much known about the Google's Intentions with Android at that time.

At the end of 2007 the Open Headset Alliance revealed their plan to develop open standards for mobile devices including Android as their first product. First commercially distributed device running Android was HTC Dream which was released almost one year later.

Android has gone through numerous changes which were divided into different API levels. The figure 3.1 shows the amount of distribution for major APIs. We will be focusing the implementation on Android 5.0 and above (Lollipop), which natively supports all technologies we need.

3.1.2 Architecture

Android uses a stack of software components divided into four layers – see figure 3.2.

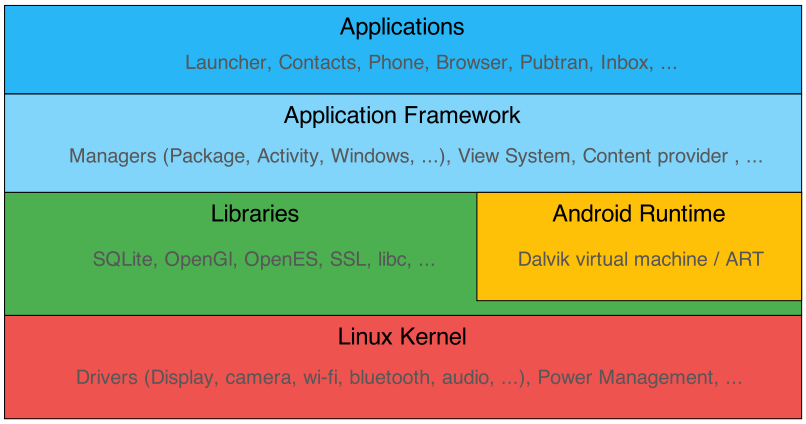


Figure 3.2: Android OS architecture layers.

Kernel

Android OS is based on Long Term Support releases of Linux kernel. The most recent Android (Lollipop) has Linux kernel in version 3.10 but the version also depends on hardware and the device itself.

It contains hardware drivers, power management capabilities and other low level services. Linux and Android aim to include Android hardware drivers and specific features into Linux kernel, so they could both use the same kernel without a lot of modifications.

Middleware

On top of Linux kernel there is a middleware layer which contains libraries together with Android Runtime both written in C language. Libraries provide capabilities for SQLite,

OpenGL, libc and many other low level functions. Android Runtime contains core libraries and special virtual machine for running Android applications. From the first versions of Android the only virtual machine (VM) available was Dalvik VM. It runs a modified Java byte code which focuses on memory efficiency and it uses Just In Time compilation (JIT)[1]. Together with Android 4.4 the new runtime called ART was introduced as an experimental feature and it is natively supported in Android 5.0. It brings Ahead Of Time compilation (AOT)[2] and a better garbage collection.

Application framework

The Application framework layer provides high level software components that can be used directly by applications. They are exposed as Java classes with well documented API, so it is easy to include them into applications. It provides capabilities such as Window, Activity and Resource managers and many other services forming the Android operating system capabilities.

Applications

The application layer is the one where all applications are installed. It can directly use components from Application framework or use Java Native Interface and implement functions in native C/C++ code.

3.1.3 Components

The Application framework provides a set of reusable components which let you create very rich applications. This section is just an overview of these components without a lot of implementation and design details. For further information about each component see Google developer guides [2].

Android Manifest

Android Manifest is not truly a component but it has to be included in every application. It is structured XML[3] file, which contains information about the application itself. Android uses this file to determine application's package, components, required permissions, system version restrictions, list of linked libraries and many other things.

Intents and Intent filters

The Intent is a messaging object used for asynchronous communication between application components. The messages can be sent between components from the same applications as well as components between different applications.

The Intent can contain additional data inside Bundle object which may be used by the receiver of Intent.

[1] Applications are translated into native code every time they are launched.

[2] Applications are translated into native code in time of installation. This increases the performance over memory efficiency

[3] XML stands for Extensible Markup Language.

Three main things that the Intent is used for:

- **To start an Activity:** Basic mechanism to start a single Activity. Intent contains information about which activity should be started and how.
- **To start a Service:** Services can be started very much alike the Activities but they run without the access to a user interface thread.
- **For delivering a broadcast:** A broadcast message can be received by many applications at the same time. The system events such as charge state change or connectivity change are broadcasted and they are available to all broadcast receivers.

Activities

An activity represents a single screen of application tied to a user interface so that a user can interact with another user through graphical elements and touch gestures. Activity's life–cycle is very important because it illustrates how the Activity interacts with the system as shown in figure 3.3.

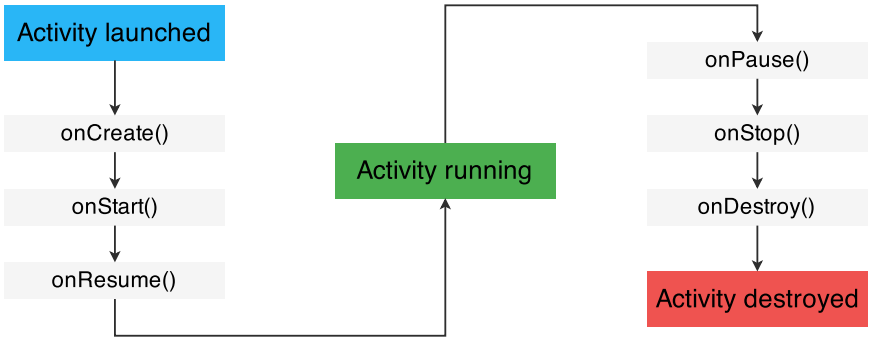


Figure 3.3: Activity's simplified lifecycle.

The system creates Window before entering in `onCreate()` method so we can place Activity's User Interface (UI) with `setContentView(int viewID)` method. Our implementation uses the life–cycle to acquire (inside `onStart()`) and release (inside `onStop()`) WakeLock object which is used to prevent device from sleeping while capturing the media. You can read more about Activities in a developer guide [1].

Fragments

Fragments were introduced in Android 3.0 as a concept of reusable small groups of graphical elements tied together and enhanced with additional logic. Multiple fragments can be placed into one Activity to build a multi–pane layouts or they can be used in Dialogs. Fragments can also be nested in Android 4.2 and above. Alike Activities, Fragments also have their own life–cycle.

Services

A service component is not tied to the user interface. This component was designed for long lasting operations which should be performed in background in order not to slow down the

application. It can be run in the main application process or in a separate one. Activity can bind the service to be able to communicate with it or it can just send Intents to it.

Service can be started in two modes. It can be bounded or unbounded. Unbounded services are explicitly started and stopped, while bounded services are automatically created when Activity binds to it and they are destroyed when all activities unbound.

We use a special case of unbound service which is called IntentService. It allows us to perform short tasks in background, which is ideal for handling push notifications (more in chapter 4) from server and react to them. IntentService's simplified life–cycle is shown in figure 3.4.

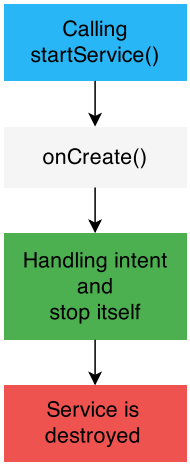


Figure 3.4: IntentService's simplified life–cycle.

For further information about Services see developer guide [10].

Content Providers

Content providers add the possibility to share structured data among applications. There are many content providers built into the system which expose contacts, media, SMS and other useful information. Developers can use this information in their own applications or provide their own content.

App Widgets

App Widget is a small part of the application that can be easily embedded into other applications. These components have specific user interface and they are periodically updated. The Widgets are extensively used in launcher applications to serve important information directly to a user without the need of launching the application itself.

Processes and Threads

When application is started, the system creates new Linux process with a single thread – "main" thread. Starting another application's component does not create another process or thread unless specified otherwise. The main thread is designed to serve short lasting operations that manipulate UI, but it is not suitable for long lasting operations such as socket communication or media playback because it causes UI to lag, which breaks down

the user experience (UX). Any non instant operations should be moved into a separate thread or a separate process which does not block the main thread of the application.

3.1.4 Development tools

Android development process has recently gone through a lot of changes. New official Integrated Development Environment (IDE) was introduced together with a new Android build system which brought new possibilities for building application variants.

Figure 3.5: Android studio and Gradle logos.

