Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Rekonfigurace mobilní aplikace na základě kontextu uživatele

Diplomová práce

Autor: Bc. Ondřej Schneider

Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Hradec Králové srpen 2021

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 13.8.2021 Ondřej Schneider

Poděkování:

Děkuji doc. Ing. Filipu Malému, Ph.D. za odborné vedení při zpracování této diplomové práce.

Anotace

Cílem této práce je studium možností rekonfigurace mobilní aplikace na základě změn v kontextu uživatele, konkrétně na základě změn v datové propustnosti sítě. Nejprve jsou představeny pojmy důležité k pochopení problematiky, mezi tyto pojmy se řadí síťové architektury klient-server a peer-to-peer, dále tenký a tlustý klient a v neposlední řadě modularita jakožto základní prostředek pro rekonfiguraci. Následně je představen koncept tzv. hybridního klienta. Součástí tohoto konceptu je návrh vzorové mobilní aplikace a centrálního serveru coby komunikačního prostředníka sloužícího k podpoře některých funkcí hybridního klienta. Cílovou platformou pro návrh a následný vývoj klienta je operační systém Android OS. Teoretický koncept je následně implementován a otestován na několika zařízeních. Výsledkem je plně funkční systém, jenž řeší problematiku dynamické rekonfigurace za běhu aplikace takovým způsobem, aby nebyl narušen chod aplikace a negativně ovlivněn cílový uživatel systému. Navržený hybridní klient reaguje na změny v datové propustnosti sítě, což přináší celou řadu využití v praxi. Koncept je nicméně možné dále rozšiřovat či upravit tak, aby reagoval v podstatě na jakýkoliv podnět z uživatelského kontextu daného mobilního zařízení.

Annotation

Title: Configuring mobile application based on the user's context

The aim of this work is to study the possibilities of reconfiguration of a mobile application based on changes in the user's context, specifically based on changes in the data throughput of the network. First, concepts important for understanding the issue are introduced, these concepts include client-server and peer-to-peer network architectures, then thin and thick client and, last but not least, modularity as a main tool of reconfiguration. Subsequently, the concept of the hybrid client is introduced. This concept includes design of mobile application and central server used to support some functions of the hybrid client. The target platform for the design and subsequent development of the client is the Android OS. The theoretical concept is then implemented and tested on several devices. The result is a fully functional system that solves the issue of dynamic reconfiguration while the application is running in such way that the application is not

disrupted and the target user of the system is not negatively affected. The proposed hybrid client responds to changes in data throughput of the network, which brings a wide range of uses in practice. However, the concept can be further extended or modified to respond to any stimulus from the user's context of a given mobile device.

Obsah

1 Úvod

Důležitost této diplomové práce vyplývá ze stále se zvyšujícího počtu mobilních aplikací, kdy platí, že velká část těchto aplikací je závislá na internetovém připojení, neboť fungují na bázi komunikace architektury klient-server. Počet mobilních aplikací, jež jsou dostupné ke stažení v hlavních obchodech s aplikacemi – tzv. *„app-stores“*, přesáhl ve čtvrtém kvartálu roku 2020 šest milionů a tři sta tisíc (1). Stejně tak neustále stoupá počet mobilních zařízení využívajících internetové připojení, kdy dle (2) tento počet přesáhl v roce 2021 již 4,32 miliard. Dokonce platí, že tento způsob využití internetu, tedy prostřednictvím mobilních zařízení, stoupá na úkor užívání internetové sítě tradičními desktopy. Podle (3) je v roce 2021 až 90 % mobilního připojení (průzkum byl prováděn v USA) využíváno právě v mobilních aplikacích, a nikoliv v internetovém prohlížeči, což vypovídá o velké důležitosti vyžití internetu v souvislosti s používáním mobilních aplikací.

V současnosti většina obydleného území má k dispozici velmi kvalitní síťové pokrytí. Nicméně stále se mohou vyskytnout místa (často v přírodě či méně osídlených oblastech) s horší dostupností mobilní sítě, kdy může dojít až k úplné ztrátě připojení. S touto problematikou se nabízí následující otázky: Co když internetové připojení není k dispozici? Je možné zajistit alespoň částečnou funkcionalitu mobilních aplikací při nedostatečném připojení k mobilní síti, jsou-li funkce aplikací závislé na serveru, který je k dispozici pouze skrze tuto síť? Tyto otázky nabývají velkého významu především v dnešní době, kdy se objevují termíny jako *internet věcí*, *všudypřítomné* a *všude dostupné informace a služby*, *chytrá prostředí*, *zařízení* či *chytré interakce*. Všechny tyto relativně stále nové technologické směry kladou velké nároky na mobilní aplikace a zařízení a rovněž vyvolávají závislost na internetovém připojení, které se zdá být nezbytným prvkem. Je tedy velmi důležité, aby mobilní aplikace byly schopny vykonávat svoji činnost i v omezených podmínkách jako je oblast s horším přístupem k datům apod. Tato problematika by se však neměla týkat pouze stavu datové propustnosti sítě, ale obecně *kontextu uživatele* mobilního zařízení, neboť mobilní zařízení dnes nabízí velkou variabilitu dat související s uživatelským kontextem, kdy tato data jsou získána z velkého množství různých senzorů. Informace z dat potom mohou být využity k různým účelům souvisejících s danou problematikou, které jsou rovněž probírány v této práci. Jedním ze

stěžejních senzorů je lokalizační modul například v podobě *GPS modulu*, pomocí něhož lze s určitou přesností lokalizovat polohu uživatelova zařízení. Takováto informace může být využita například k analýze datové propustnosti na základě zeměpisných souřadnic, kdy je možné vytipovat místa či oblasti, ve kterých se dají předpokládat horší možnosti připojení a je možné k tomu mobilní aplikaci uzpůsobit. Tedy připravit ji na fungování v *„offline“* režimu, a to za pomoci určité rekonfigurace aplikace takovým způsobem, aby byly zachované funkce aplikace i bez připojení k serveru.

Mobilní aplikace mohou dnes poskytovat dokonce velmi kritická data a mohou sloužit i v oblasti zdravotnictví. Jeden takový příklad uvádí studie (4), v níž je zmíněna aplikace, která monitoruje stav pacienta pomocí mnoha externích senzorů a zasílá tyto data na nemocniční server, kde doktor má možnost získávat ze zaslaných dat důležité informace a mít tak přehled o zdravotním stavu pacienta. Je jasné, že takováto aplikace je závislá velmi významně na datovém připojení a těžko by mohla fungovat v nějakém „offline“ režimu, tedy rekonfigurovat se tak, aby mohla fungovat bez připojení k síti. Nicméně zde by se dalo uvažovat o možnosti *peer-to-peer komunikace*, které se rovněž tato práce věnuje. Bylo by totiž možné nalézt jiné zařízení v okolí jakožto přístupový bod sítě, které má lepší podmínky připojení a prostřednictvím tohoto zařízení komunikovat se serverem. I tento případ vypovídá o důležitosti zabývat se tématem rekonfigurace aplikace v souvislosti se stavem připojení k internetové síti.

2 Cíl práce

Všechny zmíněné otázky a problémy v úvodu vedou k jedné nejzásadnější otázce celé této práce: *Je možné aplikaci dynamicky rekonfigurovat, tedy měnit její funkcionalitu za běhu, na základě změn v kontextu uživatele?*

Primárním cílem této práce je tedy navrhnout a následně implementovat aplikaci, která by byla schopna z běžného tenkého klienta v architektuře klient-server vlastní rekonfigurací vytvořit klienta tlustého, jenž bude schopný zajišťovat funkce aplikace v případě nedostatečné datové propustnosti. Jinak řečeno získá určité funkce serveru tak, aby mohl v omezené míře fungovat bez přístupu k serveru. Výsledkem pak je modulární aplikace, která je schopna aplikační moduly zavádět, spouštět a ukončovat dynamicky za běhu aplikace. Tato aplikace by ale také měla být schopna v určitých situacích server zastoupit. To znamená poskytovat služby serveru ostatním tenkým klientům v okolí, jež ztratili připojení se serverem. Vzhledem k tomu, že zmíněné požadavky jsou poměrně komplexní, je vhodné krom samotné aplikace současně navrhnout celou architekturu, která umožní těmto hybridním klientům navzájem komunikovat a spolupracovat při případné rekonfiguraci na základě změny jejich kontextu. Pro tyto účely tato diplomová práce vychází z teoretického konceptu navržené architektury v práci (5). Je jasné, že návrh a implementace hybridního klienta je ovlivněna použitou technologií a platformou, pro kterou je aplikace vyvíjena. Cílem tedy je také nalézt a vyřešit (je-li to možné) technologická omezení, která se vážou s vývojem navrženého systému, a především s vývojem cílové mobilní aplikace jakožto hybridního klienta. Pro účely této práce a pro vývoj cílové aplikace byl vybrán Android OS jakožto nejpoužívanější operační systém pro mobilní zařízení (6) a implementační technologie Java a Android SDK.

3 Teoretická část

Teoretická část je rozdělena do dvou segmentů. První se věnuje základním pojmům, které mají zásadní význam pro tuto diplomovou práci. Druhý je zaměřen na architekturu Android OS, neboť, jak již bylo zmíněno, právě tento operační systém byl vybrán v prakticko-teoretické a praktické části pro návrh a vývoj hybridního klienta se schopností rekonfigurace za běhu na základě změn uživatelského kontextu.

3.1 Základní pojmy

Nejprve je vhodné osvětlit základní pojmy. Veškeré pojmy zmíněné v této kapitole jsou dále používány v celé diplomové práci a mají pro ni zásadní význam.

3.1.1 Typy klientů

Klienta lze chápat jako program či aplikaci běžící na operačním systému určitého zařízení (zpravidla desktop či mobilní zařízení) (7). Klient je schopný vykonávat specifickou množinu funkcí a poskytovat tak uživateli konkrétní službu či více služeb. K tomu, aby splnil svůj účel musí takový klient komunikovat se serverem, případně s různými webovými službami prostřednictvím internetové sítě – zde se uvádí architektura klient-server (viz kapitola *3.1.2.1 Architektura klient-server*). Komunikačními protokoly pro přenos dat a zasílání zpráv mezi klientem a serverem jsou nejčastěji HTTP a jeho bezpečnější verze HTTPS (7). Klienti však mohou komunikovat i sami mezi sebou – v tomto smyslu se hovoří o architektuře či komunikaci peer-to-peer (viz kapitola *3.1.2.2 Architektura peer-to-peer*). Na základě chování a nabízených funkcí či služeb lze rozdělit klienty na tenké, tlusté, případně hybridní.

3.1.1.1 Tenký klient

Aplikace představující tenkého klienta bývá navržena tak, aby měla malou velikost a snadnou implementaci, která se většinou týká získávání dat ze serveru a odesílání dat na server. Veškeré zpracování dat a složitější implementační mechanismy včetně naprosté většiny aplikační logiky jsou součástí implementace serveru, se kterým tenký klient komunikuje skrze síť (viz kapitola *3.1.2.1 Architektura klient-server*). (8, 9) Stejně tak veškerá data jsou uložena na serveru (7). Krom komunikace se serverem by hlavním úkolem klienta mělo být zobrazení dat a interakce s koncovým uživatelem, tedy

měl by ztělesňovat prezentační vrstvu (8). Koncept tenkého klienta umožňuje zjednodušit síťovou architekturu klient-server či její design, dále zlepšit výkon a snížit náklady – vzhledem k jednoduchosti a snadné implementaci není vyžadován výkonný hardware na straně klientů (7). Naopak z jednoduchého konceptu plyne hlavní slabá stránka tohoto typu klienta a tou je úplná závislost na serveru, neboť právě ten obsahuje zásadní implementační části a veškerá data, bez čehož je takto navržený klient v podstatě funkčně nepoužitelný.

3.1.1.2 Tlustý klient

V modelu tlustého klienta je rovněž klientská aplikace zodpovědná za úlohy prezentační vrstvy a komunikaci se serverem. Nicméně klientská část je zodpovědná také za zpracování a transformaci dat před zobrazením dat uživateli, zatímco server mívá relativně jednoduchou implementaci, jež spočívá zpravidla ve správě přístupu k datům. Lze tak říci, že součástí klienta je krom prezentační také významná část aplikační vrstvy (tlustý klient může mít i vlastní lokální uložiště dat, které však bývá kapacitně více omezeno), kdežto server bývá zodpovědný pouze za datovou komponentu – ukládání a údržba dat. (8) Tlustý klient díky tomu není tak závislý na serveru, neboť komunikace je omezena jen na předávání dat či informací týkajících se datového uložiště na serveru (9). Důsledkem chování takového klienta však bývá větší velikost a zároveň náročnost na hardware klientského zařízení.

3.1.1.3 Hybridní klient

Označení hybridní klient je pro tuto práci naprosto zásadní, neboť hlavním cílem je pokusit se takový model klienta navrhnout a implementovat. Proto je důležité uchopit pojem vhodným způsobem pro účely diplomové práce. Hybridní klient se poměrně často zmiňuje v souvislosti se síťovými architekturami. Studie (10) představuje návrh hybridní architektury, která kombinuje prvky architektur klient-server a peer-to-peer. Návrh je podrobněji rozebrán v kapitole *3.1.2.3 Hybridní architektura.* Nicméně souvislost se síťovými architekturami není zcela vhodné uchopení pojmu hybridního klienta pro záměry této práce. Mnohem vhodnější se zdá být náčrt významu, jenž je zmíněný v článku (11). Hybridní klient je zde představen jako model klienta, v jehož chování a implementaci jsou vlastnosti jak tenkého, tak tlustého klienta.

V práci (5) je představen koncept hybridního klienta v podobě tzv. *„m klienta“*. M klient může být představen i jako samostatný typ klienta, nicméně pro jednodušší pochopení problematiky byl v této práci zařazen do skupiny hybridních klientů a to proto, že se může chovat jako tenký i tlustý klient. Takovýto klient se za běžných podmínek chová jako tenký klient architektury klient-server. Avšak je připraven svoji podobu změnit a rekonfigurovat se na základě určitých významných událostí. Tyto události jsou vyvolány změnou kontextu uživatele, nejčastěji změnou v datové propustnosti sítě, skrze kterou m klient komunikuje se serverem. Dojde-li k vyvolání události je klient schopný se dynamicky rekonfigurovat (způsob jakým rekonfiguraci činí je uveden v kapitole *4.3 Návrh architektury hybridního klienta*, konkrétně v podkapitole *4.3.3 Modul dynamického zavádění knihoven*) ze stavu tenkého klienta do podoby klienta hybridního či dokonce tlustého. V tomto případě je schopný fungovat buď zcela offline, tedy bez přístupu k serveru nebo je schopný server zastoupit a poskytnout služby serveru jiným klientům, kteří rovněž ztratili spojení se serverem.

Koncept m klienta, respektive jeho klíčové vlastnosti byly použity rovněž v cílové aplikaci v této diplomové práci. Konkrétní způsob chování navrženého klienta je blíže specifikován v prakticko-teoretické části, a to v kapitole *4.1.1 Podoby a role hybridního klienta*. Toto paradigma zdá se být nejvhodnějším uchopením modelu hybridního klienta pro záměr této diplomové práce a hybridní klient bude dále takto uvažován.

Koncept tloustnutí a hubnutí

V souvislosti s hybridním klientem je možné vytyčit dva důležité pojmy – tloustnutí a hubnutí hybridního klienta. Proces ztloustnutí spočívá v rozšíření funkcionalit klienta, kdy z tenkého klienta postupně vzniká určitá forma klienta tlustého. V praxi tento proces spočívá v zavádění modulů do aplikace. Opakem je proces zhubnutí, při kterém klient ztrácí určité schopnosti a funkce, respektive jsou z aplikace odstraňovány aplikační moduly s přidanou funkcionalitou. Z tlustého klienta se tak může stát opět klient tenký. Tento koncept vychází z práce (5) a je klíčový pro definici chování cílové vzorové hybridní aplikace.

3.1.2 Typy síťových architektur

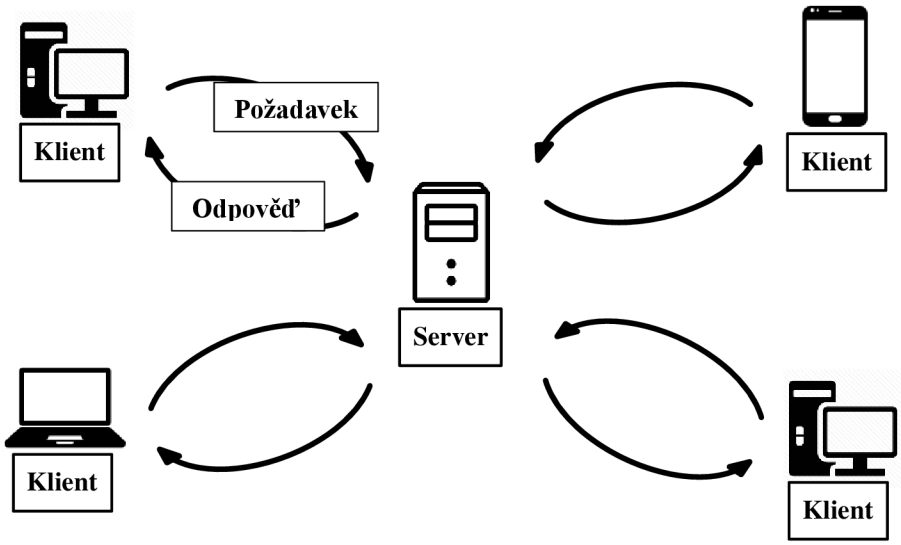
Síťová architektura je chápána jako logické a fyzické uspořádání komponent a služeb (procesů) v počítačové síti. Komponenty sítě tvoří klienti, přenosové zařízení, software a komunikační protokoly či samotná infrastruktura (kabelová nebo bezdrátová) přenosu dat mezi komponentami. Služby jsou představovány funkcemi a možnostmi jednotlivých komponent. (8, 12) Architektura poskytuje sadu modelů, které popisují nejen rozvržení různých systémů, ale také pomáhají identifikovat výhody a nevýhody daného nasazení a zda je nasazení vhodné pro konkrétní množinu aplikací (8). V (8) je síťová architektura označována pod názvem *„architektonický styl distribučních systémů“*.

V následujících kapitolách budou představeny dvě stěžejní síťové architektury – klient-server a peer-to-peer. Dále bude ilustrováno, že oba tyto přístupy lze kombinovat a vytvořit jakousi hybridní architekturu na bázi principů obou architektur.

3.1.2.1 Architektura klient-server

Architektura klient-server je velmi populární ve všemožných distribuovaných systémech a je vhodná pro širokou škálu aplikací. Jak je znázorněno na *obrázku 1*, model se skládá ze dvou komponent – klienta a serveru. Tyto dvě komponenty spolu interagují prostřednictvím počítačové sítě za využití daného komunikačního protokolu. (8) Centrálním prvkem a jádrem celé architektury je server v podobě zpravidla velmi výkonného počítače, jehož úkolem je spravování a poskytování prostředků či služeb každému klientovi, který si o ně zažádá (12). Obecně platí, že o tyto služby žádá více klientů a server musí být přizpůsoben efektivně obsluhovat veškeré příchozí požadavky od různých klientů. Tato myšlenka má významný vliv na design serveru a klientů. Hlavní úlohou klienta je tedy zasílání požadavků na server a následně přijímání či zpracování odpovědí ze serveru. Kdežto server musí být schopný naslouchat požadavkům od klientů a na jejich základě zasílat příslušné odpovědi. (8) V architektuře klient-server je nastavena určitá míra odpovědnosti obou komponent. Ta se odvíjí od schopností a možností klienta vykonávat příslušné funkce, kdy rozlišujeme dva základní typy klientů (8) – tenký a tlustý, případně jejich kombinaci – hybridní. Typy klientů jsou důkladněji popsány v kapitole *3.1.1 Typy klientů*.

Model klient-server je ideální použít ve scénáři *one-to-many*, kde lze informace a služby centralizovat a přistupovat k nim prostřednictvím jediného přístupového bodu a to serveru (8). Výhody takového konceptu jsou: Možnost přistupovat na server kdekoliv a na různých platformách; žádné omezení na malý počet výpočetních stanic; centrální řízení prostředků a zabezpečení dat na serveru. Nevýhodou naopak může být situace, při níž dojde k výpadku centrálního serveru a tím pádem k výraznému omezení funkcionalit celého systému. Architektura se rovněž může stát poměrně nákladná vzhledem k nutnosti serveru a různých síťových zařízení jako jsou směrovače, přepínače a mnoho dalších, které zajišťují spojení mezi serverem a klienty. S tímto problémem souvisí i personál potřebný k efektivní údržbě a zajištění funkcí serveru. (12)

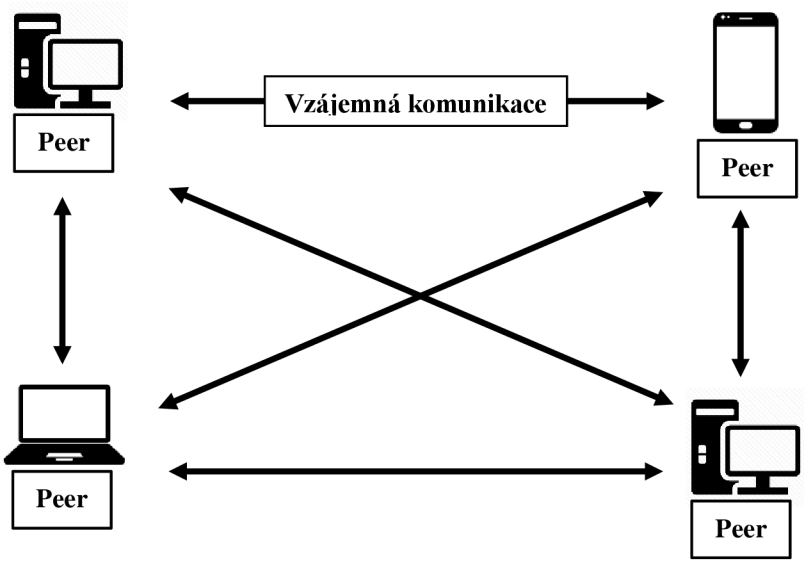


Obrázek 1 - Schéma architektury klient-server

3.1.2.2 Architektura peer-to-peer

Model peer-to-peer, jenž je znázorněný na *obrázku 2*, zavádí symetrickou architekturu, ve které neexistuje žádná hierarchie a všechny komponenty zvané *peers* jsou tak považovány za rovnocenné, díky čemuž mají stejné schopnosti a možnosti ve využívání dostupných zdrojů v počítačové síti (8, 12). V kontextu s architekturou klient-server se dá říci, že komponenta modelu peer-to-peer, tedy každý peer, zahrnuje funkce jak klienta, tak i serveru. Přesněji řečeno chová se jako server, neboť naslouchá a zpracovává požadavky ostatních klientů (peerů) v síti a zároveň jako klient, protože stejně tak může požadavky ostatním klientům (peerům) zasílat. (8)

Vzhledem k tomu, že v této formě síťové architektury neexistuje žádný centrální prvek a všechny komponenty jsou si rovnocenné, je vhodným využitím vysoce decentralizovaný systém, který lze lépe škálovat podle dimenze počtu komponent, respektive peerů (8). Z takto nastíněného modelu plynou následující přednosti: Není vyžadován žádný centrální server, což znamená, že systém je méně nákladný; pokud jeden peer přestane fungovat, není narušena funkčnost ostatních zařízení v síti. Na druhé straně koncept peer-to-peer architektury přináší i některé problémy: Zabezpečení a zálohování dat je nutné provádět na každém jednotlivém zařízení (peeru); a v neposlední řadě, zvyšuje-li se počet zařízení, pak narůstá i počet problémů spojených s výkonem, zabezpečením a přístupem u jednotlivých komponent systému. (12)



Obrázek 2 - Schéma architektury peer-to-peer

3.1.2.3 Hybridní architektura

Oba zmíněné typy síťových architektur je možné v případě potřeby kombinovat a vytvořit tak síť na bázi prvků a vlastností obou architektur. Jeden z příkladů, jakým způsobem formovat a použít hybridní architekturu je popsán ve studii (10). Cílem studie bylo navrhnout systém či architekturu umožňující spolupráci několika týmů z různých oborů pro vývoj *BIM (Building Information Model* neboli *Informační model budovy)*. Architektura měla být navržena tak, aby umožnila kontrolu nad BIM všem týmům a zároveň členům každého týmu pracovat souběžně a kooperativně.

