VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

OPTIMALIZOVANÝ STREAMING VIDEA NA PLATFORMĚ ANDROID

OPTIMIZED VIDEO STREAMING ON ANDROID PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. FRANTIŠEK SPURNÝ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. VLADIMÍR VESELÝ, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/20128/2016/xspurn04

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Spurný František, Bc.**
Obor: Počítačové sítě a komunikace
Téma: **Optimalizovaný streaming videa na platformě Android**
Optimized Video Streaming on Android Platform
Kategorie: Počítačové sítě

Pokyny:

1. Nastudujte problematiku streamingu videa a zjistěte jaké síťové protokoly lze na ni využít.
2. Analyzujte vhodnost použití konkrétních technologií s ohledem na co nejoptimálnější přenos videa; zaměřte se zejména na vícecestnost transportních protokolů, použití komprese a kodeků.
3. Navrhněte aplikaci pro mobilní platformu Android, která bude sloužit pro práci se soubory v cloudu a bude podporovat optimalizovaný přenos videa.
4. Navrhněte serverovou aplikaci, která bude zprostředkovávat vysílání optimalizovaného video streamu.
5. Navrženou mobilní a serverovou aplikaci podle doporučení vedoucího implementujte.
6. Proveďte ověření funkčnosti, částečně pomocí automatických testů "Espresso". Analyzujte výsledky a diskutujte další možné rozšíření aplikace.

Literatura:

- Paul Blundell and Diego Torres, Learning Android Application Testing, Pact books, 2015, ISBN 9781784395339.
- A. Ford, C. Raiciu, M. Handley, O. Bonaventure, TCP Extensions for Multipath Operation with Multiple Addresses (RFC 6824), IETF, 2013.
- R. Stewart, Stream Control Transmission Protocol (RFC 4960), IETF, 2007.
- J. Bankoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, Y. Xu, VP8 Data Format and Decoding Guide (RFC 6386), IETF, 2011.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3 včetně.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Veselý Vladimír, Ing., Ph.D.**, UIFS FIT VUT
Konzultant: König Michal, Ing., AVG
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Předmětem této práce je analýza a popis možností optimalizace streamování videa a následný návrh klient/server aplikace, která optimalizované streamování vykonává v praxi. Klientskou část představuje aplikace mobilní platformy Android, která mimo streamování poskytuje i funkcionalitu souborového „manažera“ nad adresáři a soubory uloženými v cloudu.

Abstract

The subject of this term project is the analysis and description of possibilities in video streaming optimization, and the design of a client/server application to implement optimized video streaming in practice. The client side of the application is realized with a mobile application for the Android platform, which in addition to video streaming implements the functionality of a file manager over directories and files saved in the cloud.

Klíčová slova

Android, adaptivní streamování, optimalizace, VP9, WEBM, MPEG-DASH

Keywords

Android, adaptive streaming, optimalization, VP9, WEBM, MPEG-DASH

Citace

SPURNÝ, František. *Optimalizovaný streaming videa na platformě Android*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Veselý Vladimír.

Optimalizovaný streaming videa na platformě Android

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing, Vladimíra Veselého, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
František Spurný
23. května 2017

Poděkování

Chtěl bych poděkovat Ing, Vladimírovi Veselému, Ph.D. za poskytnutí odborné pomoci při vypracování diplomové práce.

Obsah

Kapitola 1

Úvod

Sledování videa na mobilním telefonu se s ohledem na neustále a nezadržitelně rostoucí rozvoj technologií v této oblasti stalo naprosto běžnou činností značné části dnešní populace. Velkým hitem v této oblasti se stávají služby jako Youtube či Netflix, které umožňují koncovému uživateli sledovat video, které není lokálně uloženo na jeho telefonu. Tím dávají uživateli možnost mít vždy dostupné nepřeberné množství obsahu. Jediným pomyslným háčkem je nutnost být připojen k síti Internet.

Připojení k Internetu je u mobilních zařízení běžné, ovšem kvalita tohoto připojení nemusí být a není vždy ideální. Právě z tohoto předpokladu vychází zadání této práce, která se primárně zaměřuje na optimalizaci přenosu videa po síti. Díky případné optimalizaci je totiž možné snížit požadavky na kvalitu připojení, a tím uživateli zpřístupnit videa i v situacích, kdy jeho připojení není ideální. V návaznosti na zvolené téma je výstupem této práce mobilní aplikace platformy Android, která umožňuje přijímat a přehrávat video vysílané ze vzdáleného serveru, jenž je taktéž navržen a implementován v rámci této práce.

Přehrávání a příjem videa jsou ovšem pouze částí funkcionality výsledné aplikace, ta se bude dále zabývat zprostředkováním služeb cloudového uložiště společnosti Master Internet, která zaštiťuje vývoj. Uživateli bude tudíž jejím prostřednictvím umožněno provádět klasické souborové a adresářové operace nad daty uloženými mimo mobilní zařízení v cloudu.

V textu práce jsou popsány nejen použité technologie a jejich názvosloví, ale je zde i nastíněno odůvodnění jejich výběru. Velký důraz je kladen na obecné zasvěcení do problematiky a popsání problému v souvislostech.

Kapitola 2 se zabývá úvodem do problematiky streamování a popisuje základní terminologii a principy.

Následující kapitola 3 dekomponuje problematiku streamování na jednotlivé součásti. Pro každou součást jsou uvedeny a popsány nejpoužívanější technologie, které jsou případně následně porovnávány. Z jejich výčtu je pak zdůvodněn výběr konkrétních technologií. Krom samotných technologií jsou popsány i některé obecné principy a postupy, které jsou na konkrétních implementacích nezávislé.

Kapitola 4 popisuje funkcionalitu a technologie umožňující samotný přenos videa po síti od serveru až k cílovému uživateli.

Návrh nejen aplikace, ale i serverů je popsán v kapitole 5. Ta vychází z teoretických znalostí popsaných v kapitolách předcházejících a má relativně široký záběr. Nastiňuje jak návrh architektury serverů, tak například grafické zpracování aplikace a mnohé další.

Následuje dvojice kapitol zabývajících se popisem samotné implementace, a to jak mobilní aplikace, tak streamovacího serveru. Tyto kapitoly obsahují nejen ukázky kódu, ale hlavně se snaží nastínit myšlenky v „pozadí“ jednotlivých implementačních řešení.

Kapitola 8 pojednává o použitých způsobech testování mobilní aplikace. Je popsán nejen automatický, ale také uživatelský neboli manuální přístup k testování a jsou zde zhodnoceny jejich vlastnosti.

Kapitola 2

Streamování

Pod pojmem streamování v oblasti informačních technologií rozumíme situaci, ve které uživatel sleduje digitální video, respektive poslouchá digitální zvukovou stopu za pomocí počítačové sítě. Streamování dává uživateli možnost sledovat/poslouchat požadované video/zvuk bez nutnosti ho celé v předstihu ze sítě stáhnout.

2.1 Streamování videa

Streamované video, je takové video, které je konstantně prezentováno a odesíláno koncovému uživateli nějakým poskytovatelem. Alternativním přístupem k streamování, je stažení videa a jeho následné přehrávání. Tyto dva přístupy mají své klady a zápory, které si nyní nastíníme:

- *Streamované video:*
  - Uživatel může začít sledovat video, aniž by musel přenést velký objem dat (pouze úvodní inicializace přenosu).
  - Pokud uživatel chce vidět pouze část videa, bude mu poskytnuta prostřednictvím streamování pouze tato část, což v praxi znamená znatelnou úsporu s ohledem na přenos dat.
  - Kvalita a plynulost streamovaného videa je úzce závislá na kvalitě síťového připojení na cestě k poskytovateli obsahu.
  - Streamování videa vyžaduje přidanou režii nutnou pro synchronizaci poskytovatele a přehrávače videa na straně uživatele.
- *Stažené video:*
  - Hlavní nevýhodou je to, že video je nutné stáhnout před samotným sledováním. To může s ohledem na kvalitu aktuálního síťového připojení způsobit vznik velké časové prodlevy mezi tím, než uživatel o video poskytovatele požádá, a okamžikem kdy ho může sledovat.
  - Kvalita videa a plynulost jeho přehrávání nejsou přímo ovlivněny kvalitou síťového připojení (limitací může být výkonnost hardware a možnosti použitého přehrávače).
  - Při stahování videa není potřebná další přidaná režie. Jedná se o standardní stahování jako u jiných libovolných dat.

Aplikace vytvářená v rámci této práce poskytuje uživateli oba následující přístupy k videu uloženému v cloudovém uložišti. Ovšem zásadní důraz bude kladen právě na streamování videa a jeho optimalizaci, jelikož s ohledem na mobilní platformu a její specifika, klady vysoce převažují zmíněné zápory. Což dokazuje i popularita tohoto přístupu v posledních letech (viz oblíbenost služeb jako Youtube, Netflix, aj.).

2.1.1 Specifika mobilní platformy

Mobilní platforma má svá zásadní specifika, kterými se liší od platformy stolních počítačů. Nejvýraznější z nich budou uvedeny v následujícím výčtu:

- Velikost displejů mobilních zařízení je typicky řádově menší než u monitorů pro stolní počítače.
- Rozdílné bývá i rozlišení, ale toto specifikum bylo postupem času téměř anulováno, protože v dnešní době není žádnou výjimkou mobilní zařízení s rozlišením displeje Full HD [43].
- Připojení k síti je realizováno pomocí bezdrátové komunikace, a to typicky pomocí technologie Wi-Fi a nebo mobilního připojení.

Velikost displeje a rozlišení bude mít typicky vliv pouze na to jakou *maximální* kvalitu videa je vhodné streamovat na koncové zařízení. Jednoduchým příkladem může být zbytečnost streamovat video Full HD na telefon s HD [43] rozlišením displeje.

Na to, v jaké kvalitě a s jakou plynulostí bude video streamováno, má tedy zásadní vliv kvalita a typ připojení k síti. Z tohoto důvodu je nutné snížit co nejvíce paměťové nároky videa při zachování přijatelné kvality, respektive rozlišení, a to za pomoci *komprese*. Video lze ovšem komprimovat jen do určité míry, tudíž je nutné, aby kvalita a rozlišení videa reflektovalo kvalitu připojení.

O tom, jakým způsobem lze video adaptovat na „míru“ danému mobilnímu zařízení a kvalitě jeho aktuálního připojení k Internetu, bude velkým tématem v následujícím textu.

2.1.2 Živé streamování / streamování uloženého obsahu

Pro úplnost je ještě nutné streamování rozdělit na dvě základní kategorie a to takzvané „živé“ streamování a streamování uloženého obsahu. Při „živém“ streamování poskytovatel uživateli zprostředkovává video události, která právě probíhá. To znamená, že na jednom konci je zařízení pro přenos videa, které v reálném čase zasílá skrze poskytovatele tento obsah ke koncovému uživateli, respektive uživatelům.

V rámci této práce se ovšem omezíme na streaming již u poskytovatele uloženého obsahu, což přesně odpovídá definici výsledné aplikace, která umožňuje zprostředkovávat obsah uložený v cloudu.

Kapitola 3

Protokoly

Streamované video a sním spojenou zvukovou stopu je nejprve nutné komprimovat, a to za pomoci jednoho z video, respektive audio, kódovacích formátů. Dále jsou video a zvuková stopa „zabaleny“ v „bitstream“ kontejneru a následně přenášeny mezi poskytovatelem a koncovým uživatelem za pomoci transportního protokolu [17]. O jednotlivých technologiích, respektive protokolech, umožňujících uskutečnit tento proces a jejich alternativách budeme hovořit v následujícím textu.

3.1 Video kódovací formáty

Jedná se o formáty určené pro ukládání a přenos digitálního videa. Kromě schopnosti video uchovávat je jejich hlavním cílem video komprimovat neboli snížit jeho velikost při zachování pokud možno maximální možné kvality. Kódovací formáty typicky dělíme na dvě základní kategorie, a to na bezeztrátové a ztrátové.

U bezeztrátových formátů nedochází ke ztrátě žádné obrazové informace, jinak řečeno, nedochází k snižování kvality videa, což je velice pozitivní vlastnost, která si ovšem vybírá svou daň, a to v podobě velice malého kompresního poměru těchto formátů. Typický kompresní poměru u bezeztrátových formátů dosahuje hodnot kolem $1:2$ ([45] strana 38).

Tento kompresní poměr je nedostatečný i vzhledem k výše uvedeným specifikům mobilní platformy. Dále se tudíž budeme bavit výhradně o ztrátových formátech. V závislosti na použitém algoritmu a míře komprese u nich dochází v určité míře ke ztrátě obrazových informací, ale díky tomu dosahují řádově vyšších kompresních poměrů. Ztrátové formáty vychází z předpokladu, že lidské oko není dokonalé a určité obrazové nedostatky ani nepostřehne. Tudíž i přes to, že dochází ke ztrátě dat, může video na uživatele působit co se týče obrazové kvality stejně, nebo alespoň velice srovnatelně, jako video komprimované bezeztrátovým formátem.

3.1.1 Nutnost komprese

Video je pouze posloupností obrazů (dále pouze jen jako „*rámce*“), které jsou zobrazeny za sebou s dostatečnou frekvencí (minimální hodnota se typicky uvádí 24 rámců za sekundu) tak, aby vytvořily iluzi pohybu. V následujícím příkladu budeme zvažovat přenos videa s následujícími parametry:

- Kvalita videa bude Full HD to znamená, že každý rámec bude mít rozlišení 1920×1080 pixelů.

- Snímková frekvence bude 30 snímku za sekundu.
- Barevná hloubka rámců bude 10 bitů.

Pro plynulé streamování následujícího videa by byla potřebná šířka pásma B_{min}:

$$B_{min} = 1920 \cdot 1080 \cdot 30 \cdot 10 = 622.08 Mb/s = 77,76 MB/s$$

Tyto požadavky na šířku přenosového pásma jsou naprosto nepřijatelné, a to nejen vzhledem k šířkám přenosových pásem jednotlivých mobilních technologií pro připojení k Internetu, ale také z důvodu, že drtivá většina uživatelů má omezený celkový datový přenos a rychlost připojení od svého mobilního operátora (*Fair usage policy (FUP)*).

3.1.2 Kódování/Dekódování

Kódovaní (komprese) a dekódování (dekomprese) jsou dvě základní operace, o kterých se v následujícím textu budeme bavit.

Kódováním rozumíme proces převodu zdrojových dat videa na data zakódovaná za pomoci video kódovacího formátu. Tato operace bývá typicky značně časově náročná, ale není to velká komplikace, protože kódovaní se provádí pouze jednou pro každý video soubor a je prováděno na straně serveru, a nikoliv u uživatele.

Opačnou operací je dekódování, při kterém dochází ke zpětnému převodu zakódovaného videa a výsledná data jsou v podobě bitového proudu poskytnuta přehrávači, který je prezentuje cílovému uživateli. Tato operace je, co se týče efektivity streamování a přehrávání videa na mobilních zařízeních, klíčová, a to z důvodu, že je prováděna pokaždé, když uživatel video přehrává. Důležité tedy je, aby byla operace dekódování co nejlépe optimalizovaná tak, aby nebránila plynulému přehrávání přijímaného videa.

3.1.3 Součásti video sekvence

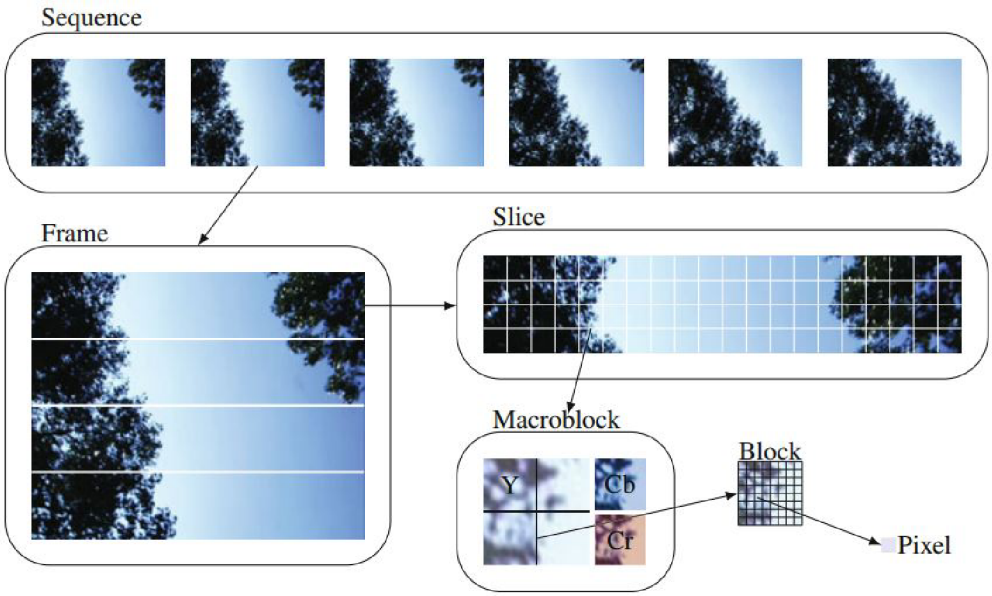
Pokud kodér/dekodér označíme jako blokově-orientovaný [29] znamená to, že každý rámec je místo na jednotlivé pixely rozdělen na bloky o pevné či proměnlivé velikosti, které nazýváme makrobloky. Při kódování jsou tedy veškeré výpočty prováděny na makroblocích, což umožňuje razantní snížení výpočetní náročnosti.

Na obrázku 3.1 jsou zobrazeny tyto základní součásti video sekvence:

- *Sekvence* – určitý počet po sobě jdoucích rámců může být seskupen jako nezávislá sekvence neboli GOP (group of pictures).
- *Rámec* – jeden obraz videa nazýváme rámcem.
- *Plátek* – každý rámec je rozdělen na několik plátků.
- *Makroblok* – makroblok obsahuje dílčí bloky s různými významy a jedná se o základní jednotku rozdělení rámce.

3.1.4 H.264 (AVC – Advanced Video Coding)

Prvním zástupcem video kódovacích formátů, které si představíme, je H.264. Jedná se v dnešní době o nejrozšířenější video kódovací formát. Je blokově orientovaný a v literatuře bývá často označován jako AVC – Advanced Video Coding nebo též jako MPEG-4.



Obrázek 3.1: Ukázka dělení video sekvence. [29]

Tento formát je standardizován [13] a chráněn řadou patentů [2], což znamená, že při komerčním využití je nutné platit organizaci MPEG LA a dalším vlastníkům patentů. Ovšem na videa streamovaná po síti Internet, která jsou zdarma k dispozici pro cílového uživatele, se tato povinnost nevztahuje, což je právě případ využití pro účely této práce.

Mezi jeho klíčové vlastnosti [46] a využité technologie patří:

- Efektivní zpracování prokládaného videa kódováním prokládaných polí jako samostatných obrazů a nebo kódování každého 16×32 pixelového regionu.
- Prostorová mezi-rámcová predikce s volitelnou velikostí bloku a směrovým filtrováním.
- Vylepšené pohybově-kompenzační mezi-rámcové předpovědní techniky.
- Podpora monochromatického obrazu, ale i jiných možností chromatického podvzorkování ([18] kapitola 3.2) například 4:2:0, 4:2:2 a 4:4:4.
- Podpora barevné hloubky rámců v rozmezí od $8b$ až po $14b$.
- Podporuje celou škálu rozlišení od hodnot 128×96 pixelů až po 4096×2304 pixelů.

3.1.5 H.265 (HEVC – High Efficiency Video Coding)

H.265 je blokově orientovaným video kompresním standardem [27] a jedná se o následníka H.264. Často bývá označován také jako *HEVC – High Efficiency Video Coding*. Oproti svému předchůdci dosahuje H.265 citelně vyššího kompresního poměru při zachování srovnatelné obrazové kvality.

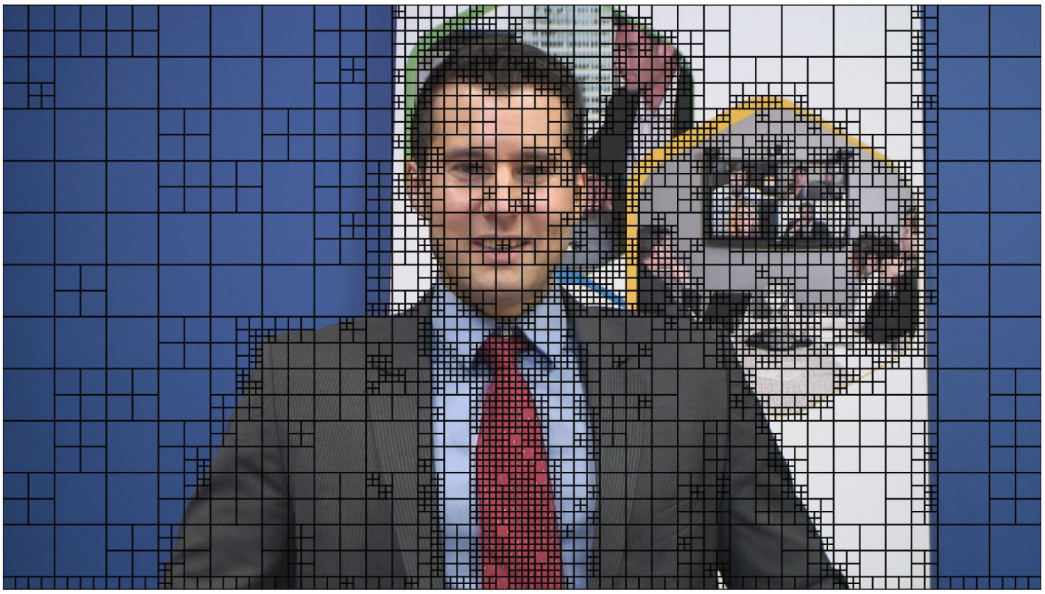
Ovšem jeho velkou nevýhodu, a pro potřeby cílové aplikace nevýhodu kritickou, je situace kolem jeho patentů. H.265 je chráněn řadou patentů [6] [10], které jsou vlastněny celou řadou společností a organizací. Poplatky za využití patentovaných částí kodeku jsou

velmi vysoké, a tudíž i přes svou nepochybnou kvalitu nelze tento video kódovací formát využít pro účely této práce.

3.1.6 VP9

VP9 je taktéž blokově orientovaný video kompresní formát, který je vyvíjen jako přímý konkurent a „soupeř“ pro formát H.265. Na rozdíl od něj se ovšem jedná o otevřený „royalty-free“ formát, což znamená, že jej lze používat zcela zdarma, a to i pro komerční účely.