Android Studio

The Google developer team announced Android Studio on Google I/O conference in May 2013. First IDE completely dedicated to Android development. It is built on top of IntelliJ IDEA community edition from JetBrains and together with a new Android build system it is much more oriented to project scalability and maintenance than the previous combination of Eclipse IDE with Android Developer's Tools. At the end of 2014, Google released stable Android Studio in version 1.0 together with Lollipop Software Development Kit (SDK).

Android build system

Android build system brings complex project configuration which allows to build, test, package and run applications. It is based on Gradle but specifics for Android are implemented with Android Gradle plugin.

Gradle is powered by Groovy Domain Specific Language and as a language for automation it really makes the build process self independent and automatic. It combines the power and flexibility of Ant together with dependency management and plug–ins from Maven and much more.

Android Gradle plug–in adds Android Manifest merging capabilities, build types and flavours, code obfuscation, signing configurations and support for other specifics of Android platform.

3.2 Android's WebView

WebView is Android View for loading Hyper Text Markup Language content. It was tied together with operating system and could not be updated until Android Lollipop. Then it was made available through Google Play as an application called Android System WebView.

3.2.1 WebRTC in WebView

WebRTC is available in WebView since version 36.0.0 which was shipped with Android Lollipop. We cannot use WebRTC technology in native applications with devices running older Android versions because the older WebView does not support it.

3.2.2 Using the WebView

WebView can be used like any regular Android View in XML layout or instantiated programmatically. You can then call `loadData()` or `loadUrl()` functions to display or download the content.

If we want to use WebRTC in the WebView we have to enable JavaScript and set up WebChromeClient which handles permissions when the application asks for user media.

Enabling JavaScript:

```
WebSettings webSettings = webViewInstance.getSettings();
webSettings.setJavaScriptEnabled(true);
```

Setting the WebChromeClient for permission handling:

```
webViewInstance.setWebChromeClient(new WebChromeClient() {
  @Override
  public void onPermissionRequest(final PermissionRequest r) {
    // r.grant(request.getResources()); allow the request
    // r.deny(); deny the request
  }
});
```

3.3 Existing applications

There are good applications on Google Play store which have the ability to work with a camera and stream its content. The problem with existing solutions is that they are based on older technologies like pure Real Time Streaming Protocol (RTSP) which does not solve the problem of NAT and firewall restrictions. There is no problem with using such applications in close self managed environment but they can hardly be used in real world usage multimedia server infrastructure.

This section shows two applications which were used as inspiration for design, usability and used technologies at the beginning of the theoretical preparation for the thesis.

3.3.1 Spydroid–ipcamera

Spydroid–ipcamera is a very simple application for audio and video streaming. It is built on top of libstreaming [1] library. It has its own RTSP server implementation for simple streaming to RTSP clients. It also includes the possibility to start HTTP server, which can provide more settings for the stream.

This application is open sourced but not maintained at the time of writing. It lacks the support for delivering the content into a cloud or the possibility to view the stream on a web page.

[1] Simon Guigui and contributors – https://github.com/fyhertz/libstreaming/.

3.3.2 IP Webcam

IP Webcam has the same functions as Spydroid–ipcamera and adds much more complex settings and better HTTP server implementation. It also lacks the ability to view the stream on a web page.

There are no implementation details available – this is a proprietary software actively developed.

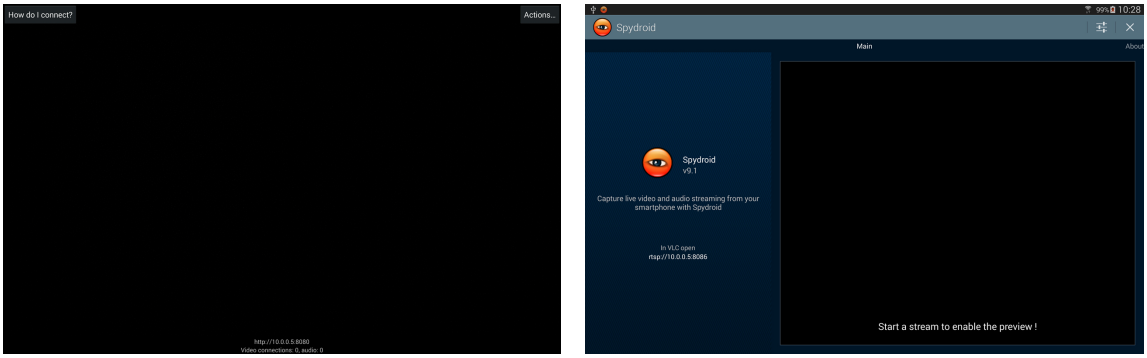


Figure 3.6: Screenshots with UI of current applications.

The user interface in both applications is very simple. It shows only details for connection and video preview as shown in figure 3.6.

Chapter 4

Google Cloud Messaging

Google Cloud Messaging (GCM) for Android is a free service allowing to send messages from server to GCM–enabled applications. These messages can be used to initiate some action or to send messages with content to a specific application.

This service is an important part of the resultant system. It allows us to start Android application and initiate streaming just in time when playback is requested.

Google Cloud Messaging is supported on Android with version 2.2 (Froyo) as minimum.

4.1 Architecture overview

GCM implementation consists of three main components which communicate between each other as shown in figure 4.1.

4.1.1 Components

There are three components in GCM.

Figure 4.1: Google Cloud Messaging components interaction

GCM connection Servers
This is the main component placed in between the client applications and application server which provides connectivity and services for sending messages to client applications. These servers are private and owned by Google Inc.. Currently there is HTTP and XMPP protocol support.

Application server
This component has to be implemented. Its main purpose is to receive registration IDs from clients so that it can use them later on for sending messages to them via GCM connection servers.

Client application
GCM–enabled Android application has to be registered on GCM connection servers to get identification. It can then receive downstream messages from a server or send upstream messages to a server[1].

4.1.2 Credentials

Identification tokens are used for authentication and to ensure that the message reaches the correct destination.

Sender ID
This is the number of projects registered in Google API console. It is used to identify a server when sending messages and also for client registration to allow messages from specific server.

Sender auth token
This token is used for application server authentication. It has to be included in the headers of POST request.

Application ID
Application ID (package name) is used for applications on registering to GCM on Android platform. This ID should be used to ensure that messages are sent to the right application.

Registration ID
This ID is generated upon client application registration from GCM connection servers and it is essential for client application identification for sending a message.

4.2 Sending Messages from application server

Application server has essential role in GCM implementation. It has to be able to communicate with GCM–enabled clients and GCM connection servers. It has to be capable of collecting registration IDs from clients and save them so that they can be used later on when sending messages.

4.2.1 Types of GCM Connection servers

There are two types of GCM connection servers which differ from each other by the used communication protocol — HTTP and XMPP. The capabilities are also different so it is possible to use them separately or both at the same time.

[1] Upstream messages are possible to be sent only when using XMPP protocol.

The HTTP implementation can send only downstream (cloud–to–device) synchronous messages with maximum payload size 4KB of data. The payload can be plain text or JSON object and it also supports multicast messages (JSON only).

The XMPP implementation on the other hand can send both downstream and upstream (device–to–cloud) messages. The messages are sent asynchronously over the persistent XMPP connection and the response is also received asynchronously. The format of the response is JSON object representing acknowledgement (ACK) or negative–acknowledgement (NACK). XMPP supports only JSON message format which is encapsulated in XMPP message and it does not support multicast messages.

4.2.2 Sending messages

In order to send a message, we have to follow these four steps:

1. Application server sends a message to GCM connection servers.
2. The message is stored on GCM connection server and enqueued for further processing.
3. GCM connection server sends the message to online devices immediately or waits until they get online.
4. GCM–enabled device receives the message.

When creating the message request you have to specify the target and add some extra properties or payload to the message.

Target
This is the required part of the message. It is represented by registration ID of GCM–enabled client application.

Options
The message may contain additional options which specify the message behaviour or lifetime. Some of the options:

1. `collapse_key` – Only one message with the same `collapse_key` will be delivered when device is offline even if a server sent more of them.
2. `delay_while_idle` – This option indicates that the message should be delivered in time that the device is active.

Payload
Extra data can be sent together with a message. They should be included in parameter `data` and they are optional. Maximum size of payload is 4KB.

4.3 Receiving Messages on Client

Client application has to follow certain steps to be able to receive messages via GCM services. Firstly it needs to be registered for GCM, tell the application server about it and finally wait for incoming messages.

This section does not fully cover the client application implementation details. For further information see [6].