Jádrem architektury je globální server, ke kterému jsou připojeny hostitelské počítače každého týmu označené jako lokální servery. V každém týmu tedy musí být

jeden hostitelský počítač sloužící jako místní server, který je připojen ke globálnímu serveru pro výměnu a aktualizaci návrhových dat v BIM a pro zachování konzistence dat v týmu. Spojení mezi specializovanými týmy je tak založeno čistě na bázi architektury klient-server. Propojení členů ve stejném týmu je však realizováno strukturovanou peer-to-peer sítí. Díky takto organizované spolupráci, mohou členové jednoho týmu pracovat souběžně, zatímco každý tým pracuje v jeden okamžik pouze na své části BIM modelu, která mu byla přidělena na základě specializace daného týmu. (10)

3.1.3 Modularita

Modularitu lze považovat za jeden ze základních principů programování (13). Je určena především ke kontrole složitosti softwarového systému, neboť umožňuje logické rozdělení softwarového designu tak, aby byl lépe pochopitelný a snadno udržovatelný (13, 14). V modularizaci je užito přístupu *„rozděl a panuj“*, kdy dochází k rozložení celistvého systému na sadu volně spojených, avšak soudržných modulů. Nejčastějšími principy takového rozložení jsou: Lokalizace souvisejících dat a metod; rozložení systému na části se zvládnutelnou velikostí; vytvoření acyklických závislostí a omezení závislostí na minimum. (14) Každý modul, jakožto součást systému, by měl mít ideální velikost – například v souvislosti s modularitou v *OOP (objektově orientované programování)* by třída neměla mít příliš mnoho metod či příliš komplikovanou implementaci (14).

Problémem při takto definovaném procesu modularity může být stále se zvyšující počet závislostí v aplikaci či systému při přidávání dalších nových modulů (15). Ne vždy je rovněž žádoucí, aby systém byl závislý na přítomnosti všech modulů. Alternativou je potom vytváření zcela nezávislých modulů – viz. následující kapitola *3.1.3.1 Modulární aplikace*.

3.1.3.1 Modulární aplikace

Výsledkem procesu modularity je modulární aplikace, která se skládá z několika implementačních částí. Cílem této práce je, aby aplikační části či moduly rozšiřovaly funkcionalitu aplikace. Čili aplikace by měla být schopna fungovat i bez rozšiřujících modulů a neměla by na nich být, jakkoliv závislá. Stejně tak samotné moduly by měly být na sobě vzájemně nezávislé. Díky tomuto designu nezávislých modulů, lze aplikaci dynamicky rekonfigurovat. Jinými slovy lze definovat chování aplikace přidáváním či

odebíráním nezávislých aplikačních celků. Ty jsou označovány různými způsoby – nejčastěji se používají označení jako *modul*, *plugin (plug-in)*, *balíček (bundle)* či *doplněk (add-on)* (15). Další nespornou výhodou dle (15) pak je nezávislý vývoj aplikace a všech modulů. Nenastávají tak již problémy s implementačními změnami, kdy by změna v jednom modulu mohla narušit funkčnost ostatních modulů či samotné aplikace. Takto definovaný typ architektury modulární aplikace je blíže specifikován v (15).

3.1.4 Kontext uživatele

Uživatelský kontext je jeden ze základních konceptů v tzv. *Ubiquitous computing* (*všudypřítomné výpočty*) (16). Pojem má souvislost se zvyšujícími se schopnostmi mobilních zařízení či mobilních telefonů. Ty jsou dnes označovány pod pojmem *smartphone* a mají krom funkčních operačních systémů a spousty dalších funkcí rovněž vestavěné senzory. Mezi senzory patří: GPS modul (lokalizace), kamera (snímání obrazu), mikrofon (snímání zvuku), světelný senzor, senzory pro snímání tlaku a teploty, dále snímače směru (gyroskop) a snímače pohybu (akcelerometr). Dají se připojit i další externí senzory, jež nejsou v zařízení vestavěny. Smartphony mají rovněž schopnost tato data efektivně získávat a dále zpracovávat. Při analýze a zpracování se často využívají i metody strojového učení k osvojení uživatelských vzorů chování nebo situací v uživatelském prostředí. (17)

Množina získaných informací ze senzorů pak pomáhá utvářet uživatelský kontext, jenž je dle (16) definovaný následovně: Kontext je jakákoliv informace, kterou lze použít k charakterizaci situace, jež se váže k určité entitě. Touto entitou je osoba, místo nebo objekt, který je považován za relevantní vzhledem k interakci mezi uživatelem a aplikací, a to včetně uživatele a samotných aplikací.

V oblasti všudypřítomných výpočtů se uživatelský kontext využívá v tzv. *context-aware systems* (*kontextově-uvědomělé systémy*). Tyto systémy využívají kontext kdykoliv a kdekoliv k poskytování informací a služeb v závislosti na uživatelském chování a uživatelově okolí – tomuto procesu se říká rovněž *adaptace systému na uživatelský kontext*, tedy systém se přizpůsobuje potřebám uživatele (16). Příklady kontextově-uvědomělých systému jsou uvedeny například v (16) nebo v (18).

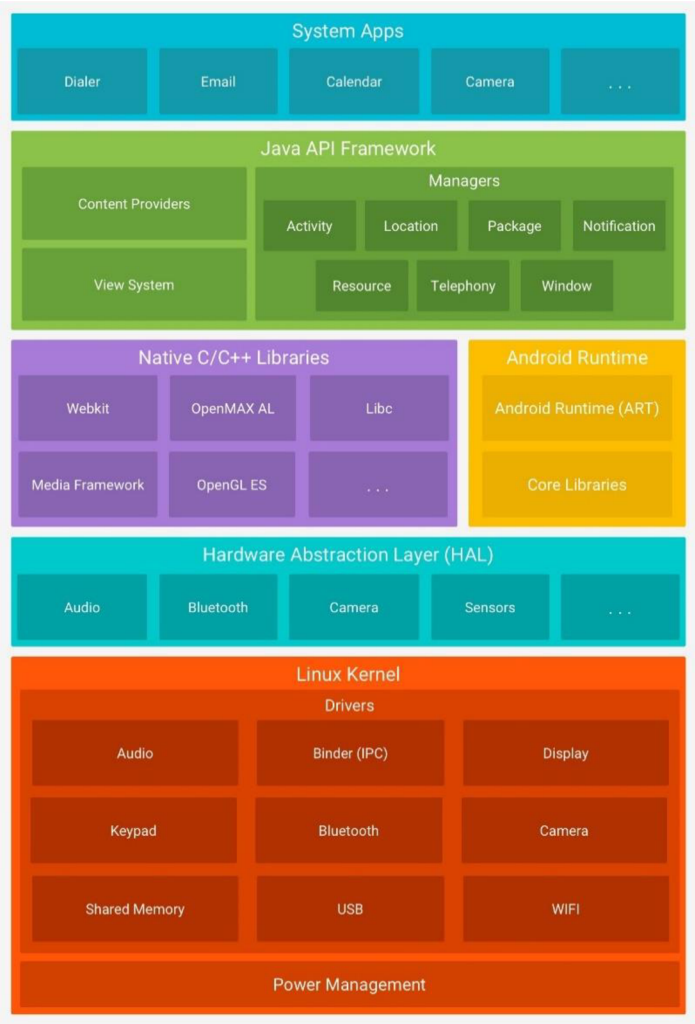
V této diplomové práci je kontext uživatele využit v rámci vzorové funkcionality cílové hybridní aplikace, a především v rámci navrhované architektury k podpoře

možnosti ztloustnutí klienta pro účely peer-to-peer komunikace mezi blízkými klienty, kdy jsou využívány lokalizační informace klientských zařízení – více v *4 Prakticko-teoretická část*.

3.2 Android OS architektura

Úkolem této kapitoly je stručné seznámení s architekturou operačního systému Android. Nejprve je představeno kompletní schéma architektury. Následně jsou velmi stručně popsány jednotlivé hlavní komponenty se zaměřením na modul *Java API Framework*, který je zásadní pro vývoj aplikace na platformě Android. Z tohoto modulu jsou poté vybrány a představeny balíčky API, jež jsou použity ve vzorové aplikaci. Poslední kapitola se týká modularity a toho, jak je chápána a definována v prostředí Android OS, neboť modularita jakožto základní kámen dynamické rekonfigurace vzorové aplikace je zásadním nástrojem pro dosažení cíle této diplomové práce.

3.2.1 Schéma a popis modulů



Obrázek 3 - schéma architektury Android OS, zdroj: www.developer.android.com, 2021

Na *obrázku 3* lze vidět schéma architektury operačního systému Android, kde jsou vyobrazeny hlavní součásti systému. Základem platformy je *linuxové jádro* obsahující veškeré hardwarové ovladače a modul pro řízení spotřeby či výkonu. Použití linuxového jádra umožňuje také systému Android využívat klíčové funkce zabezpečení a výrobcům zařízení umožňuje vyvíjet ovladače pro již známé jádro. *Android Runtime*, jakožto komponenta vytvářející běhové prostředí v Android OS, využívá linuxové jádro pro správu vláken či nízko úrovňovou správu paměti. (19)

Další komponentou při postupu od nejnižší po nejvyšší úroveň je *hardwarová abstrakční vrstva* (*Hardware Abstraction Layer – HAL*). Ta poskytuje standartní rozhraní, která odkrývají možnosti jednotlivých hardwarových součástí daného zařízení pro

manipulaci s hardwarem ve vyšší vrstvě – *Java API Framework*. HAL se skládá z několika knihovních modulů, kdy každý z nich implementuje jedno rozhraní pro jeden typ hardwarové komponenty (například modul pro Bluetooth, kameru atd.). Při vyžádání přístupu k určitému typu hardwaru v *Java API Framework*, je načten knihovní modul pro tuto hardwarovou komponentu, prostřednictvím něhož je možné s komponentou dále pracovat. (19)

Jak již bylo zmíněno, další vrstva *Android Runtime (ART)* vytváří běhové prostředí pro aplikace běžící na Android OS. Od verze operačního systému 5.0 (API úroveň 21) běží každá aplikace ve svém vlastním procesu a se svou vlastní instancí ART. ART je navržen pro zařízení s malou pamětí a spouští soubory ve formátu *DEX*, což je formát navržený pro spouštění aplikací v operačním systému Android. Formát DEX je optimalizován pro minimální paměťovou náročnost. (19) Má-li být tedy vytvořená aplikace spustitelná na platformě Android, musí být nejprve přeložena do formátu DEX, který je schopný ART přeložit do strojového kódu cílového zařízení. ART se skládá z několika částí – mezi nejdůležitější patří *JIT (just-in-time)* či *AOT (ahead-of-time) kompilátor*, optimalizovaný *garbage collector*, podpora *debuggingu* (Android uvádí dokonce vylepšenou formu debuggování) či *překladač DEX souborů* do více kompaktního strojového kódu. Předchůdcem ART byl tzv. *Dalvik*, který je se svým následovníkem plně kompatibilní. (19, 20)

Značné množství komponent a služeb systému Android (například HAL či ART) vyžaduje nativní kód v podobě nativních knihoven psaných v jazycích *C* nebo *C++* (viz komponenta *Native C/C++ Libraries* na *obrázku 3*). K těmto funkcionalitám lze přistupovat buď pomocí knihoven, které jsou součástí komponenty *Java API Framework*, případně přímo pomocí jazyků C/C++ za využití sady nástrojů *Android NDK*. Příkladem použití knihovny z komponenty *Java API Framework* může být manipulace s 2D nebo 3D grafikou ve vytvářené aplikaci, kdy je možné pro volání nativních funkcí využít knihovny *Java OpenGL API*. (19)

Z hlediska vývoje aplikací na platformě Android je nejdůležitější komponentou systému *Java API Framework*, neboť jejím prostřednictvím je k dispozici celá sada funkcí Android OS, a to v podobě jazyku *Java*. Celé API má několik základních stavebních prvků. Prvním z nich je systém zobrazení (*View System*), který slouží k budování uživatelského rozhraní (*User Interface – UI*) aplikací. Tento systém nabízí velké

množství různých *GUI* (*Graphical User Interface – grafické uživatelské rozhraní*) komponent včetně integrovaného webového prohlížeče. (19) Dalším prvkem je *Resource Manager*, jehož úkolem je přístup k souborům, které se označují jako *Resources* – jedná se o další soubory a statický obsah, který je využíván v kódu aplikací. Nejčastěji se jedná o definici layoutu jednotlivých obrazovek, tedy rozložení a vlastnosti GUI komponent, či o definici řetězců, které jsou součástí UI. (21) Třetím klíčovým prvkem je *Notification Manager*. Ten umožňuje aplikacím zasílat tzv. notifikace – jedná se o zprávu zobrazenou systémem Android mimo UI aplikace. Taková zpráva obsahuje zpravidla určitou informaci o událostech v aplikaci. (22) Možná nejdůležitějším modulem v *Java API Framework* při vývoji aplikací je *Activity Manager*, neboť spravuje životní cyklus samotné aplikace, respektive aktivit, ze kterých je aplikace poskládána. *Aktivita* je naprosto stěžejní komponenta každé aplikace na platformě Android a chod aplikace je založen právě na skládání aktivit a na jejich životních cyklech. (19, 23) Více o aktivitách v (23). Posledním prvkem jsou *Content Providers*, díky nimž mohou aplikace přistupovat k datům jiných aplikací, potažmo sdílet svá vlastní data. (19)

Operační systém Android je dodáván se sadou základních systémových aplikací (například pro správu emailů, SMS zpráv či procházení internetu nebo třeba kontaktů). Tyto aplikace představují poslední modul architektury Android OS, respektive modul na nejvyšší úrovni. Systémové aplikace fungují jako aplikace pro běžné uživatele, ale také jako klíčové funkce, ke kterým mají vývojáři přístup ze svých vlastních aplikací. Není tak kupříkladu nutné vytvářet nový aplikační modul pro zasílání zpráv SMS, nýbrž je možné využít již zavedené systémové aplikace, případně jiné již nainstalované aplikace třetí strany. (19)

3.2.2 Důležité moduly Java API Framework pro účely práce

Komponenta *Java API Framework* poskytuje veškeré funkce a služby systému Android, a to jak systémovým aplikacím, tak vývojářům vlastních aplikací (19). Jednotlivé funkcionality jsou rozděleny do samostatných balíčků a dál děleny dle konkrétnějších funkcí či služeb. Pro implementaci vzorové aplikace bylo použito několik významných balíčků. Za zmínku stojí především klíčový balíček *android.app*, jenž zapouzdřuje všechny důležité funkce týkající se aplikačního modelu (24). Spadá zde mimo jiné výše zmíněná zásadní aplikační komponenta – *Aktivita* či komponenta zvaná

Fragment, která může definovat určitou část dané Aktivity, včetně jejího souvisejícího UI (25). Součástí tohoto balíčku jsou rovněž důležité komponenty, respektive Java třídy z dalšího důležitého balíčku *android.content*, bez kterého by nebylo možné manipulovat s daty. Čili prostřednictvím nástrojů tohoto balíčku lze realizovat přístup k datům a jejich publikování v aplikaci. Třetím důležitým balíčkem pro samotný chod aplikace je *android.os* poskytující základní služby operačního systému, včetně komunikace mezi vnitřními komponentami aplikace. Pro přístup k senzorům a následnému získávání snímaných dat slouží balíček *android.hardware*. Veškeré základní informace o mobilním zařízení, typu a stavu připojení k síti zaštiťují API spadající pod *android.telephony* a pro definici GUI jsou k dispozici komponenty z *android.view* potažmo *android.widget*. Posledním důležitým použitým balíčkem je *android.database* pro práci s *SQLite* databází. (24)

Na závěr této kapitoly je nutné zmínit, že některé aplikační komponenty (především *Aktivity* a *Fragmenty*) byly použity z podpůrné soustavy knihoven *Android Jetpack*, jakožto nástupcem *Android Support Library*, které jsou k dispozici pod jmenným prostorem (*namespace*) s označením *androidx*. Úkolem těchto knihoven je „best-practise“ implementace napříč různými verzemi Android OS. Výsledkem jsou pak aplikace kompatibilní s různými verzemi operačního systému a s různými zařízeními, jejichž dalšími přednostmi jsou menší náchylnost k chybám a jednodušší implementace bez nadbytečného kódu u základních funkcí, jako je navigace mezi *Aktivitami*, správa jejich životního cyklu a podobně. (26) Podrobnější informace vztahující se ke konkrétním nástrojům a implementaci jednotlivých balíčků jsou podrobněji rozepsány i s ukázkami kódu v praktické části (kapitola *5 Praktická část*).

3.2.3 Modularita v Android OS

Tato podkapitola vychází z teoretického konceptu modularity, jenž byl nastíněn v kapitole *3.1.3 Modularita*, respektive v její podkapitole *3.1.3.1 Modulární aplikace*. Modul je tak vnímán jako zcela nezávislá aplikační součást, kterou je možné dynamicky zavádět, a naopak odebírat za běhu aplikace. Tento teoretický návrh je nyní třeba uchopit z ryze praktického pohledu, a to na základě možností a nástrojů, které nabízí vývoj aplikací na platformě Android. Možností je vícero, nicméně nástroje a funkcionality zde zmíněné souvisí s návrhem vzorové hybridní aplikace, respektive s jejím modulem pro

dynamické zavádění knihoven (viz kapitola *4.3.3 Modul dynamického zavádění knihoven* v prakticko-teoretické části). Vzhledem k tomu, že pro implementaci hybridní aplikace bude využit programovací jazyk Java, jakožto primární jazyk používaný pro vývoj aplikací na této platformě (viz komponenta *Java API Framework* v podkapitole *3.2.1 Schéma a popis modulů* této kapitoly), je vhodné se nejprve podívat na problematiku modularity z pohledu Javy.

Jednotlivé moduly budou vlastně Java knihovnami. To znamená, že budou obsahovat sadu tříd či rozhraní, jež budou poskytovat funkcionalitu daného modulu. Jak již bylo zmíněno tyto moduly by měly být zaváděny dynamicky za běhu aplikace. Zde však nastává problém, neboť je nutné vyřešit, jakým způsobem načítat deklarace tříd dynamicky.

V Javě je zdrojový kód psaný formou souborů ve formátu *.java* (27). Nicméně tyto soubory nejsou spustitelné, neboť k tomu, aby mohly být interpretovány do podoby strojového kódu konkrétního zařízení prostřednictvím nástroje *JVM (Java Virtual Machine)*, musí být zkompilovány do podoby *bytecode* souborů ve formátu *.class*. Díky této formě překladu zdrojového kódu je možné spustit program v Javě na jakékoliv platformě s přítomností JVM. (27, 28) Krom JVM je součástí *JRE (Java Runtime Environment)*, tedy běhového prostředí pro spouštění Java aplikací (29), také klíčový prvek pro modularitu *Java ClassLoader*, pomocí něhož je možné načítat Java třídy dynamicky za běhu do JVM. Zásluhou této komponenty tak běhové prostředí nemusí vědět o implementačních souborech se zdrojovým kódem. (30) K deklarovaným atributům a metodám dynamicky načtených tříd za využití *ClassLoaderu* je pak možné přistupovat prostřednictvím funkce, která je součástí Javy a nazývá se *Java reflexe (Java Reflection)*. Dle (31) reflexe umožňuje vykonávanému programu prozkoumat sebe sama a manipulovat s vnitřními vlastnostmi programu. Zdroj (32) přímo uvádí, že díky této funkci je možné prozkoumat a modifikovat třídy, metody, rozhraní či atributy za běhu programu či aplikace bez nutnosti uvádět závislosti, jejich import do aplikace či další speciální konfigurační soubory. Stačí pouze importovat balíček *java.lang.reflection*, který obsahuje veškeré implementační nástroje související s reflexí v Javě (32).