Tento formát je vyvíjen firmou Google, což znamená, že má velmi silnou podporu [9] v operačním systému Android, který je též vyvíjen touto společností. Mezi jeho hlavní vlastnosti [26] a silné stránky patří:



Obrázek 3.2: Ukázka dělení rámce na bloky o proměnlivé velikosti.

- Podporuje rozdělení na bloky o proměnlivé velikosti (viz obr. 3.2), přičemž rozdělení probíhá následovně:
  - Rámec je rozdělen na bloky o velikosti 64×64 pixelů, které nazýváme super bloky.
  - Super bloky pak mohou být dále děleny na bloky menší, až na minimální velikost 4×4 pixely. Toto dělení probíhá za pomoci datové struktury nazývané *rekurzivní quadtree* [39].
- Podporuje asymetrickou diskrétní sinus transformaci.
- Má tři rozdílné a přepnutelné subpixelové interpolační filtry.
- Má oproti svému předchůdci VP8 výrazně vylepšené následující technologie:
  - kódování entropie,
  - kódování a odsazení pohybových vektorů vůči jejich referenčním rámcům,
  - osmi-pixelová preciznost pohybových vektorů.

3.2 Porovnání video kodeků

Výše jsme si stručně popsali nejpoužívanější video kódovací formáty, ty ovšem přímo porovnávat nebudeme. Porovnání podrobíme jejich konkrétní softwarové implementace, které nazýváme *video kodeky*. Kodek je tedy software, který umožňuje video jak kódovat (komprimovat), tak i dekódovat (dekomprimovat).

Při porovnávání video kodeků se omezíme na tři klíčové vlastnosti: kvalita v závislosti na bitrate, rychlost kódování a rychlost dekódování videa.

3.2.1 Video kvalita v závislosti na bitrate

To, jakou kvalitu bude mít zakódované video při určité hodnotě bitrate[1], je typicky považováno za naprosto klíčovou a rozhodující vlastnost. Postupy pro porovnání kvality videa dělíme následovně:

- *Objektivní kvalita videa* [35] Postupy pro objektivní vyhodnocení kvality videa využívají matematické modely, které se snaží předvídat lidský pohled na kvalitu daného snímku videa. Typicky hodnotí za pomocí metrik a kritérií, která lze měřit objektivně což znamená, že jsou vyjádřitelná matematicky a existuje možnost je automaticky vyhodnocovat za pomoci počítačového programu.
- *Subjektivní kvalita videa* [34] Jak již název napovídá, zde je kvalita videa posuzována zcela subjektivně, to znamená, že uživatel, respektive skupina uživatelů, za pomoci určitého hodnotícího systému porovnává kvalitu dvou či více videí zakódovaných různými kodeky.

V dalším textu budou prezentovány výsledky a nastíněn postup objektivního porovnání následujících video kodeků:

- Pro formát *HEVC (H.265)* byl zvolen referenční kodek, který budeme dále označovat jako *HEVC*.
- Taktéž pro formát *VP9* byl zvolen referenční kodek, který budeme dále označovat jako *VP9*.
- Formát *H.264* byl v prezentovaném testu kódován za pomoci nejpoužívanějšího „open-source“ kodeku *x264*.

Při porovnávání byla použita metoda pro objektivní porovnání kvality videa, a to konkrétně PSNR.

3.2.2 PSNR (Peak signal-to-noise ratio)

PSNR [47] neboli „špičkový poměr signálu k šumu“ vyjadřuje poměr mezi maximální silou signálu a šumem, který ovlivňuje přesnost jeho reprezentace. Signálem v kontextu porovnávání kvality videa rozumíme původní nekomprimované video a šum vyjadřuje chyby, které do původního signálu „zanese“ komprese. Hlavní výhodou PSNR je, že se snaží aproximovat lidský pohled a vnímání kvality videa.

[1] Udává počet bitů nutný pro uchování určité délky zakódovaného videa. Typicky bývá uváděn jako počet bitů nutných pro uchování jedné vteřiny, to znamená, že základní jednotkou je bit/s.

PSNR je definováno pomocí *střední kvadratické chyby*, která je pro nekomprimovaný monochromatický snímek videa I a pro odpovídající komprimovaný monochromatický snímek videa J, oba o rozměrech $m \times n$ bodů, vyjádřena následujícím vztahem:

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i,j) - K(i,j)]^2$$

Poté definujeme PSNR jako:

$$\begin{aligned} PSNR &= 10 \cdot log_{10} \left(\frac{MAX_I^2}{MSE} \right) \\ &= 20 \cdot log_{10} \left(\frac{MAX_I}{\sqrt{MSE}} \right) \\ &= 20 \cdot log_{10}(MAX_I) - 10 \cdot log_{10}(MSE) \end{aligned}$$

Kde MAX_I vyjadřuje maximální hodnotu pixelu snímku, tedy pokud máme osmi bitovou barevnou hloubku $MAX_I = 255$. Obecně tedy platí, že při barevné hloubce B bitů platí $MAX_I = 2^B - 1$.

V případě prezentovaného testu byl u videí využit barevný prostor YUV, tudíž výpočet probíhal tak, že byl obraz rozdělen na jednotlivé kanály barevného prostoru. V případě YUV na jasovou složku Y a barevné složky U a V. A je PSNR vypočteno pro každou složku samostatně.

3.2.3 Výsledky a parametry testu

Pro porovnání byla použita následující trojice video souborů:

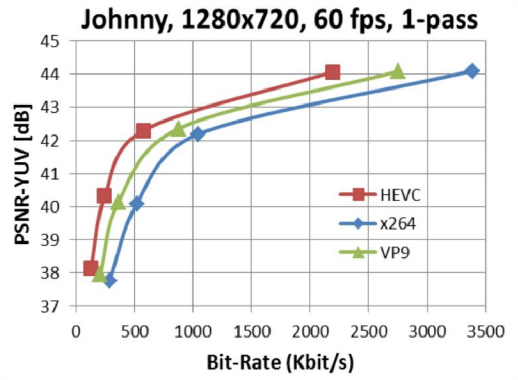
Jméno video sekvence	**Rozlišení**	**Snímková frekvence**
FourPeople	1280x720px	60fps
Johny	1280x720px	60fps
KristenAndSara	1280x720px	60fps

Tabulka 3.1: Seznam porovnávaných video souborů.

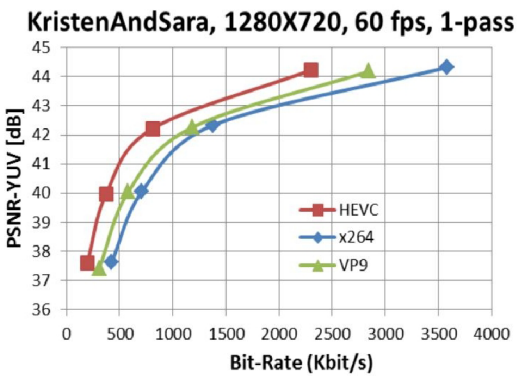
Před samotným představením a zhodnocením výsledků porovnání je vhodné zmínit, že testy probíhaly na procesoru Intel Core i5 CPU, 2,4GHz a k dispozici byla operační paměť o velikosti 4 GB RAM. Veškeré další detaily a metodologii měření lze najít v citovaném materiálu (viz [21]).

Následující grafy ukazují výsledky porovnání kvality zdrojových video souborů a video souborů co vznikly po jedno-průchodovém zakódování za pomoci kodeků HEVC, VP9 a x264.

Z výše uvedených licenčních důvodů (viz 3.1.5) video kompresní formát HEVC (H.265), respektive jeho kodek HEVC, při dalším rozboru výsledků nebudeme zvažovat i přes jeho nepopiratelnou kvalitu. Z grafů lze snadno vyčíst, že v závislosti na bitrate dosahuje kvalita videa vyjádřená pomocí síly signálu v dB velmi srovnatelných hodnot u dvojice zbývajících zvažovaných kodeků x264 a VP9. I přesto, po sumarizaci výsledků testů nám vychází, že kodek VP9 potřebuje v průměru o $12,5\%$ nižší bitrate pro zachování stejné kvality (porovnání pomocí PSNR).



Obrázek 3.3: Výsledný graf testu kvality na video souboru „Johny“.



Obrázek 3.4: Výsledný graf testu kvality na video souboru „KristenAndSara“.

	VP9 vs. x264 [%]			
Sequences / PRNS_YUV [dB]	**22**	**27**	**32**	**37**
FourPeople	13506	16438	17557	18480
Johny	9945	11791	13082	13869
KristenAndSara	11018	12717	12996	13759
Průměr	11489	13648	14545	15369
Celkový průměr	**13762**			

Tabulka 3.2: Porovnání rychlosti zakódování videí s odpovídajícím PSNR$_Y UV$

3.2.4 Rychlost kódování videa

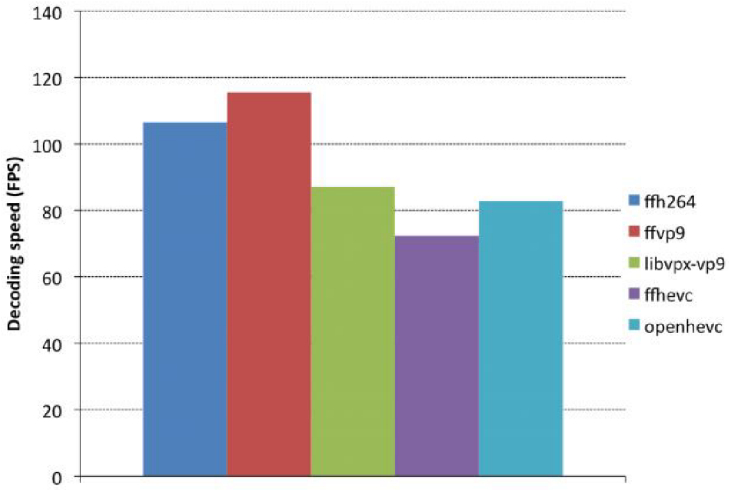
Rychlost kódování není kritická, tak jako kvalita videa, ale je vhodné ji zvážit. Kódování videa bude probíhat na serveru a jeho kratší trvání sníží dobu, po kterou bude server vytížen. Při vyšším počtu videí, které bude nutné zakódovat, může tedy rychlost a efektivita kódování videa mít zásadní vliv na vytížení serveru.

Jak ukazuje tabulka 3.2 rozdíl v rychlosti kódování mezi x264 a VP9 je diametrální. V presentovaném testu [20] byla doba kódování H.264 za pomocí x264 průměrně 130× kratší než doba potřebná pro zakódování odpovídajícího videa s kodekem VP9.

3.2.5 Rychlost dekódování videa

U porovnání kvality videa a měření rychlosti kódování, jsme se zatím bavili pouze o kodérech videa. Nyní se podíváme na rychlost nejpoužívanějších dekodérů videa. Pro dekódování formátu H.264 byl u prezentovaného testu využit kodek *ffh264* a pro formát VP9 bude porovnáván standardní kodek *libvpx-vp9* a kodek *ffvp9*.

Pro video soubory, jejichž kvalita dosahovala srovnatelných hodnot PSNR, dekódované za pomocí výše uvedených kodeků byly naměřeny velmi srovnatelné hodnoty. Zajímavým výsledkem ovšem je, že při využití kodeku *ffvp9* bylo dekódování videa ve formátu VP9 více jak o 5% rychlejší než *ffh264* dekódující H.264.



Obrázek 3.5: Graf zobrazující výsledky testu porovnávajícího rychlost dekódování.

3.2.6 Volba video kódovacího formátu

Na základě výše uvedených porovnání a vlastností byl pro účely této práce uznán jako vhodný formát *VP9*. Tyto hlavní argumenty, respektive klíčové vlastnosti rozhodly o jeho výběru:

- Jedná se o otevřený software, který není „svázán“ patenty a licencemi.
- Dosahuje velmi dobrých poměrů kvality videa k hodnotám bitrate.
- I přesto, že se jedná o formát nové generace, dosahuje přijatelných rychlostí při dekódování.
- Má silnou podporu v rámci operačního systému Android.

3.2.7 VP9 podrobněji

Tento blokově orientovaný video formát je nástupcem formátu VP8 [16] a je vyvíjen společností Google. To znamená, že má pochopitelně silnou podporu na platformě Android, ale je též hojně využíván na webu. Tato skutečnost je velmi příznivá, protože paralelně s mobilní aplikací vyvíjenou v rámci této práce vzniká i aplikace webová, která musí streamování videa podporovat taktéž.

VP9 umožňuje reprezentovat videa s velikou škálou rozlišení dokonce i vyšších jak definice *ultra-high-definition video (UHD)* a podporuje následující barevné prostory Rec. 601, Rec. 709, Rec. 2020, SMPTE-170, SMPTE-240, a sRGB.

Profil	Barevná hloubka	Podvzorkování barvonosných složek
0	8 bit	4:2:0
1	8 bit	4:2:2, 4:4:4
2	10 nebo 12 bitů	4:2:0
3	10 nebo 12 bitů	4:2:2, 4:4:4

Tabulka 3.3: Profily video kódovacího formátu VP9.

Obsahuje předdefinované kódovací profily. Jedná se o sady možností, respektive nastavení, které jsou určeny pro různé „třídy“ aplikací a využití. Jak je vidět v tabulce 3.3, k dispozici máme celou škálu profilů od základního profilu 0 (minimální nastavení pro hardwarové implementace) až po pokročilý profil 3.

S profily úzce souvisí další pojem a tím jsou *úrovně (levels)*. Úrovně specifikují sadu určitých pravidel a omezení, která určují jakousi výkonnost dekodéru pro daný profil. Formát VP9 definuje 14 základních úrovní, a to pro profily 0 (8 bit) a 2 (10 bit).

Úroveň	**Maximální bitrate (1000 bits/s)**	**Maximální velikost CPB (1000 bits)**	**Minimální kompresní poměr**	**Velikost snímku @ Počet snímků za sekundu**
1	200	400	2	$256 \times 144@15$
1.1	800	1000	2	$384 \times 192@30$
2	1800	1500	2	$480 \times 256@30$
2.1	3600	2800	2	$640 \times 384@30$
3	7200	6000	2	$1080 \times 512@30$
3.1	12000	10000	2	$1280 \times 768@30$
4	18000	16000	4	$2048 \times 1088@30$
4.1	30000	18000	4	$2048 \times 1088@60$

Tabulka 3.4: Úrovně video kódovacího formátu VP9.

V tabulce 3.4 jsou uvedeny všechny, pro naše potřeby využitelné úrovně a základní parametry. Pro úplnost je ještě nutné popsat význam parametru *maximální velikosti CPB*, ten určuje maximální datovou velikost čtyř po sobě jdoucích snímků.

VP9 má stále rostoucí hardware podporu přímo v mobilních zařízeních. Například u současného[2] nejprodávanějšího Android zařízení Samsung Galaxy S7 lze najít plnou hardware podporu pro dekódování videa.

3.3 Audio kódovací formáty

Audio kódovací formáty slouží pro uchování a přenos digitálního zvuku. Formátů je široká škála a můžeme je dělit na formáty kompresní/nekompresní, respektive ztrátové/bezztrátové. Dále se budeme bavit pouze o formátech ztrátových kompresí. I přes to, že digitální zvuková stopa není ani zdaleka tak paměťově náročná na uchování, respektive na přenos, jako digitální video, lze i zde za pomoci vhodně zvoleného formátu při zachování přijatelné kvality dosáhnout výrazného snížení paměťových nároků. To v důsledku znamená, že při streamování můžeme snížit požadavky na kvalitu, respektive šířku přenosového pásma.

3.3.1 MP3

MP3 [38], někdy označovaný také jako *MPEG-1 Audio Layer III*, je ztrátovým audio kompresním formátem, který je tak široce rozšířený, že je de facto považován za standardní formát pro streamování a uchování zvuku, tudíž je široce podporovaný na veliké škále zařízení. Je navržen tak, aby výrazně snížil velikost a množství dat pro uchování zvukové stopy (dosahuje velmi dobrých kompresních poměrů).

[2]Informace platná k datu 11.12.2016.

Komprese probíhá na principu takzvaného *vjemového kódování* (perceptual coding) [33], to znamená, že dochází k redukování, respektive aproximování, určitých částí zvukového signálu, které jsou mimo rozlišovací schopnosti lidského sluchu.

Za slabinu mp3 bývá považována komprese mluveného slova, kdy jsou při zvolené vyšší kompresi znatelné nedostatky, a to až do té míry, že je potlačena počáteční, respektive koncová, slabika u jednotlivých slov.

3.3.2 AAC

AAC [28] neboli *Advanced audio coding* je též formátem pro ztrátovou kompresi zvukové stopy. Byl navržen jako nástupce MP3, a jak ukazuje i test uvedený v kapitole 3.4, dosahuje při srovnatelných kódovacích parametrech lepších výsledků, co se týče kvality zvuku.

AAC používá „wideband“ audio kódovací algoritmus, který má za úkol co nejvíce snížit paměťové nároky, k tomu využívá následující dvě kódovací strategie:

- Komponenty signálu, které jsou pro lidský sluch neslyšitelné, jsou odstraněny (podobně jako u formátu mp3).
- Redundance v zakódovaném zvukovém signálu jsou odstraněny (zvláště u hudebních audio souborů, se určité části skladeb mohou opakovat).

Jedná se o standardizovaný formát, který je plně podporován [12] na platformě Android.

3.3.3 Opus

Opus [44] je ztrátový audio kódovací formát, který je navržen tak, aby byl vhodný pro zakódování libovolné zvukové stopy, a přitom by dosahoval dostatečně nízké latence pro hlasovou komunikaci v reálném čase. Tato jeho vlastnost, respektive zaměření, je výhodná nejen při hlasové komunikaci, ale také právě při streamování, protože jak nám i níže uvedený test 3.4 dokazuje, Opus umožňuje přenášet zvukovou stopu v přijatelné kvalitě s využitím velmi malé šířky přenosového pásma. Opus kombinuje využití *SILK* algoritmu, hlasově orientovaného lineárně prediktivního kódování a nízko latenčního *CELT* (Constrained Energy Lapped Transform) algoritmu.

Zkratka(název)	**Šířka audio pásma**	**Efektivní vzorkovací frekvence**
NB (narrowband)	4 kHz	8 kHz
MB (medium-band)	6 kHz	12 kHz
WB (wideband)	8 kHz	16 kHz
SWB (super-wideband)	12 kHz	24 kHz
FB (fullband)	20 kHz [nb 1]	48 kHz

Tabulka 3.5: Tabulka povolených vzorkovacích frekvencí.

Další zajímavé parametry formátu Opus si představíme v následujícím výčtu:

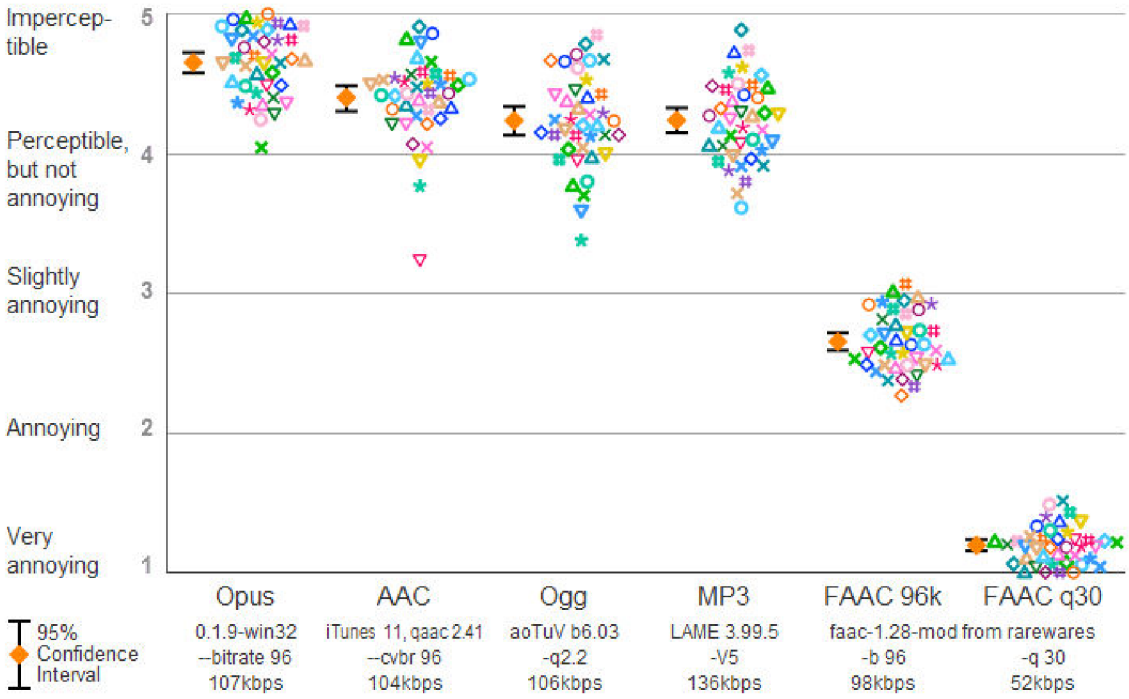
- Podporuje konstantní, ale také proměnlivý kódovací bitrate v rozmezí od *6 kbit/s* až po *510 kbit/s*.
- Velikost rámců se pohybuje v rozmezí *2.5 ms* po *60 ms*.
- K dispozici je 5 rozdílných vzorkovacích frekvencí (viz tabulka 3.5).

- Proud zvuku může mít až *255 audio kanálů*.
- Má velmi nízkou latenci a to *26.5 ms* v základním nastavení.

3.4 Porovnání audio kodeků

Samotný software implementující konkrétní audio kódovací formát nazýváme kodek. Při porovnávání kodeků nás bude výhradně zajímat poměr kvality digitální zvukové stopy a jejího bitrate. Na rozdíl od digitálního videa a jeho kodeků, zde testy probíhají výhradně subjektivní a nazýváme je *poslechové testy*.

Pro dosažení co největší objektivity zde budeme prezentovat výsledky rozsáhlého nezávislého testu [14] nejpoužívanějších audio kodeků, ke kterým jsou dostupná veškerá data.



Obrázek 3.6: Graf prezentující výsledky poslechového testu.

Uživatelé hodnotili sadu 40 zvukových stop, mezi kterými byla jak hudba, tak mluvené slovo v různých jazycích. Skladby byly zakódovány za pomocí různých audio kodeků ovšem s téměř shodnými hodnotami bitrate a dodatečnými nastaveními. Kvalita byla uživateli ohodnocena u jednotlivých zvukových stop známkami 1-5 s následujícím významem:

- *5* - Komprese je nerozeznatelná.
- *4* - Komprese je rozeznatelná, ale není rušivá.
- *3* - Komprese je lehce rušivá.
- *2* - Komprese je rušivá.
- *1* - Komprese je velmi rušivá.