4.3.1 Import Google Cloud Messaging API

GoogleCloudMessaging API provides the simplest way of working with GCM on Android. It is one of Google Play Services modules and it can be easily included in Gradle based projects like this:

```
dependencies {
  compile "com.google.android.gms:play-services-gcm:7.3.0"
}
```

4.3.2 Android Manifest update

GoogleCloudMessaging API needs some permission to be able to work. This permission has to be set inside Android Manifest file.

- `android.permission.INTERNET`
- `android.permission.GET_ACCOUNTS`
- `{PACKAGE_NAME}.permission.C2D_MESSAGE`

  Where `{PACKAGE_NAME}` is the application ID.

It is the best practice to receive messages via BroadcastReceiver and pass it to Service which requires another Manifest modifications. BroadcastReceiver has to specify the permission for receiving messages `com.google.android.c2dm.permission.SEND` and it also has to provide Intent filter:

```
<intent-filter>
  <action android:name="com.google.android.c2dm.intent.RECEIVE" />
  <category android:name="{PACKAGE_NAME}" />
</intent-filter>
```

4.3.3 Registering for Google Cloud Messaging

Client application has to register itself to GCM in order to be able to receive messages. The registration ID which is returned should be sent to application server, saved there and kept in secret. When it is not possible to send the ID to server, the client application should unregister itself from GCM.

The Registration process should be repeated when the client application was updated or restored from backup.

The whole process and ID propagation is not instant and can take a couple of minutes. After it is finished the client can receive the first message.

```
GoogleCloudMessaging gcm = GoogleCloudMessaging.getInstance(context);
String registrationID = gcm.register(SENDER_ID);
```

`SENDER_ID` is the API application ID from Google API console and `context` is an instance of Context class which can be obtained from Activity.

Chapter 5

System for streaming multimedia data from Android devices

The previous chapters described important parts of the Android IP camera system from a theoretical point of view. This chapter concentrates on the design and implementation details of the resultant system, which consists of application server and Android application.

System requirements are analysed at the beginning followed by the system architecture and component implementation details. Communication, Google Cloud Messaging, libraries used and user interface design are in the second part of this section.

The project is maintained on GitHub repository[1] and I hope I will be able to implement the feature ides in near future.

5.1 Requirements

The goal is to create a system which allows a user to stream video from Android device and play this video in the web browser. The video stream should be started only when it is requested and it should be possible even if the device is hidden behind NAT. The stream will be broadcast so it is important not to request the user media from the user who wants to playback the stream. The application has to provide usable user interface and basic stream settings and password protection.

Android application has to:

- be able to get a stream from camera,
- start a stream when requested,
- have simple user interface,
- protect a stream with password,
- allow to change stream properties.

[1] Android IP camera GitHub repository: https://github.com/JanChvala/thesis-ipcam

Application server has to:

- be able to process requests for a stream,
- show the requested stream to a user,
- have simple user interface,
- set a password for the stream.

These requirements are fulfilled and implemented as described in the next sections.

5.2 System architecture design

The system architecture consists of the application server and Android application. Handling push notifications[1] also requires GCM servers (see chapter 4) to be involved and WebRTC requires signalling servers to establish direct communication between peers[2].

The whole system overview with all its components can be seen in figure 5.1.

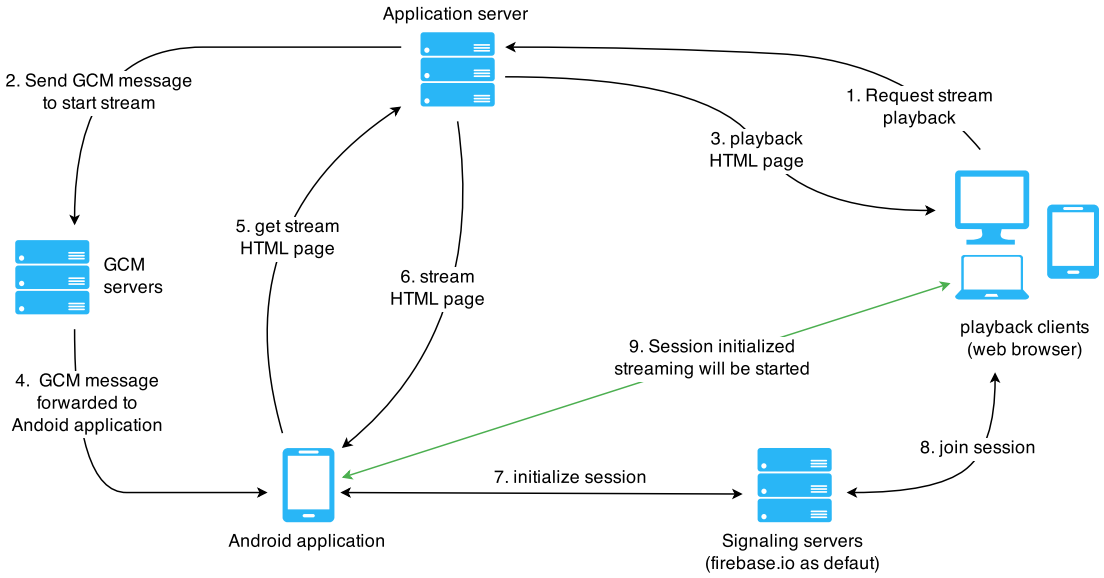


Figure 5.1: Android IP camera system architecture.

5.2.1 Application server

The application server is responsible for serving static HTML files streaming HTML page and playback HTML page, handling client application registration (using simple REST[1] API) and sending GCM messages to a client.

[1]Simple messages which are sent (pushed) from the server to the client application.
[2]Default signalling servers for RTCMultiConnection library are http://firebase.io.
[1]Representational State Transfer.

Server back–end[2] is implemented with Node.js framework and these libraries:

- **Node.js** - Platform built on top of Chrome's JavaScript runtime.
- **express**[3] - Minimalist web framework for server core functionality and routing.
- **node-gcm**[4] - Node.js library for handling Google Cloud Messaging.
- **passport**[5] - Node.js library used for HTTP request authentication.
- **mongoose**[6] - Node.js library used for working with MongoDB.

Front–end is implemented using these technologies:

- **HTML 5** - HyperText Markup Language - standard markup language for creating web pages.
- **CSS 3.0** - Cascading Style Sheets - handles the visual style of web page written in markup language.
- **AngularJS 1.4** - JavaScript web application framework.
- **RTCMultiConnection.js** - WebRTC library for JavaScript.
- **firabase.js** - JavaScript library used for WebRTC signalling.

The core of the front–end application was generated with Yeoman[7] and its gulp-angular[1] generator. These tools do more than just generating the structured base application but also provide more features such as ready–to–use development server or minification and code obfuscation for production build.

The application server implementation is important for the ability to start the stream when it is requested. It was developed using WebStorm IDE with student licence.

5.2.2 Android application

Client application for streaming the data from Android device is implemented in Java and runnable on Android version 4 (Ice Cream Sandwich) and above.

When running on Android 5 (Lollipop) and above the WebRTC is handled inside the application using native WebView component but the lack of native WebRTC support in WebView component prior to Lollipop versions puts restrictions to the resultant application. On Android 4 we use external web browser to handle the WebRTC. It is up to the user to choose which browser will handle the Intent. Chrome 42.02311 and Firefox 38.0.0 were tested as a compliant web browser capable of WebRTC.

WebRTC is crucial technology for the resultant system and cannot work without it. Using the demo application puts more restrictions and usability issues but it is working as expected.

Implementation is dependent on the following libraries:

[2] Implementation of database, REST API and sending GCM messages.
[3] express: http://expressjs.com/
[4] node-gcm: https://github.com/ToothlessGear/node-gcm
[5] passport: http://passportjs.org/
[6] mongoose: http://mongoosejs.com/
[7] Scaffolding tool for web applications. http://www.yeoman.io.
[1] The gulp-angular generator creates AngularJS base project with configured Gulp build system.

- **Google Play Services** - The important module of Google Play Services is the *play–services-gcm*[2] which provides the ability to work with GCM services.
- **AndroidAnnotations**[3] - Open–source framework for speeding–up Android application development by generating boilerplate parts using a custom Java preprocessor.
- **Support libraries** - Libraries providing back–port compatibility and additional utility classes.
- **StandOut**[4] - Library for creating floating UI elements using service.
- **Retrofit**[5] - Library used for creating REST API clients from Java interfaces.
- **Material dialogs**[6] - Implementation of Android dialogs with respect to Material Design guidelines.
- **Material**[7] - Implementation of Material Design components for Android.