Při vývoji na platformě Android v jazyce Java se rovněž kompilují soubory se zdrojovými kódy *.java* do *bytecode* souborů s koncovkou *.class*. Android však k dalšímu překladu aplikací nevyužívá JVM nýbrž Dalvik, respektive ART v novějších verzích

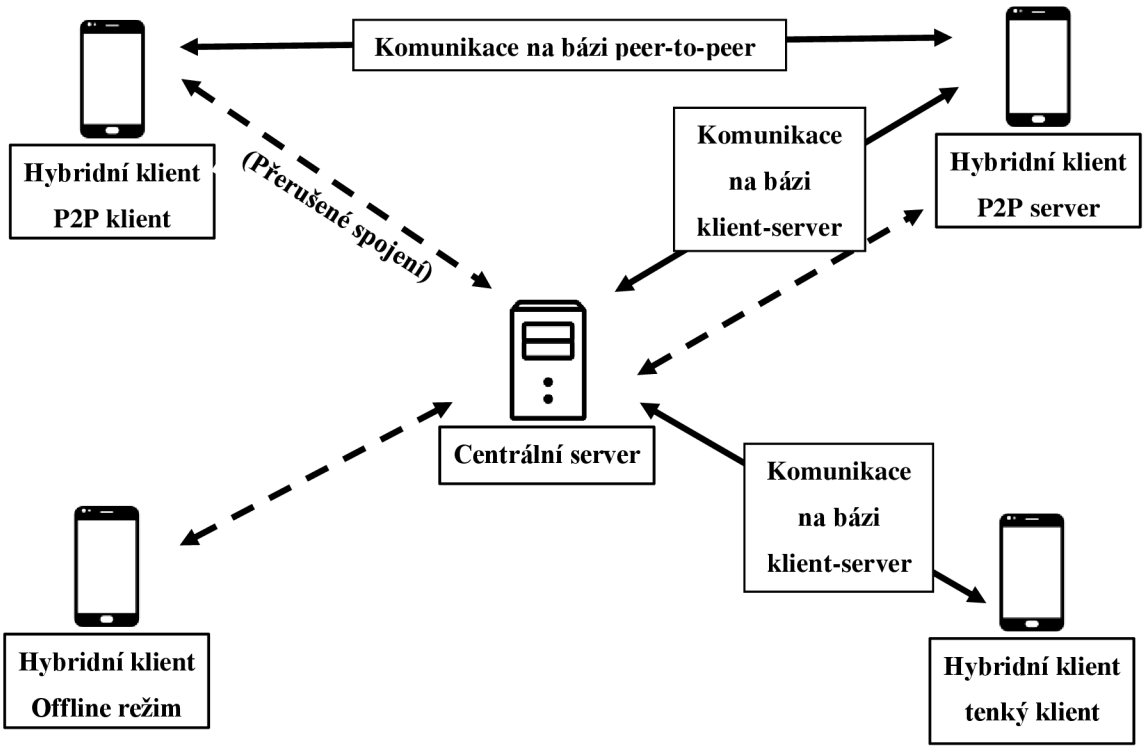
systému (více v podkapitole *3.2.1 Schéma a popis modulů*). Jedná se o virtuální stroje vytvořené přímo pro účely interpretace aplikací v operačním systému Android, které jsou schopny číst *bytecode* v podobě *DEX* souborů. (33) Tento odlišný princip překladu přináší problém, neboť pro dynamické načítání tříd není možné využít tradičního Java *ClassLoaderu*. Android však přichází s alternativou v podobě komponenty *DexClassLoader* (34), která je schopná dynamicky načítat třídy z *.apk* souborů (35), což je souborový formát pro distribuci a instalaci Android aplikací obsahující mimo jiné zkompilované *.DEX* soubory. *DexClassLoader* je tak možné využít k vykonávání kódu, který není nainstalovaný jako součást aplikace (34). Jedná se proto o ideální nástroj pro načítání tříd z externích modulů. Pro přístupu k atributům, metodám či konstruktorům načtených tříd je možné přistupovat i v tomto případě pomocí *Java reflexe*.

4 Prakticko-teoretická část

Prostřední prakticko-teoretická část diplomové práce se zabývá především návrhem architektury mobilní aplikace v podobě hybridního klienta tak, jak byl nastíněn v kapitole *3.1.1.3 Hybridní klient*. K tomu, aby byly splněny všechny deklarované cíle této práce (viz kapitola *2 Cíl práce*) je však nejdříve nutné navrhnout a specifikovat architekturu v hlubším kontextu. To znamená zahrnout v ní i centrální server, který bude mít řídící schopnosti a skrze který budou moci klienti vzájemně spolupracovat, kdy tato jejich součinnost vede k dalším možnostem využití dynamické rekonfigurace klienta na základě nejen vlastního kontextu, nýbrž i kontextu okolních klientů.

4.1 Návrh architektury systému a komunikace

Na následujícím obrázku (*obrázek 4*) lze vidět schéma architektury celého systému a komunikace mezi jednotlivými komponentami:



Obrázek 4 - schéma architektury systému

Architektura vyobrazena na *obrázku 4* je založena na teoretickém návrhu v práci (5). Nicméně tento teoretický koncept je nutné vhodně modifikovat tak, aby zcela odpovídal reálné implementaci na platformě Android a podmínkám či prostředí, ve kterém bude funkčnost systému a jednotlivých komponent testována. Součástí tohoto

architektonického návrhu je naznačení formy komunikace a síťových architektur, které jsou v systému využity.

Centrální server jakožto jádro celé architektury má dvě hlavní role. První je řídící role, ve které slouží jako správce veškerých informací o všech klientech, kteří se do systému registrovali a zároveň důležitý komunikační prostředek pro spolupráci klientů, respektive jejich vzájemnou komunikaci za účelem dosažení požadovaného chování, které odpovídá pojetí hybridního klienta v této práci (tato využití centrálního serveru jsou zmiňována dále v práci). Druhou rolí centrálního serveru je role informačního systému (dále jen IS), kdy server poskytuje klientům určité služby, na kterých je klient zpravidla závislý. Funkce IS mohou být samozřejmě různé, vzorová funkcionalita, jež byla navržena čistě pro experimentální a demonstrační účely, je popsána v kapitole *4.2 Funkce vzorového informačního systému* níže. Jak lze vidět na *obrázku 4*, server komunikuje s klienty na bázi síťové architektury klient-server. V běžné architektuře klient-server za předpokladu dosažení všech funkcí bez omezení je často žádoucí či přímo nutné, aby server byl vždy k dispozici pro potřeby klienta (především jedná-li se o koncept tenkého klienta). Cílem navržené architektury a konceptu hybridního klienta je tuto závislost do co nejvyšší míry eliminovat, neboť je brán v potaz předpoklad ztráty připojení se serverem.

4.1.1 Podoby a role hybridního klienta

V architektuře na *obrázku 4* jsou rovněž zaznamenány všechny klíčové funkcionality a možnosti hybridních klientů. Jinými slovy všechny role, které může hybridní klient teoreticky zastávat díky takto navržené architektuře a klientově vnitřní sktruktuře (viz *4.3 Návrh architektury hybridního klienta*). V praxi by nedávalo smysl, aby každý klient mohl naplňovat všechny role, neboť některé se svými službami a možnostmi překrývají (výjimku tvoří vzorová aplikace této práce, neboť je určena pro otestování všech variant). Proto platí, že každý hybridní klient má implementované funkce, které realizují pouze některé z těchto rolí. Při dalším popisu jednotlivých rolí bude zřejmé, že role se liší mimo jiné implementační složitostí, a především náročností na zdroje a hardware mobilních zařízení. Některé role dokonce nemusí být na určitých zařízeních vůbec realizovatelné – například mohou chybět technologie pro podporu peer-to-peer komunikace, na které jsou závislé role P2P serveru a P2P klienta. Schopnosti

klienta se proto mohou odvíjet kupříkladu od mobilního uživatelského kontextu, konkrétně hardwarových možností příslušného mobilního zařízení. Má-li hybridní klient běžet například na zařízení se slabším výkonem, je žádoucí, aby v systému figuroval spíše v roli P2P klienta, neboť právě tato varianta nevyžaduje pro své fungování tak náročné funkce a operace. Role, které mohou být naplňovány příslušným hybridním klientem by měly být dány počáteční konfigurací, jež reflektuje možnosti každého hybridního klienta.

4.1.1.1 Hybridní klient v roli tenkého klienta

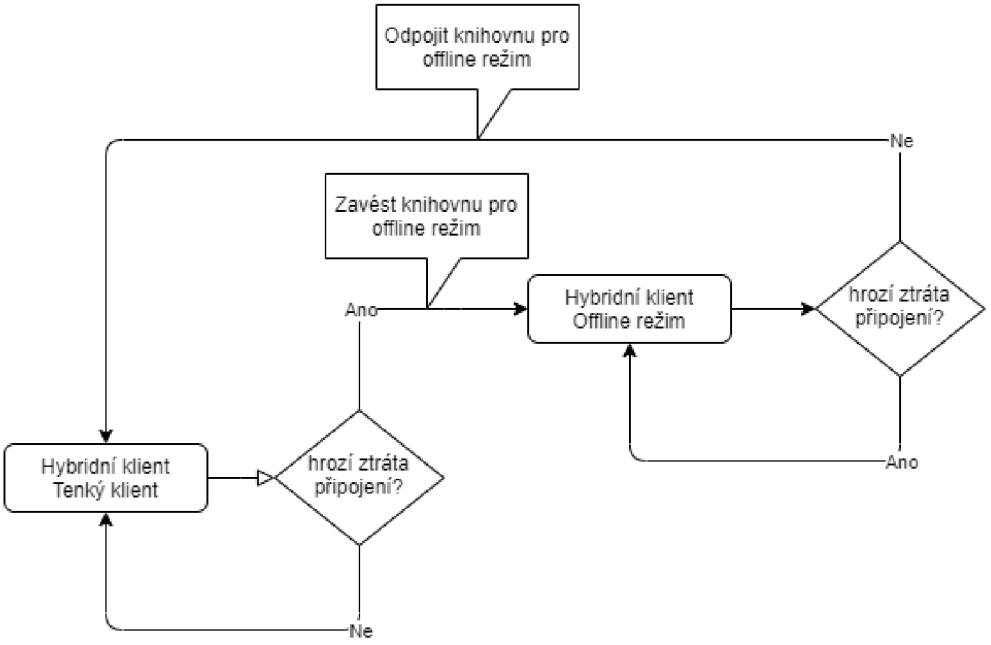
První, pravděpodobně nejjednodušší, podobou hybridního klienta je klient tenký. Jedná se o výchozí stav každého hybridního klienta, kdy aplikaci je nutné po instalaci registrovat do systému pro zajištění její plné funkcionality, a to právě prostřednictvím centrálního serveru. Existuje tedy předpoklad, že každá aplikace či každý hybridní klient je na počátku svého fungování schopen komunikovat s centrálním serverem. Ten poskytuje klientovi nejen určité funkce IS, nicméně také informace o ostatních klientech v síti a rovněž poskytuje komunikačního prostředníka pro jednotlivé klienty, neboť jeho prostřednictvím si klienti mohou zasílat požadavky (viz například požadavek tloustnutí do role P2P serveru v kapitole *4.1.1.3 Hybridní klient v roli P2P serveru*). Aby klient obsahoval stále aktuální data je potřeba neustálá pravidelná komunikace, tedy opakující se požadavky na server a následné odpovědi, ve kterých může být vložena dodatečná informace o požadavku na rekonfiguraci od jiného klienta. Hybridní klient v roli tenkého klienta rovněž pravidelně zasílá na centrální server informace o svém aktuálním stavu a poloze. Tyto informace jsou zásadní především pro role související s peer-to-peer komunikací (více v kapitolách *4.1.1.3 Hybridní klient v roli P2P serveru* a *4.1.1.4 Hybridní klient v roli P2P klienta*). Konkrétní podoba komunikace mezi tenkým klientem a centrálním serverem stejně jako obsah zasílaných zpráv či dat je předmětem kapitoly *4.3.5 Komunikační modul*. S využitím peer-to-peer architektury se pojí také další předpoklad role tenkého klienta a tím je dostupnost knihovny pro peer-to-peer komunikaci ihned po instalování hybridního klienta. Architektura na *obrázku 4* je navržena tak, aby hybridní klient měl vždy tuto knihovnu již v uložišti a nemusel ji stahovat při požadavku na tloustnutí, a to jak do podoby P2P serveru, tak do podoby P2P klienta. Díky tomu je proces přecházení do režimu peer-to-peer komunikace rychlejší a bezproblémovější. Více informací o obecné struktuře hybridního klienta lze nalézt v kapitole *4.3 Návrh architektury hybridního klienta*.

Pojmenování stavu hybridního klienta „tenký klient“ v pojetí architektury znamená, že takový klient je striktně závislý na dostupnosti serveru bez možnosti jakékoliv ztráty připojení. Tenký klient zde však nemusí být chápán jako aplikace představující pouze prezentační vrstvu tak, jako tomu může být v tradičním pojetí konceptu tenkého klienta (viz *3.1.1.1 Tenký klient*), nýbrž může vykazovat i prvky klienta tlustého (zejména větší množství aplikační logiky). Role má vlastně představovat běžnou podobu mobilní aplikace, jež je závislá na internetovém připojení a na připojení k serveru, bez jehož přítomnosti je vlastně funkčně nepoužitelná. Vzhledem k tomu, že tato diplomová práce řeší problematiku přizpůsobení se aplikace či klienta na změnu kontextu uživatele především v souvislosti s datovou propustností sítě, ostatní role hybridního klienta představují vlastně možnosti řešení ztráty připojení k centrálnímu serveru, respektive předejití tomu, aby došlo k omezení funkcí, případně úplnému kolapsu klienta v případě, že by k takové ztrátě došlo nebo potenciálně mohlo dojít.

4.1.1.2 Hybridní klient v roli klienta v offline režimu

První důležitou rolí, kterou by měl být schopen hybridní klient zastávat a která je prvním cílem této diplomové práce, je role klienta v tzv. *„offline režimu“*. Jedná se o první způsob (a možná způsob nejjednodušší vzhledem k implementaci takového klienta), jak rekonfigurovat hybridního klienta z role klienta tenkého (tak jak byl popsán výše) do role tlustého klienta, který je schopný fungovat zcela nezávisle na centrálním serveru. K dosažení takové podoby však klient nutně musí získat alespoň některé (z hlediska funkcionality nejdůležitější) funkce serveru. Jinými slovy potřebuje jakýmsi způsobem ztloustnout – nabrat aplikační logiku (viz kapitola *Koncept tloustnutí a hubnutí*). Je jasné, že takový klient je výpočetně a paměťově mnohem náročnější, což může být nežádoucí, a to především v kontextu mobilních zařízení, na které se práce zaměřuje. I přes fakt, že mobilní zařízení dnes mohou být velmi výkonná, stále se jedná o zařízení menších rozměrů oproti tradičním desktopům a vzhledem k velkému množství aplikací, které na něm mohou současně běžet, je vhodné zdroje zařízení zbytečně nezatěžovat, není-li to zcela nutné. Zde tak vyvstává důležitá otázka a to: V jaké situaci by měl klient tento proces tloustnutí provádět, potažmo kdy je možné opět zhubnout do podoby tenkého klienta? Odpověď by měla být řízena dostupností centrálního serveru. V případě, že je server dostupný a zároveň nehrozí žádná potenciální ztráta připojení, není nutné, aby se klient nacházel v podobě tlustého klienta. Naopak hrozí-li ztráta připojení

se serverem je nutné, aby klient přešel co nejdříve do formy tlustého klienta. Z této odpovědi plyne nutnost zavést na hybridního klienta modul, jenž by byl schopen měřit datovou propustnost sítě, na základě níž, hybridní klient vyhodnotí, zda spustit proces ztloustnutí či zhubnutí (samozřejmě také na základě současného stavu klienta). Tento modul je v architektuře hybridního klienta na *obrázku 7* označen jako *modul vyhodnocení datové propustnosti sítě*. Proces rekonfigurace klienta, tedy tloustnutí a hubnutí je prováděn prostřednictvím *modulu pro dynamické zavádění knihoven*, respektive *centrálního modulu*. Oba tyto moduly jsou rovněž součástí architektury hybridního klienta (moduly jsou blíže specifikovány v kapitole *4.3 Návrh architektury hybridního klienta*). Chování hybridního klienta, který je schopen ztloustnout do role klienta v offline režimu tak lze zjednodušeně popsat následovně:



Obrázek 5 - workflow chování hybridního klienta v offline režimu

Obrázek 5 lze interpretovat takto: Každý hybridní klient je na počátku v roli tenkého klienta. Modul datové propustnosti pravidelně vyhodnocuje, zda se zásadně zhoršuje datová propustnost sítě, tedy zda hrozí ztráta připojení se serverem. Pokud nehrozí, tak klient pokračuje ve své běžné činnosti. To znamená, že plní roli, kterou má přidělenou v rámci funkcionalit IS a zároveň neustále vyhodnocuje stav sítě. Pokud hrozí ztráta spojení se serverem, modul pro dynamické zavádění knihoven stáhne knihovnu pro offline režim z centrálního serveru, uloží ji do lokálního uložiště, zapíše do seznamu stažených knihoven a tuto knihovnu následně zavede a spustí (kompletní proces je popsán

v kapitole *4.3.3 Modul dynamického zavádění knihoven*). Po úspěšném zavedení knihovny klient funguje v offline režimu, tedy je schopný naplňovat funkce IS s určitou mírou omezení (viz dále) bez přítomnosti serveru. Následnou snahou klienta je zpětný proces transformace zpět do podoby tenkého klienta, a to v nejkratším možném čase (z důvodů snížení kladených požadavků na zdroje zařízení a omezených funkcí IS). Z tohoto důvodu modul datové propustnosti stále vyhodnocuje stav sítě a jakmile je server znovu k dispozici, tedy dle terminologie *obrázku 5* již nehrozí ztráta připojení, je odpojena knihovna pro offline režim a klient přechází do počáteční podoby tenkého klienta.

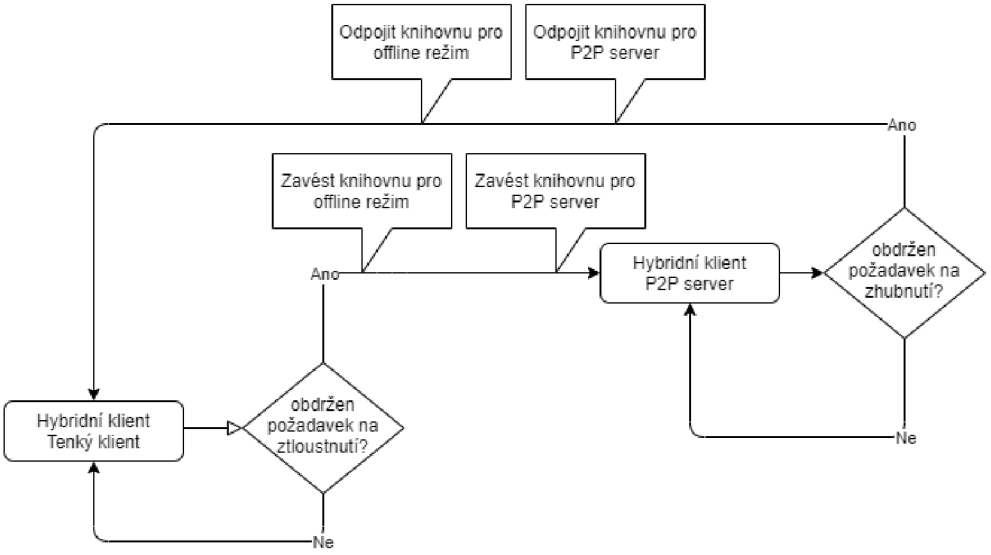
Architektura rovněž umožňuje manuální rekonfiguraci hybridního klienta, jež je nezávislá na automatickém procesu vyhodnocování datové propustnosti sítě. V tomto případě je změna role klienta vyvolána interakcí uživatele, který může dát pokyn k ztloustnutí i zhubnutí klienta. Manuální rekonfigurace nachází využití v případech, kdy uživatel ví, že se blíží do oblasti, ve které se dají očekávat zhoršené podmínky připojení k mobilní síti, potažmo pokud se domnívá, že připojení bude stabilní a nechce čekat na proces automatické rekonfigurace, neboť modul vyhodnocení datové propustnosti vyhodnocuje kvalitu připojení s určitou časovou prodlevou. Dalším využitím mohou být ryze testovací účely chování a funkčnosti hybridního klienta. No a v neposlední řadě manuální rekonfigurace přináší možnost zavést a spustit nebo naopak ukončit jakoukoliv další externí funkcionalitu v podobě knihovny, a to pouze na základě požadavku od uživatele.

Jak již bylo naznačeno, v podobě klienta v offline režimu je potřeba počítat s jistými omezeními, neboť cílem není plnohodnotné zastoupení serveru a dost možná by to ani nebylo možné, protože server často pracuje s daty ostatních klientů, ke kterým nemá klient přístup. Příkladem je vzorová funkcionalita IS navržená pro účely této práce, která je popsána v kapitole *4.2 Funkce vzorového informačního systému*. Základním prvkem této funkcionality je zobrazování ostatních klientů na mapě s hodnotami z jejich senzorů, kdy je evidentní, že tato data po určité době od ztráty připojení se serverem přestanou být relevantní. Řešením může být peer-to-peer komunikace (viz kapitola *3.1.2.2 Architektura peer-to-peer*) mezi hybridními klienty, kterou se zabývají zbylé dvě role hybridního klienta – role P2P serveru a P2P klienta (viz *obrázek 4*). Tyto role budou následně rozebrány.

4.1.1.3 Hybridní klient v roli P2P serveru

Na rozdíl od role hybridního klienta v offline režimu se jedná o podstatně komplexnější variantu hybridního klienta s více možnostmi v souvislosti se ztrátou připojení k centrálnímu serveru a jeho možném zastoupení. V podstatě se může vyskytovat ve dvou podobách. První podoba představuje hybridního klienta jako komunikační uzel či prostředníka komunikace mezi serverem a klienty, kteří ztratili připojení k serveru (tito klienti jsou označeni rolí P2P klient – viz *obrázek 4*). Takovýto klient se chová jako tenký klient – přijímá služby od serveru na bázi komunikace klient-server. Nicméně zároveň je schopný rekonfigurovat se do podoby, ve které může naslouchat klientům ve svém okolí a zprostředkovávat jim služby serveru na základě peer-to-peer komunikace. Tato varianta vede k řešení problému neaktuálních dat, který byl zmiňován v souvislosti s rolí klienta v offline režimu. Tedy hybridní klienti, jež mají přístup ke klientovi v takovéto podobě a komunikují s ním na bázi peer-to-peer architektury (klienti v roli P2P klienta), obsahují vždy aktuální data ze serveru.

Druhá varianta hybridního klienta v roli P2P serveru je velmi podobná, avšak liší se tím, že klient po rekonfiguraci opravdu představuje plnohodnotný server, tak jako tomu je v architektuře klient-server. To znamená, že není již pouze prostředníkem, nýbrž plnohodnotným zástupcem centrálního serveru, tudíž se od něho očekávají v podstatě stejné schopnosti jako od centrálního serveru. Tato varianta je podstatně jednodušší nejen implementačně ale především architektonicky – nemusí se řešit spousta potenciálních problémů, které mohou nastat například při výpadku spojení mezi takovýmto klientem a centrálním serverem, což je reálná situace, jež mohla nastat v první zmíněné variantě. Stejně jako v případě předešlé varianty běžný tenký klient při ztrátě připojení se serverem vlastně ani nepozná, že již nekomunikuje s centrálním serverem, ale právě s tímto hybridním klientem v roli P2P serveru. Zároveň tak není nutné, aby všichni hybridní klienti, kteří se nachází v určité oblasti a ztratili připojení k serveru, tloustli do podoby offline režimu a zatěžovali tak zbytečně mobilní zařízení. Právě tato varianta je součástí návrhu a implementace hybridního klienta a je předmětem *obrázku 6*.