Graf zobrazuje průměrné hodnocení kvality skladeb pro sadu použitých kodeků. Jednotlivé skladby jsou na grafu zobrazeny jako geometrické tvary různých barev. Veškeré parametry a data získaná v testu lze nalézt v citovaném dokumentu [14].

3.4.1 Volba audio kódovacího formátu

Z výše prezentovaného testu vzešel vítězně audio kódovací formát *Opus*, respektive jeho stejnojmenný kodek. To ovšem není jeho jediná výhoda:

- Jedná se o standardizovaný [44] formát organizací IETF.
- Je navržen pro použití s video kódovacím formátem VP9, to znamená, že oba mohou být spolu zabaleny do kontejneru *WebM*.
- Jedná se o otevřený formát, který je šířen pod BSD licencí [23].
- Má podporu [12] v systému Android, a to konkrétně od verze 5.0 (Lollipop).

3.5 Video kontejnery

Kontejnery slouží k „obalení“ souboru nebo datového toku, který obsahuje více multimediálních dat. V našem případě bude tedy kontejner sloužit k uchování zakódovaného videa a zvukové stopy. Obecně může kontejner obsahovat více videí i zvukových stop, ale také například různé titulky a další jiná metadata. Jednotlivé formáty kontejnerů se typicky odlišují právě v schopnosti uchovat konkrétní typy multimediálních dat.

Práce s kontejnery se dělí na dvě základní součásti:

- Na straně serveru dochází díky softwaru, který nazýváme *muxer* k „zabalení“ zakódovaného videa a zvukové stopy do kontejneru.
- Na druhém konci komunikace, na koncovém zařízení, dochází naopak k „rozdělení“ kontejneru zpátky na video a zvukovou stopu za pomoci *demuxeru* (někdy označován též jako splitter).

3.5.1 Matroska

Matroska [32] je standardizovaný formát kontejneru, jehož hlavní výhodou je jeho otevřenost. Všechny specifikace jsou volně dostupné a drtivá většina implementací je „open source“. Kontejner Matroska může uchovávat neomezené množství audio a video stop, titulků a rastrových obrázků v rámci jednoho souboru.

Matroska se typicky využívá v kombinaci s video kódovacím formátem H.264 a audio kódovacími formáty AAC a VORBIS. Tudíž se nejedná o kontejner vhodný pro uchování videa, respektive zvukové stopy ve formátech, které jsme zvolily, tedy VP9, respektive OPUS.

3.5.2 WebM

WebM [15] je formát video souborů, který je vyvíjen společností Google, šířen pod *BSD licencí* a jeho soubory mají příponu *.webm*. Vzhledem k streamování je ovšem zajímavější WebM kontejner, který je inspirován kontejnerem *Matroska* a podporuje video formát *VP9*

a audio formát *Opus*. To znamená, že je přesně určen pro uchovávání videí a zvukových stop zakódovaných za pomoci formátů, které byly pro výslednou aplikaci vybrány, tedy VP9 a OPUS.

V operačním systému Android má WebM nativní podporu [12], a to od verze 4.4 (KitKat), tudíž by mělo být jeho využití i s ohledem na licenci bezproblémové.

Kapitola 4

Síťový přenos

V okamžiku, kdy video a zvuková stopa jsou zakódovány a zabaleny v kontejneru, je nutné výsledná data přenést po síti k cílovému zařízení. Zásadní roli v tomto přenosu hrají protokoly *transportní* vrstvy [17], které zajišťují transparentní přenos dat a mohou zaručovat i spolehlivý přenos.

4.1 UDP

Pro streamování lze využít protokol *User Datagram Protocol (UDP)* [36], který dokáže zasílat streamovaná data jako posloupnost menších paketů. Jeho výhodou je relativní jednoduchost a efektivita, nedokáže ovšem zaručit spolehlivý přenos, což v praxi znamená, že při přenosu může docházet ke ztrátě dat. Poté záleží na daném připojení, jehož kvalita přímo ovlivňuje plynulost příjmu a přehrávání streamovaného videa.

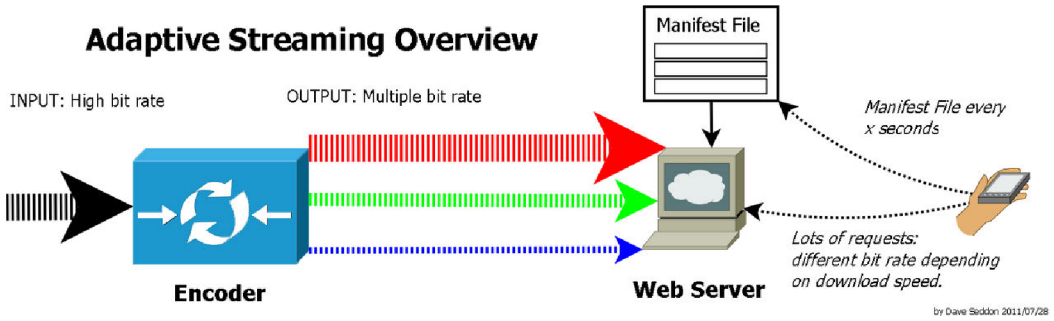
Transportní protokol UDP se typicky pro účely streamování využívá v kombinaci s dvojicí protokolů aplikační vrstvy:

- *Real-time Transport Protocol (RTP)* [40] Tento protokol zajišťuje takzvané „end-to-end“ doručování paketů zvukových a obrazových dat po síti (obecně může sloužit pro libovolná data). Zajišťuje identifikaci, sekvenční číslování, označení časovými razítky a monitoruje doručení.
- *RTP Control Protocol (RTCP)* [40] Pracuje v úzké spolupráci s RTP a jedná se o kontrolní protokol, který monitoruje probíhající přenos.

UDP v kombinaci RTP a RTCP se typicky využívá pro živé streamování, kde je nejvyšší důraz kladen na latenci a občasné výpadky obrazu jsou přijatelné. Dále se ovšem budeme zabývat jiným přístupem ke streamování, který se souhrnně označuje jako adaptive bitrate streaming a je vhodnější pro streamování uloženého obsahu.

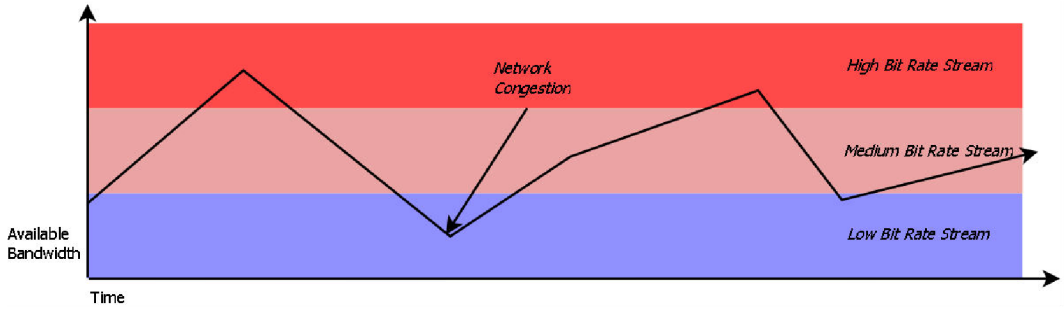
4.2 Adaptive bitrate streaming

Adaptive bitrate streaming [42] lze volně přeložit jako „streamování s adaptivním bitrate“. Jedná se o technologii, která v reálném čase detekuje rychlost a kvalitu připojení k Internetu společně s vytížením procesoru na koncovém zařízení a na základě získaných údajů upravuje, respektive adaptuje, kvalitu video streamu. Kvalita je adaptována za pomocí přepínání mezi různými hladinami bitrate.



Obrázek 4.1: Diagram popisující základní principy adaptivního streamování.

Aby bylo využití této technologie možné, je nutné zdrojové video na straně serveru vícenásobně zakódovat s různými hodnotami bitrate tak, aby vzniklo více zakódovaných video souborů, respektive potencionálních video streamů. Poté je každý stream rozdělen na malé, několika vteřinové, segmenty. Klient je následně informován za pomoci takzvaného *manifest* souboru o existenci různých streamů s odlišnými hodnotami bitrate a také o délce segmentu.



Obrázek 4.2: Ukázka fungování adaptačního algoritmu.

Na obrázku 4.2 je nastíněno fungování takzvaného *adaptačního algoritmu* [30], jehož princip je zjednodušeně popsán v následujícím odstavci.

Na začátek klient volí stream s nejnižší kvalitou, respektive nejnižší hodnotou bitrate. Pokud ovšem zjistí, že kvalita a rychlost jeho připojení je vyšší než požadavky právě přenášeného videa, zažádá si o stream ve vyšší kvalitě, a takto pokračuje až do té doby, než bude přenášen stream optimální kvality k danému připojení. Naopak, pokud se kvalita připojení zhorší, klient zažádá o stream s nižší kvalitou tak, aby předešel zhoršení plynulosti přehrávání streamovaného videa. Výsledkem využití této technologie by tedy měl být plynulý stream po celou dobu přenosu, a to i na nestálém připojení.

4.2.1 Dynamic Adaptive Streaming over HTTP

Dynamic Adaptive Streaming over HTTP (adaptivní dynamické streamování přes HTTP), někdy též označovaný jako *MPEG-DASH*, je první mezinárodně standardizovanou [22] technologií pro streamování s adaptivním bitrate.

Tato technologie, jak již její název napovídá, přenáší streamované video za pomoci aplikačního protokolu *HTTP* [24]. Protokol HTTP, ovšem na rozdíl od výše uvedené dvojice RTP a RTCP, pracuje v kombinaci s protokolem transportní vrstvy *Transmission Control*

Protocol (TCP) [37]. Díky využití TCP jako protokolu transportní vrstvy jsou garantovány následující funkce a vlastnosti:

- Dochází k navazování spojení mezi dvojicí komunikujících stran (klient – server).
- Je garantováno spolehlivé doručování ve správném pořadí. To ovšem může zvyšovat latenci spojení, protože je nutné provádět opětovné zasílání.
- Poskytuje *end-to-end flow control* neboli kontrolu „proudu“, což znamená, že odesílatel nebude vysílat data rychleji, než je příjemce dokáže přijímat.
- Má také mechanismy, které předchází zahlcení spojení, to v praxi znamená, že množství přenášených dat by nemělo překročit určitou hraniční mez, po jejímž překročení by mohlo dojít k zahlcení.

MPEG-DASH pracuje v souladu s výše uvedenými principy streamování s adaptivním bitrate (viz kapitola 4.2). Jeho hlavní výhodou oproti konkurenčním technologiím *HTTP Live Streaming*, *HDS* nebo *Smooth Streaming* je, že je nezávislý na kodeku použitém na zpracování videa, tudíž ho lze použít v kombinaci s H.265, H.264, VP9 a mnoha dalšími. Další zásadní vlastností a výhodou je jeho nativní podpora [4] této technologie v operačním systému Android.

4.2.2 HTTP Live Streaming

HTTP Live Streaming (HTTP živé streamování), často označováno jako *HLS*, je proprietárním streamovacím protokolem společnosti *Apple*. Princip jeho fungování je velmi podobný jako u výše uvedeného MPEG-DASH. To znamená mimo jiné i to, že veškerá komunikace je vykonávána za pomocí protokolu HTTP. Mezi zajímavé funkcionality HLS patří:

- Ve své specifikaci obsahuje i standardní šifrovací mechanismus, který využívá algoritmu AES.
- Umožňuje kromě video a zvukové stopy přenášet i titulky.
- Zajímavostí je přítomnost takzvaného *trikového módu*. Jedná se o systém, který dovoluje během streamování simulovat vizuální zpětnou vazbu a dojem z rychlého přetáčení videa, známý z analogových přehrávačů video kazet.

Podpora HLS je velmi široká, a to jak v podobě nepřeberného množství přehrávačů, tak i v nativní podpoře v operačních systémech firmy Apple (iOS a OS X) a také v operačních systémech konkurenčních společností (Windows a Android).

4.2.3 HTTP Dynamic Streaming

HTTP Dynamic Streaming (HTTP dynamické streamování) je proprietárním protokolem společnosti *Adobe*. Tato technologie je úzce navázána na software vyvíjený touto společností, a to *Flash Player* (od verze 10.1), respektive *Flash Media Server*.

Pro streamování je využit jeden ze dvojice protokolů HTTP a RTMP. Největší odlišností od konkurenčních protokolů je právě možnost využití protokolu *RTPM* (Real-Time Messaging Protocol), který je na rozdíl od protokolu HTTP proprietárním protokolem společnosti Adobe (dříve společnosti Macromedia). Hlavním účelem vzniku tohoto protokolu

byl právě přenos Flash videa po internetu, a to s možností využít při přenosu šifrování pomocí TLS/SSL.

Pro využití v této práci je HTTP Dynamic Streaming nevhodný, a to z důvodu jeho uzavřenosti (jedná se o proprietární software) a úzké návaznosti na video formát *Flash*.

4.2.4 Smooth Streaming

Posledním z řady protokolů pro adaptivní streamování je Smooth Streaming. Jedná se o proprietární protokol společnosti *Microsoft*, která ho taktéž standardizovala.

V principu své činnosti je tento protokol velmi podobný výše uvedeným protokolům, a to naznačuje i to, že přenos streamovaných dat je uskutečněn za pomoci protokolu HTTP.

Díky takzvanému *Porting kitu* je možné tento protokol využít v kombinaci nejen s platformou Windows, ale také Android a iOS.

4.3 Optimalizace na transportní vrstvě

Další prostor pro optimalizaci streamování se nachází ve zvolení jiného transportního protokolu než UDP či TCP. V následujícím textu budou uvedeny dvě alternativy. Bohužel v kontextu této práce jsou uvedeny pouze pro úplnost, protože ani jeden z těchto protokolů nemá podporu v operačním systému Android. Dá se tedy očekávat, že v budoucnu se v Androidu jejich podpora objeví, protože konkurenční platforma, respektive operační systém iOS, od společnosti Apple podporuje protokol MPTCP již od verze 7 (tedy od 18. září 2013).

4.3.1 MPTCP

MPTCP [25] neboli *Multipath* TCP je alternativou k protokolu TCP a je s ním zpětně kompatibilní. Jeho hlavní myšlenkou a cílem je, umožnit, aby data zasílaná pomocí TCP spojení mohla být přenášena více cestami současně, díky čemuž lze lépe využít síťovou infrastrukturu. To může vést k získání širšího přenosového pásma.

Další důležitou funkcionalitou, kterou MPTCP přináší, je že v případě výpadku jednoho z využitých připojení není samotné TCP spojení přerušeno. To v praxi znamená, že pokud bylo video přenášeno za pomocí MPTCP s využitím mobilního připojení a Wi-Fi, tak nedochází při výpadku jednoho k výpadku celého TCP spojení, ale streamování pokračuje bez výpadku dále za pomoci zbývajícího připojení.

4.3.2 SCTP

SCTP [41] neboli *Stream Control Transmission Protocol* je taktéž protokolem transportní vrstvy zajišťující spolehlivý a spojovaný přenos podobně jako TCP. Jeho hlavní odlišností je možnost vytvořit několik nezávislých kanálů, které jsou přepravovány po síti paralelně. V terminologii SCTP nazýváme spojení *asociací*. Po navázání asociace po ní můžeme přenášet několik nezávislých proudů, které nejsou vzájemně ovlivněny (selhání a následná náprava chyby v jednom z proudů nijak neovlivní proudy ostatní). Mezi hlavní výhody SCTP patří:

- *Multihoming* umožňuje komunikačnímu uzlu vystupovat v rámci asociace pod více IP adresami (vzniká více cest), přičemž jedna je vždy brána jako primární a jsou na ni odesílána data.

- Dochází k sledování stavu jednotlivých cest v rámci kanálu a v případě, že má cesta k primární adrese časté problémy s dostupností, je jako primární zvolena jedna z adres záložních.

Kapitola 5

Návrh řešení

Následující kapitola pojednává o aspektech návrhu mobilní aplikace pro platformu Android, jenž představuje klientskou část výsledného řešení. Poté je nastíněn návrh API (application programming interface), které slouží k takzvanému „volání“ metod vykonávajících souborové a adresářové operace na serveru a také návrh serverů, jenž se starají o samotné streamování videa.

5.1 Návrh mobilní aplikace

Výsledná mobilní aplikace slouží pouze jako jakýsi „prohlížeč“ dat uložených na serveru a umožňuje provádět za pomoci komunikačního rozhraní volání souborových operací na serveru. Z toho vyplývá, že na straně klienta není potřeba navrhovat a posléze implementovat složitou podpůrnou logiku, jelikož téměř veškerá logika je implementována na straně serveru. Dále se tedy budeme bavit výhradně o návrhu aplikace z hlediska grafického uživatelského rozhraní (GUI), jehož kvalita bude přímo odpovídat kvalitě výsledné aplikace.

5.1.1 Drátový model

Prvotním a naprosto elementárním krokem návrhu GUI bylo vytvoření drátového modelu (viz příloha C). Ten nevyobrazuje jednotlivé grafické detaily ani konkrétní vzhled rozhraní, ale jasně nastiňuje rozmístění jednotlivých prvků, jejich základní funkcionalitu a v neposlední řadě dělí nejen GUI, ale také celou aplikaci na jednotlivé logické celky, které budeme označovat jako aktivity. Pod pojmem *aktivita* [1] v terminologii operačního systému Android rozumíme, velmi zjednodušeně řečeno, „jednu obrazovku“. Aktivita shromažďuje jak logiku, tak vzhled a definici GUI, proto se jedná o vhodnou jednotku dělení celku aplikace na menší části.

V následujícím textu se zaměříme na popis návrhově nejzajímavějších a funkcionálně nejzásadnějších aktivit a prvků GUI s ohledem na jejich rozhraní a funkcionalitu.

5.1.2 Zobrazení souborů a adresářů

Základní funkcionalitou je zobrazení jednotlivých souborů, respektive adresářů, a to s možností tyto položky řadit, vyhledávat v nich a také přecházet mezi různými adresářovými úrovněmi. Aktivita zobrazující soubory a adresáře poskytuje dva odlišné módy zobrazení:

- *Zobrazení v mozaice* je vhodné pro prohlížení adresářů obsahujících multimediální soubory (fotky, videa) nebo dokumenty, a to díky tomu, že jsou zobrazeny náhledové

obrázky. Počet sloupců mozaikového zobrazení se adaptuje podle šířky obrazovky zařízení. Minimálně jsou zobrazeny 2 sloupce, avšak na tabletech a telefonech s širokým displejem může být zobrazeno sloupců více (maximálně však 5).

- *Zobrazení v seznamu* je naopak vhodné pro procházení většího množství souborů, u kterých pro uživatele není tak důležitý jejich náhled, ale spíše typ a název.

5.1.3 Výběr souborů a adresářů

Tak, aby bylo možné provádět operace nad soubory a adresáři nejen pro jednotlivé prvky, ale i nad výběrem, je nutné navrhnout pro aktivitu zobrazení mód výběru. Tento mód je vyvolán dlouhým stlačením položky mozaiky, respektive seznamu, a nebo za pomoci položek „výběr“ a „výběr všeho“ v menu aktivity. Poté, co je mód výběru aktivován, může uživatel kliknutím na libovolnou položku provést její výběr a nebo naopak výběr zrušit.

Na spodní části aktivity je v době, kdy je mód výběru aktivní, zobrazena lišta, která dává uživateli možnost vykonat operaci nad vybranými soubory a adresáři.

5.1.4 Kontextová menu

Jeden z velkých problémů při návrhu GUI je způsoben možností uživatele provádět značné množství souborových a adresářových operací, a to skrze tlačítka „napojená“ na jejich volání. Problémem je, kde tato tlačítka zobrazit a přitom neblokovat veliké množství prostoru. Jako vhodné řešení jsou zvolena takzvaná kontextová menu.

První kontextové menu pracuje v kombinaci s takzvaným plovoucím akčním tlačítkem (Floating action button), které je jedním z mnoha prvků, jenž nám nabízí designový jazyk *Material design* [8]. Po kliknutí na toto tlačítko, které je k dispozici na aktivitě pro zobrazení souborů a adresářů, je zobrazena kontextová nabídka umožnující provést následující operace:

- Vytvořit nový adresář.
- Nahrát soubory.
- Vyfotit fotku.
- Nahrát video.
- Zaznamenat zvukovou stopu.

Druhé kontextové menu je dostupné pro každý soubor a adresář po kliknutí na tlačítko menu vyobrazené pomocí tří nad sebou ležících teček. Toto menu umožňuje uživateli zvolit operaci, která má být provedena nad souborem, respektive adresářem, pro který bylo kontextové menu zobrazeno.

Obě zmíněná kontextová menu je možné zavřít poklikem na libovolnou plochu mimo menu. V aplikaci se vyskytuje více kontextových menu. Zde jsou uvedeny pouze dvě základní a pro funkci naprosto nezbytná.

5.1.5 Hamburger menu

Dalším z klasických GUI prvků Material design, které byly při návrhu aplikace využity, je takzvané *hamburger menu*. Tento prvek si svůj název vysloužil podle ikonky vyobrazující tři nad sebou ležící čáry, které připomínají právě hamburger. V aplikaci je toto menu, které se vysouvá z levé strany aktivity, využito pro více účelů:

- Zobrazuje aktuálně přihlášeného uživatele, přičemž společně s jeho jménem je zobrazen i takzvaný „avatar“ neboli obrázek jeho profilu.
- Zobrazena je též informace o tom, kolik prostoru má daný uživatel v jeho cloudovém uložišti obsazeno a kolik prostoru má k dispozici celkem.
- Nabízí možnost navigace (přechodu) mezi jednotlivými aktivitami.
- Jsou zde k dispozici odkazy na další možnosti jako:
  - Nastavení aplikace.
  - Informace o aplikaci.
  - Hodnocení aplikace.
  - Možnost zaplatit si rozšíření kapacity úložného prostoru.

5.1.6 Nahrávání zvuku

Na tvorbu fotografií a videozáznamů lze použít systémové aktivity, na jejichž používání je uživatel zvyklý. Ovšem u nahrávání zvuku tuto možnost nemáme, a tudíž je nutné navrhnout vlastní řešení.

K nahrávání zvuku slouží dvou-krokový dialog (modální okno nad aktivitou). V prvním kroku je možné nahrát samotnou zvukovou stopu a po ukončení nahrávání je zobrazen druhý krok dialogu, který umožňuje nahraný zvuk přehrát a případně uložit na cloudové uložiště.