Android Studio 1.2 together with Gradle 2.3 and Android SDK Build-tools 22.0.1 served as the base of the development environment.

5.3 Application server capabilities and implementation

This section describes individual parts of the application server and their implementation in more detail. The beginning focuses on the database and REST API. The end of this section describes the RTCMultiConnection JavaScript library and then it presents some parts of source code and UI of streaming and playback pages.

5.3.1 Database

There are barely any requirements for the database so it does not matter what database is used for such simple usage. We use MongoDB[1] which is NoSQL document–oriented database. The advantage is that we can work with object models. We work with *mongoose* library for *Node.js*.

Device object model definition:

```
// loading mongoose library
var mongoose = require('mongoose');

// Define our device schema
var DeviceSchema = new mongoose.Schema({
    name: String,
    code: String,
    gcmRegistrationId: String,
```

[2]Google Play Services setup: http://developer.android.com/google/play-services/setup.html.
[3]AndroidAnnotations library: http://androidannotations.org/
[4]StandOut library: https://github.com/sherpya/StandOut/
[5]Retrofit library: http://square.github.io/retrofit/
[6]Material dialogs: https://github.com/afollestad/material-dialogs
[7]Material: https://github.com/rey5137/material
[1]MongoDB: https://www.mongodb.org/

```
    lastUpdate: Date
});
```

Saving device into database:

```
// create new instance of device
var device = new Device();

// TODO: edit device properties

// Saving the device into database
device.save(function (err) {
  // handle error or do something on success
});
```

Finding device by its code:

```
Device.find({code: deviceCode}, function (err, devices) {
  // handle errors or do something with device
});
```

5.3.2 Representational State Transfer API

The application server provides simple REST API. It allows to register Android application after successful registration to Google Cloud Messaging servers. It also provides endpoint for starting registered device using Google Cloud Messaging service.

POST /api/devices

This request registers device. It expects the body of the request to be this JSON object:

```
{
  name: String,
  gcmRegistrationId: String
}
```

Server takes the property `name` and `gcmRegistrationId` and creates a new object representing the device:

```
{
  name: String,
  gcmRegistrationId: String,
  code: String,
  lastUpdate: Date
}
```

Object's property `lastUpdate` is filled with actual date of registration and value of property `code` is generated pseudo random string with eight characters.

GET /devices/:device-code/start–stream

This request is used to start streaming from device with specific `:device-code` code. When this request is invoked then the application server tries to find device by its code in the database. Then it sends a message to the device using the `gcmRegistrationId` token.

This request is invoked using jQuery[2] from within the playback page.

```
$.get("/api/devices/" + deviceCode + "/start-stream",
  function(response) {
    console.log(response);
  }
);
```

5.3.3 RTCMultiConnection

RTCMultiConnection[3] is open–sourced library developed under MIT license[4]. It is wrapping RTCPeerConnection JavaScript API (from the same developer) which handles the Peer Connection establishment and provides many features which make the development with WebRTC easier.

We use this library for WebRTC session initiation, acquiring local media stream, joining the created session and transferring the stream using Peer connection. This library also handles signalling by default[1] so we do not need to manage additional servers.

Channels and rooms

It uses a simple concept of virtual channels with rooms. A channel is the main container for rooms which are represented as sessions. There can be multiple sessions in the same channel. Each session has its initiator (the peer who opened the room) and participants (who joined the room). Channels, rooms and peers have their own unique identifiers.

```
// including library
<script
   src="//cdn.webrtc-experiment.com/RTCMultiConnection.js">
</script>

// signalling library
<script
   src="//cdn.webrtc-experiment.com/firebase.js"></script>

// joining the room with roomID this is inside script tag
var connection = new RTCMultiConnection(roomID);
```

For our purposes, the channel is unique for every device and contains only one room. Both channel and room identifiers are based on device hash which is generated on the server side. The streaming device is in the role of an initiator and the clients which play the stream are participants.

Session connection constraints

Session connection is affected by media constraints. There is one option in RTCMultiConnection library which does not occur in WebRTC specification – boolean constraint `oneway`.

[2]jQuery is JavaScript framework: https://jquery.com/
[3]Muaz Khan - RTCMultiConnection: http://www.rtcmulticonnection.org/
[4]MIT license in Open source initiative: http://opensource.org/licenses/MIT
[1]RTCMultiConnection uses Firebase servers for signalling by default: http://firebase.com.

This property allows peers that only want to playback the stream not to be prompted with local media permission because the local media is never demanded. This option is very useful in our case for broadcasting the stream.

```
// setting the session for video only
connection.session = {
  audio: false,
  video: true,
  oneway: false
};
...
// example of setting FullHD resolution
connection.media.min(1920, 1080);
connection.media.max(1920, 1080);
```

Local media resolution

Dealing with resolution can be a little tricky when working with WebRTC. There are three parameters for `getLocalMedia()` function which affect the actual quality of video that is being transmitted. There are `resolution`, `frameRate` and `bandwith`. All those attributes are the base values for codec settings.

Resolution is the most tricky one. The parameter is used only for initial settings for video source (e.g. front facing camera) to provide media with that particular resolution. This value is unchangeable in the future for the same session. A new session has to be created in case of the need for resolution to be changed. The `frameRate` is self–explaining — It is the number of frames per seconds which should be transmitted. Finally, it is the bandwidth. It is the limitation for network usage for this specific stream.

These attributes put demands on the codec which, in case of WebRTC, is the VP8[1] open–sourced codec. It tries to maximize the quality of transmitted media while keeping considerable bandwidth. This is harder for rapidly changing scenes rather than for static ones. WebRTC is trying to keep the frame rate so it sacrifices the resolution of the video during the streaming to satisfy the frame rate and bandwidth. This is noticeable and could be a limitation for some use cases when the resolution prioritizes over the frame rate.

Working with session

Both peers, the one who streams and the one who wants to play the stream, connect to the signalling channel. We could omit this for participants and simply joint the session but they would not get notified when a new session is ready and it would not be possible to distinguish whether the session exists or not. Connecting to the signal brings the ability to hook up to `onNewSession` which is fired when a participant is connected to signalling channel and the initiator opens the session.

```
// all users connect to channel
connection.connect();
...
// session initiator (Android device) opens the session
```

[1] More information about VP8 codec can be found on the WebM project page: http://www.webmproject.org/

```
connection.open({ sessionid: connection.channel });
...
// participants are informed when initiator opens the session
var sessions = {};
connection.onNewSession = function(session) {
  // if we have joined the session the skip
  if (sessions[session.sessionid] return;
  sessions[session.sessionid] = session;
  connection.join(session.sessionid);
};
```

By joining the session the connection between peers starts to be negotiated. Upon successful negotiation the Peer Connection is established.

Displaying stream

RTCMultiConnection provides useful callbacks for important events. One of these callbacks is `onstream` callback which is invoked when any stream becomes available. We can then show the stream using `video` HTML tag.

Example of inserting local media stream into HTML DOM:

```
connection.onstream = function (stream) {
  // we put the media element inside div with 'video' id
  var videos = document.querySelector('#videos');
  // automatically play the video - both stream and playback
  stream.mediaElement.autoplay = true;
  // disabling controls for streaming
  stream.mediaElement.controls = false;

  // put the element inside videos container
  videos.appendChild(stream.mediaElement);
};
```

Closing the stream when nobody is watching

Another callback used is `onleave` callback. This callback notifies one peer that the other one has left the session. It allows the application to close any previously created sessions and release camera and other resources.

Reaction on `onleave` event:

```
connection.onleave = function(e) {
  // the numberOfConnectedUsers represents the amount
  // of users in this session
  if(connection.numberOfConnectedUsers < 2) {
    // I am the only one in here. Leaving.
    closeConnection();
  }
}
```

5.3.4 Playback page

Playback page is the first place where all the streaming starts. The RTCMultiConnection is initialized and a check for existing session is made. When the streaming is not active, HTTP request to start the device is invoked[1] and the user has to wait for streaming device to open a session. When the session exists, the RTCMultiConnection connects to it. After that the session is joined and the stream data can be transmitted.

Figure 5.2 shows the sequence diagram of this process.