Obrázek 6 - workflow chování hybridního klienta v roli P2P serveru

Je evidentní, že schéma na *obrázku 6* má spoustu podobností se schématem chování hybridního klienta na *obrázku 5*. Nejzásadnějším rozdílem mezi klientem v roli P2P serveru a klientem fungujícím v offline režimu je způsob vynucování rekonfigurace, tedy tloustnutí či hubnutí klienta. Je zřejmé, že motivací rekonfigurace klienta v tomto případě není stav datové propustnosti sítě, nýbrž požadavek na ztloustnutí od jiného okolního klienta, který ztrácí připojení k centrálnímu serveru. Zde je však otázkou, jakým způsobem tento požadavek předat. V této situaci je vhodným využitím centrální server, neboť právě ten je (dle *obrázku 4*) centrálním prvkem, který komunikuje se všemi klienty v systému a zároveň má o všech klientech přehled – zná jejich stavy, možné budoucí stavy, do kterých se potenciálně mohou klienti dostat, polohu a spoustu dalších důležitých dat.

Celý proces tloustnutí pak může vypadat tak, že klient v roli tenkého klienta, který ztrácí připojení k serveru (jeho modul vyhodnocení datové propustnosti zaznamená zhoršující podmínky), zašle požadavek na server o ztloustnutí klienta v jeho okolí do podoby P2P serveru. Pokud server odpoví pozitivně, to znamená, že existuje blízký klient, jenž má požadovanou schopnost nebo je již v roli P2P serveru, pak klient přejde do stavu P2P klienta (více o této roli v následující kapitole *4.1.1.4 Hybridní klient v roli P2P klienta*). Jestliže se blízký klient (potenciální P2P server) nachází stále v základní podobě, to znamená v roli tenkého klienta, pak takovému klientovi přijde v odpovědi na pravidelný požadavek získání aktuálních dat z centrálního serveru požadavek na ztloustnutí do podoby P2P serveru. Dle schématu chování na *obrázku 6* a uvedené definici

chování tohoto typu klienta je nejprve zapotřebí, aby převzal schopnosti serveru, tedy ztloustl do podoby serveru – zavedl knihovnu pro offline režim. Jakmile je schopný poskytovat služby serveru je nutné zajistit komunikační schopnosti pro spojení a předávaní dat s okolními klienty v roli P2P klienta. Zde je potřeba připomenout, že každý tenký klient obsahuje již P2P knihovnu ve svém uložišti, to znamená knihovnu již není nutné stahovat, nýbrž pouze zavést. Po úspěšném zavedení může dojít k navázání spojení s P2P klientem a zahájení běžné komunikace s tím, že P2P klient nyní vystupuje v roli běžného tenkého klienta a P2P server v roli centrálního serveru. Zpětná rekonfigurace klienta z podoby P2P serveru do tenké varianty může být vyvolána například požadavkem P2P klienta nebo více P2P klientů na zhubnutí, neboť se jim podařilo navázat spojení s centrálním serverem a služby klienta v roli P2P serveru nejsou dále potřeba (v případě, že je připojeno více P2P klientů, proces zhubnutí proběhne až po obdržení požadavků od všech klientů). Zhubnutí je v tomto případě definováno jako odpojení obou knihoven – jak pro offline režim, tak pro P2P komunikaci.

Ať už je použita jakákoliv varianta hybridního klienta v roli P2P serveru, využití peer-to-peer architektury vždy přináší velké množství výhod a možností v problematice hybridního klienta, nicméně klade vyšší nároky na architekturu a samotnou implementaci klienta, potažmo centrálního serveru.

4.1.1.4 Hybridní klient v roli P2P klienta

Poslední rolí, která je součástí architektury systému na *obrázku 4* a kterou může hybridní klient zastávat je role P2P klienta. Ta opět souvisí s využitím peer-to-peer architektury pro vzájemnou komunikaci mezi klienty a představuje doplněk pro roli P2P serveru. Role představuje běžného tenkého klienta, jenž ztrácí připojení k centrálnímu serveru (modul vyhodnocení datové propustnosti vykazuje zhoršené podmínky připojení) a místo toho, aby ztloustl do podoby klienta v offline režimu, tak si zachová svoji strukturu tenkého klienta a pouze zavede knihovnu pro peer-to-peer komunikaci, kterou má každý tenký klient k dispozici. Současně se zavedením knihovny zasílá požadavek na centrální server ohledně ztloustnutí některého klienta v blízkosti se schopností poskytnout tomuto tenkému klientovi služby serveru skrze peer-to-peer komunikaci (tedy požadavek na klienta v roli P2P serveru, potažmo na klienta, který je schopný tuto roli zastávat čili rekonfigurovat se do takové podoby). Při obdržení pozitivní odpovědi od centrálního serveru, spustí hybridní klient činnost knihovny peer-to-peer komunikace a pokusí se

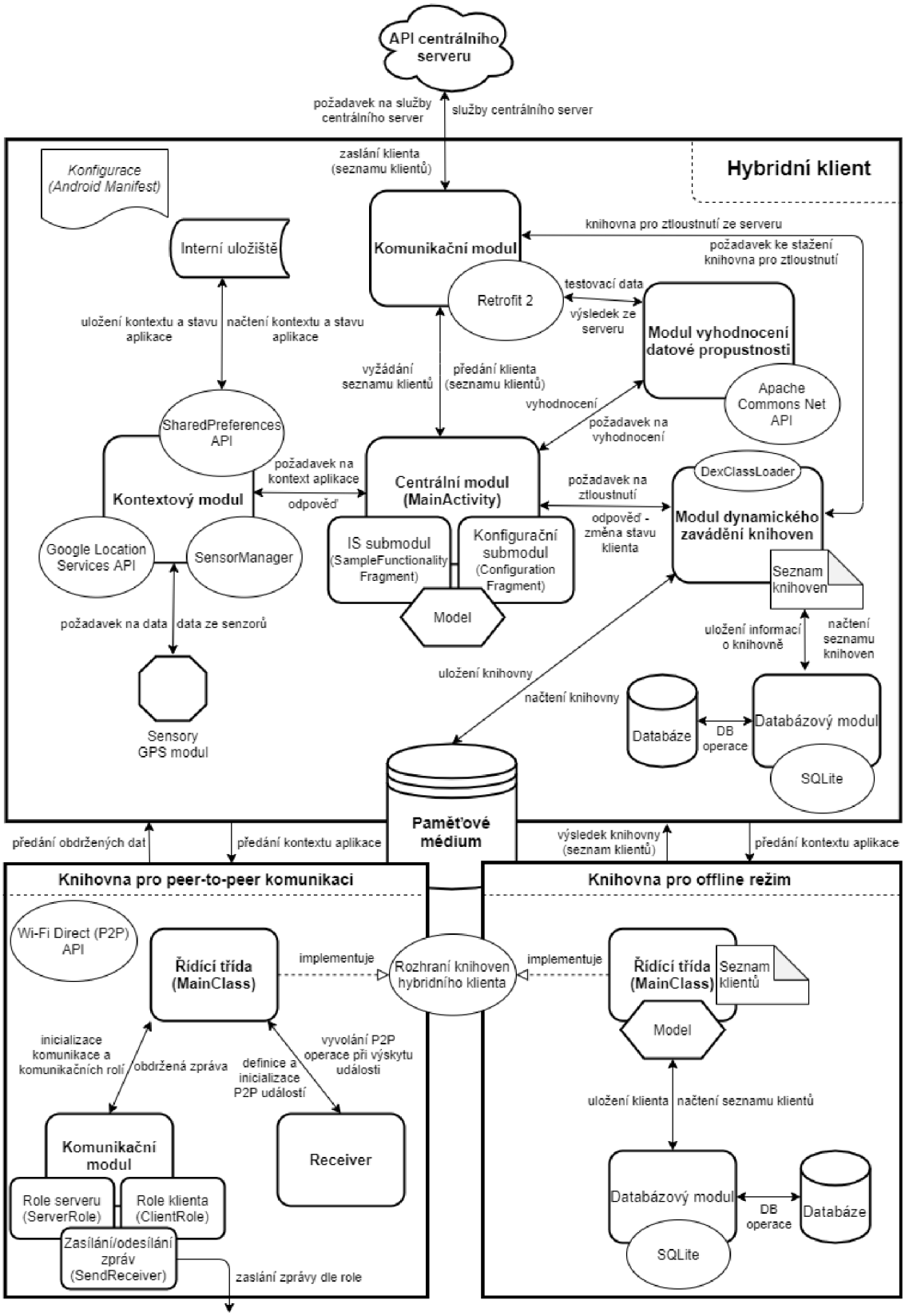
navázat spojení s novým potenciálním serverem v podobě hybridního klienta v roli P2P serveru. V okamžiku, kdy modul vyhodnocení datové propustnosti zaznamená výrazné zlepšení propustnosti sítě, zašle klient informaci na klienta v podobě P2P serveru, se kterým úspěšně komunikuje, o tom, že již má možnost spojení s centrálním serverem a klienta jakožto zástupce serveru dále ke své činnosti nepotřebuje. Hybridní klient v roli P2P klienta pak může po ukončení běhu knihovny peer-to-peer komunikace úspěšně přejít zpět do počáteční podoby tenkého klienta. Obě role založené na peer-to-peer architektuře mohou, stejně jako v případě klienta v offline režimu (viz kapitola *4.1.1.2 Hybridní klient v roli klienta v offline režimu*), ztloustnout do své podoby také prostřednictvím manuální rekonfigurace, tedy nezávisle na zmíněných automatizovaných procesech tloustnutí. Důvody této možnosti zůstávají totožné jako v případě klienta v offline režimu.

4.2 Funkce vzorového informačního systému

Pro demonstraci využití navrženého konceptu hybridního klienta, potažmo z části i celé architektury systému, byl navržen vzorový informační systém, jehož funkce jsou rozděleny mezi hybridního klienta a centrální server. Celý IS si lze představit jako systém meteostanic, které jsou různě rozmístěny po světě. Namísto tradičních meteostanic jsou využity mobilní zařízení, které jsou schopny snímat teplotu a tlak v okolí či prostředí zařízení a také mobilní zařízení lokalizovat za pomoci lokalizačního modulu. Myšlenka vychází ze schopností dnešních mobilních zařízení v souvislosti s monitorováním uživatelského kontextu (viz kapitola *3.1.4 Kontext uživatele*) a možností vestavěných senzorů většiny zařízení s operačním systémem Android (36), neboť právě Android představuje cílovou platformu pro hybridního klienta. Výhodou tohoto využití mobilních zařízení je relativně snadná a jednotná implementace pro všechna zařízení na platformě Android a žádná potřeba dodávání externích senzorů či jiných subsystémů. Mobilní zařízení je schopné data ze senzorů získat, zobrazit uživateli a zaslat na jiné zařízení (centrální server či jiného hybridního klienta), což jsou veškeré funkce důležité pro účely navrženého IS. Úkolem hybridního klienta čili mobilní aplikace v rámci IS je zaznamenání ambientní teploty a ambientního tlaku zároveň se zaznamenáním polohy klienta, zobrazení hodnot uživateli a následné odeslání získaných dat na centrální server. Další funkcí je zobrazení mapy, na které jsou umístěni ostatní hybridní klienti v systému

s jejich hodnotami ze senzorů (je rovněž nutné pravidelně získávat data o ostatních klientech ze serveru). Z uvedených úkolů a funkcí hybridního klienta plyne hlavní účel centrálního serveru a tím je centrální uložiště, přístup k němu a komunikace s klienty v systému. Základními službami centrálního serveru v souvislosti s IS jsou tedy: registrace a správa klientů, ukládání a udržování informací o klientech (identifikátor, hodnoty teploty a tlaku a informace o poloze) a poskytnutí těchto informací všem klientům v systému. Z implementačního hlediska poskytuje *CRUD* operace nad centrální databází s informacemi všech klientů prostřednictvím *REST API* (více o REST API například v (37)). Výhodou je, že takto koncipované funkce centrálního serveru naleznou uplatnění nejen v kontextu služeb IS, ale také v jeho řídící funkci v rámci navržené architektury na *obrázku 4* tak, jak bylo popsáno v kapitole *4.1 Návrh architektury systému a komunikace*. Hlavní předností takto navrženého IS je dobře patrná a snadná testovatelnost navržené architektury a jednotlivých rolí hybridního klienta (viz kapitola *4.1 Návrh architektury systému a komunikace*, potažmo *4.1.1 Podoby a role hybridního klienta*), stejně tak vnitřní architektury hybridního klienta a centrálního serveru (viz kapitola *4.3 Návrh architektury hybridního klienta* a kapitola *4.4 Návrh architektury centrálního serveru*).

4.3 Návrh architektury hybridního klienta



Obrázek 7 - schéma architektury hybridního klienta

Na *obrázku 7* lze vidět schéma architektury hybridního klienta. Návrh vychází z teoretické podoby hybridního klienta (dle terminologie práce (5) je hybridní klient označen jako tzv. *„m klient“* – viz kapitola *3.1.1.3 Hybridní klient*), jež je nastíněna v práci (5). Nicméně tato teoretická vize je upravena tak, aby odpovídala reálné implementaci vzorové aplikace na platformě Android. To znamená, že kromě aplikačních modulů či komponent (podoba zaoblených obdélníků) jsou součástí návrhu rovněž zásadní použité technologie (knihovny či API), které byly využity pro dosažení požadované funkcionality u jednotlivých aplikačních modulů či v rámci externích modulů v podobě dynamických knihoven – například *Wi-Fi Direct API* u knihovny pro peer-to-peer komunikaci (technologie jsou vyobrazeny jako elipsy). V návrhu není vyobrazena pouze architektura samotného hybridního klienta tak, jak se vyskytuje v počáteční fázi či v roli tenkého klienta, ale součástí návrhu jsou i dvě nejdůležitější externí dynamické knihovny, sloužící k realizaci zbylých rolí, které může hybridní klient zastávat (viz kapitola *4.1.1 Podoby a role hybridního klienta*). Knihovna pro offline režim slouží pro potřeby klienta v offline režimu a bez knihovny pro peer-to-peer komunikaci se neobejdou role závislé na této formě komunikace – role P2P serveru a P2P klienta. Dále budou blíže představeny hlavní moduly hybridního klienta a obě zmíněné knihovny.

4.3.1 Centrální modul

Centrální modul je jádrem hybridního klienta a hlavní řídící modul. V podstatě řídí veškerou funkcionalitu klienta, a to jak v rámci konceptu hybridního klienta, tak z pohledu vzorového IS. Jakákoliv funkce klienta je tedy prováděna prostřednictvím centrálního modulu s tím, že k vykonávání dílčích částí dané funkce jsou využívány ostatní moduly klienta, které má centrální modul k dispozici. Přesněji řečeno centrální modul požádá určitý aplikační modul k poskytnutí nějaké služby na základě schopností a poskytovaných služeb tohoto aplikačního modulu a na základě úkolu, jenž má hybridní klient plnit. Nejdůležitějším úkolem centrálního modulu z hlediska konceptu hybridního klienta je udržování a modifikace stavu či podoby klienta. Právě tento modul zodpovídá za provádění veškerých operací spojených s rolemi, které má hybridní klient zastávat (všechny role jsou popsány v kapitole *4.1.1 Podoby a role hybridního klienta*). Každá role je spjata s určitou externí knihovnou, kterou hybridní klient v okamžiku změny role (ztloustnutí) zavádí pomocí modulu pro dynamické zavádění knihoven (viz kapitola *4.3.3*

Modul dynamického zavádění knihoven). Centrální modul dále rozhoduje o dalším běhu knihovny, čímž řídí aktuální stav a podobu hybridního klienta – přerušení běhu a následné znovuspuštění dané knihovny, případně úplné ukončení chodu knihovny při procesu hubnutí klienta. To, jakým způsobem hybridní klient (centrální modul) pracuje a přistupuje k externím knihovnám, stejně jako jejich obecná struktura je podrobněji popsáno v kapitole *4.3.6 Externí knihovny*.

Ve vzorové aplikaci je tento modul reprezentován hlavní aktivitou (*MainActivity*), která bývá součástí každé aplikace na platformě Android a představuje obvykle první funkci či obrazovku, jež je k dispozici po otevření dané aplikace (23). Modul obsahuje dva submodely – IS submodul (ve vzorové aplikaci v podobě *SampleFunctionalityFragment*) a konfigurační submodul (ve vzorové aplikaci *ConfigurationFragment*). Tyto submoduly lze na platformě Android realizovat pomocí komponenty zvané *Fragment*, která je přímo definována jako dílčí část chování Aktivity (25). První dílčí submodul se věnuje čistě funkcionalitám IS tak, jak byly navrženy v kapitole *4.2 Funkce vzorového informačního systému*. Jeho úkolem je tedy zajistit získání dat ze senzorů, zobrazení dat uživateli a uložení získaných hodnot do datové struktury (objektu) klienta, který reprezentuje kontext a stav příslušného hybridního klienta – za tímto účelem spolupracuje s kontextovým modulem a pro uložení potažmo aktualizaci stavu klienta využívá třídu, jež je umístěna v komponentě Model (viz *obrázek 7*). Součástí této komponenty je rovněž Fragment pro vykreslení mapy s markery označující ostatní klienty.

Druhý submodul slouží k procesu manuální rekonfigurace hybridního klienta (tato problematika byla rozebrána v kapitole *4.1.1 Podoby a role hybridního klienta*, respektive v její podkapitole *4.1.1.2 Hybridní klient v roli klienta v offline režimu*). Po dokončení procesu manuální rekonfigurace je vždy důležité uložit změněný stav klienta opět do datové struktury (objektu) příslušného hybridního klienta – tato datová struktura je vždy předána komunikačnímu modulu a odeslána na centrální server (více o komunikaci klienta a serveru v komunikačním modulu v kapitole *4.3.5 Komunikační modul*).

4.3.2 Modul vyhodnocení datové propustnosti

Prvním doplňkovým klíčovým modulem, se kterým centrální modul pracuje a který je naprosto nezbytný pro fungování hybridního klienta je Modul vyhodnocení datové

propustnosti. Funkce tohoto modulu jsou volány v pravidelných časových intervalech, a to podle potřeby zjištění datové propustnosti a vyhodnocení stavu připojení k centrálnímu serveru, neboť právě parametr datové propustnosti a dostupnosti serveru spadá pod jeden z faktorů vyvolání automatické rekonfigurace hybridního klienta, konkrétně v souvislosti s rolemi klienta v offline režimu a P2P klienta (viz kapitoly *4.1.1.2 Hybridní klient v roli klienta v offline režimu* a *4.1.1.4 Hybridní klient v roli P2P klienta*). Proces vyhodnocení se skládá ze dvou fází. První fáze je založena na principu pravidelného odesílání testovacích dat na server s přidáváním časových razítek a následném měření času potřebného pro odesílání těchto dat a získávání následné odpovědi. Jinými slovy měří se čas potřebný k nahrání dat na server (lze označit jako *upload*) a poté čas potřebný k získání dat ze serveru (zde lze použít výraz *download*). Součet těchto hodnot pak představuje definici jedné hodnoty datové propustnosti sítě. Ve druhé fázi je vykonáváno vyhodnocení datové propustnosti, kdy je ukládáno vždy pět posledních měření do fronty a vypočítán jejich průměr. Samotné vyhodnocení je reprezentováno známkou, pro kterou platí, že čím vyšší hodnota známky, tím horší datová propustnost. Průměr hodnot ve frontě je při každém vyhodnocení porovnán s aktuální (poslední získanou) naměřenou hodnotou a na základě tohoto porovnání se buď inkrementuje nebo dekrementuje známka (zhorší nebo zlepší propustnost sítě). Kompletní výše navržený proces vychází z procesu vyhodnocení datové propustnosti v práci (5) a lze ho popsat těmito kroky:

Hybridní klient:

1. Vytvoření testovacích dat určité velikosti (ve vzorové aplikaci náhodný řetězec dané délky)
2. Získání aktuálního času (problémy a řešení při získání času a synchronizace času u serveru a klienta podrobněji v podkapitole *5.2.2 Problémy při vývoji* kapitoly *5.2 Modul vyhodnocení datové propustnosti*) – zde využito *Apache Commons Net API* (implementace rozebrána v kapitole *5.2.3 Vzorová implementace*)
3. Přidání časového razítka hybridního klienta před odesláním (pro výpočet upload času) a poté odeslání dat na centrální server

Centrální server:

4. Získání aktuálního času a přidání časového razítka po obdržení zprávy (pro výpočet upload času) – od času odečtena délka procesu získávání aktuálního času (z důvodu vyšší přesnosti a problematiky získávání času z NTP serveru)
5. Přidání časového razítka před odesláním zpětné odpovědi klientovi – bez odečtu času procesu získání aktuálního času (pro výpočet download času – tento krok není nutný, opět pouze z důvodu vyšší přesnosti a problematiky získávání přesného času)

Hybridní klient:

6. Přidání posledního časového razítka při obdržení odpovědi ze serveru (pro výpočet download času)
7. Výpočet download a upload hodnoty z časových razítek a jejich sečtení, čímž je vytvořena aktuální hodnota datové propustnosti
8. Je-li fronta naměřených hodnot prázdná
    a. Nastavení známky na 1
    b. Jinak výpočet průměru hodnot ve frontě, porovnání s aktuální naměřenou hodnotou a je-li aktuální hodnota větší než průměr
        i. Inkrementace známky o jedničku
        ii. Jinak dekrementace známky o jedničku
9. Pokud je ve frontě více než 5 hodnot
    a. Odstraní se první hodnota z fronty
10. Přidání aktuální hodnoty na konec fronty

Veškeré důvody, proč byl zvolen právě tento proces jsou zmíněny v práci (5). Nicméně bezesporu důležitými výhodami jsou relativně snadná implementace, a především nižší nároky na hardware mobilních zařízení (o důležitosti šetření zdrojů mobilních zařízení v souvislosti s hybridním klientem byla řeč již v kapitole *4.1.1.2 Hybridní klient v roli klienta v offline režimu*). Další předností je zvolená forma vyhodnocení v podobě známky, jejíž hodnotu určuje porovnávání předešlých hodnot, respektive jejich průměru, s aktuální naměřenou hodnotou. Díky postupnému formování

známky na základě posloupnosti několika měření totiž nezávisí vyhodnocení tolik na přesnosti jednotlivých měření a výpočtu rychlosti či doby datového přenosu mezi centrálním serverem a hybridním klientem. Z tohoto důvodu lze považovat krok 5 ve výše popsaném procesu, tedy přidání dalšího časového razítka, jako nepovinný. Stejně tak jako odečítání doby procesu získávání aktuálního času od získaného času v kroku 4.

Hodnota známky jakožto vyhodnocení stavu datové propustnosti sítě je po ukončení celého procesu předána zpět do centrálního modulu, který na základě této hodnoty rozhoduje o tom, zda spustit či nespustit proces rekonfigurace klienta, a to buď do podoby klienta v offline režimu nebo do role P2P klienta (záleží na možnostech daného klienta). Proces rekonfigurace hybridního klienta je realizován pomocí modulu dynamického zavádění knihoven.