5.2 Návrh API

Implementace serverové logiky zabývající se souborovými a adresářovými operacemi není předmětem této práce, ale návrh aplikačního programového rozhraní neboli API a operací, které bude na straně serveru potřeba implementovat, je v následujícím textu popsán. Rozhraní je implementováno pomocí *representational state transfer (REST)* [19] rozhraní.

5.2.1 REST

REST je architektura rozhraní, která je vhodná pro snadný přístup ke zdrojům na serveru. Zdroji budeme v kontextu této práce rozumět jednotlivé souborové a adresářové operace. Volání jsou uskutečněna pomocí protokolu HTTP a jednotlivé zdroje jsou identifikovány následujícím způsobem:

- *URI (Uniform Resource Identifier)*, která jasně identifikuje cestu k danému zdroji a slouží též k identifikaci zdroje.
- Zdroje se stejnou URI mohou být dále logicky děleny pomocí zvolené *HTTP metody* (například GET, PUT, POST, DELETE, ale i další).

Každý HTTP dotaz může obsahovat data v hlavičce a také ve svém těle ve formátu JSON. Podobně i odpověď na dotaz může obsahovat JSON data ve svém těle a musí obsahovat také číselný kód, který mimo jiné říká, jestli byl dotaz zpracován správně.

5.2.2 Apiary

Při návrhu byl využit webový nástroj Apiary[1], který je určen právě na vývoj, respektive návrh, API. Tento nástroj umožňuje definice jednotlivých prvků rozhraní přehledně zapisovat a na základě tohoto zápisu:

- Vytváří přehlednou a jasně strukturovanou dokumentaci.
- Umožňuje provádět testovací volání jednotlivých metod rozhraní.
- Vytváří automatické testy rozhraní, které lze periodicky spouštět.

5.2.3 Navržené API

V tabulce 5.1 je ukázka navrženého API, a to konkrétně výčet REST metod pro vykonávání adresářových operací. Kompletní tabulka zahrnující všechny metody API bude obsažena v přílohách. Sloupec s označením *sufix URI* ukazuje pouze koncovou část celé URI, která specifikuje požadovanou metodu. Zbývající část URI je URL odkazující na server poskytující REST metody.

URI sufix	**HTTP metoda**	**Popis**
/folders	POST	Vytvoří nový adresář v zadané cestě.
/folders	PUT	Přejmenuje adresář v zadané cestě a kontroluje validitu nového jména.
/folders	DELETE	Smaže adresář v zadané cestě, a to i s veškerým jeho obsahem.
/folders/content	POST	Vrací JSON objekt obsahující veškeré informace o obsahu adresáře.
/folders/copy	POST	Zkopíruje adresář i s jeho obsahem do zadaného cílového adresáře.
/folders/move	POST	Přesune adresář i s jeho obsahem do zadaného cílového adresáře.

Tabulka 5.1: Přehled REST metod vykonávajících adresářové operace.

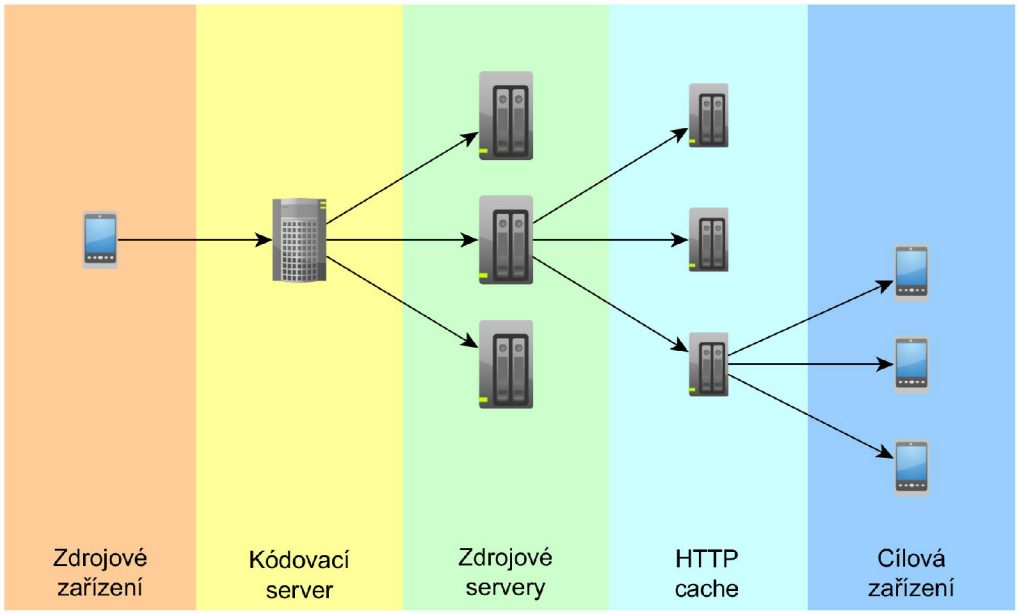
5.3 Návrh serverů pro streamování videa

Jak již nadpis napovídá, v procesu streamování videa nebude vystupovat pouze jeden server, ale v ideálním případě jich bude celá řada, kdy každý z nich má specifickou úlohu. V následujícím textu se zaměříme nejen na použitou architekturu serverů, ale také na vnitřní architekturu klientské aplikace.

[1] K dispozici na: https://apiary.io/

5.3.1 Architektura serverové části

Obrázek 5.1 popisuje typickou architekturu kolekce serverů, které zajišťují při vzájemné spolupráci streamování videa s adaptivní hodnotou bitrate.



Obrázek 5.1: Typická architektura serverové infrastruktury.

Funkcionalita jednotlivých součástí je následující:

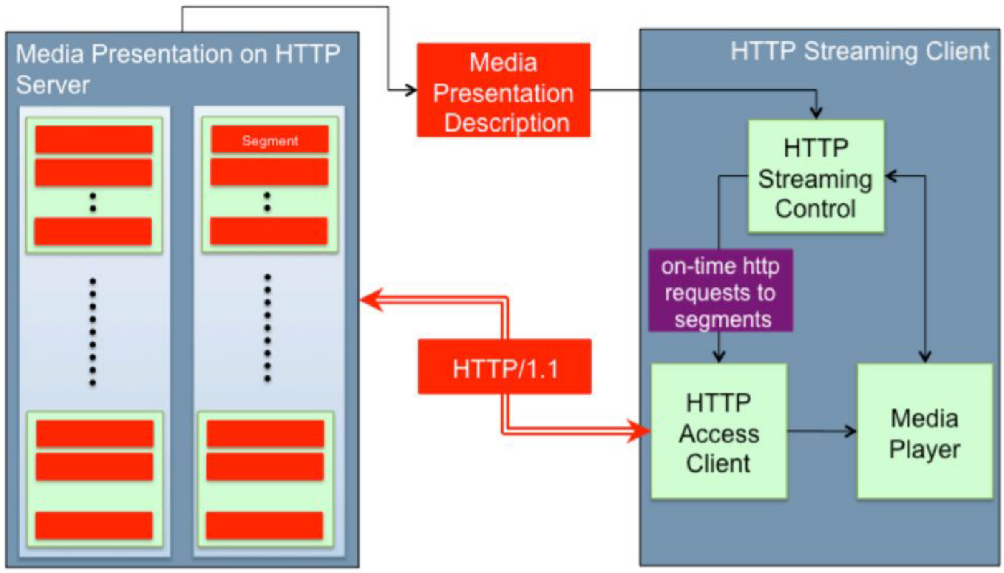
- *Zdrojové zařízení* je zdrojem dat, respektive videa, které je ze zařízení nahráno na kódovací server.
- *Kódovací server*, někdy též označovaný jako přípravný server, se stará o příjem videí, která jsou nahrávána ze zdrojových zařízení. Každé přijaté video musí následně zakódovat v souladu s principy streamování videa s adaptivní hodnotou bitrate. To znamená, že dochází k vícenásobnému zakódování zdrojového videa s různým nastavením hodnoty *bitrate*. Zakódována videa jsou následně dělena na menší segmenty a tím vzniká takzvaný soubor adaptivního videa, ve kterém jsou k dispozici veškerá data pro následné streamování. Tento soubor je přesunut na jeden ze zdrojových serverů.
- *Zdrojový server* slouží pro uchování zakódovaných video souborů a pokud se klientem požadovaný video soubor nenachází na jednom z HTTP cache serverů, stará se i o samotné streamování videa.
- *HTTP cache* servery se starají o snížení zátěže na zdrojové servery tím, že streamovaná videa dočasně ukládají u sebe, a tudíž mohou funkci zdrojových serverů částečně nahradit. Taktéž mohou zlepšit kvalitu streamování videa, protože se tyto servery typicky mohou nacházet, co se týče síťového připojení, „blíže“ ke klientovi, a tudíž je k nim i lepší připojení.

V rámci této práce se budeme zabývat implementací kódovacího serveru, který po zpracování videa přenese cílová data za pomocí výše uvedeného API právě na servery sloužící

pro ukládání dat. Dále bude implementován server, který bude zabezpečovat samotné streamování uložených video souborů a komunikaci související s fungováním streamování.

5.3.2 Architektura klientské části

Detailnější pohled na architekturu klienta a komunikační kanály využívané při klient-server komunikaci nám dává schéma 5.2.



Obrázek 5.2: Ukázka architektury klientské části a komunikačních kanálů.

Jak lze vidět, na straně serveru je nachystáno více variant videa a také v rámci každé varianty jednotlivé segmenty. Nastíněny jsou dva komunikační kanály:

- *Hlavní HTTP kanál*, který slouží pro samotný přenos streamovaného videa a komunikaci spojenou s tímto přenosem.
- *Kanál pro metadata* slouží pro přenos doplňujících metadat, respektive informací, spojených s adaptivním streamování. Typicky jsou zde přenášeny takzvané *manifest* soubory (viz kapitola 4.2).

Schéma 5.2 popisuje i vnitřní architekturu klientského software starajícího se o adaptivní streamování, ta se skládají ze tří základních částí:

- *Kontrola HTTP streamu* se stará o kontrolování plynulosti streamovaného videa a případně upravuje vlastnosti streamu (žádá server o stream s vyšším, nebo naopak nižším bitrate). Dále zprostředkovává pro přehrávač logiku přehrávání, jako například pauza, posun v čase a spuštění videa.
- *HTTP přístupový klient* zajišťuje samotnou HTTP komunikaci se serverem a poskytuje bitový proud streamovaného videa přehrávači.

- *Multimediální přehrávač* přijímá příchozí bitový proud a presentuje video koncovému uživateli. Pracuje také v kombinaci s kontrolou HTTP streamu, tak aby znal parametry videa, které má přehrávat a mohl streamování na základě požadavků uživatele ovládat.

Kapitola 6

Implementace klientské aplikace

Tato kapitola se bude věnovat implementaci klientské aplikace, kterou představuje mobilní aplikace platformy Android. Pro její vývoj byl použit objektově orientovaný programovací jazyk Java, verze 7 v kombinaci se značkovacími jazyky *XML* a *JSON*. Vývoj probíhal na operačním systému Windows 10 a zvoleno bylo nejrozšířenější vývojové prostředí pro vývoj Android aplikací *Android Studio*. Pro účely zálohování a takzvaného „verzování“ byl využit verzovací systém *Git* v kombinaci s nástrojem *GitLab*.

V následujícím textu bude popsán způsob implementace důležitých nebo implementačně zajímavých funkcionalit klientské aplikace.

6.1 Klient-server komunikace

Zásadní roli při fungování aplikace představuje komunikace se serverem, přesněji řečeno s jeho *REST* rozhraním. Veškeré souborově-adresářové operace jsou vykonávány právě na straně serveru a do klientské aplikace jsou následně pouze „promítnuty“.

Tato komunikace byla implementována pomocí knihovny *Retrofit*[1] verze *2.2.0*, která umožňuje velmi elegantně a poměrně snadno realizovat volání REST rozhraní.

6.1.1 Definice dotazů

Prvním krokem byla tvorba rozhraní *StorageRestInterface*, které slouží pro snadnou definici jednotlivých dotazů.

```
@POST("storage/v1/folders")
Call<Path> createNewFolder(
        @Header("Authorization") String accessToken,
        @Body Path path);
```

Listing 6.1: Definice dotazu *„createNewFolder“*.

Z výše uvedeného ukázkového kódu lze vyčíst pevně danou strukturu dotazu:

- Definice začíná anotací (*@POST*), která specifikuje, jaká dotazovací metoda protokolu HTTP bude využita.
- Specifikována je unikátní část URL (*"storage/v1/folders"*).

[1] K dispozici na: http://square.github.io/retrofit/

- Návratovým typem musí být odpovídající *POJO* (viz kapitola 6.1.2) objekt (*Call<Path>*).
- Parametry metody korespondují s parametry odpovídající REST metody. Musejí být vždy anotovány tak, aby bylo jednoznačně určeno, či se daný parametr *předává* v těle dotazu, hlavičce dotazu, anebo prostřednictvím parametrů URL.

6.1.2 Model komunikace

Tělo odpovědi serveru u většiny dotazů představuje JSON objekt, který je při příjmu odpovědi transformován za pomocí knihovny GSON na takzvaný *Plain old Java object* (dále POJO). POJO představuje klasický objekt jazyka Java, jehož atributy přímo odpovídají atributům JSON objektu.

Tato konverze funguje i opačným směrem, pokud dotaz na server má neprázdné tělo. Obsah těla dotazu je typicky také JSON objekt, tudíž se při vykonávání takového dotazu na něj POJO objekt převádí.

Pro každý možný JSON objekt musí v aplikaci existovat odpovídající třída umožnující instanciovat příslušný POJO objekt. Všechny tyto třídy souhrnně označujeme *modelem* komunikace.

6.1.3 Vykonání dotazu na server

Knihovna *Retrofit* umožňuje vykonávat 2 různé druhy dotazování na server:

- *Synchronní* dotazování na server blokuje další vykonávání programu do té doby, než obdrží odpověď od serveru, anebo spojení neskončí po vypršení takzvaného „time-outu“.
- *Asynchronní* dotaz čeká na odpověď serveru v samostatném vlákně, tudíž nedochází k blokování, ovšem následnou odpověď je taktéž nutné zpracovat asynchronně.

I přes vyšší složitost implementace bylo nutné pro dotazování zvolit *asynchronní* přístup, jelikož platforma Android neumožňuje blokování na hlavním vlákně (někdy též označovaném jako *„user interface“* vlákno).

```
Call<Path> call = createNewFolder(accessToken, path);
call.enqueue(new Callback<SearchResponse>() {
        @Override
        public void onResponse(Call<Path> call, Response<Path>
           response) {
                if (response.isSuccessful()) {
                        // response ok
                }
        }

        @Override
        public void onFailure(Call<Path> call, Throwable t) {
                // call failed
        }
});
```

Listing 6.2: Asynchronní volání REST metody *„createNewFolder“*.

Jak naznačuje výše uvedená ukázka kódu, metody *onResponse*, respektive *onFailure*, jsou asynchronně volány v okamžiku úspěšného, respektive selhaného, dotazu na server. Po každém dokončeném volání následuje určitá reakce, která je závislá na úspěšnosti daného volání (například zobrazení nové složky, zobrazení chybového hlášení a podobně).

6.1.4 Zpřístupnění API

Třída *StorageRestHandler* obsahuje statické metody, které zprostředkovávají jednotlivé dotazy na server. Kdekoliv v kódu aplikace je tedy možné provést volání jedné z těchto statických metod, a tudíž i vykonat dotaz na server. Tímto způsobem je zpřístupněno API serveru a došlo také k logickému oddělení kódu, který se stará o provádění dotazů na server.

Jediným předpokladem pro správné fungování této implementace je nutnost jednorázového zavolání metody *StorageRestHandler.init()* při spuštění aplikace. Tato metoda se stará o korektní inicializaci knihoven *Retrofit* a *GSON* potřebných pro vykonávání dotazů.

6.1.5 Download a upload

Operace *download* (stažení souboru) a *upload* (nahrání souboru na server) se od ostatních souborových operací liší tím, že délka jejich vykování může být značná. Z tohoto důvodu je vhodné tyto operace vykonávat v samostatném vlákně tak, aby provádění těchto operací neblokovalo uživatele a ten mohl dále aplikaci používat.

Dalším požadavkem na implementaci těchto operací, je odloučení od životního cyklu aplikace. To v praxi znamená, že i v okamžiku, kdy uživatel nemá aplikaci otevřenou na „popředí“, může download, respektive upload, pokračovat dále.

Dále se zaměříme konkrétně na operaci download, na níž si popíšeme způsob implementace. K implementaci operace download byla využita třída *DownloadManager*, která představuje systémovou službu určenou právě pro stahování souborů, a to za pomocí protokolu *HTTP*.

```
DownloadManager dm = (DownloadManager)
    getSystemService(Context.DOWNLOAD_SERVICE);
DownloadManager.Request request = new
    DownloadManager.Request(Uri.parse(BASE_URL +
    "storage/v1/files/multi"));

request.addRequestHeader("Content-type", "application/json");
request.addRequestHeader("Authorization", accessToken);
request.addRequestHeader("Path", pathToFile));

enqueue = dm.enqueue(request);
```

Listing 6.3: Ukázka práce s *DownlodManagerem*.

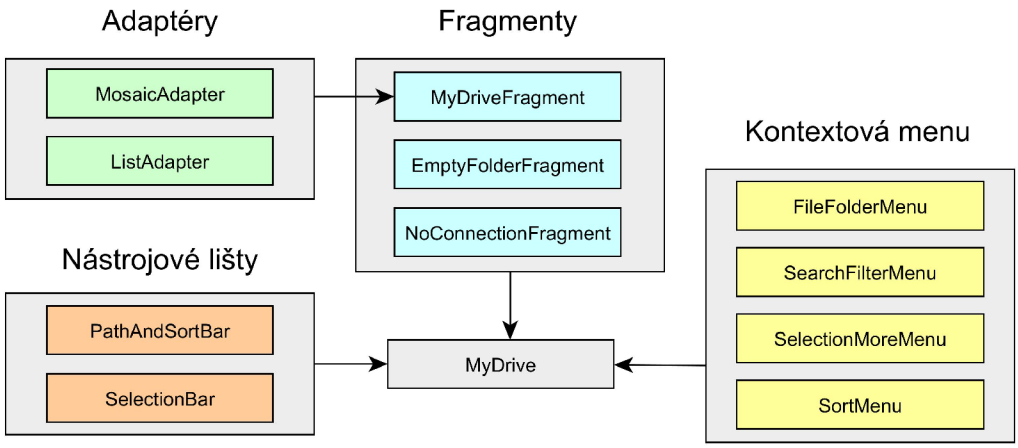
Výše uvedená ukázka kódu ukazuje základní inicializaci *DownlodManageru* a následné vykonání dotazu na server. Dále se v kódu nachází *BroadcastReceiver*, který slouží pro „zachytávání“ broadcastových zpráv zasílaných napříč Android systémem. Pokud je zachycena zpráva typu *DownloadManager.ACTION_DOWNLOAD_COMPLETE* víme, že bylo dokončeno stahování souboru. Uživatel je o dokončení stahování informován pomocí notifikace. Ta je ovšem zobrazena po celou dobu stahování a zobrazuje podstatné informace o průběhu stahování.

Operace upload je implementována obdobně, a tudíž není její bližší popis nutný.

6.2 Aktivita MyDrive

Od implementace sítové komunikace se dostáváme k implementaci grafického uživatelského rozhraní aplikace.

MyDrive je klíčovou aktivitou, která se stará o zobrazení obsahu a zprostředkovává veškeré ovládací prvky umožnující provádět souborově-adresářové operace. Z důvodů rozsahu a vyšší komplexity této aktivity, ji bylo nutné rozdělit na více logických celků.



Obrázek 6.1: Architektura aktivity MyDrive.

6.2.1 Layout aktivity

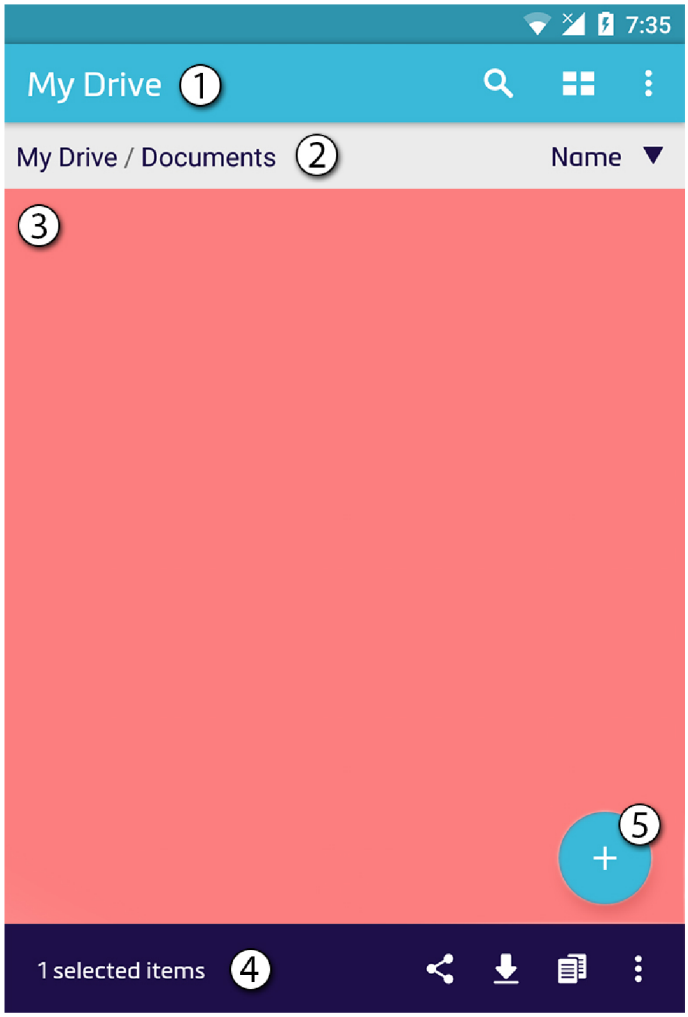
Layout definuje rozložení jednotlivých grafických prvků (v praxi se setkáme s označením „pohled“ a v dalším textu ho budeme užívat) na dané aktivitě. Při implementaci aktivity je návrh a tvorba layoutu základní činností, na kterou pak úzce navazuje samotný kód. V dalším textu bude popsán layout aktivity *MyDrive* tak, aby bylo možné se při popisu implementace aktivity odkazovat na jednotlivé součásti tohoto layoutu.