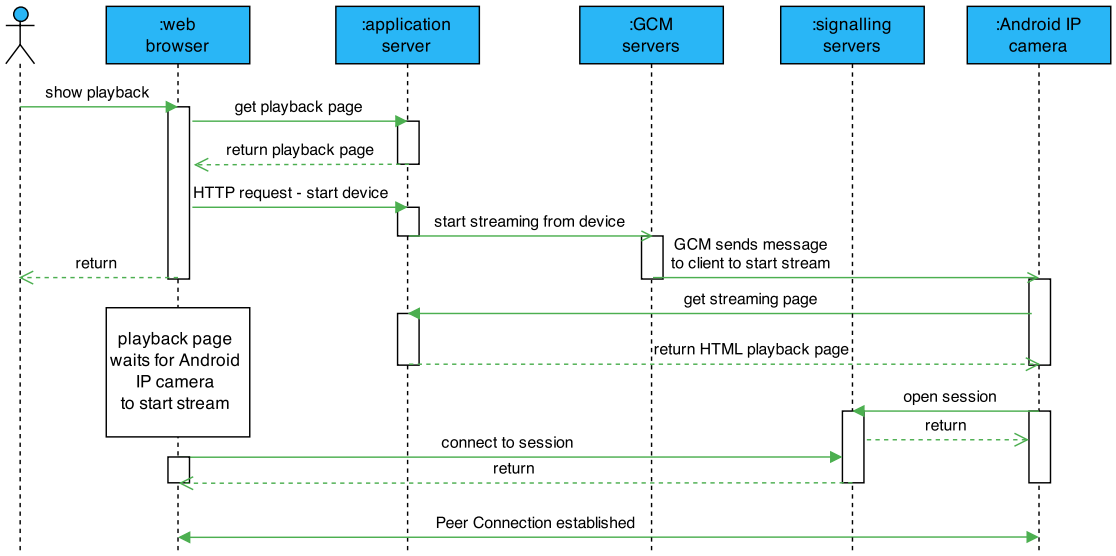


Figure 5.2: Communication sequence when requesting the stream.

Screenshot in figure 5.3 shows the user interface when the Android application starts the streaming. It is very simple and shows only maximized video stream with button to end the preview and get back to a home page of demo application[1].

5.3.5 Stream page

The streaming page is very much alike the playback one. It connects to a channel but instead of waiting for the session it simply opens a new one. We suppose that the session identifier will be unique for each device so we should not get conflict when opening session. We do not want to interact with the streaming page so the video controls are disabled and there is also no button for ending the stream.

The streaming page can be used from any device or web browser supporting WebRTC technology - it is not limited to Android.

Android application requests streaming page when it is prompted to start the stream. It opens the streaming page with proper parameters for RTCMultiConnection initialization. More information about streaming and user interface is in the section 5.4.

[1]It was empirically researched that checking for session presence takes around one to two and a half of a second. The request is send after three seconds of session inactivity.

[1]Demo application will be available on: http://ipcam.janchvala.cz/

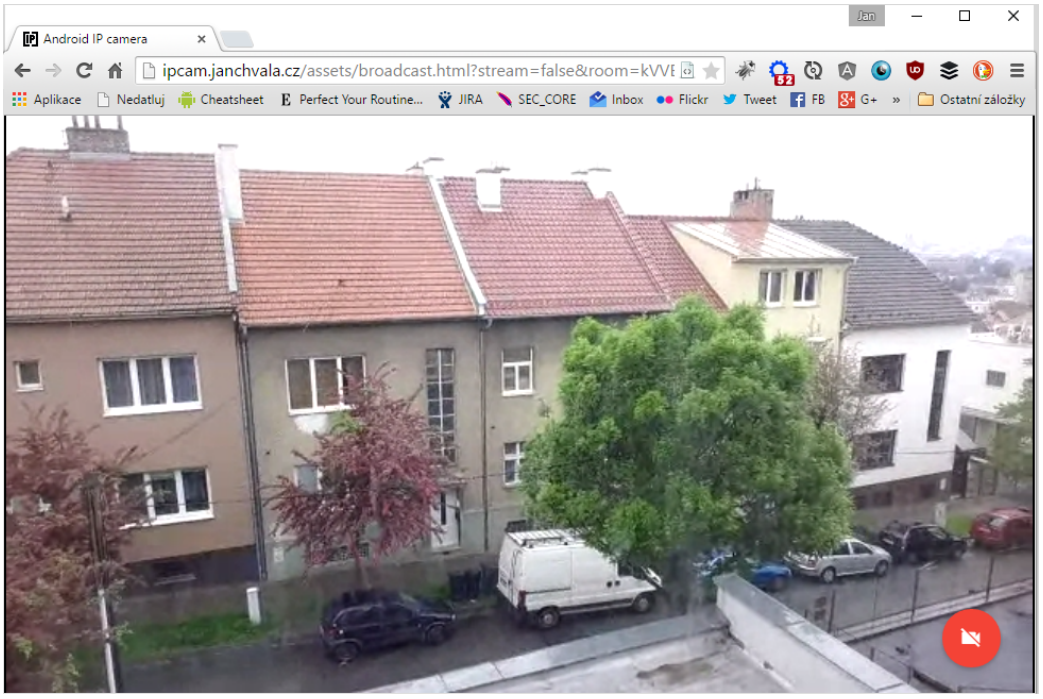


Figure 5.3: Showing user interface of playback page.

5.4 Android application implementation and design

This section covers the implementation details of Android application which is responsible for registration to Google Cloud Messaging, registration to the application server and streaming multimedia data using WebRTC technology.

First part of this section shows the registration process together with the behaviour until the application is ready to stream. It is followed by the user interface design explanation. The rest of the section is dedicated to handling the GCM messages and streaming.

5.4.1 Application life–cycle

The most important part of the application life–cycle is the registration process that is shown in figure 5.4. After first launch, the application registers to Google Cloud Messaging services, then it sends the registration information to the application server.

If this process is successful, then it is ready to receive requests to start streaming. Setting a password for stream is required and it is forced upon successful registration.

When the device is ready to stream, it waits for incoming GCM messages. They are handled by *BroadcastReceiver* and passed to *IntentService* which decides whether to start the Service[1] that is actually streaming or send Intent to show streaming page in web browser.

```
// pseudo code in Java for decision about the way of streaming
public static void startStreaming() {
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
    // Lolliopop's WebView supports WebRTC - show Service
    startStreamingService();
```

[1]The Service contains UI with WebView component and streaming page loaded into it.

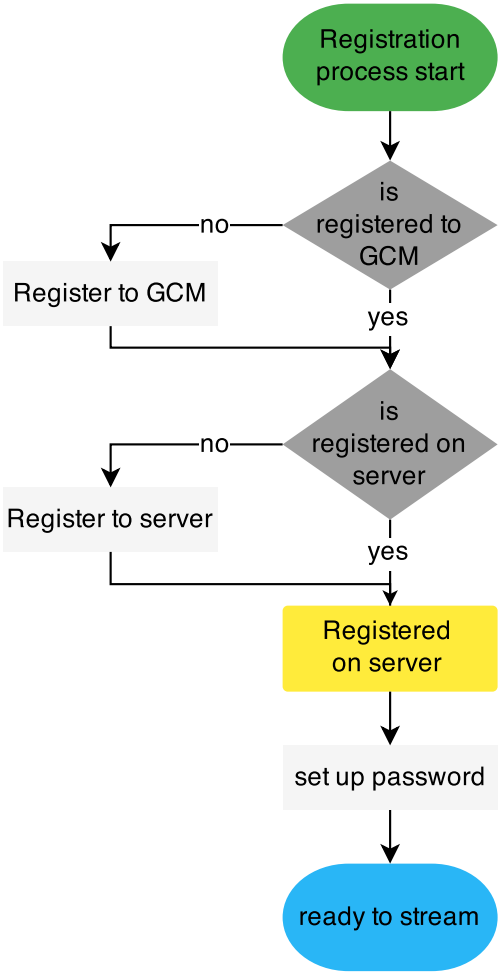


Figure 5.4: Application registration process.

```
    } else {
        // pre-lollipop devices has to show stream in browser
        sendIntentToShowStreamPageInWebBrowser();
    }
}
```

5.4.2 User interface design

The user interface is very simple. The UI design was created on the basis of Material design specification[2] and the application does not require almost any interaction with the user except setting the password which is required (but may be generated by the application).