4.3.3 Modul dynamického zavádění knihoven

Modul dynamického zavádění knihoven je nejdůležitějším doplňujícím modulem vnitřní struktury hybridního klienta. Jeho úkolem je provést samotnou rekonfiguraci klienta procesem zavedení knihovny. Chod modulu je vždy inicializován požadavkem z centrálního modulu, který vyhodnocuje (za využití dalších modulů), zda změnit stav aplikace či nikoliv. Nicméně konkrétní typ knihovny k zavedení stejně jako impuls pro zahájení procesu zavedení je dán rolí, do které má hybridní klient ztloustnout. V případě role klienta v offline režimu je zavedena knihovna pro offline režim a rekonfigurace je řízena vyhodnocením, respektive známkou zaslanou centrálnímu modulu modulem vyhodnocení datové propustnosti (fungování tohoto modulu je popsáno v předchozí kapitole *4.3.2 Modul vyhodnocení datové propustnosti*). Známka je využita i v případě rekonfigurace do role P2P klienta, avšak zaváděnou knihovnou je nyní knihovna pro peer-to-peer komunikaci. Tato knihovna je rovněž použita při transformaci klienta do poslední role, tedy do role P2P serveru, zde však podnět k tloustnutí nevychází z modulu vyhodnocení datové propustnosti, nýbrž komunikačního modulu, který v odpovědi na pravidelné požadavky na data (seznam klientů) z centrálního serveru zaznamená požadavek na ztloustnutí do podoby P2P serveru a předá tuto informaci centrálnímu modulu, jenž následně požádá modul dynamického zavádění knihoven o zavedení knihovny pro peer-to-peer komunikaci. Při každém úspěšném zavedení jakékoliv

knihovny je na centrální modul zaslána odpověď v podobě žádosti o změnu stavu (role) hybridního klienta.

Kompletní proces zavedení určité knihovny je iniciován požadavkem centrálního modulu, jehož obsahem je název knihovny, která má být zavedena. Jak lze vidět na *obrázku 7* v návrhu architektury hybridního klienta, modul pro dynamické zavádění knihoven má k dispozici seznam knihoven, jenž je uložen v *SQLite* databázi a ke kterému přistupuje prostřednictvím databázového modulu. Tento seznam obsahuje všechny knihovny, které již byly staženy a jsou součástí interního uložiště (paměťového média) hybridního klienta. Každý záznam o stažené knihovně zahrnuje tyto informace: Název knihovny, název *apk* souboru (pro účely zavedení skrze *DexClassLoader* – více v kapitole *5.3 Modul dynamického zavádění knihoven* v praktické části), název hlavní třídy (v architektuře na *obrázku 7* hlavní třída označena jako řídící třída) a stručný popis knihovny. První fází procesu je porovnání názvu knihovny pro zavedení se všemi uloženými názvy knihoven v databázi.

Pokud je knihovna v databázi obsažena (shoduje se název zaváděné knihovny s některým z názvů knihoven v databázi) stačí provést pouze proces zavádění. Pokud ale knihovna není v databázi, je nejprve nutné ji stáhnout z centrálního serveru, provést dekomprimaci (každá knihovna je k dispozici na centrálním serveru ve formátu ZIP pro snazší přenos mezi serverem a klientem) a uložit rozbalené soubory na paměťové médium hybridního klienta (vyhrazená složka *„libs“* v adresářovém prostoru vzorové aplikace. Poté jsou uloženy veškeré informace o knihovně ze souboru, představujícího deskriptor knihovny do seznamu knihoven v databázi (každá knihovna musí obsahovat soubor se všemi důležitými informacemi o knihovně pro účely dynamického zavádění knihoven – více o deskriptoru v kapitole *4.3.6 Externí knihovny*). Následně je již možné knihovnu zavést.

Celý proces zavádění včetně implementace ve vzorové aplikaci je uveden v praktické části v kapitole *5.3 Modul dynamického zavádění knihoven*. V principu se načte řídící třída dané knihovny pomocí komponenty *DexClassLoader* (základní prvek modularity na platformě Android – viz kapitola *3.2.3 Modularita v Android OS*), vytvoří se instance této třídy a uloží do datové struktury všech aktuálně zavedených knihoven, prostřednictvím níž může centrální modul přistupovat ke všem zavedeným knihovnám a řídit jejich chod na základě volání metod společného rozhraní (v *obrázku 7* označeno jako

„Rozhraní knihoven hybridního klienta“) hybridního klienta (více v kapitole *4.3.6 Externí knihovny*). Toto společné rozhraní pro všechny externí knihovny je využito i v poslední fázi dynamického zavádění knihoven, tedy při samotném nastartování běhu knihovny – to se provádí zavoláním metody *start()* na instanci zaváděné knihovny. Velkou výhodou společného rozhraní je jednotný přístup ke všem externím knihovnám, nicméně problém nastává ve chvíli, kdy určitá knihovna vyžaduje zvláštní přístup – například v podobě vícero vstupních dat nezbytných pro zajištění chodu knihovny (problém knihovny pro peer-to-peer komunikaci). Konkrétní problémy, které jsou spojeny se společným rozhraním externích knihoven a s praktickou realizací takto navrženého modulu dynamického zavádění knihoven, a které se vyskytly při vývoji vzorové aplikace, jsou rozebrány v kapitole *5.4.2 Problémy spojené s externími knihovnami*, respektive *5.3.2 Problémy při vývoji*.

4.3.4 Kontextový modul

Kontextový modul může a nemusí být implementován jako samostatný modul (v případě vzorové aplikace jsou jeho funkce součástí implementace centrálního modulu). Nicméně do tohoto modulu je možné zařadit docela obsáhlou množinu funkcí, proto je v architektuře hybridního klienta vyčleněn jako samostatný modul.

Kontextový modul slouží v podstatě jako jakýsi správce uživatelského kontextu a veškerého kontextu, který se váže k hybridnímu klientovi. V případě uživatelského kontextu jde především o data ze senzorů (tlakového a teplotního) pro účely funkcí IS a data z lokalizačního modulu (zpravidla GPS modul, v implementaci pro získání aktuální polohy použito *Google Location Services API*), která slouží navíc i pro potřeby hybridního klienta, a to konkrétně pro inicializaci procesu tloustnutí klienta do role P2P serveru. Jak již bylo zmíněno v kapitole *4.1.1.3 Hybridní klient v roli P2P serveru* podnět k rekonfiguraci klienta do této role přichází od jiného blízkého klienta v okolí, kdy pro určení vzdálenosti mezi klienty jsou využity právě zeměpisné souřadnice získané lokalizačním modulem.

Obecný kontext hybridního klienta, jenž je také velmi důležitý, se týká především aktuální role, kterou klient zastává a jednoznačného identifikátoru klienta (dále jen ID). Aktuální roli využívá centrální modul k definování chování klienta a k určení zobrazovaného obsahu, tedy k formování GUI. ID se využívá k identifikování hybridního

klienta v rámci systému. Vzhledem k tomu, že zajištění jedinečnosti klienta je pro správné fungování systému naprosto zásadní, je potřeba, aby byla zajištěna různost ID všech klientů v maximální možné míře. Zároveň je vhodné, aby algoritmus pro generování těchto ID nebyl příliš výpočetně náročný, neboť hybridní klient je i tak již dost komplexní a náročný na cílové mobilní zařízení (stále platí snaha minimalizovat zátěž na zdroje zařízení). Z tohoto důvodu je ID hybridního klienta tvořeno předponou *„hk“* (označení hybridního klienta) a následuje tzv. *IMEI (International Mobile Equipment Identity)* tak, jak je navrženo v práci (5). Dle (38) se jedná o unikátní číslo identifikující mobilní zařízení při připojení k mobilní síti. IMEI se váže k samotnému zařízení, a nikoliv k SIM kartě, což znamená, že je možné zařízení identifikovat či vystopovat nezávisle na vložené SIM kartě (38). Pokud by se nepodařilo IMEI z jakéhokoliv důvodu získat, zašle se na server pouze předpona „hk“ a centrální server vygeneruje pseudonáhodné číslo (tato varianta není příliš pravděpodobná, proto nejsou kladeny tak velké nároky na generátor a délku generovaného čísla). Jak ID, tak aktuální role jsou ukládány persistentně do interního uložiště na základě jednoduchého uložiště na bázi klíč-hodnota, které nabízí *SharedPreferences API*.

4.3.5 Komunikační modul

Komunikační modul umožňuje hybridnímu klientovi komunikovat s centrálním serverem v podobě HTTP komunikace. Modul zprostředkovává komunikační služby nejen centrálnímu modulu, ale rovněž modulu pro vyhodnocení datové propustnosti a modulu pro dynamické zavádění knihoven. Úkolem modulu je zasílání HTTP požadavků na centrální server a následné zpracování odpovědí ze serveru. Za tímto účelem je využita knihovna *Retrofit 2*. V případě služeb pro centrální modul se jedná o pravidelné zasílání (provedeno pomocí *TimerTask*) informací (kontextu) hybridního klienta a požadavku na aktuální seznam všech klientů – především pro účely IS, konkrétně pro zobrazení klientů na mapě tak, aby jejich poloha, hodnota teploty a tlaku byly co nejaktuálnější. Modul vyhodnocení datové propustnosti využívá komunikační modul k pravidelnému odesílání náhodných dat na centrální server a k získání časových razítek z odpovědi serveru (podrobně popsáno v kapitole *4.3.2 Modul vyhodnocení datové propustnosti*). Poslední funkcí tohoto modulu je stažení příslušné knihovny ze serveru při procesu tloustnutí na

základě požadavku modulu pro dynamické zavádění knihoven (zde se nepodařila implementace pomocí knihovny *Retrofit* – viz *5.3.2 Problémy při vývoji*).

Je zřejmé, že služby komunikačního modulu lze využívat pouze v případě připojení k centrálnímu serveru, tedy pokud se hybridní klient nachází v roli tenkého klienta, případně P2P serveru (tato varianta role P2P serveru není součástí implementace hybridního klienta ve vzorové aplikaci, nicméně takto navržená architektura hybridního klienta ji umožňuje realizovat).

4.3.6 Externí knihovny

Všechny externí knihovny, které mají být dynamicky zavedeny na takto navrženého hybridního klienta musí splňovat určité požadavky a jsou tak na ně kladeny určité nároky. Zaprvé je důležité zajistit efektivní přenos knihovny (všech souborů, které se ke knihovně vážou) skrze internetovou síť. Z tohoto důvodu je každá knihovna komprimována do formátu *„.zip“*. Po dekomprimaci je v hlavním souboru reprezentujícím danou knihovnu *apk* soubor, jenž obsahuje veškeré funkce knihovny (zkompilované všechny třídy, rozhraní apod.) a je zaváděn modulem pro dynamické zavádění knihoven skrze komponentu *DexClassLoader* (více v implementační praktické části v kapitole *5.3 Modul dynamického zavádění knihoven*). Dále je zde obsažen soubor označený jako *descriptor.txt*. Jedná se o textový soubor, v němž se nachází veškeré důležité informace o knihovně (popis knihovny – deskriptor). První položkou v deskriptoru je název knihovny, který je použit jako název vytvořeného adresáře v uložišti hybridního klienta při stažení knihovny ze serveru. V tomto adresáři jsou umístěny všechny soubory příslušné knihovny. Další dvě klíčové položky v popisujícím souboru slouží k samotnému načtení knihovny – jsou to názvy *apk* souboru a hlavní (řídící) třídy, skrze kterou je řízen chod knihovny. Přesněji řečeno hybridní klient komunikuje s externí knihovnou (řídí její běh) pouze skrze tuto třídu. Je nežádoucí, aby se ke každé knihovně, respektive její hlavní třídě přistupovalo jiným způsobem. Za tímto účelem je dobré navrhnout určitou deklaraci pravidel chování pro všechny hlavní třídy externích knihoven. Proto bylo navrženo společné rozhraní (*kód 1*), které musí implementovat každá hlavní třída, čímž se zavazuje k dodržování jednotného způsobu řízení běhu knihovny. Zbylé položky v souboru *descriptor.txt* jsou zejména informativní – jedná se o stručný popis fungování (či smyslu, účelu) a verzi knihovny.

```
import android.content.Context;

public interface LibraryLoaderInterface {

   int start(String path, Context context);

   int stop();

   int resume(String path, Context context);

   int exit();

   String getDescription();
}
```

Kód 1 - Rozhraní knihoven hybridního klienta

Výše navržené rozhraní pro externí knihovny hybridního klienta vychází z obdobné ideji v práci (5), nicméně je rozšířeno o vstupní parametry v metodě *start()*, která slouží k nastartování běhu knihovny. Parametry slouží k předání kontextu hybridního klienta do prostředí externích knihoven, což se ukázalo jako velmi vhodný až nezbytný krok při implementaci externích knihoven. První parametr představuje cestu k adresáři dané knihovny v uložišti hybridního klienta a nachází uplatnění například v těle metody *getDescription()* vracející popis fungování a účelu knihovny. Druhým vstupním parametrem je aplikační kontext. Bez tohoto atributu by nebylo možné přistupovat ke vnitřnímu kontextu hybridního klienta a manipulovat s jeho stavem (souvisí s platformou Android – problém by nastal například při vytváření či mazání databáze, nutné kontrole povolení od uživatele při přístupu k hardwaru či určitým funkcím zařízení atd.). Problém spojený se společným rozhraním je přiblížen v kapitole *5.4.2 Problémy spojené s externími knihovnami*. Další metodou je metoda *exit()* pro ukončení běhu knihovny. Zde je důležité především uvolnění prostředků mobilního zařízení, se kterými knihovna pracovala a které při své činnosti využívala. Poslední metody *stop()* a *resume()* slouží k zastavení, respektive znovuobnovení běhu knihovny. Všechny řídící metody společného rozhraní vrací celočíselnou hodnotu reprezentující úspěšnost provedení dané metody (ve vzorové aplikaci definovány dva možné výstupy – 0 pro úspěšné provedení a 1 pro chybné a neúspěšné provedení). Je vhodné poznamenat,

že tyto hodnoty informují o úspěšnosti vykonání kódu příslušné metody, a nikoliv o správném chodu knihovny. Dále budou představeny dvě základní knihovny nezbytné pro naplnění definovaných rolí hybridního klienta v kapitole *4.1.1 Podoby a role hybridního klienta*.

4.3.6.1 Knihovna pro offline režim

Tato knihovna slouží k realizaci podoby klienta popsané v kapitole *4.1.1.2 Hybridní klient v roli klienta v offline režimu*. Cílem role je zastoupení funkcí centrálního serveru do takové míry, aby klient byl schopný na omezený čas fungovat bez přítomnosti serveru. Z kapitoly *4.2 Funkce vzorového informačního systému* je patrné, že jedinou stěžejní funkcí serveru v rámci vzorového IS je správa klientů, díky čemuž je struktura této knihovny relativně snadná (složitost architektury knihovny je samozřejmě závislá na funkcích centrálního serveru, což znamená, že může být mnohem komplikovanější). U klienta v offline režimu se navíc předpokládá úplná izolace od ostatních prvků v systému, tudíž stačí pouze schopnost udržování seznamu všech klientů v uložišti hybridního klienta. Jádrem knihovny je tedy řídící (hlavní) třída, která implementuje metody společného rozhraní, čímž je řízeno fungování knihovny. Součástí implementovaných metod jsou veškeré funkce serveru v rámci IS (především persistentní uložení seznamu klientů, aby bylo možné vždy zobrazit klienty na mapě) a pro přístup k databázi je vyčleněn databázový modul.

Knihovna je však využita rovněž pro naplnění role klienta z kapitoly *4.1.1.3 Hybridní klient v roli P2P serveru*. V této podobě je úkolem klienta zastoupit centrální server nejen z pohledu funkcionalit IS, ale rovněž z pohledu samotného konceptu hybridního klienta a celého navrženého systému z *obrázku 4*. Takovýto klient by měl představovat jakýsi záložní centrální server, který bude sbírat a udržovat informace o okolních klientech, kteří ztratili připojení k serveru (opět využit seznam klientů – nyní však navíc modifikován). Z této úlohy hybridního klienta plyne jeho povinnost synchronizace získaných dat s centrálním serverem ihned při obnoveném připojení, neboť centrální server by měl vždy obsahovat co nejaktuálnější data o všech klientech v systému. Synchronizace se serverem by tak měla proběhnout ihned při procesu hubnutí klienta (nejsou již vyžadovány jeho služby – všechna zařízení jsou opět připojena k serveru) a proto je součástí implementace metody *exit()*, tedy ukončení běhu knihovny – zhubnutí hybridního klienta zpět do podoby tenkého klienta.

4.3.6.2 Knihovna pro peer-to-peer komunikaci

Bez této knihovny by nebylo možné uskutečnit podoby klienta z kapitol *4.1.1.3 Hybridní klient v roli P2P serveru* a *4.1.1.4 Hybridní klient v roli P2P klienta*. Jejím úkolem je zajištění komunikace právě mezi těmito rolemi. Za tímto účelem je využita technologie *Wi-Fi Direct*, jež umožňuje přímé spojení více zařízení s technologií *Wi-Fi*, aniž by se tato zařízení připojovala k tradiční domácí, kancelářské nebo veřejné síti (39). Využití technologie Wi-Fi oproti ostatním variantám (například *Bluetooth*) nabízí určité výhody. Jednou z nich je například větší maximální vzdálenost mezi zařízeními potřebná pro úspěšné připojení (právě oproti Bluetooth) (40). Android nabízí pro vývoj peer-to-peer komunikace na bázi *Wi-Fi Direct* soubor funkcí, který je označen jako *Wi-Fi Direct (peer-to-peer či P2P) API* (40).

Struktura knihovny je uzpůsobena právě zmíněným implementačním technologiím platformy Android – z tohoto důvodu je zde velmi úzké spojení mezi architekturou knihovny a platformou Android. Implementované metody společného rozhraní v řídící třídě se starají o inicializaci, přerušení, znovuspuštění a zastavení komunikačního procesu (registrace receiverů, správa vláken a důležitých objektů pro komunikaci – více v praktické části v kapitole *5.4.4 Knihovna pro peer-to-peer komunikaci*). Krom řídící třídy je v knihovně komunikační modul, sloužící k definici chování rolí v komunikaci – role serveru a klienta (ekvivalentní k roli P2P serveru a P2P klienta v této práci) a k vymezení podoby či formy zasílaných zpráv. Komunikační modul slouží také k zasílání a přijímání zpráv, kdy po obdržení dané zprávy je zpráva předána do řídící třídy, která se postará o další předání do prostředí aplikace (hybridního klienta). Poslední důležitou komponentou této knihovny je *P2P Receiver*, který umožňuje naslouchat všem důležitým událostem v souvislosti s peer-to-peer komunikací a který vyvolává veškeré operace (ve skutečnosti je součástí implementace hned několik receiverů, tento je však naprosto stěžejní). Zjednodušená struktura včetně komunikace mezi jednotlivými komponentami je součástí *obrázku 7* a konkrétní implementační kroky jsou popsány v kapitole *5.4.4 Knihovna pro peer-to-peer komunikaci* a v jejich podkapitolách v praktické části.

4.3.7 Android Manifest

Android Manifest představuje hlavní konfigurační soubor každé aplikace na platformě Android, neboť popisuje základní informace o aplikaci pro operační systém Android, jeho nástroje pro sestavení aplikace (*Gradle*) a obchod s aplikacemi *Google Play* (41). Dle (41) musí být v manifestu uvedeno následující: Název balíčku aplikace, komponenty aplikace (nejčastěji *Aktivity*), veškerá oprávnění, která aplikace potřebuje pro přístup k chráněným částem systému a hardwarové a softwarové funkce, které aplikace vyžaduje. V souvislosti s modularitou, tedy dynamickým zaváděním dalších aplikačních částí se jeví problematické hlavně udávání komponent a oprávnění, neboť hybridní klient takto musí mít přehled o struktuře externích knihoven a jejich požadavcích na určité systémové prvky. Z pohledu této práce není udávání komponent problematické, protože v implementaci externích knihoven jsou používány pouze tradiční Java třídy či rozhraní, které nepotřebují být pro svou správnou funkci nikde deklarovány. Pokud by však byla vytvořena další externí knihovna (což navržená architektura umožňuje), jež by využívala jakoukoliv aplikační komponentu platformy Android, musela by být tato komponenta uvedena nejen v manifestu dané knihovny, ale také přímo v manifestu hybridního klienta. Tuto myšlenku ostatně potvrzuje práce (42). Problém související s nutností uvádět veškerá oprávnění se však již této práce týká, neboť součástí implementace knihovny pro peer-to-peer komunikaci jsou požadavky na povolení uživatele související s technologií Wi-Fi či se získáním aktuální polohy zařízení (viz kapitola *5.4.4.2 Vzorová implementace* v praktické části). Veškerá tato povolení je proto nutné uvést navíc v manifestu hybridního klienta.