Následující výčet popisuje nejvýznamnější součásti layoutu (číslování odrážek koresponduje s číslováním v obrázku 6.2):

1. *Panel nástrojů* (Toolbar) obsahuje nadpis aktivity a zobrazuje menu. Některé položky menu (v závislosti na volném prostoru) jsou zobrazeny přímo na panelu, zbytek je „schovaný“ v kontextovém menu.

2. *Panel cesty a řazení* slouží k zobrazení aktuální adresářové cesty, ve které se uživatel nachází, zároveň ve své pravé části obsahuje kontextové menu umožnující konfiguraci typu řazení obsahu (podle jména, typu a data modifikace).

3. *Fragmenty* celá část naznačená červenou barvou je určená pro zobrazení jednoho z fragmentů. Fragment představuje určitou část uživatelského rozhraní a přidružené logiky, která tvoří samostatný celek. Aktivita v sobě může mít zobrazen jeden či více



Obrázek 6.2: Ukázka layoutu aktivity MyDrive.

fragmentů, přičemž fragmenty mají svůj vlastní životní cyklus a lze je tedy dynamicky na aktivitě zobrazovat či nahrazovat. Tohoto principu bude hojně využito, kdy v různých situacích bude v této části layoutu zobrazen odlišný fragment.

4. *Panel výběru* je viditelný pouze při aktivním módu výběru (viz kapitola 6.2.4) a umožňuje uživateli zvolit operaci, která má být provedena nad výběrem.

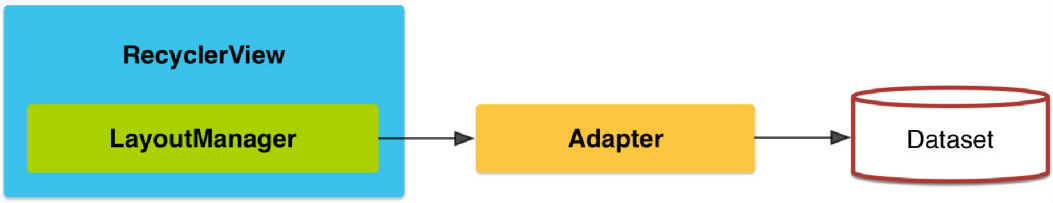
5. *Floating action button*, což je možné volně do češtiny přeložit jako „plovoucí akční tlačítko“, slouží pro zobrazení kontextového menu zpřístupňujícího možnosti, jako je nahrání souboru, pořízení fotografie a mnohé další.

6.2.2 Fragment MyDriveFragment

Stěžejní činností při fungovaní klientské aplikace je zobrazování obsahu souborového systému.

Souborový systém, respektive jeho adresářová struktura, odpovídá logické struktuře *stromu*. Každý uzel tohoto stromu odpovídá právě jedné úrovni souborového systému.

O zobrazení obsahu jedné úrovně se stará fragment *MyDriveFragment*, a to s využitím takzvaného *RecyclerView*. *RecyclerView* je pohled systému Android, který slouží k zobrazování větších objemů dat, v nichž musí být možné efektivně „scrolovat“.



Obrázek 6.3: Zobrazovací prvek RecyclerView.

RecyclerView ke svému správnému fungování vyžaduje několik součástí (viz obr. 6.3), jejichž význam je popsán v následujících podkapitolách.

LayoutManager

LayoutManager je „zodpovědný“ za to, jakým způsobem budou jednotlivé položky v RecyclerView zobrazeny. Layout managerů existuje celé řada, ale pro použití v této aplikaci byla zvolena dvojice:

- *LinearLayoutManager* dokáže položky zobrazit v seznamu jehož prvky se nachází pod sebou. Toto zobrazení si můžeme přirovnat k tabulce obsahující pouze jeden sloupec a neomezené množství řádků.
- *GridLayoutManager* umožňuje zobrazovat jednotlivé položky v takzvaném „gridu“, respektive mozaice. Použijeme-li znovu přirovnání k tabulce, jedná se o tabulku s pevně daným počtem sloupců (typicky je počet sloupců větší než 1) a neomezeným počtem řádků.

Za pomoci této dvojice „manažerů“ dokážeme vyhovět požadavkům v návrhu aplikace a umožníme uživateli zobrazit soubory a složky v *seznamu*, respektive *mozaice*.

		Orientace	
		portrait	landscape
Velikost displeje	small	1	2
	normal	2	4
	large	4	6
	xlarge	6	8

Tabulka 6.1: Počty sloupců *GridLayoutManager*.

Zajímavostí je optimalizace mozaikového zobrazení, kdy je *GridLayoutManageru* nastaven počet sloupců v závislosti na velikosti a orientaci displeje (viz tabulka 6.1). Díky této optimalizaci se mozaikové zobrazení přizpůsobí konkrétnímu zařízení koncového uživatele. To například u tabletů umožňuje velmi efektivně procházet velká množství adresářů a souborů.

Adaptér

V adaptéru je „ukryta“ logika zobrazení jednotlivých položek seznamu, respektive mozaiky. Pro účely zobrazení obsahu byly vytvořeny dva základní adaptéry:

- *ListAdapter* zajišťuje korektní zobrazení „layoutu“ položky seznamu, jeho naplnění hodnotami a implementuje *RecyclerViewClickListener*, díky němuž je možné zachytávat akci kliknutí na položku.
- *MosaicAdapter* se stará o zobrazení položky mozaiky. Ta má ovšem jedno zásadní specifikum oproti položce seznamu a to, že zde má položka adresáře jiný layout jako položka souboru. Bylo zde nutné tedy implementovat metodu *getItemViewType*, která pro danou pozici navrátí typ dané položky (*ItemType.FOLDER*, respektive *ItemType.FOLDER*). Na základě zjištění typu položky dojde v metodě *onCreateViewHolder* k zobrazení a naplnění hodnotami správného „layoutu“. Obdobně u detekce kliknutí na položku mozaiky dochází k rozlišení, zdali bylo kliknuto na položku adresáře nebo souboru.

Dataset

Dataset neboli *datovou sadu* zde představuje JSON objekt získaný za pomocí volání REST metody *getOrderedFolderContent*.

```
{
    "path": "root/subfolder",
    "folders": [
        {
                        "name": "folder_name",
                        "lastModified": 14672639840300
                        ...
        }
    ],
    "files": [
        {
            "name": "file_name.txt",
            "lastModified": 1467263999000,
                        ...
        }
    ]
}
```

Listing 6.4: Ukázkový JSON objekt.

Jak lze vidět na výše uvedeném ukázkovém JSON objektu, jsou v něm obsažena veškerá potřebná data pro zobrazení obsahu. Nachází se v něm atribut *path*, který definuje cestu k adresáři, jehož obsah má býti zobrazen. Dále se v tomto objektu nachází dvojice polí, které se sestávají z objektů reprezentujících jednotlivé adresáře, respektive soubory.

6.2.3 Zástupné fragmenty

K zobrazení obsahu je využit fragment *MyDriveFragment*, ovšem v některých situacích není obsah (dataset) k dispozici. V těchto případech je zobrazen jeden z takzvaných „zástupných“[2] fragmentů.

EmptyFolderFragment je fragment, který je zobrazen v případech, kdy je adresář, pro který má být obsah zobrazen, prázdný (neobsahuje žádné adresáře a soubory).

Před každým libovolným voláním REST rozhraní serveru je ověřena konektivita (dostupnost připojení k internetu, respektive serveru). Pokud není připojení k dispozici, místo obsahu je zobrazen fragment *NoConnectionFragment*, který uživatele informuje o vzniklé situaci a prostřednictvím zobrazeného tlačítka „nastavení“ mu nabízí možnost přejít do nastavení připojení a konektivitu zajistit.

Zároveň se zobrazením tohoto fragmentu je inicializován *NetworkStateReceiver*. Ten slouží pro zachytávání systémových zpráv informujících o změnách v síťovém připojení. Pokud je zachycena jakákoliv změna, dojde k opětovnému ověření konektivity a v případě, že je připojení dostupné, je zobrazen požadovaný obsah.

6.2.4 Mód výběru

Některé adresářově-souborové operace lze provádět nad více jak jedním adresářem, respektive souborem, tudíž je nutná přítomnost takzvaného módu výběru, při kterém lze označit více položek a operaci pak vykonat na nich.

Uživatel tento mód vyvolá dlouhým kliknutím na položku seznamu nebo mozaiky, případně za pomocí položek „výběr“ a „vyber vše“ kontextového menu panelu nástrojů. Poté, co je mód aktivován, je upravena logika akce kliknutí na položku seznamu nebo mozaiky. Nyní kliknutí na neoznačenou položku vyvolá její označení, což se graficky vyznačuje tím, že je položka zabarvena na modro.

Během aktivního módu selekce je vytvořena dvojice *bitových (boolean)* polí *selectedFiles* a *selectedFolders*, která jsou inicializována na hodnotu *0 (false)* pro všechny jejich položky. Tato pole mají stejný počet položek jako odpovídající pole datové sady (viz kapitola 6.2.2). V okamžiku, kdy uživatel vybere libovolnou položku, je hodnota na odpovídající pozici v jednom z polí změněna na *1 (true)*, čímž dojde k zapamatování výběru.

V okamžiku, kdy uživatel označí požadované adresáře a soubory, může prostřednictvím zobrazeného panelu výběru zvolit jednu z operací, které chce nad výběrem vykonat.

6.2.5 Kontextová menu

Kontextová menu jsou v rámci klientské aplikace hojně využívána a k jejich tvorbě byla využita komponenta systému android *PopupMenu*. Ta se stará o tvorbu menu zobrazeného v modálním okně, které je „přikotvené“ k nějakému pohledu. Právě takové menu označujeme jako *kontextové*.

6.3 Operace

Prozatím bylo popsáno jakým způsobem dochází k zobrazení obsahu a grafických prvků pro ovládání a také jakým způsobem lze volat REST metody dostupné na serveru. Mezi těmito vrstvami se ovšem nachází spojující mezivrstva, kterou označíme souhrnným názvem

[2] V anglické terminologii často označováno jako „placeholder“.

„operace“. Právě tato mezivrstva „vdechne“ klientské aplikaci život a umožní uživateli vykonávat požadované operace nad zobrazenými adresáři a soubory.

Následující text se bude věnovat jednotlivým „skupinám“ operací a nastíní některé aspekty jejich implementace.

6.3.1 Souborově-adresářové operace

Souborově-adresářovými operacemi rozumíme ty operace, které jsou nutné pro základní manipulaci se soubory a adresáři. Typicky se jedná o operace jako upload, download, přesunutí, kopírování, přejmenování nebo odstranění, a to jak nad jedním, ale typicky i nad více adresáři, respektive soubory.

Funkcionalita souborově-adresářových operací neboli tříd představujících vrstvu mezi grafickým uživatelským rozhraním a voláním serveru, spočívá hlavně v shromažďování všech potřebných informací k zformulování validního dotazu na server.

Operací tohoto typu je v klientské aplikaci větší množství, tudíž není možné detailně popsat fungování všech. V dalším textu se zaměříme na konkrétní příklad, který dá nahlédnout do typické implementace těchto operací.

Tvorba adresáře

Logiku tvorby nového adresáře implementuje třída *CreateFolderJob*. Ta musí shromáždit veškerá data potřebná pro správnou formulaci dotazu na server.

Základem je získání uživatelova přihlašovacího tokenu. Ten je uložen v takzvané *„klíčence“* na zařízení a byl získán při přihlášení uživatele. Tento token je ostatně potřebný pro libovolné volání serveru, bez něj je dotaz považován jako neoprávněný a je serverem odmítnut.

Dalším parametrem je aktuální cesta souborového systému, v které se uživatel nachází v době, kdy chce vytvořit nový adresář. Tato cesta je vždy uložena v takzvaných *Shared Preferences* pod identifikátorem "actual.folder.path".

```
private static final String FOLDER_PATH = "actual.folder.path";
private static final String ROOT = "/";

public static String loadString(Context context, String key, String
    defaultValue) {
        SharedPreferences preferences =
            context.getSharedPreferences(PREFERENCES,
            Context.MODE_PRIVATE);
        return preferences.getString(key, defaultValue);
}

public static String loadFolderPath(Context context) {
        return loadString(context, FOLDER_PATH, ROOT);
}
```

Listing 6.5: Ukázka kódu pracujícího s *Shared Preferences*.

Shared Preferences přináší jednoduchý způsob, jak uložit menší množství dat perzistentně s ohledem na životní cyklus aplikace, to znamená, že takto uložená data zůstávají zachována i v okamžiku, kdy aplikace není aktivní a při jejím opětovném spuštění, jsou opět

k dispozici. K načtení aktuální cesty je využita metoda využívající *Shared Preferences*, a to *loadFolderPath*.

Posledním, ale nejdůležitějším parametrem je název adresáře, který má být vytvořen. Pro jeho získání je nutné zobrazit jednoduchý dialog, který obsahuje *EditText* pohled, do kterého může uživatel zapsat název adresáře. Tento název musí být validován tak, aby neobsahoval žádný ze zakázaných znaků.

O validaci se stará metoda *Validator.validateFolderName*, která kontroluje zadaný název nad následujícími kritérii:

1. Název adresáře nesmí být prázdný.
2. Může obsahovat bílé znaky, ale nemůže se skládat pouze z nich.
3. Některé znaky jsou v názvu adresáře zakázané, a to konkrétně ", \, |, <, >, :, *, ?, %.

```
public static final String NAME_REGEX = "^[^\\/?%*:|\"<>]+$"
if (!name.trim().isEmpty() && s.matches(NAME_REGEX)) {
        // valid folder name
}
```

Listing 6.6: Kontrola validity názvu adresáře.

Tato validace je implementována pro kritérium 2. tak, že jsou z názvu odstraněny všechny bílé znaky za pomocí metody *trim* a poté je zkontrolována jeho délka. Kritéria 1. a 3. jsou kontrolována regulárním výrazem.

V okamžiku, kdy je k dispozici validní název složky, cesta souborového systému a přístupový token, je možné formulovat dotaz na server. Pokud je dotaz, respektive volání, REST metody úspěšné, je znovu načten obsah zobrazovaného adresáře, který již obsahuje nově vytvořený adresář.

6.3.2 Pořízení fotografie

Uživatel má k dispozici možnost vytvořit fotografii přímo v aplikaci, přičemž po jejím pořízení je tato fotografie ihned uploadována na server.

```
private File createImageFile() throws IOException {
        String timeStamp = new
           SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
        String imageFileName = "Image_" + timeStamp + "_";
        File storageDir =
           context.getExternalFilesDir(Environment.DIRECTORY_PICTURES);
        return File.createTempFile(imageFileName, ".jpg",
           storageDir);
}
```

Listing 6.7: Metoda *createImageFile()* sloužící k tvorbě nového souboru pro uložení fotografie.

Ještě před zachycením fotografie je nutné připravit soubor, do kterého bude uložena. Název souboru musí být unikátní, k tomu slouží zařazení časového razítka.

Pro pořízení fotografie je využita systémová aktivita *MediaStore.ACTION_IMAGE_CAPTURE*. Ta je spuštěna za pomocí volání metody *startActivityForResult*, to znamená, že systémová aktivita fotoaparátu bude zobrazena a její mateřská aktivita čeká na výsledek. Po přijetí výsledku je v předem vytvořeném souboru uložena fotografie, kterou je možné nahrát na server.

Obdobným způsobem je implementován záznam videa s tou změnou, že je volána systémová aktivita *MediaStore.ACTION_VIDEO_CAPTURE* a připravený soubor má odlišný formát.

6.3.3 Získání náhledových obrázků

Velmi specifickou operací je načítání náhledových obrázků. Každý soubor podporovaného typu má jak v mozaikovém, tak i při zobrazení v seznamu namísto ikony zobrazen právě náhledový obrázek (typické označení bývá *thumbnail*).

K získávání obsahu jednotlivých úrovní souborového systému ze serveru dochází v klientské aplikaci poměrně často (pokaždé, když je úroveň změněna). Jedná se ovšem pouze o jednoduchý JSON objekt s minimální velikostí, tudíž to nepředstavuje vysokou zátěž na síťové připojení. Náhledové obrázky, i přes své nižší rozlišení (256×256 pixelů), ovšem při vyšším počtu souborů, představují velmi znatelnou zátěž.

Jejich opětovné načítaní by zatěžovalo síťové připojení a vedlo by k nadměrné spotřebě dat. Je tedy nutné náhledové obrázky ukládat do „cache“ paměti, tak aby nedocházelo k jejich stahování opakovaně, ale vždy pouze jednou.

Kešování a mnohé další funkcionality poskytuje knihovna *Picasso*. Tu bylo možné pro účely této aplikace téměř bez problémů využít. Jediná zásadní modifikace spočívala v implementaci vlastního *RequestHandleru*, tedy třídy, jenž poskytuje metody sloužící k tvorbě dotazu na server. V novém RequestHandleru, jehož implementace se nachází v třídě *ThumbnailRequestHandler*, je oproti defaultnímu přidána možnost vložit do dotazu na server hlavičky. Ty jsou potřebné, protože se v nich mimo jiné nachází přihlašovací token, který je nezbytný pro pro provedení libovolného dotazu na server.

6.4 Oprávnění

Operační systém Android doznal od verze 6 velké změny ve způsobu jakým jsou aplikaci přidělena oprávnění. Dříve bylo možné libovolná požadováná oprávnění vložit do souboru *AndroidManifest.xml* a uživatel všechna tato oprávnění potvrzoval ještě před samotnou instalací aplikace. Od verze 6 je nutné o některá povolení (ta pro uživatele nejcitlivější) žádat za běhu aplikace.

Zápis do externí paměti (WRITE_EXTERNAL_STORAGE) – toto oprávnění je nutné zejména pro operace stahování a při vytváření multimediálních souborů (foto, video, zvuková stopa).

Nahrávání zvuku (RECORD_AUDIO) – toto oprávnění je potřebné z důvodu, že k nahrávání zvuku dochází přímo v rámci klientské aplikace. V případě, že by k nahrávání zvuku docházelo v systémové aktivitě, stačilo by pouze oprávnění pro zápis externí paměti. Této zákonitosti je využito právě při pořizování fotografií a videa, jelikož pro tuto dvojici operací jsou využity systémové aktivity, není nutné se dotazovat na další speciální povolení.

Samotné „dotazování“ na oprávnění je implementováno zobrazením systémového dialogu, který uživatele informuje o tom, jaká oprávnění aplikaci poskytuje a jaká jsou s poskytnutím daných práv spřažena rizika.

6.5 Přehrávač videa

Funkcionalita, která má klientskou aplikaci odlišit od konkurenčních aplikací, je adaptivní streamování videa. Na straně klienta je tudíž potřebné implementovat přehrávač, který adaptivní streamování videí podporuje a umožňuje jej naplno využít.

Zvoleným přehrávačem, jenž bude upraven pro potřeby této aplikace, je *ExoPlayer*[5]. Jedná se o alternativu k defaultnímu přehrávači systému Android s názvem *MediaPlayer*.

Hlavní výhodou ExoPlayeru je jeho silná podpora pro přehrávání adaptivně streamovaných videí, a to za pomocí technologií *DASH* a *SmoothStreaming*.

Mezi další nesporné výhody patří jednoduchá upravitelnost, rozšiřitelnost a možnost aktualizace za pomocí standardních „Play store“ aktualizací. To v praxi znamená, že klientská aplikace bude mít k dispozici vždy nejnovější verzi přehrávače, a to bez nutnosti zásahu vývojáře v podobě vydávání nových aktualizací.

6.5.1 Inicializace přehrávače

Tvorba přehrávače spočívá v správném instanciování třídy *SimpleExoPlayer* a skládá ze tří základních kroků.

```
// 1. krok
BandwidthMeter bandwidthMeter = new DefaultBandwidthMeter();
TrackSelection.Factory videoTrackSelectionFactory =
    new AdaptiveTrackSelection.Factory(bandwidthMeter);
TrackSelector trackSelector =
    new DefaultTrackSelector(videoTrackSelectionFactory);

// 2. krok
LoadControl loadControl = new DefaultLoadControl();

// 3. krok
SimpleExoPlayer player =
    ExoPlayerFactory.newSimpleInstance(context, trackSelector,
        loadControl);
```

Listing 6.8: Inicializace přehrávače *SimpleExoPlayer*.

- *1. krok:* Jako první je vytvořen *TrackSelector*. Ten slouží pro volbu nejvhodnější video, respektive audio, stopy a zajišťuje tak adaptivní chování streamu. Tak, aby mohla probíhat volba nejvhodnější stopy, musíme mít k dispozici nějakou metriku, která bude určovat jakousi míru „vhodnosti“. Pro tyto účely je využit *BandwidthMeter*, který periodicky měří aktuální dostupný bandwidth a TrackSelector na jeho základě může volit stopu s vhodným bitrate.
- *2. krok:* O takzvaný „buffering“ videa se stará třída *LoadControl*. Bufferovaní videa umožňuje vyrovnat se s kolísáním kvality síťového připojení nebo dokonce s krátkými výpadky a dovoluje video přehrávat plynule i přes tyto nedostatky.
- *3. krok:* V momentě, kdy je připraven *TrackSelector* a *LoadControl*, je možné instanciovat *SimpleExoPlayer*, který bude zaštiťovat veškerou logiku přehrávače, kterou budeme v klientské aplikaci potřebovat.

6.5.2 Nastavení stopy k přehrávání

V okamžiku, kdy je vytvořen a inicializován přehrávač, je možné ho „naplnit“ obsahem. Obsahem rozumíme video připravené k adaptivnímu streamování, jenž je dostupné na unikátní URI adrese.

Pro získání URI ke konkrétnímu video souboru, je nutné zavolat REST metodu *streamGetStatus* (viz kapitola 7.3.3). Ta „vrací“ přímo požadovanou URI, anebo v případě, že video ještě nebylo zakódováno, procentuální informaci o stavu kódování.

Uživatel může v módu výběru získat URI i k více označeným video souborům. Z tohoto důvodu musíme při tvorbě stopy, respektive stop, k přehrávání počítat i s možností, že je k dispozici pole URI adres.

```
MediaSource[] mediaSources = new MediaSource[uris.length];
for (int i = 0; i < uris.length; i++) {
        mediaSources[i] = new DashMediaSource(uri,
           buildDataSourceFactory(false),new
           DefaultDashChunkSource.Factory(mediaDataSourceFactory),
           mainHandler, eventLogger);
}
MediaSource mediaSource = mediaSources.length == 1 ?
   mediaSources[0] : new ConcatenatingMediaSource(mediaSources);
player.prepare(mediaSource, !haveResumePosition, false);
```

Listing 6.9: Konstrukce objektu třídy *MediaSource*.