The registration process takes place after the first launch of the application. Individual UI screens of it are shown in the figure 5.5. The registration is automatic and when the

[2]Material design specification:
http://www.google.com/design/spec/material-design/

servers are reachable, it takes only a couple of seconds until the password setting dialog is shown, which indicates that the registration was finished.

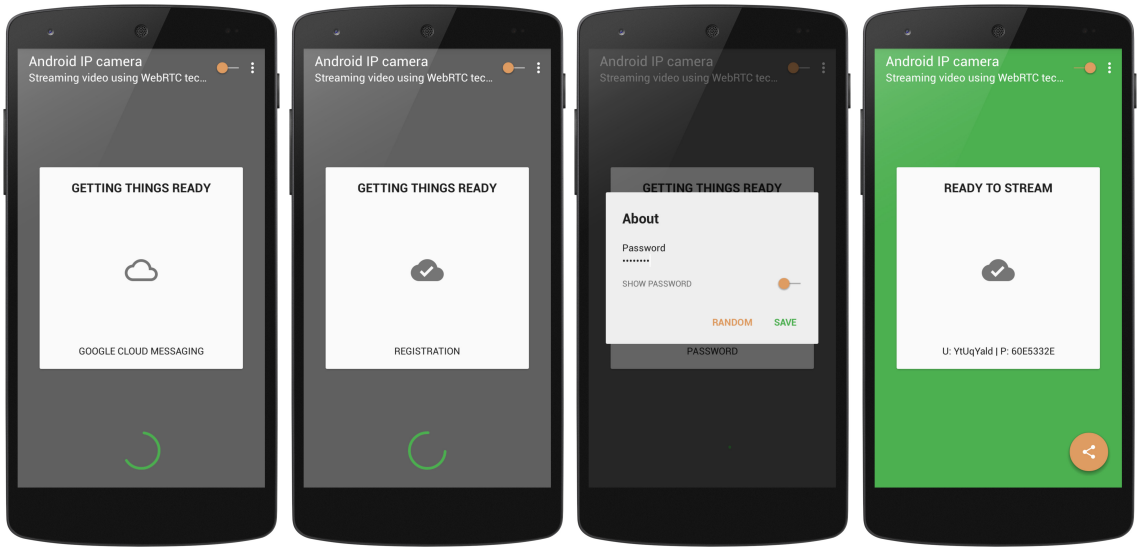


Figure 5.5: Screenshots of application's registration process. From left: GCM registration, registration to application server, password setting and ready to stream screen.

After the registration is completed the user has the possibility to enable the incoming GCM requests for the streaming. This is done with the switch located in Toolbar at the top of each screen, see the last image in figure 5.5. The state of the application is indicated in Card Fragment and also with the background colour which is indicating the state for the first sight.

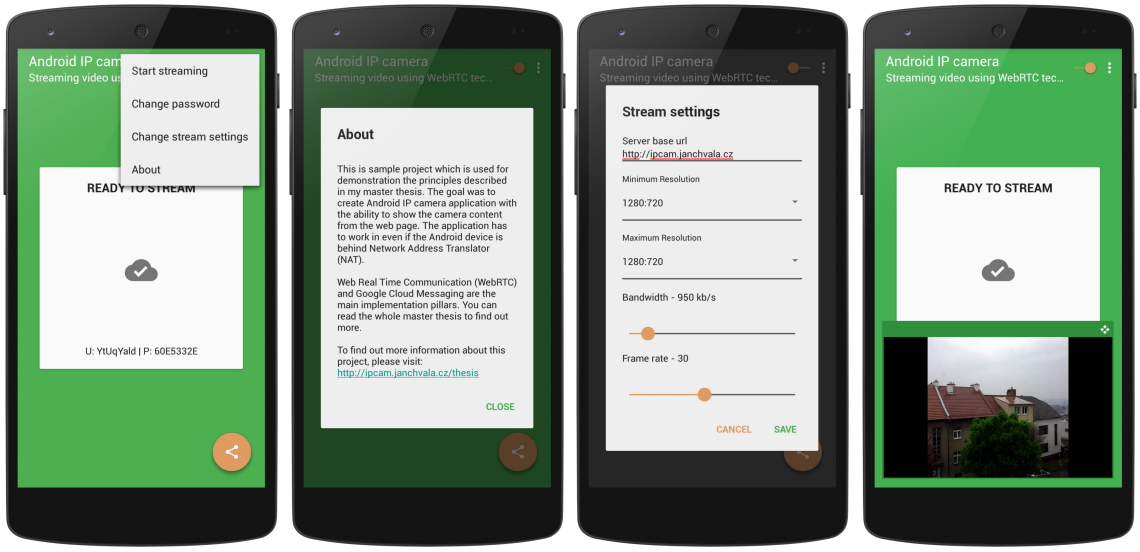


Figure 5.6: Screenshots of application during registration process and streaming. Last screenshot also shows streaming notification.

The application provides some actions for the user. Quick action to share the stream URL is accessible from FAB[1] in the left bottom corner of the screen while the others are located in the overflow menu[1]. With these actions the user can easily change the password the same way he initialized it, change stream parameters, start the streaming explicitly or show simple information about the application.

It is also possible to change the server base url which is useful if application url changes. Server registration is reset and new request for streaming code is triggered by changing this option.

Last screen shot shows the notification and also tablet layout user interface. The user has the possibility to stop the streaming or share the streaming URL via notification's buttons. The notification is shown in figure 5.7 but it is shown only on Android Lollipop when streaming with native WebView component.

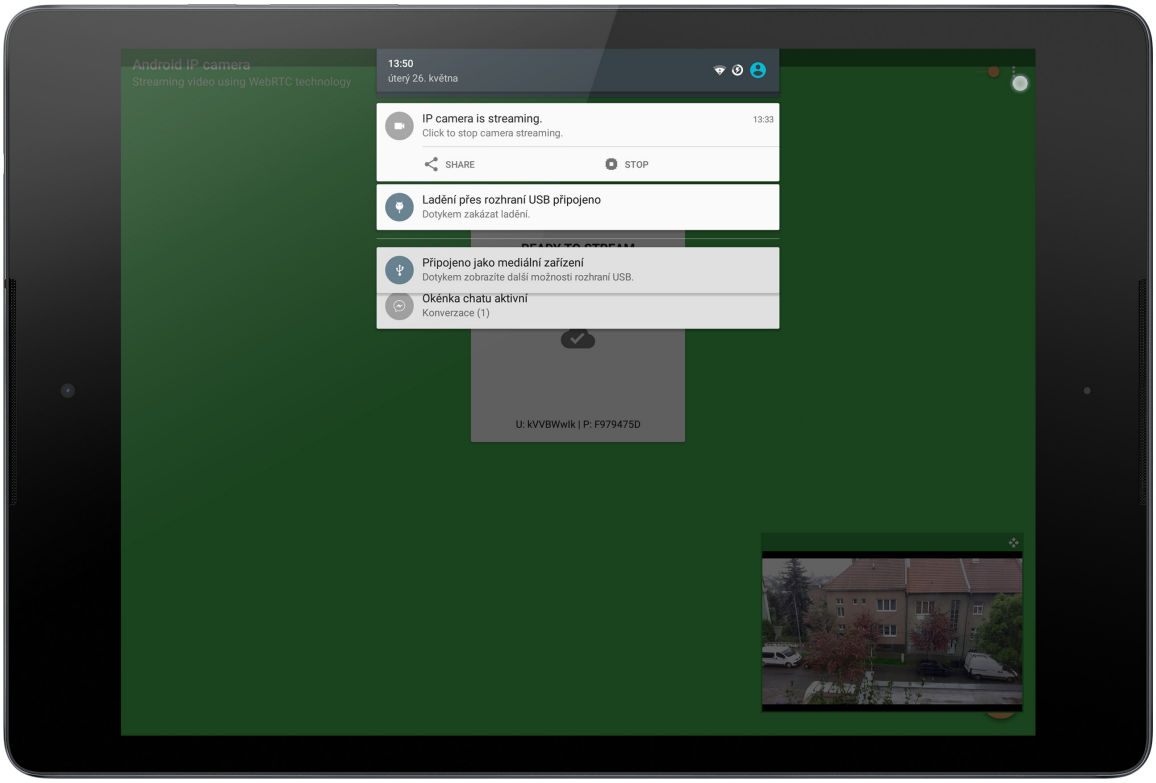


Figure 5.7: Screenshots of notification during streaming. Tablet UI.