4.4 Návrh architektury centrálního serveru

Architektura centrálního serveru se zaměřuje především na plnění funkcí či služeb v rámci vzorového IS navrženého v kapitole *4.2 Funkce vzorového informačního systému*. Zároveň jsou implementovány stěžejní funkce centrálního serveru jako řídícího prvku a komunikačního prostředníka ve spojení s navrženou architekturou (*obrázek 4*) pro podporu konceptu hybridního klienta. Značnou výhodou je, že funkce pro oba tyto účely jsou si velmi podobné, což umožnilo architekturu významně zjednodušit. Mezi tyto funkce patří správa a udržování aktuálního seznamu klientů a z velké části i vytváření či registrování nových klientů (u centrálního serveru jako řídícího prvku je navíc potřeba

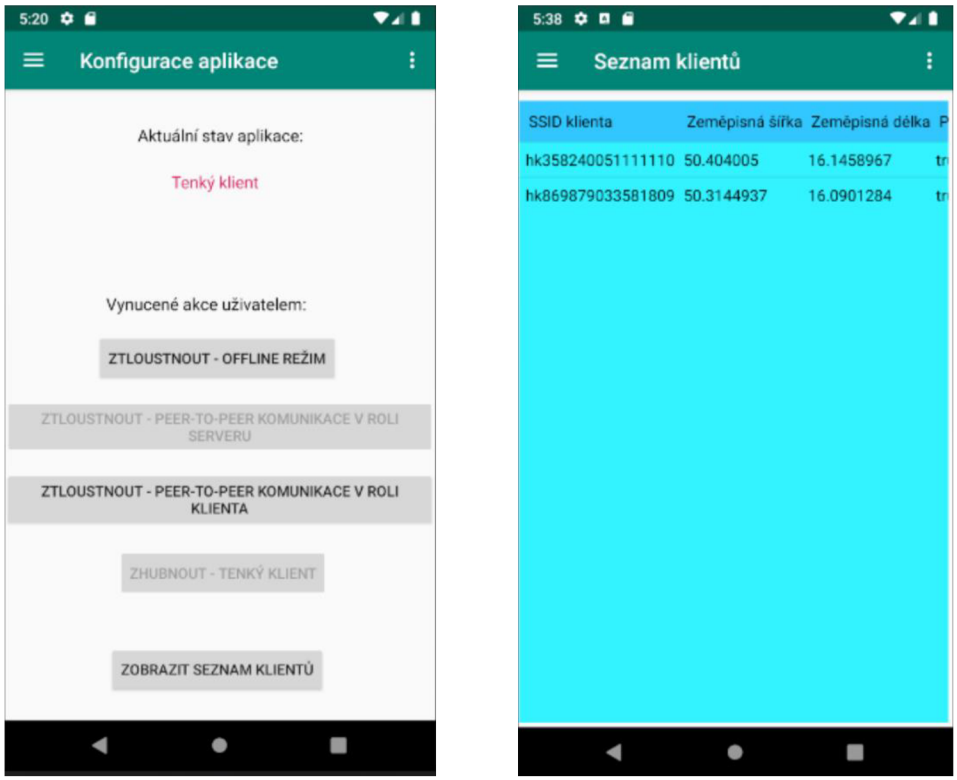
generovat ID v případě, že se ho nepodařilo vytvořit na straně klienta, což by v rámci funkce IS nebylo nutné). V implementaci centrálního serveru byla vynechána jedna důležitá funkce, která souvisí s rolemi P2P serveru a P2P klienta. Jedná se o rozhodování o zahájení procesu ztloustnutí klienta do podoby P2P serveru, které je iniciováno klientem v roli P2P klienta (funkce je specifikována v kapitole *4.1.1.3 Hybridní klient v roli P2P serveru*). Centrální server je v tomto případě zprostředkovatelem komunikace mezi klienty. Implementace by mohla vypadat následovně: Nejprve by bylo nutné implementovat odchytávání požadavku na ztloustnutí jak na straně centrálního serveru (požadavek od klienta v roli P2P klienta), tak na straně potenciálně vhodného hybridního klienta (požadavek od centrálního serveru). S tím, že důležitou součástí funkcionality je výpočet vzdálenosti mezi klienty, kontrola aktuální či možné budoucí podoby klientů (obě informace k dispozici na klientovi i serveru) a následné vyhodnocení, zda existuje takový klient, který splňuje požadavky pro rekonfiguraci do podoby P2P serveru (je dostatečně blízko a je toho schopný). Pokud se takový adept nalezne, pak je potřeba ho o ztloustnutí požádat. Dalo by se rovněž polemizovat o rozdělení implementačních částí, neboť většinu z nich je možné umístit jak na hybridního klienta, tak na centrální server (vždy k dispozici seznam klientů). Nicméně nejvhodnější variantou se zdá být právě centrální server, a to především kvůli jeho větším výpočetním možnostem a snahou maximálního šetření zdrojů mobilních zařízení jakožto hybridního klienta. Tato funkce byla v implementaci (jak centrálního serveru, tak hybridního klienta) vynechána ne proto, že by návrh řešení či jeho implementace byla příliš komplikovaná, ale z toho důvodu, že tato práce se zaměřuje především na návrh hybridního klienta v podobě mobilní aplikace, kdy server slouží zejména k načrtnutí dalších možností využití tohoto konceptu.

Centrální server představuje *Spring Boot* (43) aplikaci v podobě *REST API*. Detailní popis architektury aplikací implementovaných ve frameworku *Spring* není předmětem této práce, proto je dále zmíněn jen stručný výčet nejdůležitějších komponent *Spring* frameworku a jejich účel, neboť implementace centrálního serveru je členěna do balíčků právě na základě typů komponent, respektive jejich rolí v aplikaci (výjimku tvoří balíček *service*). První balíček (*controller*) je tvořen komponentami s označením *Controller* (44). V terminologii architektury hybridního klienta by tento balíček představoval komunikační modul. Jeho úlohou je komunikace s klienty (přijímání HTTP požadavků a odeslání HTTP odpovědí), zpracování přijímaných zpráv a předání dat do dalšího

modulu, který představuje balíček *model*. Ten obsahuje modelové třídy související s informacemi o klientech v podobě tzv. *Entit* (45). Jedná se o běžné Java třídy představující data, která lze persistentně ukládat do databáze, kdy jedna entita představuje jednu tabulku v databázi a každá instance entity odpovídá jednomu řádku v tabulce (45). Pro přístup k datům (implementaci datové vrstvy) je využito *Spring Data JPA* (46), konkrétně komponenta *Repository*, prostřednictvím níž jsou prováděny veškeré operace nad daty. Poslední modul či balíček je označen jako *service*. Toto označení tentokrát nikterak nesouvisí s typem komponenty, což může být poněkud zavádějící, neboť ve *Spring* frameworku komponenta s tímto označením existuje (viz (47)). Úkolem tříd v tomto balíčku je poskytování podpůrných služeb uvnitř centrálního serveru – v případě vzorové aplikace pouze služba získání aktuálního času z NTP serveru pro potřeby formulace odpovědi na požadavek hybridního klienta ohledně otestování stavu datové propustnosti.

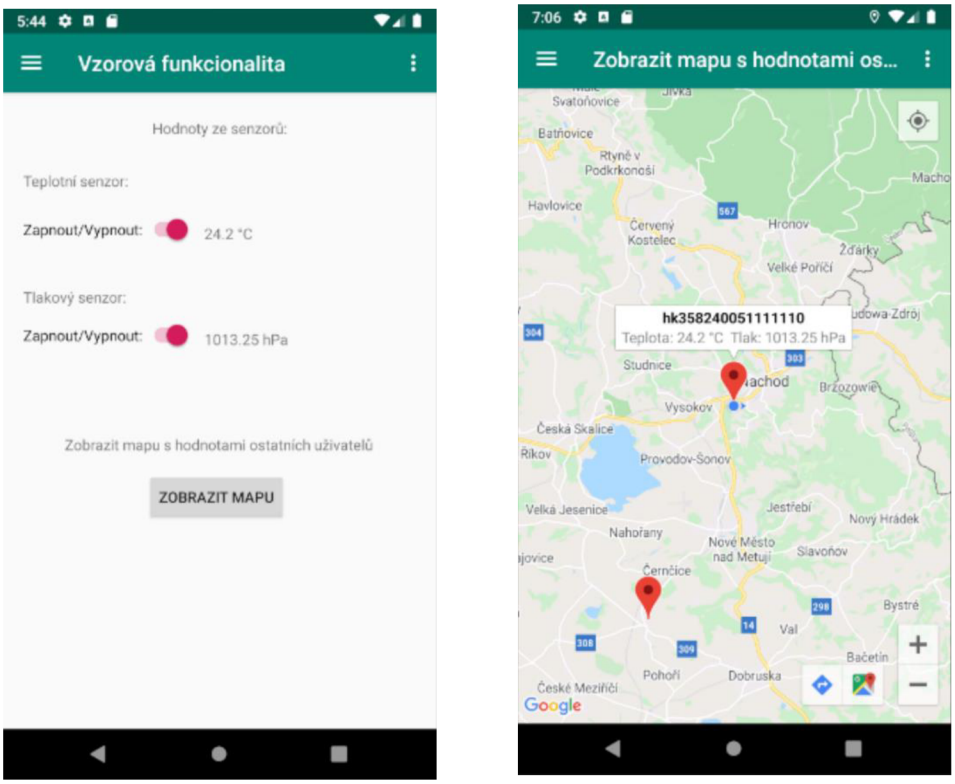
4.5 Vizualizace vzorové aplikace

Tato kapitola se věnuje náhledu obrazovek vzorové aplikace. Cílem aplikace je především otestování možností a funkčnosti konceptu hybridního klienta tak, jak byl navržen výše. Aplikace je vyvíjena takovým způsobem, aby měla co nejlepší testovací schopnosti. Z čehož plyne jednoduchý, ale výstižný a přehledný design, respektive rozvržení GUI komponent na obrazovkách. Vizualizace obrazovek je rozdělena do dvou částí, a to dle funkcí hybridního klienta. Na *obrázku 8* lze pozorovat vizualizaci funkcí týkajících se konfigurace aplikace, respektive hybridního klienta. Funkce na *obrázku 9* naopak představují služby klienta v rámci služeb IS.



Obrázek 8 – vizualizace vzorové aplikace – konfigurace aplikace (vlevo) a seznam klientů (vpravo)

Jak již bylo zmíněno v některých podkapitolách kapitoly *4.1.1 Podoby a role hybridního klienta*, vzorová aplikace by měla umožňovat statickou rekonfiguraci prostřednictvím interakce uživatele. Přesně k tomuto účelu slouží sekce „vynucené akce uživatelem“, která je součástí obrazovky vlevo na *obrázku 8*. Aktuální stav klienta je vždy zobrazen v horní části obrazovky. Dále je možné zobrazit seznam všech klientů, jenž má aplikace k dispozici – buď prostřednictvím centrálního serveru nebo od jiného klienta v roli P2P serveru. Obrazovka sloužící k vizualizaci seznamu klientů je vpravo na *obrázku 8* (v seznamu je možné se posouvat pomocí posuvníku).



Obrázek 9 - vizualizace vzorové aplikace – funkcionalita IS (vlevo) a mapa s hodnotami ostatních klientů (vpravo)

Na *obrázku 9* je znázorněna vzorová funkcionalita IS na hybridním klientovi. Obrazovka vlevo umožňuje ovládat oba stěžejní senzory, kdy v případě aktivace daného senzoru jsou získané hodnoty vypsány vedle přepínače ovládajícího stav tohoto senzoru. Následuje možnost zobrazení mapy s markery ostatních uživatelů IS, kdy po rozkliknutí příslušného markeru lze vidět krom ID klienta hodnoty ze senzorů jejich mobilního zařízení. Náhled této obrazovky je vykreslen na *obrázku 9* vpravo.

5 Praktická část

Praktická část se zabývá již konkrétními postupy při vývoji hybridního klienta, respektive vzorové aplikace. Představuje implementaci jednotlivých kroků v každé aplikační části, s čímž se pojí rovněž některé problémy, které se při vývoji jednotlivých částí aplikace vyskytly a které jsou rovněž zmíněny v této kapitole.

5.1 Nástroje pro vývoj

Nejprve je vhodné představit nástroje a technologie, které byly použity pro vývoj, nasazení a testování aplikace. Aplikace byla vyvíjena ve vývojovém prostředí *IntelliJ IDEA* od společnosti *JetBrains*. Pro nasazení a testování funkčnosti aplikace bylo využito hned několik mobilních zařízení, a to jak fyzických, tak virtuálních. Všechna užitá zařízení včetně jejich hlavních důležitých charakteristik v kontextu vývoje hybridního klienta jsou součástí *tabulky 1*. Více testovacích zařízení bylo zvoleno ze dvou důvodů. Zaprvé je tím umožněno ověření funkcionalit hybridního klienta na více verzích operačního systému Android, díky čemuž je možné ověřit, že idea hybridního klienta je realizovatelná na platformě Android v globálnějším měřítku (to znamená nikoliv pouze na jednom zařízení a na jedné verzi Android OS). Jako minimální podporovaná verze operačního systému byla zvolena verze *4.4 KitKat (úroveň API 19)*, cílovou verzí je *Android 10 (úroveň API 29)*. Druhým důvodem je složitost některých funkcí, kdy je nutné k testování využít více zařízení (například testování P2P komunikace mezi dvěma klienty).

Tabulka 1 - Charakteristika testovacích zařízení

Typ zařízení	Značka	Model	Verze Android OS	Podpora GPS	Podpora senzorů
Fyzické	Huawei	SCL-L21	5.1.1 (API 22)	Ano	Ne
Fyzické	Huawei	FIG-LX1	9.0.0 (API 28)	Ano	Ne
Virtuální	Nexus	Nexus 5	9.0.0 (API 28)	Ano	Ano

Pro testování byla nejprve vybrána dvě fyzická zařízení a to tak, aby bylo pokryto, pokud možno co největší rozmezí verzí Android OS (proto jedno starší zařízení s nižší úrovní API a jedno naopak novější s vyšší verzí OS – viz *tabulka 1*). U obou zařízení nicméně došlo k zjištění, že nepodporují oba stěžejní senzory pro účely funkcí aplikace v rámci služeb IS, tedy senzory pro snímání ambientní teploty a ambientního tlaku – atribut zařízení je v *tabulce 1* označen jako „Podpora senzorů“. Toto zjištění bylo následně ověřeno v (48) a (49) a představovalo komplikaci pro vývoj vzorové aplikace. Z tohoto důvodu byla funkcionalita ověřena v prostředí Android emulátoru na virtuálním zařízení Nexus 5, kde je možné modifikovat hodnoty většiny senzorů, jenž jsou podporovány na platformě Android (36).

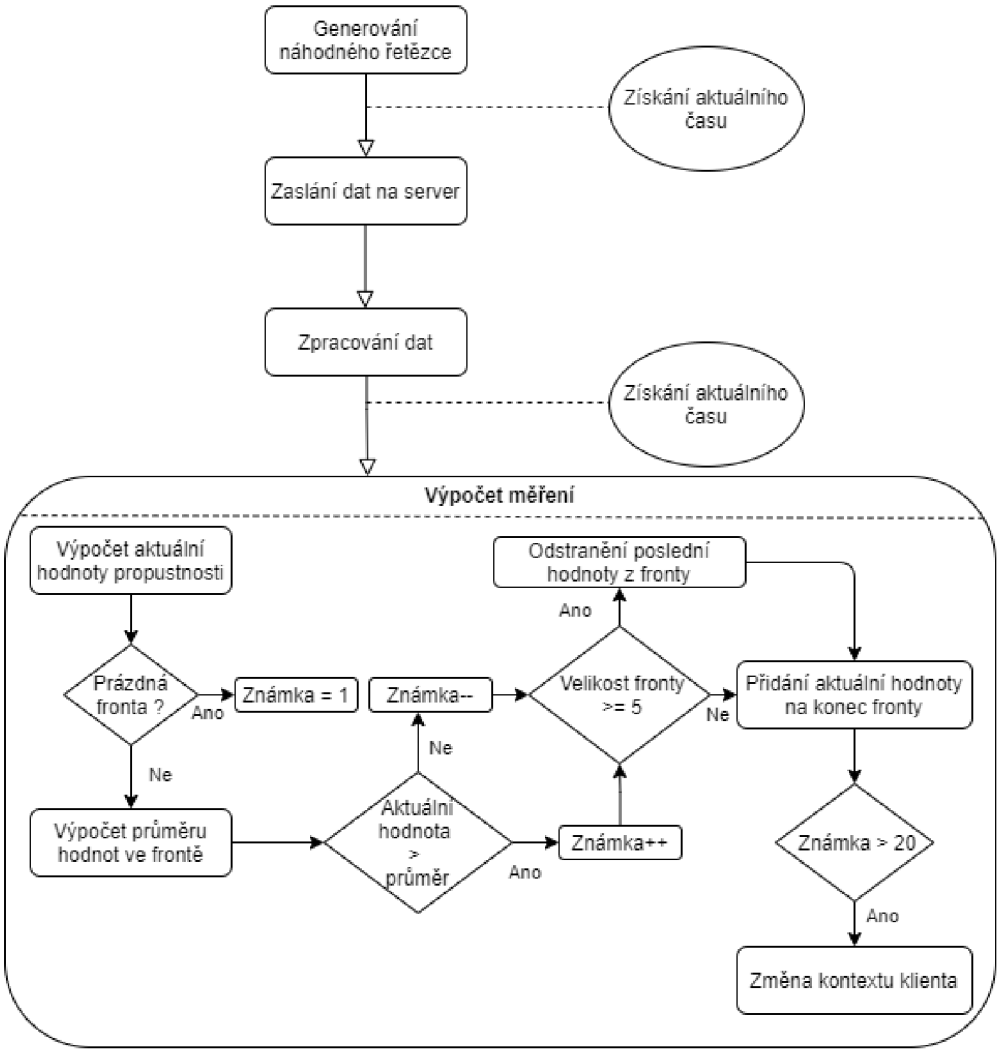
Ve zbylých podkapitolách praktické části budou představeny stěžejní implementační části, v nichž bude zmíněn postup ve vývoji (tzv. *workflow*), dále budou uvedeny případné problémy, jež se vyskytly během vývoje, a v neposlední řadě bude součástí kapitol samotná implementace ze vzorové aplikace, tedy zdrojový kód nejdůležitějších kroků z postupu vývoje dané implementační části.

5.2 Modul vyhodnocení datové propustnosti

První funkcionalitou, jejíž vývoj bude představen, je vyhodnocení datové propustnosti. Postup a samotná implementace vychází z teoretického konceptu navrženého v kapitole *4.3.2 Modul vyhodnocení datové propustnosti*, ve které je mimo jiné popsán kompletní postup celého procesu. Tato kapitola se zaměřuje pouze na stranu hybridního klienta, ve kterém je proces realizován za pomoci modulu pro vyhodnocení datové propustnosti.

5.2.1 Workflow vývoje

Všechny navržené a následně implementované kroky, které jsou znázorněny na *obrázku 10*, jsou součástí třídy, jež ve vzorové aplikaci nese název *„TestDataTimerTask“*. Při pohledu na proces vyhodnocení datové propustnosti v kapitole *4.3.2 Modul vyhodnocení datové propustnosti* a na navržené schéma na *obrázku 10* je patrné, že tato třída zaštiťuje nejen funkce modulu pro vyhodnocení datové propustnosti, ale rovněž funkce komunikačního modulu, které slouží pro podporu procesu vyhodnocení datové propustnosti (týká se zasílání a zpracování dat).



Obrázek 10 - workflow vývoje vyhodnocení datové propustnosti u hybridního klienta

5.2.2 Problémy při vývoji

Pro měření datové propustnosti je nutné zajistit synchronizaci času mezi klientskou stanicí (hybridním klientem) a centrálním serverem. K tomuto účelu bylo využito získání aktuálního času z NTP serveru za pomocí *Apache Commons Net API* (implementace rozebrána v následující kapitole, na úrovni hybridního klienta má proces na starost třída *„CurrentTimeProvider“*). V souvislosti se získáním aktuálního času z NTP serveru se však objevil problém, neboť při testování funkce se několikrát stalo, že čas ze serveru se nepodařilo získat (bylo testováno několik serverů). Konkrétně se jednalo o výjimku *java.net.SocketTimeoutException* (50), která signalizuje, že na soketu pro připojení došlo k vypršení daného časového limitu před úspěšným navázáním spojení (umělé navýšení tohoto limitu nevedlo k řešení problému). Řešením se stalo opakované

připojování k NTP serveru a snaha o znovuzískání aktuálního času v cyklu, což ovšem vedlo k umělému navyšování času a tím pádem ke zkreslení skutečného aktuálního času (nehledě na to, že proces trval na různých zařízeních různě dlouhou dobu). K úplné synchronizaci časů a k přesnějším hodnotám měření datové propustnosti tak bylo nutné, respektive vhodné (vzhledem ke způsobu vyhodnocování datové propustnosti není striktně přesný čas nutný, neboť propustnost je vyhodnocena na základě vícero po sobě jdoucích měření – viz kapitola *4.3.2 Modul vyhodnocení datové propustnosti*), aby byly do procesu zahrnuty nové časové údaje, jež představují skutečný přesný aktuální čas, tedy po odečtu časového intervalu reprezentujícího proces získávání aktuálního času z NTP serveru (pro výpočet doby procesu byl použit velmi přesný systémový čas v nanosekundách). Ve výsledku tak potom zasílaný datagram mezi klientem a serverem obsahuje celkem čtyři časová razítka. Dvě razítka pro určení hodnoty *upload* (doba zaslání dat na server) – čas odeslání dat od klienta a skutečný čas obdržení dat na serveru (tedy po odečtu). A další dvě razítka pro výpočet hodnoty *download* (doba získání dat ze serveru) – čas odeslání dat ze serveru a skutečný čas získání datagramu na klientovi (opět po odečtu). Tímto způsobem bylo docíleno přesnějších hodnot datové propustnosti.

5.2.3 Vzorová implementace

Prvním krokem v postupu vývoje na *obrázku 10* je generování náhodného řetězce. Ve vzorové aplikaci se jedná o metodu s názvem *„getRandomString(int size)“*, jež generuje řetězec z předem definované množiny povolených znaků s danou délkou řetězce v parametru metody. Pro vytváření řetězce je použita instance Java třídy *StringBuilder* a pro náhodné rozmístění znaků slouží instance třídy *Random*. Zasílání a získávání dat ze serveru spadá pod služby komunikačního modulu, který k tomuto účelu využívá knihovnu *Retrofit 2* (dále jen *Retrofit*). Vzorová implementace komunikace se serverem za využití knihovny *Retrofit* je předmětem *kódu 3*. K tomu, aby bylo možné volat služby serveru je nejprve nutné konfigurovat klíčový objekt *Retrofit* knihovny, skrze který budou všechna volání realizována a následně definovat API, jež bude odpovídat poskytovaným službám serveru – tyto implementační kroky jsou předmětem *kódu 2*.

```
// Konfigurace Retrofit objektu
// nastavení gson converteru (JSON -> Java) pro Date formát
Gson gson = new GsonBuilder()
        .setDateFormat("yyyy-MM-dd'T'HH:mm")
        .create();

// init a nastavení retrofit objektu pro připojení k serveru
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("http://10.0.2.2:8080/") // localhost alias pro AVD
        .addConverterFactory(GsonConverterFactory.create(gson))
        .build();

// rozhraní reprezentující API centrálního serveru
public interface IsCentralServerApi {
   @GET("users/all")
   Call<List<User>> getUsers();

   @Headers("Content-Type: text/plain")
   @POST("connection/test")
   Call<ResponseBody> makeTest(@Body RequestBody body);

   @Headers("Content-Type: application/json")
   @PUT("users/update")
   Call<ResponseBody> updateUser(@Body User user);

   @Headers("Content-Type: application/json")
   @POST("users/create")
   Call<ResponseBody> createUser(@Body User user);
}

// naplnění těl metod prostřednictvím retrofit objektu
IsCentralServerApi isCentralServerApi = retrofit.create(IsCentralServerApi.class);
```

Kód 2 - Příprava komunikace se serverem pomocí Retrofit knihovny

Předávaná data mezi klientem a centrálním serverem jsou zpravidla v podobě *JSON* objektů. Pokud tedy vzorová aplikace, jakožto hybridní klient chce zasílat data na server, potažmo data přijímat, je potřeba zajistit transformaci mezi Java třídami a JSON objekty. Knihovna *Retrofit* nabízí velmi elegantní řešení, kdy lze využít objekt třídy *Gson*, jenž zajišťuje převod automaticky, bez nutnosti manuálního vytváření JSON objektů v kódu Javy. V ukázce *kódu 2* je rovněž možné vidět, že tento objekt umožňuje nastavit či upravit

formu převodu (v tomto případě je definován formát pro ukládání datumu). V *kódu 2* je dále vytvořen zmíněný klíčový objekt *Retrofit* knihovny, kterému stačí pouze nastavit URL serveru a nástroj, zajišťující již zmiňovanou transformaci, tedy v tomto případě, nakonfigurovaný *Gson* objekt. Poté již stačí definovat rozhraní dle služeb serveru a naplnit těla metod rozhraní prostřednictvím metody *create()* volané na vytvořeném *Retrofit* objektu. Právě tyto metody potom slouží k provolávání služeb serveru – viz *kód* 3.