Výše uvedená ukázka kódu ukazuje proces transformace získaných URI adres na objekt třídy *MediaSource*, který v sobě nese veškeré potřebné informace pro přehrávání daného videa, respektive seznamu videí.

Nejprve dochází, pro každou jednotlivou URI, k transformaci na samostatný *MediaSource* objekt a posléze, pokud byla transformována více jak jedna adresa, dochází k instanciaci *ConcatenatingMediaSource*. Tento objekt umožňuje vytvořit seznam za sebou seřazených videí, mezi nimiž lze následně za pomocí ovládacích prvků přehrávače přecházet.

Pomocí metody *prepare* je připravený obsah „předán“ přehrávači.

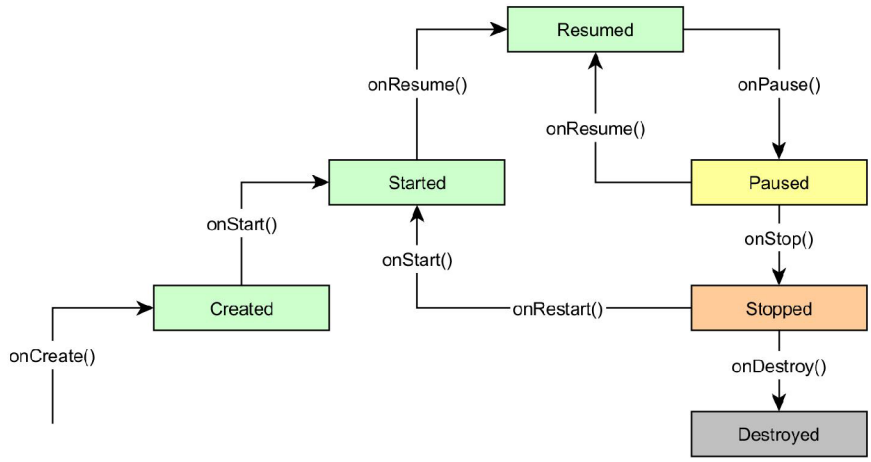
6.5.3 Pohled přehrávače

Pohled *SimpleExoPlayerView*, který zaštiťuje základní grafické rozhraní pro přehrávač, se skládá ze dvou základních částí:

- *Surface* představuje plochu, na kterou je video renderováno. To znamená, že velikost a pozice tohoto pohledu je přímo úměrná velikosti a poloze celého přehrávače.
- *PlaybackControlView* v sobě obsahuje veškeré klasické ovládací prvky nutné pro manipulaci s videem (spuštění a zastavení přehrávání, přechod na další/předchozí stopu a časová osa přehrávání).

6.5.4 Aktivita StreamPlayer

Aktivita systému Android má přesně definovaný životní cyklus[3], který je v aktivitě reprezentován voláním metod odpovídajícím jednotlivým událostem tohoto cyklu. Veškerou logiku a grafické uživatelské rozhraní přehrávače zaštiťuje aktivita *StreamPlayer*. Ta musí správně reflektovat události svého životního cyklu na chování přehrávače.



Obrázek 6.4: Životní cyklus aktivity systému Android.

V okamžiku vzniku aktivity je volána metoda *onCreate*. V ní dochází k takzvanému „napojení“ jednotlivých pohledů, to je proces, při kterém je jednotlivým pohledům z layoutu aktivity instanciován objekt, který slouží k jejich reprezentaci v kódu. Jedním z pohledů, jehož reprezentace je vytvořena, je pohled přehrávače *SimpleExoPlayerView*. Jeho reprezentace je v kódu využita například k zachycení uživatelské interakce s přehrávačem.

Metoda *onResume* je volána v okamžiku, kdy přechází aktivita do popředí, to znamená, že je viditelná pro uživatele. V této metodě dochází k inicializaci přehrávače a je nastavena stopa k přehrávání.

Při přechodu aktivity do „pozadí“, tedy v okamžiku, kdy je volána metoda *onPause*, dochází k takzvanému „uvolnění“ přehrávače. To v praxi znamená, že nad objektem přehrávače je zavolána metoda *release()* a veškeré objekty vzniklé při inicializaci přehrávače jsou nastaveny na hodnotu *null*.

Aktivita *StreamPlayer* dále implementuje rozhraní *ExoPlayer.EventListener*. Nejvýznamnější metodou tohoto rozhraní je *onPlayerError*, ta je volána, vždy když nastane při přehrávání videa chyba. Veškeré informace o chybě jsou předány metodě pomocí jejího parametru, jímž je výjimka typu *ExoPlaybackException*. Díky tomu je možné „ošetřit“ výskyt nejčastějších chyb a předejít tak pádu aplikace.

6.5.5 Spuštění aktivity StreamPlayer

V okamžiku, kdy klientská aplikace získá pomocí volání REST metody *streamGetStatus* URI adresu streamu, je tento stream reprezentován v kódu objektem třídy *UriSample*. Tato třída obsahuje metodu *buildIntent*, která slouží pro zkonstruování takzvaného „intentu“.

[3]Pro přehlednost je v diagramu 6.4 uvedena pouze zkrácená verze diagramu životního cyklu obsahující pouze nejvýznamnější události. Kompletní verze se nachází v přílohách (viz E.1).

Intent obecně slouží jako popis operace a všech jejích parametrů. V tomto případě je zkonstruován intent s následujícími parametry:

- Jako *cílová aktivita* je zvolena aktivita *StreamPlayer*.
- Akce intentu je definována konstantou *StreamPlayer.ACTION_VIEW* a specifikuje, že má být zobrazen a spuštěn streaming videa.
- Jako datový parametr je předána URI adresa manifestu.

Typicky je intent využit v kombinaci s metodou *startActivity* a slouží ke spuštění nové aktivity. V tomto případě tomu není jinak a pomocí příkazu *startActivity(uriSample.buildIntent)* lze spustit aktivitu přehrávače se streamem definovaným objektem *uriSample*.

Kapitola 7

Implementace streamovacího serveru

Streamovací server zajišťuje korektní zakódování zdrojového video souboru na video a audio soubory připravené k adaptivní streamování. Mimo to zajišťuje i komunikaci se serverem uložiště a zakódované video soubory ukládá tak, aby bylo možné jejich streamování.

7.1 Konfigurace serveru

Pro testovací účely byl využit server s následujícími „hardware“ parametry:

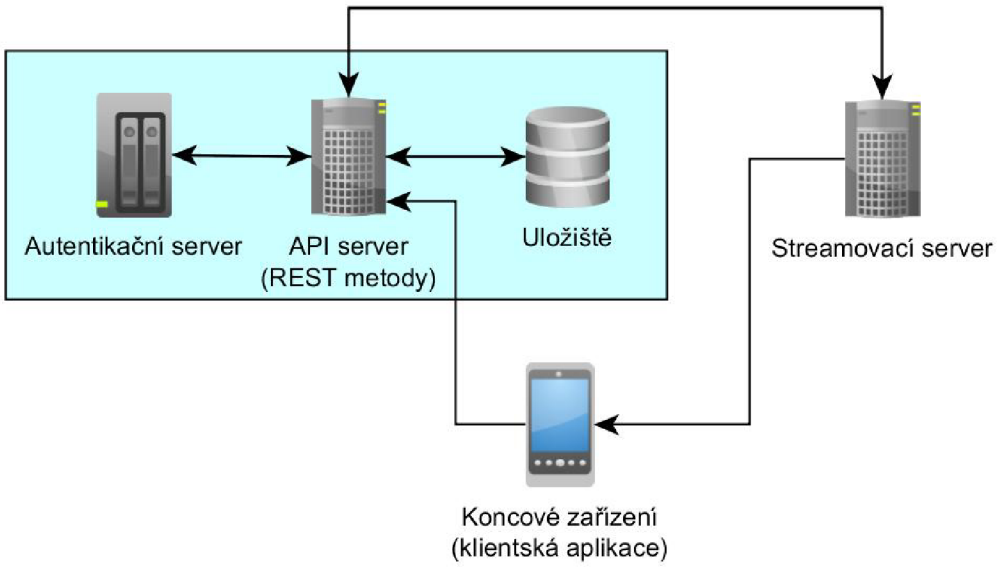
- Procesor *Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz*, který má 40 jader a na každém z nich podporuje 2 vlákna (máme tedy k dispozici 80 vláken). Vlastností procesoru, hlavně tedy počtu vláken, budeme využívat při paralelizaci kódování videa.
- Paměť RAM o velikosti 192 GB.
- 1TB SSD disk.

Na server byl „nasazen“ operační systém *Slackware 14.2 64b* a byly doinstalovány následující balíčky:

- *Java SE Runtime Environment/1.8.0_121*
- *Tomcat/8.5.12*
- *Httpd Apache/2.4.25*
- *RabbitMQ/3.6.9.*

7.2 Serverová architektura

Streamovací server nepracuje jako takzvaná „samostatná jednotka“, ale ke svému správnému fungování potřebuje infrastrukturu v podobě dalších serverů. Vazby mezi jednotlivými servery ukazuje digram 7.1. Před popisem samotného diagramu je nutné zmínit, že se jedná pouze o zjednodušenou podobu reálné architektury. Reálná síť obsahuje množství redundantních prvků, mezi nimiž dochází k load-balacingu (rozložení) zátěže.



Obrázek 7.1: Serverová architektura.

- *Autorizační server* shromažďuje všechny informace o uživatelích a jejich uživatelských účtech. Pro provedení libovolné operace na *REST serveru* je potřebné ke každému dotazu „přiložit“ platný autorizační token. Tento token lze získat právě přihlášením uživatele k autorizačnímu serveru.
- *Uložiště* je implementováno za pomocí technologie objektového uložiště *Ceph* [3]. Tudíž vystupuje jako jeden logický celek i přes to, že na hardwarové úrovni uložiště sestává z množství serverů.
- *REST server* se stará o vykonávání souborově-adresářových operací nad soubory a adresáři uloženými v uložišti. Dále zprostředkovává REST API, která umožňuje volat jednotlivé souborově-adresářové operace z klientské aplikace, a to prostřednictvím jedné z REST metod.

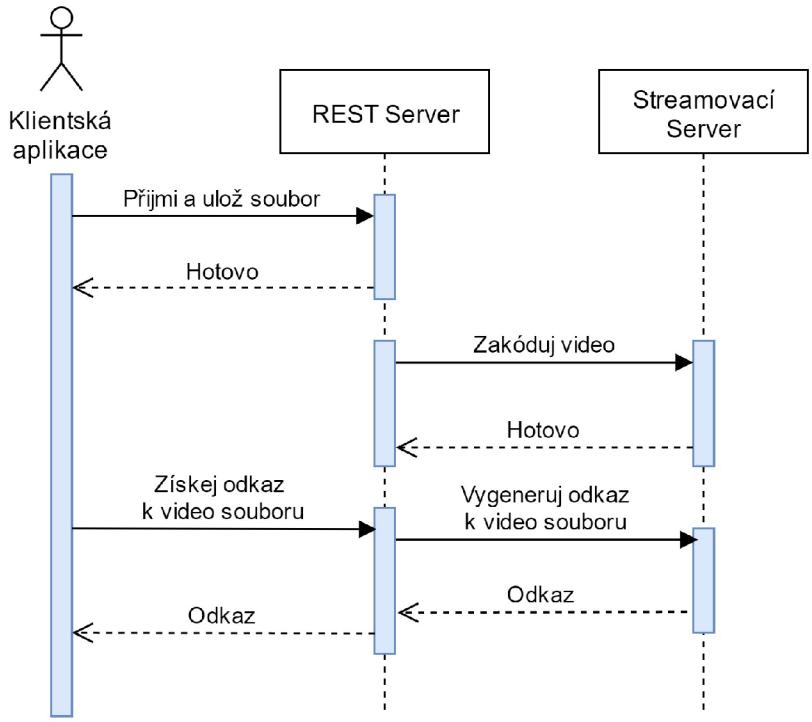
7.2.1 Model komunikace

Model komunikace nastiňuje[1] posloupnost volání a vykonávání jednotlivých operací napříč servery a klientem.

Diagram 7.2 nastiňuje typickou komunikaci při uploadu video souboru a jeho následné přípravě ke streamování. Tato komunikace se skládá ze tří elementárních kroků.

- *1. krok:* Komunikaci inicializuje klientská aplikace, ta volá metodu *uploadFile* a začíná na REST server uploadovat video soubor. Při úspěšném dokončení, ale také při selhání, je klientská aplikace informována za pomocí odeslané odpovědi.
- *2. krok:* Pokud se podařilo na REST server bezchybně uploadovat celý video soubor, tak je tento soubor uložen na úložiště a následně odeslán na streamovací server. Pro

[1] Model nepopisuje komunikace mezi REST serverem a autorizačním serverem, respektive úložištěm, a to z důvodu, že implementace REST serveru není předmětem této práce.



Obrázek 7.2: Model komunikace „Streamování“ videa.

přijetí souboru na streamovací server slouží REST metoda *uploadForStream*. V okamžiku, kdy se podaří přijmout celý zdrojový video soubor na streamovacím serveru, dochází k přípravě „adaptivního“ videa. O výsledku procesu kódování je při jeho dokončení REST server informován. Ten tuto informaci ukládá v metadatech zdrojového souboru.

- *3. krok:* V okamžiku, kdy klientská aplikace chce spustit stream k uživatelem zvolenému video souboru, musí si vyžádat URL adresu streamu. Klientská aplikace se nedotazuje přímo streamovacího serveru, ale musí dotaz směřovat k REST serveru a ten jej pouze zprostředkuje a odesílá na streamovací server. Tento zdánlivě nadbytečný krok je nutný, a to z následujících dvou důvodů:
  - Prvním důvodem je, že metody streamovacího serveru nejsou „volně“ (ze sítě internet) dostupné a ani dostupné být nemají, tak aby nemohlo dojít k jejich zneužití.
  - REST server v kombinaci s autorizačním serverem umožňuje dotazy autorizovat, a to díky tomu, že na rozdíl od streamovacího serveru má k dispozici metadata souboru. Může tedy dojít k ověření práv uživatele k manipulaci, respektive streamování, zvoleného souboru.

V případě, že je video připraveno ke streamování, je klientské aplikaci navrácen odkaz na manifest adaptivního videa. V opačném případě je navrácena procentuální informace o průběhu kódování jednotlivých kvalit video a zvukových stop.

V okamžiku, kdy má uživatelská aplikace k dispozici URL adresu na manifest probíhá dotazování na samotná data video souborů přímo na streamovací server, a to v souladu s logikou adaptivního streamování.

7.3 Implementace REST metod

Z výše nastíněného modelu komunikace vyplývá nutnost implementovat dvojici REST metod v rámci streamovacího serveru, a to *uploadForStream* a *streamGetStatus*.

7.3.1 JAX-RS & Jersey

JAX-RS [19] neboli *Java API for RESTful Web Services* je specifikace API programovacího jazyka Java, která poskytuje podporu pro vytváření takzvaných „webových“ služeb v souladu s REST architekturou.

Implementace JAX-RS, která je použita v této aplikaci, se nazývá *Jersey* [7]. Jersey je framework, který je nejen referenční implementací JAX-RS, ale přináší i přidanou funkcionalitu. Pro naše potřeby nejdůležitější přidanou funkcionalitu představuje plná podpora zpracování JSON objektů.

7.3.2 Upload souboru

Nejdůležitější REST metodou na streamovacím serveru je *uploadForStream*, ta slouží pro příjem a následné zakódování zdrojového video souboru. Její rozhraní, respektive vzniklá webová služba, je definována pomocí speciálních anotací v kombinaci s klasickou deklarací metody.

```
@POST
@Path("/uploadForStream")
@Produces("application/json" + "; charset=utf-8")
@Consumes(MediaType.APPLICATION_OCTET_STREAM)
Response uploadForStream(@HeaderParam("account") String account,
                         @HeaderParam("path") String path,
                         InputStream fileStream)
```

Listing 7.1: Definice rozhraní metody *uploadForStream*.

Z anotací lze vyčíst veškeré parametry metody *uploadForStream*. Metoda, přijímané HTTP zprávy, je *POST* a v jejím těle se nachází *APPLICATION_OCTET_STREAM*, což je MIME typ označující „proud“ binárních dat. Hlavičky obsahují dvojici parametrů, a to *account* a *path*, přičemž *account* obsahuje uživatelovo přihlašovací jméno a *path* obsahuje cestu k souboru v rámci uložiště.

URL adresa této REST metody vznikne konkatenací adresy serveru a řetězce *„/uploadForStream“*. Anotace *@Produces* nám umožňuje definovat formát odpovědi. V tomto případě se jedná o JSON objekt se zvoleným kódováním *UTF-8*.

Tak, aby mohl být prostřednictvím „proudu“ *fileStream* uložen zdrojový soubor videa na streamovací server, je nutné vytvořit (pokud již neexistuje) adresář, jehož název odpovídá přihlašovacímu jménu uživatele, který soubor uploaduje. V tomto adresáři je vytvořen soubor s názvem, který je získán z cesty obsažené v hlavičkách. Název souboru je stejný jako název odpovídajícího souboru na serveru uložiště. Do nově vytvořeného souboru jsou

ukládána data, která jsou čtena z „proudu“ *fileStream*. Dochází tedy k „ozrcadlení“ souboru z uložiště na streamovací server.

Po úspěšném přijetí souboru následuje odeslání požadavku na *kódování* do frontovacího systému (viz kapitola 7.5). Následně je odeslána odpověď směrem k REST serveru, ta informuje o úspěchu uploadu.

7.3.3 Stav video streamu

Druhou REST metodou streamovacího serveru je *streamGetStatus*. Ta slouží pro získání aktuálního stavu kódování videa, což vyžaduje zvážení následujících situací:

- Pokud má být získán status pro soubor, který nikdy kódován nebyl, je vrácen průběh kódování se záporným procentuálním stavem.
- V případě, že pro daný soubor kódování započalo, ale není dokončeno, je odeslána odpověď obsahující stav kódování jednotlivých kvalit video a audio stop.
- Poslední možností, která může nastat, je situace, kdy máme k dispozici video a audio stopy zakódované ve všech kvalitách a je možné na dotaz odpovědět pomocí odkazu na manifest adaptivního streamu.

7.4 Kódování

Klíčovou operací streamovacího serveru je *kódování videa a zvukové stopy* neboli proces, při kterém je originální zdrojové video kódováno na videa a zvukové stopy rozdílné kvality, které společně s vygenerovaným manifestem tvoří adaptivní video stream.

7.4.1 Konfigurace kódování

Veškerá logika kódování je napsána univerzálně, to znamená, že lze jednoduše konfigurovat počet a parametry jednotlivých kódovaných videí a zvukových stop.

Konfiguraci kvality videa lze definovat pomocí objektu třídy *VideoEncodingConfig*. Ten obsahuje klasické parametry pro kódování videa, jako je počet snímků za vteřinu, bitrate a rozlišení, je zde ovšem možné nastavit i počet vláken, na kterých má být video kódováno.

Označení	**Bitrate (Kbit/s)**	**Počet snímků za vteřinu**	**Rozlišení (px)**
240p	250	24	352x240
360p	500	24	480x360
480p	750	24	858x480
720p	1500	24	1280x720

Tabulka 7.1: Sada kvalit pro kódování videa.

Sadu kvalit, do kterých je tedy zdrojové video kódováno, představuje pole objektů *VideoEncodingConfig*. Pro testovací účely byla zvolena *základní* sada kvalit (viz tabulka 7.1).

Zvuková stopa byla při testování kódována pouze jedna, a to s následujícími parametry:

- Bitrate: 128 Kbit/s.
- Vzorkovací frekvence: 48 KHz.

Co se týká použitých kodeků a kontejneru při implementaci, byl respektován návrh aplikace, což znamená, že pro zakódování videa byl použit kodek *ffvp9* a pro kódování zvuku *Opus*. Kontejner použitý pro uchování video a zvukové stopy byl zvolen taktéž v souladu s návrhem, tedy *WebM*.

7.4.2 FFmpeg

FFmpeg [31] je sada open-source knihoven, které slouží pro zpracování multimediálních dat. Konkrétně knihovna *libavcodec* byla pro účely této práce hojně využita, protože právě ona se stará o kódování nejen videa, ale i zvukové stopy.

Tak, aby bylo možné zapsat kód konfigurující a spouštějící kódování jak videa, tak zvukové stopy přímo v jazyce Java byl využit *ffmpeg-cli-wrapper* [2]. Ten slouží jako rozhraní pro spuštění FFmpeg příkazů na příkazové řádce za pomocí příkazů jazyka Java.

7.4.3 Kódování videa

Před tím, než je možné spustit kódování videa, je nutné vytvořit objekt třídy *FFmpegBuilder*. Ten slouží pro nastavení všech parametrů kódování.

Jelikož je nutné vytvořit instanci *FFmpegBuilder* pro každou kvalitu videa, byla vytvořena metoda *createVideoBuilder*, která jako parametr přijímá objekt třídy *VideoEncodingConfig* a podle konfigurace, kterou obsahuje, vytvoří odpovídající *FFmpegBuilder*.

```
new FFmpegBuilder()
  .setInput(sourceFilePath)
  .addOutput(sourceFilePath + vConfig.getFileNameSuffix())
  .overrideOutputFiles(true)
  .disableAudio()
  .disableSubtitle()
  .setVideoCodec("libvpx-vp9")
  .setVideoBitRate(vConfig.getBitrate())
  .setVideoFrameRate(vConfig.getFramesPerSecond(),1)
  .setVideoResolution(vConfig.getWidth(),vConfig.getHeight())
  .addExtraArgs("-dash", 1)
  .addExtraArgs("-threads", vConfig.getNumberOfThreads())
  .done();
```

Listing 7.2: Konstrukce objektu třídy *FFmpegBuilder*.

Na výše uvedené ukázce kódu lze vidět způsob, jakým je *FFmpegBuilder* instanciován. Většina kódu je samovysvětlující, ale přesto, hlavně kvůli jeho důležité roli, budou jeho jednotlivé části v následujícím textu popsány.

- Pomocí *setInput* definujeme vstupní soubor obsahující data originálního videa. Dvojice *addOutput* a *overrideOutputFiles* naopak slouží k definici výstupního souboru kódování, a to s tím specifikem, že pokud již soubor existuje, bude přepsán. Název výstupního souboru vzniká konkatenací původního názvu vstupního souboru a takzvaného *FileNameSuffix*. *FileNameSuffix* je řetězec pevně daného formátu, který je unikátní pro každou kódovanou kvalitu videa. Jeho formát má následující strukturu: *„-ŠÍŘKA_VIDEAxVÝŠKA_VIDEA-BITRATE.webm“*.