[1]Floating Action Button – circular button introduced in Material design guidelines http://www.google.com/design/spec/components/buttons-floating-action-button.html

[1]Overflow menu is indicated with three vertically aligned dots above each other.

5.4.3 Handling Google Cloud Messages and streaming

Google Cloud Messages are processed in three steps in three different parts of the application. The message is invoked the stream is requested by the streaming page 5.3.5.

Receiving messages using BroadcastReceiver

GcmBroadcastReceiver extends *WakefulBroadcastReceiver* which is used to make device awake while receiving messages. It is registered in *AndroidManifest* (3.1.3) and it is the first part of the application which is invoked when GCM message is received.

```
@Override
public void onReceive(Context c, Intent intent) {
  // specify that GcmIntentService will handle the intent
  String serviceName = GcmIntentService_.class.getName();
  ComponentName cName =
    new ComponentName(c.getPackageName(), serviceName);
  intent.setComponent(cName);

  // Start the service and keep device awake
  startWakefulService(c, intent);

  // set the result of this receive as success
  setResultCode(Activity.RESULT_OK);
}
```

Processing messages using IntentService

After receiving the message in *BroadcastReceiver* the message is resent to *GcmIntentService* that takes proper action for the message. If it is a real GCM message (not error or deleted event) then the stream is started.

```
@Override
protected void onHandleIntent(Intent intent) {
  Bundle extras = intent.getExtras();
  if (!extras.isEmpty()) { // Initialize GCM
    GoogleCloudMessaging gcm =
       GoogleCloudMessaging.getInstance(this);

    // getting the message type
    String messageType = gcm.getMessageType(intent);
    // If this is a regular message then start stream
    String mType = GoogleCloudMessaging.MESSAGE_TYPE_MESSAGE;

    if (mType.equals(messageType)) {
      Class wClass = IpCamPreviewWindow_.class;
      Intent i = StandOutWindow.getShowIntent(this,wClass,0);
      startService(i);
    }
  }
```

```
  // Releasing the wake lock
  GcmBroadcastReceiver.completeWakefulIntent(intent);
}
```

Starting the stream Service

Stream is started using *IpCamPreviewWindow*[1]. This service creates new *Window* object which can hold UI elements and it is inflated with *WebView*. Then the *WebView* loads streaming page from application server and *WebRTC* takes care about the rest of the streaming. The code below shows the layout inflation and *WebView* initial settings.

```
@Override
public void createAndAttachView(int id, FrameLayout frame) {
  // create a new layout from body.xml
  LayoutInflater inflater = (LayoutInflater)
     getSystemService(LAYOUT_INFLATER_SERVICE);
  inflater.inflate(R.layout.ipcam_window_layout, frame, true);

  wv = (WebView) frame.findViewById(R.id.web_view_id);
  setupWebView(wv);
}

/**
 * setup WebView JavaScript, ChromeClient and load stream page
 */
protected void setupWebView(WebView wv) {
  WebSettings webSettings = wv.getSettings();
  webSettings.setJavaScriptEnabled(true);
  ...
  wv.setWebChromeClient(new WebChromeClient() {
    @Override
    public void onPermissionRequest(final PermissionRequest
       request) {
      // handling WebRTC permission request
      dispatchPermissionRequest(request);
    }
  });
  wv.loadUrl(IntentHelpers.getStreamUrl(ipCamPreferences));
}
```

To see how GCM is registered, see section 4.3.3.

[1]This window is service with specific UI elements floating above other windows in the system. This is made possible by StandOut library: https://github.com/sherpya/StandOut/

5.5 Security and authentication in demo application

The authentication and security is not crucial for demonstration of WebRTC technology but it is not desirable to allow anybody to see private streaming.

The application allows to protect device stream with a password which is set up during application first launch on Android device (it can be changed later on). This password is checked on every request from a participant to join created streaming session. The request is then accepted or rejected based on password equality.

```
// set up the password with JavaScript interface from Android
var password = Android.getPassword();

// participant requests access to session
connection.onRequest = function(e) {
  // check if the passwords equal
  if (e.extra.password !== password){
    // the passwords do not match - reject access
    connection.reject(e);
  } else { // accept the request
    connection.accept(e);
  }
};
```

There is no protection for the server REST API so anybody who can catch the device code can simply start the stream on the device. We would have to implement user management and registration on the server side to be able to protect it which is out of scope of this thesis. With that in mind the demo application is not intended to be a ready production because of the lack of server REST API protection.

5.6 The choice of WebRTC library

There were more options to choose from when deciding which library to use when working with WebRTC. A few developers try to use the WebRTC and thus they build libraries on top of this edge technology. This section will shortly describe considered libraries and the reasons why they were chosen or not.

5.6.1 PeerJS

The first option found was PeerJS[1] library. This library is very simple to use and from my point of view it is the easiest to work with. It has very simple API and it also provides the ability to use free *PeerServer* which is responsible for the signalling process.

The main problem is that this library does not support one way video streaming. This is crucial for the resultant application because it is not desirable to request permissions from the user who only wants to play the video stream.

Another problem for me concerning this library is that it seems to be stuck in development. Their master branch on GitHub repository is four months old and it does not seem to be progressing.

[1] PeerJS homepage:http://peerjs.com/

5.6.2 RTCMultiConnection v2

Another solution that was the one actually used in the resultant application was the RTC-Multiconnection[2] in its version 2.2.2. This library is well documented and allows a lot of customizations. There are also many good examples of working with this library.

The main reason for using this library is that it supports one–way multimedia streaming. This allows to request permission only from the streaming device. It also reduces the data transmission because the session does not contain any empty streams which could affect the bandwidth.

Another good reason is that Muaz Khan (The author of this library) has made it open sourced and he is very actively working on it.

5.6.3 RTCMultiConnection v3 beta

A new version of previously mentioned library was published on 1st April. I was trying to switch the resultant application to use this new version of the RTCMultiConnection library, which provides some more functionality such as build in room protection by password.

It also comes with signalling server build on top of Node.js which is the technology used for the application server. But it is still in beta and there were problems with the signalling process. The problems were making the established connection to fall apart after a couple of seconds of successful stream transmission.

There is a lot of going on around this library and we can be looking forward to seeing its improvements in the near future. I would recommend taking a look at this library for those interested in working with WebRTC.

[2]RTCMultiConnection homepage: http://www.rtcmulticonnection.org/

Chapter 6

Testing and flaws

This chapter focuses on the testing of the resultant application in real conditions. In the first part of this section there are measurements of delays and connectivity establishment. The second part points out some of the problems, which occurred during the development process.

6.1 Measuring streaming delays and stability testing

Important aspects of real–time streaming are delays. Everything takes time. Loading the page inside the browser, starting the device with GCM, acquiring local media, Peer Connection establishment and media transfer. These delays make the difference between a usable and unusable solution.

The resultant system was created to work in most cases even when the device is hidden behind NAT and to stream data only when it is needed. All the technologies that make this possible are against the speed of the process from the point where the stream is requested to the point where it is actually shown to the user.

6.1.1 Delays

The streaming application is expected to be running on local wi–fi network and to be placed stationary. In this way the streaming was tested. The requests on the other hand may be done from different networks with different capabilities. Because of that the testing requests were made from three different networks: Wi-fi, 3G and mocked 2G network[1]. For each connectivity except 2G newrok[2] there was one hundred established streams. Arithmetic mean was taken as the result for each measured value.

All the streaming was done for the best video settings:

- **resolution** - 1980 × 1020px
- **frame rate** - 30 frames per second
- **bandwidth** - 8000kb/s

[1] The 2G network was mocked by disabling 3G connectivity on mobile device.

[2] Testing of requests from 2G networks was very unstable. The connection could be established for the lowest possible streaming quality only for three times out of thirty and they were not stable. All of them broke after a couple of seconds of streaming. Therefore using the resultant application on 2G networks is not recommended.

Measured delays are:

- **stream available** - From request to media element availability. Shown in table 6.1.
- **initial video delay** - The delay between the time when stream is available and the time it is shown. Shown in table 6.2.
- **total time** - from starting the streaming page to showing the video. Shown in table 6.3.

The testing was made in two modes:

- **already running** - The stream is running and the session was established. There is no need to send GCM requests.
- **not running** - The stream has to be started with GCM messages.