Pro volání služby serveru je nejprve nutné definovat tělo požadavku na server, respektive formulovat zasílaná data a jejich formát (v *kódu* 3 jsou data odesílána jako obyčejný text, neboť se jedná o testovací data s časovými razítky, ve většině případů jsou data odesílána jako JSON objekty). V případě zasílaných dat u modulu pro vyhodnocení datové propustnosti obsahuje datagram nejprve vygenerovaný náhodný řetězec, za kterým následují časová razítka v pořadí, v jakém jsou přidávány hybridním klientem a centrálním serverem v procesu zjišťování datové propustnosti (viz předchozí kapitola *5.2.2 Problémy při vývoji*), kdy každý údaj je oddělen lomítkem. Pro volání příslušné služby serveru je použit objekt třídy *Call* a příslušná metoda z rozhraní veškerých služeb serveru, jež je volána na instanci tohoto rozhraní (ukázka rozhraní je součástí *kódu 2*), kdy do parametru volané metody je umístěn objekt reprezentující tělo požadavku. Třída *Call* má k dispozici klíčovou metodu *enqueue()*, jež zprostředkovává asynchronní volání služby serveru a informuje *Callback* třídu o odpovědi ze serveru s tím, že tato komunikace probíhá na novém vlákně (předejití výjimce *NetworkOnMainThreadException* (51), která je vyvolána v případě, že se aplikace pokusí provést síťovou operaci na svém hlavním vlákně). Právě skrze zmíněnou *Callback* třídu je možné zachytávat odpověď serveru a to jak v případě úspěšného volání (metoda *onResponse()*), tak v případě výskytu neočekáváné chyby při navázaní spojení se serverem či zpracování požadavku, respektive odpovědi (metoda *onFailure()*).

Základ naznačené konstrukce zasílání a získávání dat z *kódu 3* je použit v podstatě při každé komunikaci klienta s centrálním serverem, kdy výjimku tvoří pouze proces stažení knihovny z centrálního serveru (viz kapitola *5.3.2 Problémy při vývoji*).

```
// vytvoření objektu requestBody pro odeslání dat na server ve správném formátu
RequestBody requestBody = RequestBody.create(postedData,
MediaType.parse("text/plain"));
 // request na server
 Call<ResponseBody> call = isCentralServerApi.makeTest(requestBody);

// zpracování response ze serveru
call.enqueue(new Callback<ResponseBody>() {
   // pokud dostaneme response (nemusí být úspěšný)
   @Override
   public void onResponse(@NotNull Call<ResponseBody> call, @NotNull
   Response<ResponseBody> response) {
      // kontrola zda response je neúspěšný
      if (!response.isSuccessful()) {
         // zobrazíme chybový HTTP kód a návrat z metody
         alertMessage = "HTTP kód: " + response.code();
         return;
      }
      if (response.body() != null) {
         // uložení response ze serveru
         testDataFromServer = response.body().string();
          // zpracování response
          . . . .
       } else {
          Log.e(TAG, "Response body je null");
       }
    }
    // pokud při spojení či zpracování požadavku došlo k chybě
    @Override
    public void onFailure(@NotNull Call<ResponseBody> call, @NotNull Throwable t) {
       Log.e(TAG, "Nepodařilo se připojit k serveru: " + t.getMessage());
    }
 });
```

Kód 3 - Ukázka komunikace se serverem za použití Retrofit knihovny

Pro získání aktuálního času z NTP serveru je použito *Apache Commons Net API*. Prvně je potřeba vytvořit komunikační soket, jakožto instanci třídy *NTPUDPClient* a následně otevřít soket využitím metody *open()*. Vzhledem ke zmiňovaným problémům v předešlé kapitole *5.2.2 Problémy při vývoji* je provedeno několik pokusů o získání aktuálního času prostřednictvím metody *getTime(String název_domény_serveru)*.

Vzorová implementace i se zachycením výjimek, jež mohou nastat, je naznačena v *kódu 4*. Po dokončení komunikace s NTP serverem je nutné komunikační soket opět zavřít. Časové razítko z NTP serveru je možné získat z objektu třídy *TimeInfo*, jenž obsahuje data ze serveru, a pomocí metod *getMessage()* a *getTransmitTimeStamp()*, které umožňují získat NTP časové razítko. To je dále potřeba konvertovat na časové razítko jazyka Javy (třída *TimeStamp*), se kterým je možné v aplikaci dále pracovat. Postup získání času a následná konverze je taktéž součástí *kódu 4*. Pokud se nepodaří získat čas ze serveru ani po všech daných pokusech, je použit systémový čas zařízení.

```
NTPUDPClient timeClient = new NTPUDPClient();
try {
   timeClient.open(); // otevřít soket pro komunikaci
   // pokus o připojení na server (více pokusů)
   for (int i = 0; i < 4; i++) {
      try {
         // získání NTP serveru
         InetAddress inetAddress = InetAddress.getByName(NTP_SERVER_DOMAIN);
         timeInfo = timeClient.getTime(inetAddress);  // získání času ze serveru
         } catch (UnknownHostException uhe) {
            Log.d(TAG, "Adresa NTP serveru je neznámá: " + uhe);
         } catch (IOException ioe) {
            Log.d(TAG, "Nepodařilo se získat čas ze serveru: " + ioe);
         }
      if (timeInfo != null) {
         // získáný čas ze serveru
         timeClient.close(); // zavřít soket a ukončit komunikaci
         return new Timestamp(timeInfo.getMessage().getTransmitTimeStamp().getTime());
      }
   }
} catch (SocketException se) {
   Log.d(TAG, "Nepodařilo se otevřít soket pro komunikaci s NTP serverem: " + se);
}
```

Kód 4 - Ukázka získání aktuálního času z NTP serveru pomocí Apache Commons Net API

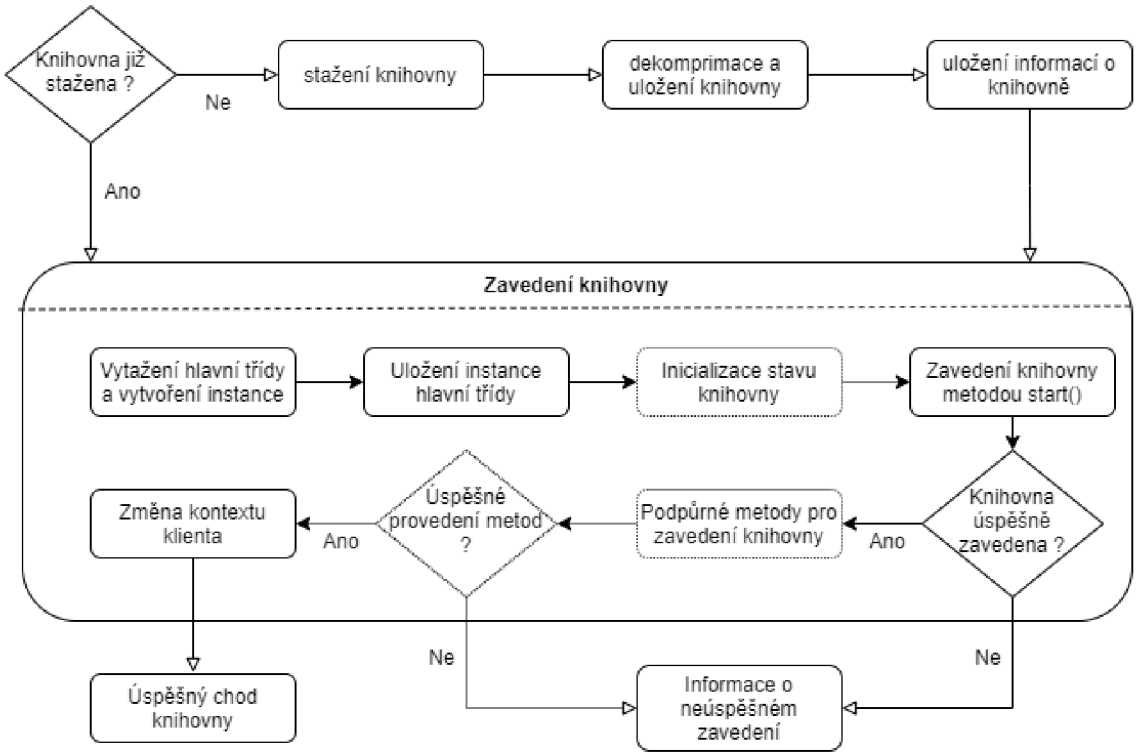
Implementace vyhodnocení měření datové propustnosti zcela přesně odpovídá implementačním krokům, které jsou znázorněny ve schématu postupu vývoje na *obrázku 10* pod názvem „výpočet měření“.

5.3 Modul dynamického zavádění knihoven

Pravděpodobně nejkritičtějším modulem hybridního klienta je modul dynamického zavádění knihoven, neboť právě jeho prostřednictvím dochází k rekonfiguraci hybridního klienta (více v kapitole *4.3.3 Modul dynamického zavádění knihoven*). Z tohoto důvodu je vhodné implementaci a fungování modulu podrobněji rozebrat.

5.3.1 Workflow vývoje

Na *obrázku 11* je znázorněn postup vývoje modulu pro dynamické zavádění knihoven. Během vývoje bylo zjištěno, že některé externí knihovny pro svoji správnou funkčnost vyžadují zavést dodatečné implementační části – tyto kroky jsou v *obrázku 11* znázorněny přerušovanou čarou a problematika je hlouběji popsána v následující kapitole *5.3.2 Problémy při vývoji*.



Obrázek 11 - workflow vývoje modulu pro dynamické zavádění knihoven

Workflow z *obrázku 11* lze rozdělit na dvě části. Tyto části jsou děleny na základě skutečnosti, zda knihovna, jež má být zavedena, již je přítomna v uložišti hybridního klienta, tedy jinými slovy již byla v minulosti zavedena či nikoliv. V případě, že knihovna ještě nebyla v minulosti zavedena, respektive stažena, proběhne proces stažení, dekomprimace (knihovna uložena na serveru ve formátu ZIP – viz kapitola *4.3.6 Externí*

knihovny) a uložení knihovny na připravené místo v uložišti, včetně vložení informací o knihovně do databáze zavedených knihoven. Zmiňovaná skupina kroků tvoří první část implementace modulu pro dynamické zavádění knihoven. Druhou částí pak je samotné zavedení knihovny skrze její hlavní třídu (proces zavádění byl podrobně popsán v kapitole *4.3.3 Modul dynamického zavádění knihoven*). Části jsou rozděleny především proto, aby byly funkčně nezávislé, tedy aby bylo možné zavést knihovnu bez stažení (postačuje ve většině případů) či naopak stáhnout knihovnu bez zavedení (Tato varianta využita například při stažení knihovny pro P2P komunikaci, která by měla být stažena, dle navrženého konceptu hybridního klienta, při zahájení fungování každého hybridního klienta).

5.3.2 Problémy při vývoji

Při vývoji vzorové aplikace bylo testováno zavádění obou stěžejních navržených externích knihoven z kapitoly *4.3.6 Externí knihovny* – knihovna pro offline režim a knihovna pro P2P komunikaci. Při testování se vyskytlo několik problémů. První problém, který se vyskytl, bylo zajištění fungování všech knihoven při zachování univerzálního způsobu řízení jejich chodu. Tento problém se podařilo z části vyřešit, neboť bylo možné zachovat univerzální rozhraní pro řízení běhu knihoven (tak jak je navrženo v kapitole *4.3.6 Externí knihovny*). Avšak bylo nutné přidat do modulu pro dynamické zavádění knihoven doplňující implementaci, jež je provedena vždy na základě zaváděné knihovny. Problém je hlouběji analyzován přímo v kapitole pojednávající o problémech spojených s vývojem externích knihoven – kapitola *5.4.2 Problémy spojené s externími knihovnami*. Toto zjištění je poměrně důležité pro budoucí využití konceptu hybridního klienta (či obecně architektury navržené v práci (5)) a to i přes to, že nikterak neovlivňuje správné fungování klienta či celého systému. Při zavádění mnoha externích knihoven by se totiž mohlo stát, že většina z nich bude vyžadovat speciální doplňující implementaci modulu pro dynamické zavádění knihoven, což může přinést nejen větší nároky na vývoj modulu, ale zároveň na chod aplikace ve smyslu náročnosti aplikace na zdroje mobilních zařízení a mohlo by tak vést ke zpomalení hybridního klienta. Druhým problémem bylo persistentní uložení instancí hlavních tříd externích knihoven. Bylo vyzkoušeno několik způsobů persistentního ukládání (JSON a XML soubor nebo již zmíněné interní uložiště *SharedPreferences*), nicméně po uložení se nepodařilo na

načtené instanci vyvolat metody prostřednictvím Java reflexe, jakožto hlavním nástrojem pro komunikaci s knihovnou, respektive její hlavní třídou (implementace použití Java reflexe je v následující kapitole *5.3.3 Vzorová implementace*). Tento problém však není tak významný pro účely hybridního klienta, neboť byl vyřešen opakovaným znovuzaváděním příslušné knihovny (instance její hlavní třídy) pokaždé, kdy je instance z paměti ztracena. Navíc v operačním systému Android je celá aplikace reprezentována hlavní aktivitou (viz kapitola *3.2 Android OS architektura*), kdy je možné efektivně reagovat na změny stavu aktivity pomocí metod pro naslouchání stavů v životním cyklu aktivity. Velkou výhodou je, že tyto metody lze velmi efektivně využít pro volání metod určených k řízení běhu knihoven, neboť jejich význam v podstatě kopíruje význam metod společného rozhraní pro externí knihovny, což značně ulehčuje řízení chodu externích knihoven na platformě Android a v podstatě maže problém nutnosti persistentního ukládání instancí knihoven. Tato spojitost, stejně jako implementace řízení chodu knihoven je rozebrána v kapitole *5.4.1 Řízení běhu knihoven – změna role hybridního klienta*. Další drobný problém představovalo stahování knihovny v podobě ZIP souboru z centrálního serveru, kdy stažení prostřednictvím *Retrofit* knihovny nebylo úspěšné. Problém se podařilo vyřešit vlastní implementací za využití třídy *java.net.HttpUrlConnection*. Všechny ostatní problémy se týkaly komunikace aplikace s externími knihovnami a byly vyřešeny použitím Java reflexe.

5.3.3 Vzorová implementace

Vzhledem ke komplexnosti modulu nebudou představeny všechny dílčí implementační kroky z *obrázku 11*, nicméně budou představeny stěžejní konstrukce procesu zavedení knihoven, tedy druhé implementační části, a to v pořadí, v jakém jsou uváděny v postupu vývoje na *obrázku 11*. Prvním krokem je tedy vytažení hlavní (řídící) třídy knihovny a vytvoření instance této třídy – viz *kód 5*.

```
// init loaderu pro zavedení knihovny
loader = new DexClassLoader (completePath, dexPath, null,
activity.getClass().getClassLoader());

// načtení hlavní třídy knihovny
Class<?> classToLoad;
try {
   classToLoad = Class.forName(myDb.getClassName(libraryName), true, loader);
} catch (ClassNotFoundException e) {
   Log.e(TAG, "Nepodařilo se nalézt zaváděcí třídu knihovny: ");
        e.printStackTrace();
        return 1;
 }
// vytvoření instance hlavní třídy knihovny
Object libraryInstance = null;
try {
   if (classToLoad != null) {
      libraryInstance = classToLoad.newInstance();
   }
} catch (IllegalAccessException | java.lang.InstantiationException e) {
      Log.e(TAG, "Nepodařilo se vytvořit instanci zaváděcí třídy knihovny: ");
      e.printStackTrace();
      return 1;
}
```

Kód 5 - načtení hlavní třídy knihovny a vytvoření její instance

Pro dynamické načtení řídící třídy je nejprve potřeba inicializovat *DexClassLoader* (problematika modularity na platformě Android byla rozebrána v kapitole *3.2.3 Modularita v Android OS*), kdy v konstruktoru jsou uváděny čtyři parametry – cesta k *apk* či *jar* souboru, který obsahuje zdrojový kód pro načtení, cesta k adresáři do kterého by měly být cachovány optimalizované soubory *.dex* (nyní deprecated a bez významu), seznam adresářů obsahujících nativní Android knihovny (není potřeba – může být null), rodičovský *ClassLoader* (v prostředí Androidu možné použít *ClassLoader* aplikačního kontextu či aktivity pro načtení tříd v rámci daného kontextu či aktivity) (34). Po této inicializaci je třída načtena prostřednictvím metody *forName()* volané na Java třídě *Class*, kdy v parametru metody je uveden název hlavní řídící třídy knihovny (vyjmut z deskriptoru knihovny a uložen do databáze po stažení souborů knihovny z centrálního serveru – viz *obrázek 11* a kapitola 4.3.3 Modul dynamického zavádění knihoven), dále

parametr, jenž definuje, zda má být třída inicializována či nikoliv (v této fázi není důležité, neboť v dalším kroku je vytvářena instance, kdy by k inicializaci stejně došlo) a naposledy *ClassLoader*, který má být pro načtení použit (zde aplikován definovaný zavaděč z předchozího kroku). Posledním krokem je vytvoření instance třídy pomocí metody *newInstance()*. V případě, že některá fáze zavádění selže, jsou moduly využívající služby modulu pro dynamické zavádění knihoven o této skutečnosti informovány chybovou návratovou hodnotou 1 (koresponduje s chybovou hodnotou metod společného rozhraní pro externí knihovny).

V případě, že se podaří úspěšně vytvořit instanci hlavní třídy zaváděné knihovny, dojde k uložení této instance tak, aby mohl být řízen chod externí knihovny centrálním modulem (potažmo dalšími moduly, ukázka implementace tohoto řízení je součástí kapitoly *5.4.1 Řízení běhu knihoven – změna role hybridního klienta*). Vzorová implementace procesu uložení instance hlavní třídy příslušné knihovny je znázorněna v *kódu 6*.

```
// uložení instance knihovny dle jejího názvu
HashMap<Integer, Object> libraries = activity.getLibraries();
// pokud se jedná o knihovnu pro offline režim
      libraries.put(MainActivity.LIBRARY_FOR_OFFLINE_MODE_POSITION, libraryInstance);
// pokud se jedná o knihovnu pro P2P komunikaci
      libraries.put(MainActivity.LIBRARY_FOR_P2P_MODE_POSITION, libraryInstance);
}
activity.setLibraries(libraries);
```

Kód 6 - ukázka uložení instance hlavní třídy externí knihovny

Jednotlivé instance jsou ukládány do vhodně zvolené datové struktury *HashMap*, kdy každá knihovna má jednoznačnou pozici v datové struktuře (svůj index v mapě). Díky tomuto přístupu je možné efektivně přidávat a odebírat instance knihoven bez toho, aby docházelo k záměnám knihoven při manipulaci s instancemi v datové struktuře, neboť indexy jsou pevně dány a nemůže dojít k žádným neočekávaným změnám. Je tak zajištěno, že centrální modul vždy přesně ví, s jakou knihovnou manipuluje. Pro vkládání se užívá metoda *put(pozice, instance_knihovny)*, pro získávání metoda *get(pozice)* a pro odstranění metoda *remove(pozice)*. Po úspěšném uložení instance následuje spuštění knihovny voláním metody *start()*, kterou každá externí knihovna musí implementovat (viz společné rozhraní v kapitole *4.3.6 Externí knihovny*).

```
// zavedení knihovny skrze metodu start()
try {
   // zavedení knihovny
   Method start = libraryInstance.getClass().getMethod("start", String.class, Context.class);
   int res = (int) start.invoke(libraryInstance, dexPath, activityContext);
} catch (NoSuchMethodException | IllegalAccessException | InvocationTargetException e) {
   Log.e(TAG, "Nepodařilo se zavést knihovnu metodou start(): " + e.getMessage());
}
```

Kód 7 - ukázka zavedení knihovny metodou start()

Konstrukce zavedení knihovny využitím instance její hlavní třídy a Java reflexe je naznačena v *kódu 7*. Z instance, respektive z hlavní třídy je vytažena metoda *start()* pomocí metody *getMethod(název_metody, typy_parametrů)*. Typy parametrů jsou deklarovány jako objekty třídy *Class*. Tímto způsobem je získán objekt typu *Method*, jenž reprezentuje metodu *start()*. Právě na tomto objektu je poté možné zavolat metodu *invoke()*, čímž dochází k vyvolání metody *start()* na základě vstupních parametrů, jež jsou do metody dodávány v parametru metody *invoke()*. Prvním parametrem metody *invoke()* je objekt, na kterém má být volání metody provedeno (v tomto případě instance hlavní třídy knihovny) a další parametry tedy slouží k definici vstupních parametrů do vyvolávané metody (dle definice společného rozhraní externích knihoven se jedná o cestu ke knihovně v interním uložišti a objekt reprezentující kontext hybridního klienta). Návratová hodnota je typu *Object*, kdy je možné provést přetypování na požadovanou formu výstupu (zde se jedná o celočíselnou hodnotu, která informuje o úspěšnosti vykonání příslušné metody – opět je možné se odkázat na definici rozhraní pro externí knihovny v kapitole *4.3.6 Externí knihovny*).

Poslední fází v běžném procesu zavedení knihovny je změna kontextu klienta, která spadá spíše pod funkce centrálního modulu, proto zde její implementace nebude podrobně rozebrána. Nicméně jedná se především o změnu hlavní řídící proměnné vzorové aplikace, podle které se řídí chování aplikace a rekonfigurace hybridního klienta (buď zavedením nebo odebráním knihovny). Novému stavu klienta je rovněž nutné přizpůsobit GUI aplikace tak, aby odpovídalo aktuálním možnostem uživatele (některé funkce mohou být omezeny). Proces odebírání knihovny, stejně jako některé prvky přizpůsobení se aplikace při rekonfiguraci jsou naznačeny v kapitole *5.4.1 Řízení běhu knihoven – změna role hybridního klienta*.

5.4 Externí knihovny

V předchozí kapitole *5.3.3 Vzorová implementace* bylo naznačeno, jakým způsobem jsou instance externích knihoven udržovány a jakým způsobem probíhá proces zavedení knihovny, tedy ztloustnutí hybridního klienta. V této kapitole bude představen obecný proces změny stavu (role) klienta tak, jak byl teoreticky navržen v kapitole *4.3.6 Externí knihovny*. Zároveň bude představena komunikace v podobě předávání dat mezi prostředím klienta a externích knihoven. Následně budou zmíněny některé problémy, které se pojí s problematikou externích knihoven a na závěr bude nastíněn vývoj obou stěžejních knihoven – knihovny pro offline režim a knihovny pro P2P komunikaci.