[2] Knihovnu lze nalézt na https://github.com/bramp/ffmpeg-cli-wrapper.

- Jelikož výstupem kódování má být pouze video, explicitně nastavíme, že nechceme zvukovou stopu ani titulky, a to pomocí *disableAudio* a *disableSubtitle*.
- Dále je nutné nastavit kodek, který bude využit pro kódování videa, a to následujícím způsobem *setVideoCodec(„libvpx-vp9“)*.
- Následuje nastavení trojice parametrů definující samotnou kvalitu videa a to bitrate, počet snímků za vteřinu a rozlišení videa.
- Parametr "-dash", nastavený na hodnotu 1, zajistí, že výstupní video soubor bude kompatibilní se standardem DASH.
- Posledním parametrem, který konfigurujeme, je počet vláken, na kterých bude kódování spuštěno, a to pomocí přepínače *„-threads“*

Poté co je vytvořena instance *FFmpegBuilder*, je spuštěno samostatné vlákno, ve kterém bude kódování probíhat. Proces kódování je spuštěn pomocí metody *executeTaskWithListener*, ta umožňuje zaregistrovat takzvaný *„listener“*. Listener je instance objektu implementující rozhraní *ProgressListener*, jehož metoda *progress* nám umožňuje sledovat pokrok při konverzi videa. V okamžiku, kdy hodnota „pokroku“ dosáhne 100%, víme, že proces kódování videa skončil a je možné ukončit běh vlákna pomocí metody *join*.

7.4.4 Kódování zvukové stopy

Proces kódování zvukové stopy je velice podobný tomu pro kódování videa. Jedinou významnou změnou vycházející z požadavků na kódování audia je nastavování vzorkovací frekvence. Při tvorbě objektu třídy *FFmpegBuilder* je vzorkovací frekvence nastavena pomocí metody *setAudioSampleRate*. V dalších aspektech je proces kódování téměř totožný.

7.4.5 Generování manifestu

V okamžiku, kdy se podaří zakódovat všechny požadované kvality videa a zvukové stopy, tak je nutné vytvořit takzvaný *manifest*. Manifest je soubor v XML formátu, který shromažďuje informace o všech dostupných kvalitách tak, aby klientská aplikace mohla vykonávat adaptivní streamování.

Z důvodů nedostatečné podpory použité knihovny *ffmpeg-cli-wrapper* nebylo možné využít její API k vygenerování korektního manifest souboru. Řešení tohoto problému předcházela nutnost pochopit to, jakým způsobem knihovna pracuje. Ta, s uživatelem instanciovaného a hodnotami naplněného objektu třídy *FFmpegBuilder*, vytvoří takzvaný *ImmutableList*, který představuje seznam parametrů příkazu příkazové řádky *ffmeg*. Příkaz s parametry uvedenými v seznamu poté vykoná na příkazové řádce, a to za pomocí metody *run()* objektu *FFmpeg*, který je součástí knihovny.

Ovšem při tvorbě manifestu knihovna *ffmpeg-cli-wrapper* vytváří nevalidní *ImmutableList*. Při jeho tvorbě dochází k záměně pořadí argumentů a tak vzniká nespustitelný příkaz.

Metoda *buildManifestCommandAndRun* slouží k vyřešení výše popsaného problému a implementuje tvorbu korektního *ImmutableList*. Její parametry představuje dvojice listů, které obsahují konfigurace video a audio stop. Na základě těchto konfigurací je zkonstruován *ImmutableList*, který obsahuje validní parametry, a to ve správném pořadí. Poté už je postup stejný jako u klasického využití knihovny *ffmpeg-cli-wrapper*, dochází tedy k vykonání příkazu na příkazové řádce, a to za pomocí metody *run()*.

7.4.6 Zpřístupnění souborů

Veškerá zakódovaná videa, zvukové stopy a vygenerované manifesty jsou uloženy v adresářích, jejichž názvy odpovídají přihlašovacím jménům uživatelů, jenž přípravu daných videí pro streamování iniciovali.

Všechny tyto adresáře se nachází v adresáři *„/var/www/htdocs/“*, který je webovým serverem (Apache/2.4.25) zpřístupněn ze sítě Internet. Veškerá data v tomto adresáři obsažená lze získat pomocí protokolu HTTP, což je podmínkou pro správné fungování adaptivního streamování.

7.5 Frontovací systém

Na server může v krátkém časovém úseku přijít velké množství požadavků na přípravu streamu, které nelze neprodleně „odbavit“. Z tohoto důvodu bylo nutné zavést takzvaný *frontovací systém*, který slouží pro řazení jednotlivých požadavků do fronty a zajišťuje jejich postupné odbavování.

7.5.1 RabbitMQ

RabbitMQ [11] je nástroj, který umožňuje implementovat na serveru zasílání zpráv a vše s touto problematikou spojené. Pro lepší představu je možné *RabbitMQ* přirovnat k fungování klasické pošty. Dopis je vložen do schránky, pošťák ho následně doručí na poštu, kde je dopis setříděn a uložený čeká na příjemce.

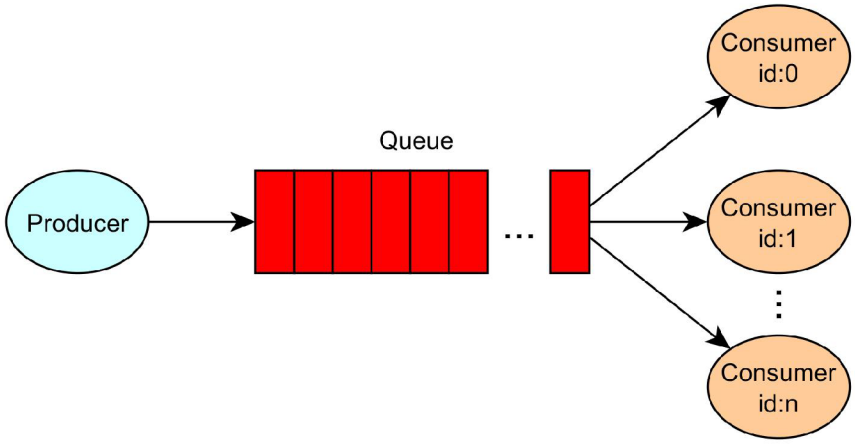
Obdobně bude pracovat i *RabbitMQ*. Uživatel zasílá požadavek na server, zde je požadavek setřízen a uložen do fronty. Ve chvíli, kdy je příjemce, tedy proces zajišťující kódování připraven dopis převzít, je tento požadavek k němu doručen.

7.5.2 Model RabbitMQ

RabbitMQ je velmi flexibilní technologie, která poskytuje množství řešení pro množství typických problémů a situací. V této práci byl zvolen návrhový vzor nazývaný *Work Queues* neboli „pracovní fronty“. Využití tohoto vzoru vede k vyhnutí se nutnosti vykonávat časově náročné operace neprodleně. Díky němu můžeme při úplné obsazenosti serveru naplánovat vykonání operace později.

Základem vzoru *Work Queues* je následující trojice komponent (viz obr. 7.3):

- *Producer* neboli producent je v našem případě REST metoda *uploadForStream*, ta po úspěšném dokončení uploadu souboru odesílá požadavek na kódování videa. Tento požadavek je reprezentován zprávou, která je zařazena do fronty.
- *Queue*, tedy fronta, slouží pro uchování zpráv, které jsou v ní řazeny pomocí systému *FIFO*.
- Poslední součástí je skupina *konzumentů* (consumers), kdy každý jeden z nich zajišťuje přípravu streamu pro jedno zdrojové video. Každý konzument, pokud není zaneprázdněn, kontroluje obsah fronty a pokud není prázdná, vybírá z ní zprávu. Přijetím zprávy začíná „práce“ konzumenta a ten nemůže přijímat další zprávy, a to do doby než práci dokončí.



Obrázek 7.3: Model vzoru *Work Queues*.

Počet konzumentů a velikost fronty je teoreticky neomezená. Omezení plynou pouze z omezení systému, na kterém je technologie RabbitMQ nasazena. V případě serveru, na kterém probíhalo testování, byl nastaven fixní počet konzumentů na 20. Probíhat tudíž mohlo 20 souběžných kódování. Velikost fronty nijak explicitně omezena nebyla.

7.5.3 Formát zpráv

Zprávy, zasílané pomocí RabbitMQ, mají textový formát, tudíž prostřednictvím nich nelze jednoduše přenášet objekty či jiné složitější jazykové struktury, ale pouze řetězec. Bylo tedy nutné zadefinovat pevný formát zpráv a vytvořit pomocné metody, které zprávy umí vytvářet, ale také zpětně zpracovat a rozdělit na původní položky. Pro formátování zpráv byla zadefinována jednoduchá pravidla:

- Každá zpráva začíná řetězcem určujícím její typ. Hodnoty řetězců pro jednotlivé typy jsou obsaženy ve výčtové třídě *MessageTypes*.
- Jednotlivé položky zprávy jsou odděleny jednoznakovým oddělovačem, který je určen konstantou *RABBIT_MQ_MESSAGE_SEPARATOR* (typicky se jedná o znak „:“).

Na serveru se setkáme se třemi druhy zpráv:

- *CONVERT_JOB* – zpráva tohoto typu slouží pro odeslání požadavku na přípravu konkrétního videa pro streamování. Ve svém těle obsahuje řetězec zprávy *unikátní id*, které je přiřazeno danému procesu kódování a plnou cestu ke zdrojovému videu.
- *STATUS_REQUEST* – pokud je zpráva tohoto typu, tak se jedná o dotaz na aktuální stav kódování. Zpráva obsahuje *unikátní id* procesu kódování, které slouží pro jeho identifikaci.
- V případě, že z nějakého důvodu musí dojít k přerušení procesu kódování v jeho průběhu je odeslána zpráva typu *ABORT_TASK*. Ta obsahuje id procesu, který má být neprodleně ukončen.

Třída *MessageHandler* obsahuje sadu statických metod právě pro generování, respektive zpracování, validních zpráv všech typů. Jedná se tedy o rozhraní fronty RabbitMQ , do níž budou vkládány pouze zprávy formátované metodami třídy *MessageHandler*.

7.5.4 Implementace producenta

Třída *RabbitMQProducer* implementuje veškerou logiku producenta, jenž se sestává ze dvou základních činností.

Před samotným začátkem produkce (zasílání zpráv do fronty) je nutné instanciovat objekty všech potřebných RabbitMQ tříd, jmenovitě *Connection* a *Channel*. Ty uchovávají například informace o adrese RabbitMQ serveru („localhost“), anebo soubor vlastností použité fronty.

Co se týče produkce, je zásadní metoda *sendMessage*, ta se stará o zaslání zprávy do fronty a kontroluje, zdali byla zpráva úspěšně frontou přijata.

7.5.5 Implementace konzumenta

Konzumentem rozumíme objekt třídy *RabbitMQConsumer*. Tato třída má jednu významnou metodu *handleDelivery*, ta je volána v okamžiku, kdy konzument vybírá z fronty zprávu. Přepsáním (označením anotací *@Override*) této metody můžeme definovat její nový obsah a tím implementovat chování konzumenta.

Jako první je nutné pomocí metody *MessageHandler.parseMessage()* rozdělit řetězec přijaté zprávy na jednotlivé položky. Poté je možné vyčíst z první položky typ přijaté zprávy.

V případě, že byla přijata zpráva typu *CONVERT_JOB*, je pomocí metody *createConvertJob* vytvořen takzvaný *convertJob*, což je objekt stejnojmenné třídy *ConvertJob*. Tento objekt zajišťuje vše, co se týče samotné konverze videa zvukové stopy a generování manifestu, přesně tak, jak bylo popsáno výše (viz kapitola 7.4).

Pokud byla přijata zpráva typu *ABORT_TASK*, dochází k neprodlenému zastavení přípravy streamu a dealokování všech použitých zdrojů. Poslední možností je přijmutí zprávy typu *STATUS_REQUEST*, ta ve svém těle obsahuje unikátní id *ConvertJob* objektu, díky němuž je možné získat aktuální procentuální průběh dané konverze.

Nezávisle na tom, jaký typ zprávy byl přijat, dochází vždy k informování fronty a producenta o tom, že byla daná zpráva z fronty vybrána a zpracována, a to za pomocí takzvané *ACK* zprávy.

Kapitola 8

Testování

Aplikace prezentovaná v rámci této práce byla vyvíjená pro společnost *Master Internet* a je určená k publikaci na distribučním kanálu *Play Store*, tedy na oficiálním obchodu s aplikacemi pro mobilní platformu Android, bylo tudíž nutné aplikaci důkladně otestovat. Při testování byly zvoleny dva postupy, a to jak automatické, tak manuální testování.

8.1 Automatické testování

Často se opakující testovací scénáře, je vhodné testovat pomocí automatických testů. Lze tak podstatně snížit časové nároky na testování.

8.1.1 Espresso

Pro potřeby automatického testování byl využit nástroj *Espresso*. Ten je vestavěn přímo do vývojového prostředí *Android Studio* a umožňuje generovat testy pomocí grafického rozhraní. Pomocí tohoto nástroje byla testována jednoduchá interakce s aplikací. Problematické ovšem bylo testovat asynchronní úkony.

V případě, že aplikace pro provedení daného úkonu vyžaduje komunikaci se serverem, což bylo velmi časté pro mnoho testovacích scénářů, nebylo využití nástroje espreso spolehlivé. Nástroj byl v době vypracování této práce ve vývoji a neměl vestavěnou podporu pro takzvané „čekání“ na asynchronní události. To bylo tedy řešeno pomocí pevně zadaného časového intervalu, po který se na výsledek asynchronní operace čekalo.

8.1.2 Monkey

Nástroj *Monkey* pracuje na principu generování pseudonáhodné sekvence takzvaných „uživatelských událostí“, tedy kliknutí, doteků a gest. Tuto sekvenci pak vykonává nad spuštěnou aplikací.

Typická délka testovací sekvence bývá v řádech tisíců událostí, a tudíž je tento způsob testování vhodný pro konečnou fázi testování, jelikož vyžaduje vysokou míru spolehlivosti aplikace ke svému úspěšnému průběhu. Aplikace si musí při testování, pomocí nástroje *Monkey*, „poradit“ s enormním zahlcením událostmi, které nemají žádnou logickou návaznost.

8.2 Manuální testování

Díky infrastruktuře poskytnuté společností *Master Internet* bylo možné provést rozsáhlé uživatelské testování. Velkou výhodou tohoto přístupu k testování je možnost ladit aplikaci nejen z hlediska spolehlivosti, ale také uživatelské přívětivosti.

K testování docházelo za pomoci systému *GitLab* v kombinaci s verzovacím systémem *Git*. Typický průběh testování byl následující:

1. V okamžiku, kdy byla při testování nalezena chyba, selhání či grafický nedostatek, je tento problém popsán v takzvaném *issue*. Issue lze chápat jako krátkou textovou poznámku, která je přiřazena konkrétnímu projektu. Tato poznámka v sobě obsahuje detailní popis problému a soupis zařízení, na kterých k němu docházelo.

2. Před samotnou implementací opravy problému popsaného v issue je vytvořena *větev* ve verzovacím systému. Tato větev má název, jenž odpovídá názvu issue, kterou řeší.

3. Na nově vytvořené větvi je implementována oprava.

4. Před sjednocením (operace *merge*) větve s opravou do hlavní větvě projektu dochází k opětovnému testování, které kontroluje, zdali došlo k opravě problému. Pokud se podařilo problém vyřešit, je sjednocení vykonáno.

Kapitola 9

Závěr

V rámci této práce byla nastíněna problematika streamování videa a její základní principy a terminologie. Byly popsány technologie a nejznámější protokoly, které lze využít právě pro implementaci streamování videa. V souladu se zadáním práce byl kladen důraz na výběr nejoptimálnějších technologií a protokolů tak, aby byly co nejlépe zohledněny specifika mobilní platformy.

Dále byl popsán návrh výsledné aplikace, a to jak klientské, tak serverové části. Navržena byla nejen grafická podoba aplikace, ale také požadovaná funkčnost jednotlivých součástí a celková architektura systému.

Následoval popis procesu implementace, při kterém byly dodrženy principy a postupy nastíněné v části návrhu aplikace. Výsledkem implementace je mobilní aplikace operačního systému Android, která byla pečlivě otestována a je připravena k publikaci na oficiálním obchodu s aplikacemi *Google Play*. Dále vznikla implementace serveru zajišťujícího vše potřebné pro adaptivního streamování. Tato implementace byla otestována na testovacím serveru a je připravena k nasazení na produkční server.

V kapitole 8 byl popsán proces testování, který ověřil a umožnil zajistit „připravenost“ aplikace k produkci.

9.1 Možná rozšíření aplikace

Další rozvoj by měl spočívat hlavně v implementaci nových funkcionalit mobilní aplikace, a to například:

- možnost sdílet soubory a adresáře jiným uživatelům,
- galerie multimediálních souborů (fotky, videa, hudba),
- podpora streamování hudebních souborů,
- vestavěný prohlížeč fotografií.

Dalším možným směrem vývoje by byla implementace *webové* aplikace umožnící uživateli využívat veškerou funcionalitu i mimo platformu *Android*.

Literatura

[1] Activity. https://developer.android.com/reference/android/app/Activity.html.

[2] AVC/H.264 FAQ. http://www.mpegla.com/main/programs/AVC/Pages/FAQ.aspx.

[3] Ceph. http://ceph.com/ceph-storage/.

[4] ExoPlayer. https://developer.android.com/guide/topics/media/exoplayer.html.

[5] ExoPlayer - Developer guide. https://google.github.io/ExoPlayer/guide.html.

[6] HEVC Advance Patent List. http://www.hevcadvance.com/pdfnew/HEVC_Patent_List_161213v2.pdf.

[7] Jersey. https://jersey.java.net/.

[8] Material design guidelines. https://material.io/guidelines/.

[9] MediaCodec. https://developer.android.com/reference/android/media/MediaCodec.html.

[10] MPEG LA HEVC Patent List. http://www.mpegla.com/main/programs/HEVC/Documents/hevc-att1.pdf.

[11] RabbitMQ. https://www.rabbitmq.com/documentation.html.

[12] Supported Media Formats. https://developer.android.com/guide/topics/media/media-formats.html.

[13] H.264 Advanced video coding for generic audiovisual services. Technická zpráva, 2003.

[14] Results of the public multiformat listening test. http://listening-test.coresv.net/results.htm#list2, July 2014.

[15] WebM Container Guidelines. https://www.webmproject.org/docs/container/, 2016.

[16] Bankoski, J.; Koleszar, J.; Quillio, L.; aj.: VP8 Data Format and Decoding Guide. RFC 6386, Internet Engineering Task Force, November 2011, doi:{10.17487/RFC6386}.

[17] Braden, R.: Requirements for Internet Hosts - Communication Layers. RFC 1122, Internet Engineering Task Force, October 1989, doi:{10.17487/RFC1122}.

[18] van den Branden Lambrecht, C.: *Vision Models and Applications to Image and Video Processing*. Springer US, 2013, ISBN 9781475734119.

[19] Burke, B.: *RESTful Java with JAX-RS 2.0*. OŔeilly Media, November 2013, ISBN 9781449361341, 1-392 s.

[20] Dan Grois, A. M., Detlev Marpe; aj.: Performance Comparison of H.265/MPEG-HEVC, VP9, and H.264/MPEG-AVC Encoders. Technická zpráva, 2013.

[21] Dan Grois, T. N., Detlev Marpe; Hadar, O.: Comparative Assessment of H.265/MPEG-HEVC, VP9, and H.264/MPEG-AVC Encoders for Low-Delay Video Applications. Technická zpráva, 2014.

[22] (DVB), D. V. B.: MPEG-DASH Profile for Transport of ISO BMFF Based DVB Services over IP Based Networks. July 2016.

[23] Dvořák, R.: *Fenomén licencí Open Source*. Diplomová práce, Právnická fakulta Masarykovy univerzity, Brno, 2010.

[24] Fielding, R.; Gettys, J.; Mogul, J.; aj.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, June 1999, doi:{10.17487/RFC2616}.

[25] Ford, A.; Raiciu, C.; Handley, M.; aj.: TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, Internet Engineering Task Force, January 2013, doi:{10.17487/RFC6824}.

[26] Grangeand, A.; aj.: VP9 Bitstream & Decoding Process Specification. Technická zpráva, March 2016.

[27] INTERNATIONAL TELECOMMUNICATION UNION: H.265 High efficiency video coding. Technická zpráva, 2015.

[28] ISO/IEC: Information technology — Generic coding of moving pictures and associated audio information — Part 7: Advanced Audio Coding (AAC) . Technická zpráva, September 2004.

[29] Juurlink, B.; aj.: *Understanding the Application: An Overview of the H.264 Standard*. Springer New York, 2012, ISBN 978-1-4614-2229-7.

[30] Konstantin Miller, G. G., Emanuele Quacchio; Wolisz, A.: Adaptation Algorithm for Adaptive Streaming over HTTP. Technická zpráva.

[31] Korbel, F.: *FFmpeg Basics: Multimedia handling with a fast audio and video encoder*. CreateSpace Independent Publishing Platform, December 2012, ISBN 9781479327836, 1-216 s.

[32] Noé, A.: Matroska File Format. Technická zpráva, January 2009.

[33] Painter, T.; Spanias, A.: Perceptual Coding of Digital Audio. Technická zpráva, 2000.

[34] Pinson, M. H.; Wolf, S.: Comparing subjective video quality testing methodologies. Technická zpráva, 2003, doi:{10.1117/12.509908}.

[35] Pinson, M. H.; Wolf, S.: Perceptual visual quality measurement techniques for multimedia services over digital cable television networks in the presence of a reduced bandwidth reference. Technická zpráva, 2008.

[36] Postel, J.: User Datagram Protocol. RFC 0768, Internet Engineering Task Force, August 1980, doi:{10.17487/RFC0768}.

[37] Postel, J.: Transmission Control Protocol. RFC 0793, Internet Engineering Task Force, September 1981, doi:{10.17487/RFC0793}.

[38] Raissi, R.: The Theory Behind Mp3. 2002.

[39] Samet, H.: The Quadtree and Related Hierarchical Data Structures. *Computing Surveys*, , č. 16, June 1984.

[40] Schulzrinne, H.; Casner, S.; Frederick, R.; aj.: RTP: A Transport Protocol for Real-Time Applications. RFC 3550, Internet Engineering Task Force, July 2003, doi:{10.17487/RFC3550}.