Table 6.1: Tables showing average **stream available** for different networks. Top table for already running streaming and bottom for GCM started streaming.

	min (ms)	max (ms)	arithmetic mean (ms)
wi–fi	1813	4354	2181
3G	1901	4523	2390
2G	-	-	-

	min (ms)	max (ms)	arithmetic mean (ms)
wi–fi	6036	9129	6639
3G	6656	9860	7183
2G	-	-	-

Table 6.2: Table showing average **initial video delay** for different networks. Top table for already running streaming and bottom for GCM started streaming.

	min (ms)	max (ms)	arithmetic mean (ms)
wi–fi	510	7702	1538
3G	922	3113	1948
2G	-	-	-

	min (ms)	max (ms)	arithmetic mean (ms)
wi–fi	464	1771	1214
3G	1051	2196	1715
2G	-	-	-

Table 6.3: Table showing average **total time** for different networks. Top table for already running streaming and bottom for GCM started streaming.

	min (ms)	max (ms)	arithmetic mean (ms)
wi–fi	2344	10823	4321
3G	3083	6745	4838
2G	-	-	-

	min (ms)	max (ms)	arithmetic mean (ms)
wi–fi	6808	10180	7854
3G	7784	11260	8898
2G	-	-	-

Both wi–fi and 3G are quite similar in the results. The stream object that is being placed into HTML DOM was accessible after two seconds when playing active stream and approximately seven seconds when it had to be started with GCM (see table 6.1). It is due to the three second delay when checking for session presence. But it is still very good time which we can count with while requesting the stream.

Video delay, which indicates the actual time that the video is behind the reality, is more different. As you can see from table 6.2 the difference in wi-fi and 3G is four to five hundred milliseconds. This initial video delay was after a couple of seconds decreased to approximately half of a second. Using an IP camera for testing purposes with video delay under one second is usable.

The 3G network is slower in comparison with wi-fi. The difference in the total time ranges from two hundred milliseconds to one second, which is an acceptable difference. There is no problem when viewing the stream from 3G networks because if the network is not capable of streaming the video in requested quality, it simply decreases the quality, so the stream does not break. The decreasing quality on 3G networks was noticeable but the streaming quality was good enough to recognize objects and people. On the other hand small text and tiny details disappeared in a low resolution.

6.1.2 Stability

The stability of the streaming was good in both networks and the connection did not break after sixty minutes of continuous streaming from one wi-fi network to the other.

The tests also included connection failures. The first testing was done by disabling the connectivity on playback device. This completely broke the streaming session and Peer Connection, which could not be restored automatically but it does not happen very often that the networking interface is completely disabled.

Blocking the connectivity by removing networking cable for very short time made the stream to stop but is was quickly restored after the connection was recovered. This indicates that the technology is capable of overcoming short time connectivity problems but it fails in case of a serious one. To overcome them, it would be possible to implement connectivity check mechanism which would initiate session restoration after detecting that the session was disconnected.

6.2 Android and its WebRTC flaws

WebRTC technology is still not fully supported anywhere. But I found some points where Android's WebView is few steps behind the latest build of web browser. Some of the points may be relevant to RTCMultiConnection library rather than to WebRTC technology itself. All the mentioned issues were tested on WebView version 39.0.0 included in Android 5.1.1 (Lollipop).

6.2.1 RTCMultiConnection ignores media constraints

RTCMultiConnection library in version 2.2.2, which is used for working with WebRTC APIs, is forcing the media constraint to be reset for mobile devices. This constraint will probably be removed in future versions.

Mobile device is detected from *user agent* in request. To workaround this without changing the library we change the value of the *user agent* when using WebView component but it is hard to change *user agent* when using web browsers for streaming.

6.2.2 Facing mode

One of the advantages of WebRTC should be the possibility for developers to choose which camera they want to use the video from. This function is so called facing mode and it can be set–up during request for local media by the standard media constraints.

Setting this property in Android leads to a faulty stream. The local media cannot be acquired at all when the facing mode is set as a mandatory constraint. The local media is acquired successfully when setting the facing mode as optional constraint but it seems to have no effect in the used camera.

The facing mode is not working neither in native WebView version 39.0.0 nor Chrome for Android version 42.02311 nor the Firefox for Android in version 38.0.1. Firefox is trying to solve this problem by asking the user to choose the source camera for local media but it requires additional interaction with the user.

6.2.3 Media element *autoplay* attribute

HTML video tag has elements which control the initial behaviour of the video playback. Attribute `autoplay` is used to automatically start the playback when the media element is inserted into the HTML DOM. This is working as expected in web browsers but Android's policy is that the user should be aware of any network data consumption so the attribute `autoplay` is ignored on the first launch and activated after user's interaction with video controls. This causes the video stream to be immediately paused (muted) after any peer demands the data. This is not particularly useful in the case when the user has no ability to interact with the device because of remote streaming.

It happens for every new peer or local media renegotiation when using RTCMultiConnection. Not only that it breaks the streaming but invalid stream events are triggered to indicate that new stream is available.

This can be solved with JavaScript's methods that can manipulate the video. There are `play()` and `pause()` methods which can be used to restart the playback when the right events occur. The resultant application hooks up the `onstream` callback and every time the invalid stream is passed to this method we simply restart the existing stream which is the only relevant one.

Chapter 7

Conclusion

The goal was to explore technologies for streaming from Android device and to create a system which would allow the user to easily stream the video content from the camera of a device to the web.

The first part of the thesis focuses on the description of WebRTC technology which is used in the resultant system for capturing the video and the streaming transmission. Then the Android operating system is described – the target platform for the implementation. It is followed by the chapter about Google Cloud Messaging service, which is used for the streaming initiation. The chapter about the design and implementation details covers all the parts of the resultant system which are necessary for the understanding of the solution as a unit.

The created system was tested with various stream settings on different networks. It was proved that this system can be used in real conditions with good video quality and acceptable delays. Some WebRTC technological flaws were discovered during the testing and implementation phase, which indicates that the technology is still a draft and it is not yet completely finished.

The resultant system was created for the demonstration of the purpose and possibilities of WebRTC technology. It is not a complex solution with a huge amount of settings, bulletproof security and stability. However, it can be used as a starting point for creating better Android IP camera application. There are a lot of features that could improve the current solution. It would be beneficial for the user to be able to change the streaming parameters remotely from the viewing page. Recording the stream would be also very useful. Finally, the authentication would have to be done better if it should go to production.

The thesis successfully demonstrated that the WebRTC technology can be used for creating the application for multimedia streaming although it has not been completely standardized yet.

Bibliography

[1] Activities, Android Developers :: Develop :: API Guides [online]. URL: http://tinyurl.com/janchvala-android-activity. May 2015.

[2] API Guides, Android Developers :: Develop [online]. URL: http://tinyurl.com/jch-android-guide. May 2015.

[3] Bringing Interoperable Real-Time Communications to the Web [online]. URL: http://tinyurl.com/janchvala-ortc. May 2015.

[4] D. C. Burnett, A. Bergkvist, C. Jennings, A. Narayanan, Editors. Media Capture and Streams [online]. URL: http://tinyurl.com/jch-webrtc-draft. 14 Apr. 2015.

[5] Dashboards, Android Developers :: About [online]. URL: http://tinyurl.com/jch-android-dash. May 2015.

[6] Google Cloud Messaging, Android Developers :: Develop :: Google Services [online]. URL: http://tinyurl.com/janchvala-android-gcm. May 2015.

[7] Huge security flaw leaks VPN users's real IP–addresses [online]. URL: http://tinyurl.com/janchvala-webrt-ip-leak. May 2015.

[8] Alan B. Johnston. *Webrtc: apis and rtcweb protocols of the html5 real-time web, third edition*. Digital Codex LLC, March 2014. ISBN: 9780985978877.

[9] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245 (Proposed Standard). Updated by RFC 6336. Internet Engineering Task Force, April 2010. URL: http://www.ietf.org/rfc/rfc5245.txt.

[10] Services, Android Developers :: Develop :: API Guides [online]. URL: http://tinyurl.com/jch-android-services. May 2015.

[11] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational). Internet Engineering Task Force, January 2001. URL: http://www.ietf.org/rfc/rfc3022.txt.

Appendix A

DVD contents

The attached DVD contains the following items:

- **/android-ip-camera** - Source code of the client application.
- **/application-server** - Source code of the application server.
- **/doc** - Folder with the thesis LaTeX source files and generated PDF file.
- **/videos** - Demonstration videos.
- **/README.txt** - File with instructions how to run the project.