[41] Stewart, R.: Stream Control Transmission Protocol. RFC 4960, Internet Engineering Task Force, September 2007, doi:{10.17487/RFC4960}.

[42] Stockhammer, T.: Dynamic Adaptive Streaming over HTTP – Design Principles and Standards. Technická zpráva.

[43] TECHNICAL, E.: *Information Paper on HDTV Formats* . EBU TECHNICAL, 2010.

[44] Valin, J.; Vos, K.; Terriberry, T.: Definition of the Opus Audio Codec. RFC 6716, Internet Engineering Task Force, September 2012, doi:{10.17487/RFC6716}.

[45] Vatolin, D.: MSU Lossless Video Codecs Comparison. Technická zpráva, MSU Video Group, 2007.

[46] Wiegand, T.; Sullivan, G. J.: The H.264/AVC Video Coding Standard. Technická zpráva, March 2007.

[47] Yusra A. Y. Al-Najjar, D. C. S.: Comparison of Image Quality Assessment:PSNR, HVS, SSIM, UIQI. *International Journal of Scientific & Engineering Research*, ročník 3, č. 8, 2012.

Přílohy

Příloha A

Obsah CD

Přiložené CD obsahuje zdrojový kód mobilní a serverové aplikace. Dále také samotnou diplomovou práci ve formátu *PDF* a její kód v LaTeX. Adresářová struktura je následující:

- *android_app* zdrojový kód Android aplikace,
- *latex_src* text práce,
- *server_app* zdrojový kód serverové aplikace,
- *dp_xspurn04.pdf* soubor diplomové práce.

Příloha B

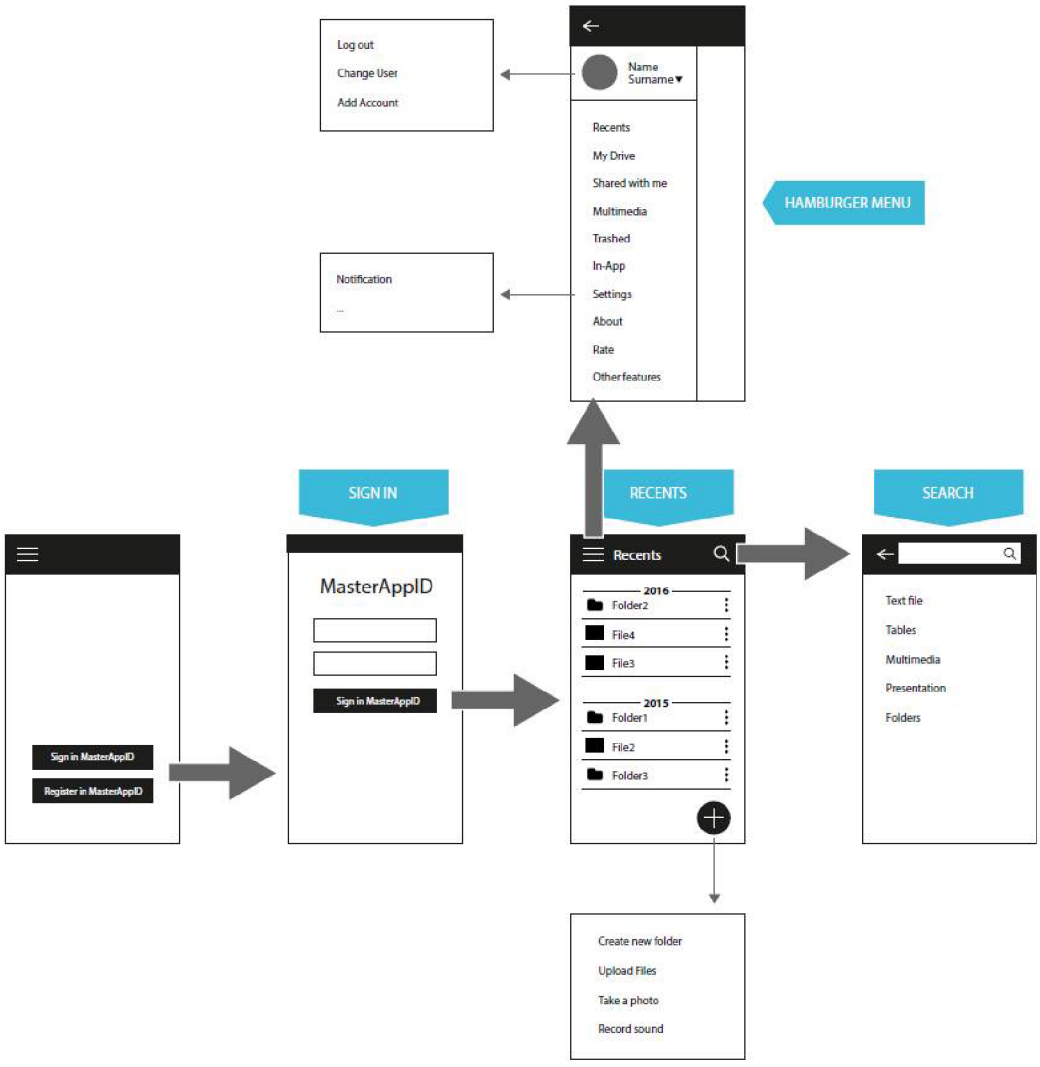
Navržené API

URI sufix	HTTP metoda	Popis
/folders	POST	Vytvoří nový adresář v zadané cestě.
/folders	PUT	Přejmenuje adresář v zadané cestě a kontroluje validitu nového jména.
/folders	DELETE	Odstraní adresář v zadané cestě a to i s veškerým jeho obsahem.
/folders/content	POST	Vrací JSON objekt obsahující veškeré informace o obsahu adresáře.
/folders/copy	POST	Zkopíruje adresář i s jeho obsahem do zadaného cílového adresáře.
/folders/move	POST	Přesune adresář i s jeho obsahem do zadaného cílového adresáře.
/files	POST	Slouží pro nahrání souboru na server (tzn. upload).
/files	DELETE	Odstraní soubor v zadané cestě.
/files	GET	Slouží pro stáhnutí souboru ke klientovi (tzn. download).
/files/change_name	PUT	Přejmenuje soubor v zadané cestě a kontroluje validitu nového jména.
/files/copy	POST	Zkopíruje soubor do zadaného cílového adresáře.
/files/move	POST	Přesune soubor do zadaného cílového adresáře.
/sharing/link	POST	Vrací veřejný odkaz na stáhnutí souboru.
/sharing/unlink	POST	Zneplatní veřejný odkaz na stáhnutí souboru.
/files/thumbnail	POST	Vrací náhledový obrázek pro zadaný soubor.

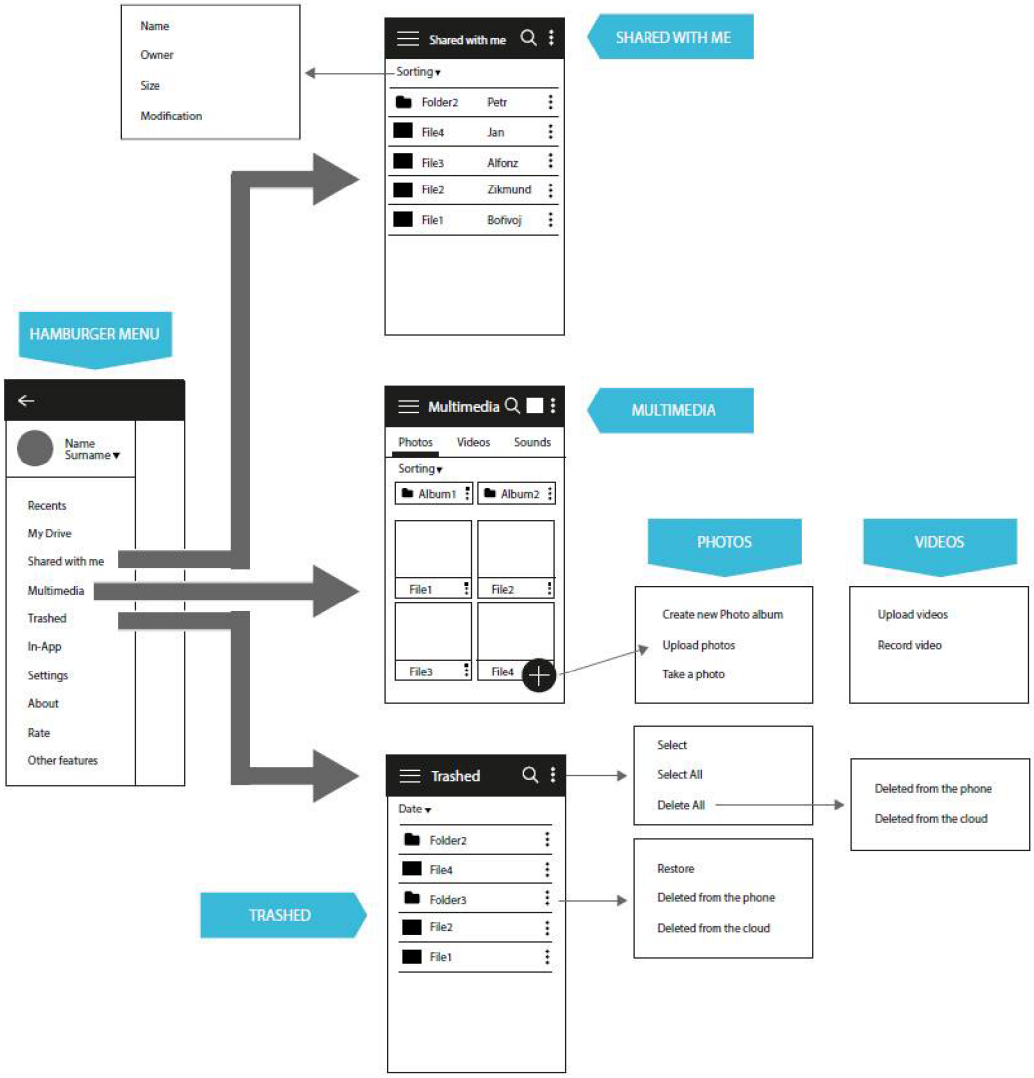
Tabulka B.1: Přehled REST metod vykonávajících souborové a adresářové operace.

Příloha C

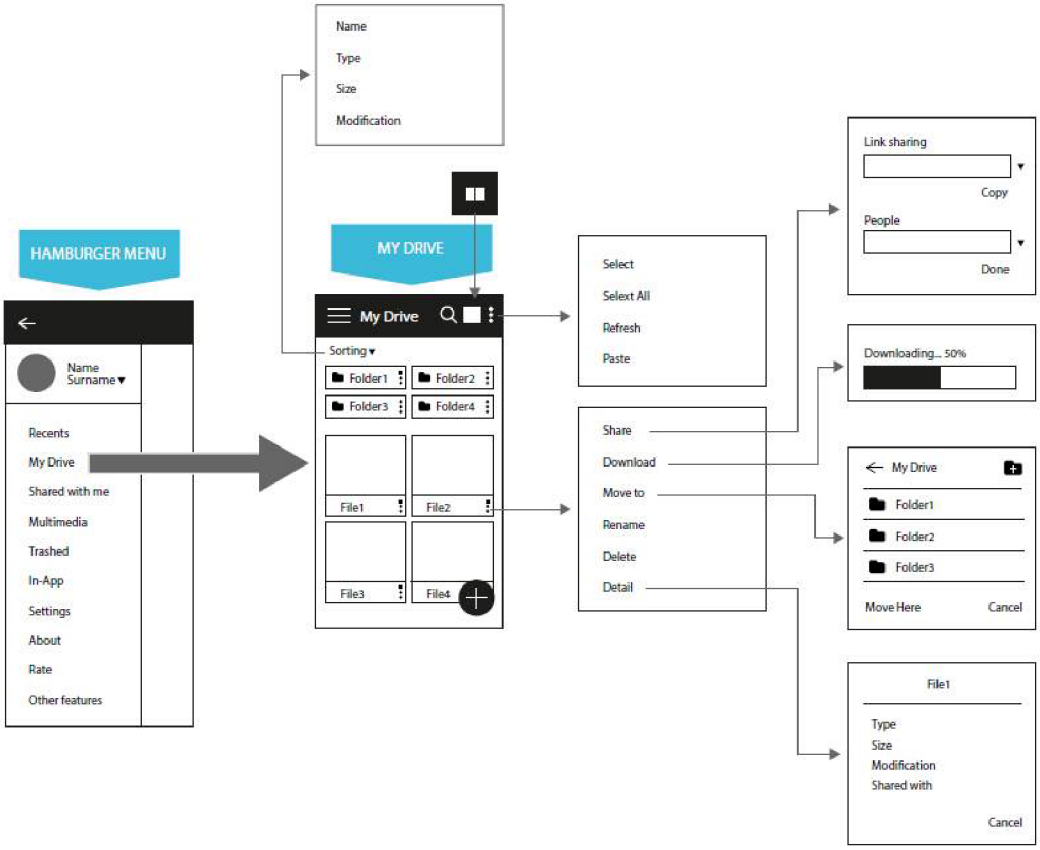
Drátový model



Obrázek C.1: Drátový model první část.



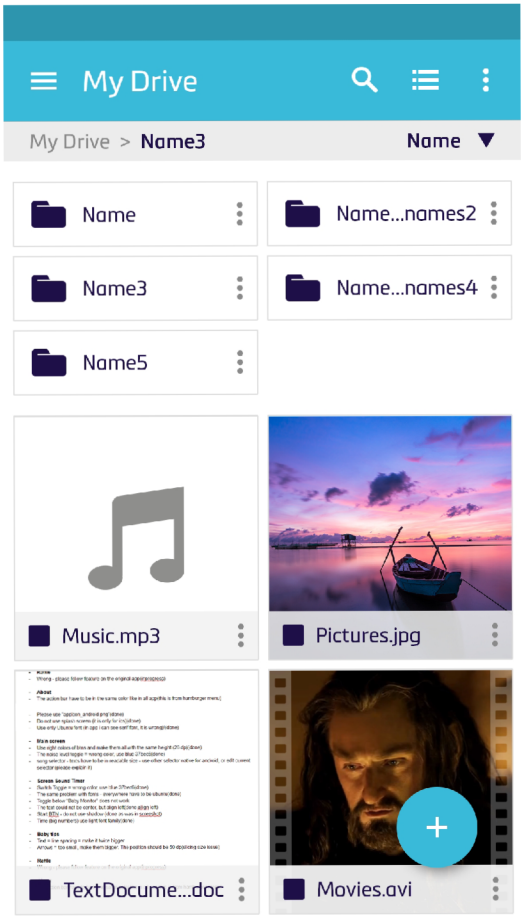
Obrázek C.2: Drátový model druhá část.



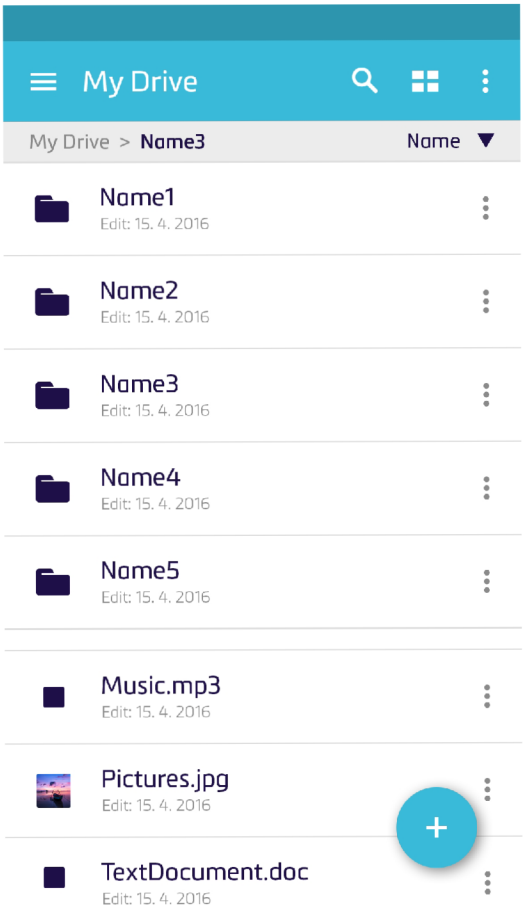
Obrázek C.3: Drátový model třetí část.

Příloha D

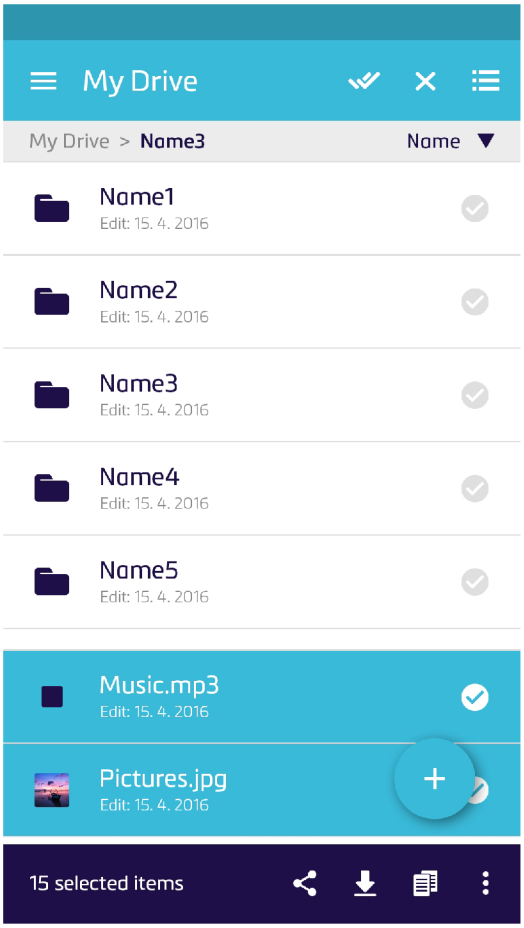
Grafický návrh aktivit



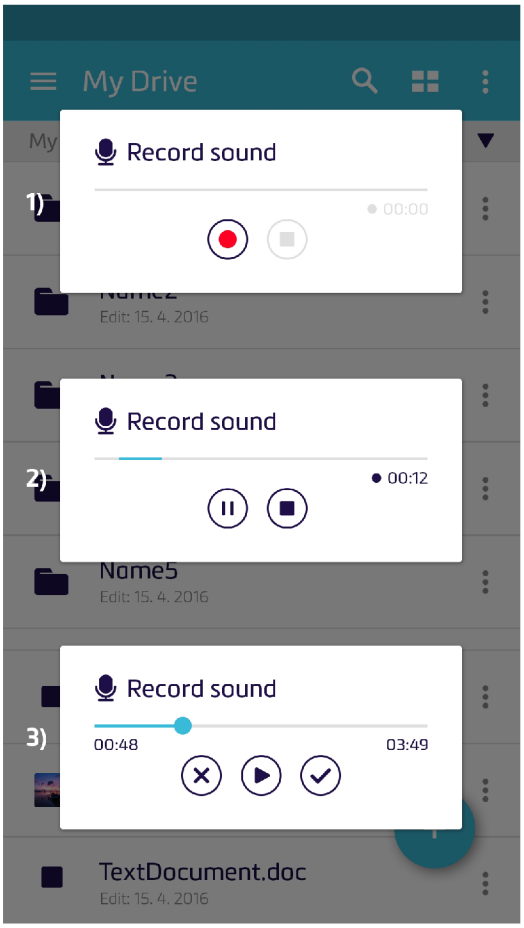
Obrázek D.1: Zobrazení souborů a adresářů v mozaice.



Obrázek D.2: Zobrazení souborů a adresářů v seznamu.



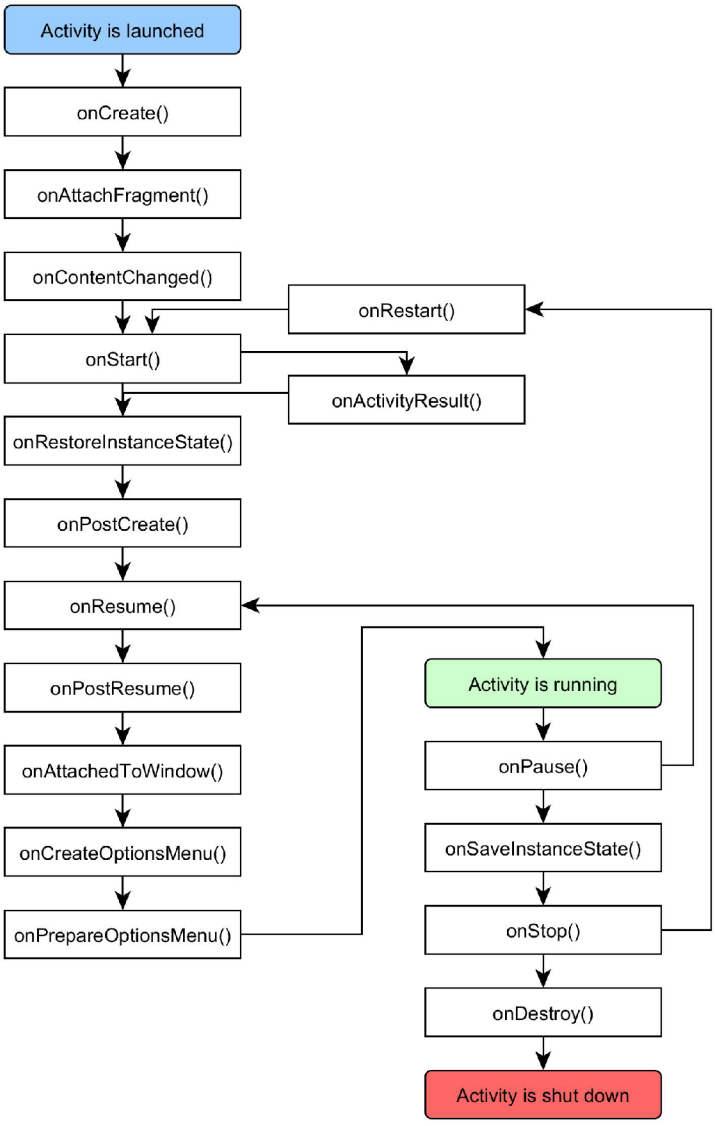
Obrázek D.3: Zobrazení souborů a adresářů v seznamu s aktivním módem výběru.



Obrázek D.4: Dialog pro nahrávání zvuku.

Příloha E

Životní cyklus aktivity



Obrázek E.1: Diagram životního cyklu aktivity systému Android.