5.4.1 Řízení běhu knihoven – změna role hybridního klienta

Řízení chodu externích knihoven tedy probíhá využitím instance hlavní třídy a metod společného rozhraní pro externí knihovny. V souvislosti s problémem persistentního uložení instancí knihoven (viz kapitola *5.3.2 Problémy při vývoji*) je důležité zmínit, že rekonfigurace neprobíhá pouze v případě inicializace od uživatele či na základě některého z impulzů pro automatické zahájení procesu rekonfigurace, jež jsou definovány v prakticko-teoretické části (kapitola *4.1.1 Podoby a role hybridního klienta*), ale rovněž při změně stavu aplikace, respektive její reprezentace v podobě hlavní aktivity. Hlavní aktivita umožňuje reagovat na změny stavu v jejím životním cyklu prostřednictvím několika metod. Spojitost mezi těmito metodami a metodami společného rozhraní externích knihoven již byla naznačena v kapitole *5.3.2 Problémy při vývoji* a je jasně patrná z *kódu 8*, v němž lze vidět konkrétní podobu hlavní aktivity vzorové aplikace v souvislosti s implementací řízení běhu knihoven na základě změn stavů hlavní aktivity (aplikace).

```
public class MainActivity extends AppCompatActivity {
   private int applicationState;
   private SharedPreferences sharedPref;

   @Override
   protected void onStart() {
      // získání stavu aplikace
      applicationState = sharedPref.getInt(getString(R.string.sh_pref_app_state), 0);
      // stažení p2p knihovny, pokud již není stažena
      . . . .
      // rozdělení chování aplikace dle jejího aktuálního stavu
      if (applicationState == 0) {
         // role tenkého klienta -> komunikace s centrálním serverem
         . . . .
      } else if (applicationState == 1) {
         // role hybridního klienta v offline režimu -> zavedení knihovny pro offline režim
         . . . . // invoke metody start() na instanci offline knihovny
      } else if (applicationState == 2) {
         // role hybridního klienta s p2p režimem v roli server -> zavedení knihovny pro p2p
         komunikaci v roli serveru && zavedení knihovny pro offline režim
         . . . . // invoke metody start() na instancích obou knihoven
      } else {
         // role hybridního klienta s p2p režimem v roli klienta -> zavedení knihovny pro p2p
         komunikaci v roli klienta
         . . . . // invoke metody start() na instanci knihovny pro p2p komunikaci
      }
   }
   @Override
   protected void onResume() {
      // znovu rozdělení chování aplikace dle jejího aktuálního stavu
      if (applicationState == 1) {
         // role hybridního klienta v offline režimu -> zavolání metody resume() na instanci
         knihovny pro offline režim
         . . . .
      }
      . . . . // to samé pro metody onPause() a onStop()
}
```

Kód 8 - ukázka řízení běhu knihoven v hlavní aktivitě

Z *kódu 8* totiž lze vyčíst následující skutečnost: Pakliže je žádoucí reagovat na změnu stavu aplikace změnou její konfigurace či jejího chování, je velmi často rovněž žádoucí zareagovat obdobně u externích knihoven. Například pokud se hlavní aktivita

(aplikace) zobrazí uživateli (tento stav je reprezentován metodou *onStart()* (23)), je potřeba provést inicializaci většiny aplikačních procesů (v podstatě zahájit chod aplikace) a zároveň spustit chod příslušné externí knihovny či více knihoven podle aktuálního stavu hybridního klienta. Tento stav je ve vzorové aplikaci vyjádřen celočíselnou proměnnou (v *kódu 8* pod označením *„applicationState“*), jež je uložena v interním uložišti *SharedPreferences*. Toto uložiště funguje na bázi klíč-hodnota, kdy pro získání požadované hodnoty se využívají různé metody, které jsou rozděleny dle datového typu hodnoty, jež chceme získat. V tomto případě se jedná o metodu *getInt(klíč, výchozí_hodnota)*. Proměnná vyjadřující stav aplikace může nabývat čtyř hodnot: 0 pro roli tenkého klienta, 1 pro roli klienta v offline režimu, 2 pro roli hybridního klienta v podobě P2P serveru a 3 pro roli hybridního klienta v podobě P2P klienta.

Stejný postup platí i pro další metody vyjadřující stav aplikace, ke kterým existuje ekvivalent ve společném rozhraní pro externí knihovny – jedná se o metody *onResume()*, *onPause()* a *onStop()*. V případě metod *onPause()* a *onStop()* dochází k určitému zastavení činnosti aplikace, neboť hlavní aktivita ustupuje do pozadí či není dále uživateli zobrazována. Některé knihovny mohou v takovémto případě vyžadovat dodatečné ukončení činnosti procesů, jež probíhají v prostředí aplikace, avšak vážou se k činnosti určité knihovny. Příkladem může být zasílání či odchytávání událostí, kdy je potřeba pro ukončení chodu aplikace a P2P knihovny odregistrovat *receiver* v prostředí klienta (implementace odregistrování je v *kódu 9* a implementace komunikace mezi knihovnou a aplikací je součástí kapitoly *5.4.1.1 Komunikace (předávání dat) mezi klientem a knihovnami*). To samé platí v případě úplného odebrání knihovny, tedy při procesu hubnutí hybridního klienta (na instanci knihovny je místo metody *stop()* volána metoda *exit()*). Vzorová implementace změny kontextu aplikace v podobě hubnutí hybridního klienta je představena v *kódu 9* (v kódu nyní již není znázorněno zachytávání výjimek, neboť je zcela totožné jako při zavádění knihovny v *kódu 7*). Konkrétně se jedná o odpojení knihovny pro P2P komunikaci a následnou změnu kontextu klienta v závislosti na nové roli – tenkého klienta. Při změně kontextu si lze povšimnout, že je kromě změny proměnné aktuálního stavu klienta a přizpůsobení GUI rovněž potřeba odregistrovat *receiver*, vyčleněný pro naslouchání při P2P komunikaci. Samotné ukončení chodu knihovny je řešeno Java reflexí a to zavoláním metody *exit()*, kdy tato implementace je totožná s implementací zavedení knihovny, jež byla popsána v kapitole *5.3.3 Vzorová*

implementace. Metody pro změnu hodnoty v uložišti *SharedPreferences* se stejně jako v případě metod pro získávání hodnoty dělí dle datového typu, kdy předpona *„get“* je nahrazena předponou *„put“*. Pro aplikaci změn se používá rozhraní *SharedPreferences.Editor*, kdy je nutné po dokončení úprav provedenou změnu potvrdit metodou *apply()* (jedná se o asynchronní volání, alternativou je synchronní volání *commit()*, jež vrací informaci o úspěšnosti zápisu (52)). Po odstranění instance knihovny z *HashMap* je nutné obnovit hlavní aktivitu a tím chod celé aplikace, aby došlo ke kompletní rekonfiguraci a aplikaci všech požadovaných změn.

```
// je-li aplikace v roli hybridního klienta s p2p komunikací
. . . .
// získání instance knihovny pro p2p komunikaci
Object p2pLibraryInstance = libraries
                    .get(MainActivity.LIBRARY_FOR_P2P_MODE_POSITION);
// odpojení a odebrání knihovny pro p2p komunikaci
Method exit = p2pLibraryInstance.getClass().getMethod("exit");
int result = (int) exit.invoke(p2pLibraryInstance);
// pokud se podaří
if (result == 0) {
  // odpojení receiveru aplikace v hlavní aktivitě (proměnná mainActivity)
  mainActivity.unregisterP2PBroadcastReceiver();
  // změna kontextu aplikace -> tenký klient
  // nastavení GUI
  . . . .
  //  změna proměnné reprezentující stav klienta
  SharedPreferences.Editor editor = sharedPref.edit(); // SharedPreferences proměnná
  deklarovaná v hlavní aktivitě
  editor.putInt(getString(R.string.sh_pref_app_state), 0); // 0 -> role tenkého klienta
  editor.apply();
  // odstranění p2p knihovny ze seznamu knihoven
  libraries.remove(1); // 1 -> pozice (klíč) knihovny pro p2p komunikaci v struktuře
  HashMap instancí knihoven
  // refresh hlavní aktivity
  mainActivity.recreate();
```

Kód 9 - ukázka odpojení knihovny pro P2P komunikaci

5.4.1.1 Komunikace (předávání dat) mezi klientem a knihovnami

Naprostá většina předávání dat je realizována pomocí vyvolávání metod využitím konstrukce Java reflexe tak, jak bylo ukázáno při zavádění (*kód 7*) či odpojení externí

knihovny (*kód 9*). Při P2P komunikaci je však potřeba zajistit předání dat i v případě, kdy knihovna pro P2P komunikaci obdrží data od jiného klienta. Tento proces je čistě asynchronní (hybridní klient neví, kdy knihovna data obdrží), tudíž je nutné informovat klienta o faktu, že data již dorazila a může být s nimi dále pracováno. Platforma Android pro tyto účely nabízí systém zasílání tzv. *Broadcasts* (53). Jedná se o zasílání zpráv mezi systémem a aplikací či mezi samotnými aplikacemi v prostředí Android OS. Tyto zprávy jsou vždy spjaty s určitou událostí (buď již definovanou, systémovou událostí nebo nově vytvořenou, vlastní událostí) a dají se zachytávat pomocí listeneru, který se označuje jako *BroadcastReceiver*. Ten lze definovat buď v konfiguračním souboru (Android Manifest – více o manifestu v kapitole *4.3.7 Android Manifest*) nebo přímo v kódu aplikace. (53) Ukázka implementace této komunikace mezi hybridním klientem a jeho externí knihovnou je představena v *kódu 10*, *kódu 11*(definice *Broadcasts* a zaslání zprávy) a *kódu 12* (definice *BroadcastReceiver*).

```
public class MainActivity extends AppCompatActivity {
   // pro naslouchání výsledků p2p knihovny
   final P2PBroadcastReceiver pBroadcastReceiver = new P2PBroadcastReceiver(this);
   . . . .
   public void registerP2PBroadcastReceiver() {
      // vytvoření a zápis události k naslouchání dle role klienta
      if (applicationState == LIBRARY_FOR_P2P_SERVER_MODE_CODE) {
         IntentFilter intentFilter = new
         IntentFilter(getString(R.string.receive_client_info_from_p2plibrary_action_name));
         registerReceiver(pBroadcastReceiver, intentFilter);
      } else if (applicationState == LIBRARY_FOR_P2P_CLIENT_MODE_CODE) {
         IntentFilter intentFilter = new
         IntentFilter(getString(R.string.receive_clients_from_p2plibrary_action_name));
         registerReceiver(pBroadcastReceiver, intentFilter);
      }
   }
}
```

Kód 10 - Hybridní klient – definice události a zaregistrování Receiveru

V *kódu 10* je znázorněna implementace vytvoření události a registrace naslouchání této události prostřednictvím *BroadcastReceiver* objektu v prostředí hybridního klienta. *IntentFilter* objekt reprezentuje událost, které má být nasloucháno. *BroadcastReceiver* je registrován metodou *registerReceiver()*, kdy v prvním parametru je uvedena událost k naslouchání a ve druhém objekt (tedy samotná instance

BroadcastReceiveru), jenž bude na událost reagovat. V případě knihovny pro P2P komunikaci, která je odesílatelem zpráv je potřeba opět vytvořit novou událost – zde je potřeba dodat, že názvy vytvořených událostí v prostředí klienta a knihovny se musí shodovat, neboť by se nejednalo o tutéž událost a zpráva zasílána knihovnou by byla aplikací ignorována. Po vytvoření události se k předávané zprávě (v podobě *Intent* objektu) připojí data k odeslání a k samotnému poslání zprávy slouží metoda *sendBroadcast(Intent_objekt_pro_odeslání)*. Ukázka implementace vytvoření a zaslání zprávy z knihovny pro P2P komunikaci je prezentována v *kódu 11*.

```
// vytvoření a zaslání Broadcast dle role knihovny (klienta) a přidání zasílaných dat
if (mode == RUN_LIBRARY_AS_SERVER_VALUE) {
   // obdržení informací o klientovi v roli P2P klienta v byte[] podobě
   byte[] receivedClientInformation = (byte[]) msg.obj;
   // předání výsledku do aplikace skrze Intent
   Intent intent = new Intent(RECEIVED_CLIENT_INFO_ACTION_NAME);
   intent.putExtra(RECEIVED_CLIENT_INFO_EXTRA_NAME, receivedClientInformation);
   mainActivity.getApplicationContext().sendBroadcast(intent);
} else {
   // obdržený seznam klientů od zařízení v roli P2P serveru v byte[] podobě
   byte[] receivedClients = (byte[]) msg.obj;
   . . . . // to samé pouze s receivedClients
}
```

Kód 11 - knihovna pro P2P komunikaci – vytvoření události a zaslání Broadcast

Posledním krokem v komunikačním procesu na bázi zasílání Broadcast zpráv je implementace zachytávání přijatých zpráv a následného reagování klienta. Při pohledu na *kód 12*, jenž tuto implementaci zaštiťuje, lze vidět, že se jedná o vlastní implementaci třídy *BroadcastReceiver*, kde veškerá funkcionalita je obsažena v metodě *onReceive(aplikační_kontext, objekt_s_obdrženou_zprávou)* určené k naslouchání daným událostem. V tomto případě jsou zachytávány dvě události – jedna pro roli P2P serveru, kdy obdržená data obsahují kontext P2P klienta a druhá pro podobu P2P klienta, v tomto případě jsou součástí přijatých dat informace o všech klientech, o nichž má komunikující klient v roli P2P serveru přehled. Událost lze získat z přijaté zprávy pomocí metody *getAction()*, díky čemuž je možné větvit chování hybridního klienta dle události, jež byla vyvolána. V *kódu 12* je představeno, jakým způsobem se zachová hybridní klient, pakliže nastanou události obdržení výsledku z knihovny pro P2P komunikaci v rolích P2P klienta a P2P serveru.

```
public class P2PBroadcastReceiver extends BroadcastReceiver {
   @Override
   public void onReceive(Context context, Intent intent) {
      // pokud nastala událost obdržení výsledku p2p knihovny v roli klienta
      // tedy seznam klientů z okolního zařízení (peer)
      if (getString(R.string.receive_clients_from_p2plibrary_action_name)
            .equals(intent.getAction())) {
         // získání výsledků z Extra objektu
         byte[] result = intent.getByteArrayExtra(
         getString(R.string.receive_clients_from_p2plibrary_extra_name));
         // převod byte[] na list klientů
         // uložení nového seznamu klientů -
         // nikoliv ze serveru, ale z okolního peer zařízení v roli serveru
         . . . .
      // pokud nastala událost obdržení výsledku p2p knihovny v roli serveru
      // tedy obdržení kontextu klienta - zařízení (peer)
      } else if (getString(R.string.receive_client_info_from_p2plibrary_action_name)
            .equals(intent.getAction())) {
         // získání výsledků z Extra objektu
         // převod byte[] na kontext klienta
         // uložení do DB - synchronizace
         . . . .
      }
   }
}
```

Kód 12 - Hybridní klient – implementace BroadcastReceiveru

5.4.2 Problémy spojené s externími knihovnami

Z výše uvedených vzorových implementací, tedy z *kódu 10*, *kódu 11* a *kódu 12* lze mimo jiné pozorovat, že pro správné fungování knihovny pro P2P komunikaci je nutné před samotným zavedením a spuštěním určit v jakém režimu má knihovna fungovat, což směřuje k prvnímu závažnému problému, jenž se při vývoji externích knihoven objevil. Nabízí se několik variant, jak tuto problematiku řešit. První varianta spočívá v přidání parametrů do spouštěcí metody *start()* ve společném rozhraní externích knihoven. V tomto bodě je však nutné položit si otázku, co by se stalo v případě, kdy by zaváděná knihovna tento parametr ke své funkci nepotřebovala, respektive jaká by byla jeho hodnota. Dalším řešením je rozvětvení funkcionalit knihovny, kdy by po zavedení knihovny následovalo vyvolání další metody dle role, ve které má být knihovna spuštěna.

Zde se již zachovává univerzální podoba společného rozhraní, nicméně komplikuje se řízení a používání knihovny v prostředí klienta (více vyvolávání metod a další používání reflexe). Proto byla vyvíjena a testována další varianta, jež se zdá být nejideálnějším řešením. Ta vede k odevzdání odpovědnosti za prováděné funkce na stranu knihovny, neboť v prostředí aplikace je nutné pouze inicializovat stav (ve vzorové aplikaci mód) knihovny před samotným spuštěním (viz schéma zavádění knihoven na *obrázku 11*). Knihovna pak sama provádí příslušný kód na základě svého stavu, jenž odpovídá roli klienta. Díky tomu se v aplikaci vyvolává pouze jedna metoda navíc, která je navíc velmi jednoduchá (pouze předání proměnných z prostředí klienta do prostředí knihovny). Skutečnost, že knihovna se zavádí jiným způsobem, než je obvyklé, je možné uvést v deskriptoru knihovny, stejně jako popis chování knihovny při dané roli hybridního klienta. Některé externí knihovny naopak mohou pro svůj chod vyžadovat vyvolání metod až po úspěšném zavedení a spuštění (na schématu v *obrázku 11* se jedná o krok „podpůrné metody pro zavedení knihovny"). To je případ i knihovny pro offline režim, kde je potřeba po spuštění metodou *start()*, která zde slouží především pro inicializaci všech datových struktur a datového uložiště, přenést seznam klientů z kontextu aplikace do prostředí knihovny, aby mohl být prostřednictvím funkcí knihovny pro offline režim uložen do vytvořené databáze. Informace o tomto způsobu řízení chodu knihovny by opět měly být součástí deskriptoru. Právě předávání seznamu klientů mezi aplikací a knihovnou pro offline režim představovalo další menší problém, neboť nebylo možné získat kompletní seznam pouze pomocí tradičního *getteru*, tak jak je běžné v programech psaných v jazyce Java. Ačkoliv třídu reprezentující kontext hybridního klienta obsahuje jak aplikace (kvůli komunikaci s centrálním serverem), tak knihovna (pro ukládání do databáze), nelze třídy považovat za shodné, neboť jsou vyvíjeny a kompilovány samostatně. Z tohoto důvodu není možné využít přetypování na třídu v daném kontextu a je potřeba užít jiného principu předání dat. Přenos seznamu z prostředí aplikace (data z centrálního serveru) do externí knihovny spočívá v postupném vyvolávání metody *addUser(atributy_předávaného_klienta)*, jež implementuje knihovna pro offline režim a která umožňuje předání jednoho klienta ze seznamu a následně ukládá tohoto klienta do databáze. Pakliže hybridní klient vyžaduje seznam klientů z databáze, je předán seznam objektů s neznámým datovým typem pro aplikaci (jedná se o třídu reprezentující klienta v knihovně) a na každém objektu jsou vyvolávány metody externí knihovny Java reflexí

(gettery třídy klienta z knihovny) pro získání atributů, ze kterých jsou následně vytvořeny objekty s datovým typem, který reprezentuje hybridního klienta v kontextu aplikace, z nichž je vytvořen seznam a s ním dále pracováno.

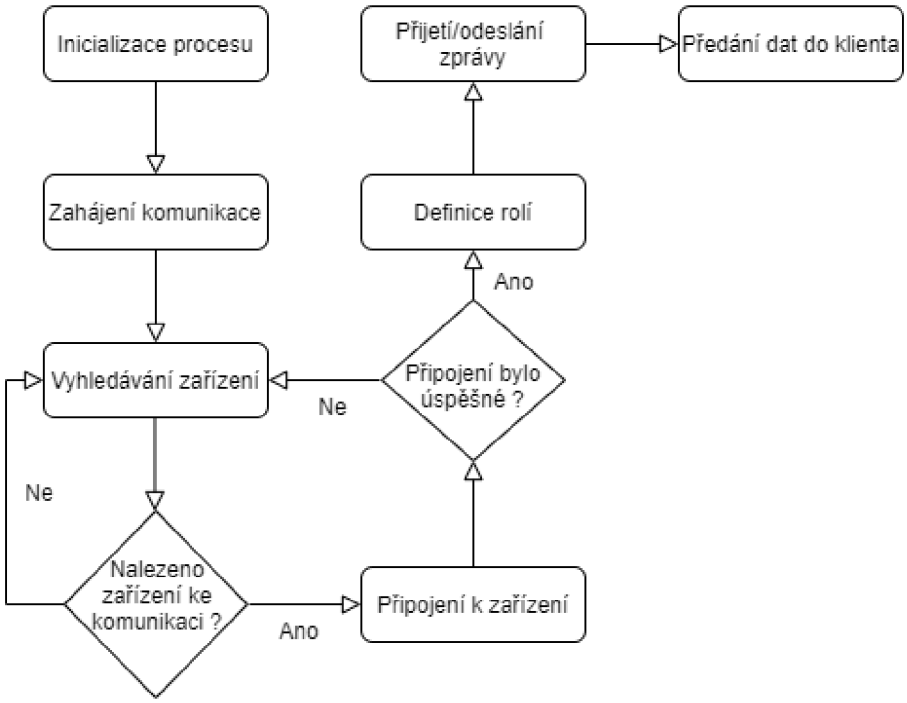
5.4.3 Knihovna pro offline režim

Jak již bylo řečeno v kapitole *4.3.6.1 Knihovna pro offline režim*, implementace knihovny pro offline režim vždy striktně závisí na službách navrženého IS. Vzhledem k IS, jenž byl navržen pro účely otestování funkčnosti konceptu hybridního klienta (popis služeb je v kapitole *4.2 Funkce vzorového informačního systému*) pomocí vzorové aplikace, má knihovna relativně snadnou implementaci, která spočívá především v manipulaci s *SQLite* databází prostřednictvím nativního balíčku *android.database*. Dalším úkolem knihovny je předávání dat, tak jak bylo popsáno v předešlé kapitole *5.4.2 Problémy spojené s externími knihovnami*, k čemuž slouží metody *getUsers()* pro předávání dat (seznamu klientů) z databáze do aplikace a *addUser()* pro získání dat (seznamu klientů) z aplikace. Poslední stěžejní implementační částí je komunikace s centrálním serverem v případě ukončení činnosti knihovny pro účel synchronizace seznamu klientů knihovny s globálním seznamem v centrálním uložišti na centrálním serveru tak, aby tento seznam byl vždy aktuální. Komunikace probíhá využitím *Retrofit* knihovny obdobným způsobem jako v případě komunikace v modulu vyhodnocení datové propustnosti v kapitole *5.2.3 Vzorová implementace*.

5.4.4 Knihovna pro peer-to-peer komunikaci

Knihovna pro peer-to-peer komunikaci má podstatně složitější strukturu a implementaci. Vnitřní struktura knihovny je znázorněna na *obrázku 7* v kapitole *4.3 Návrh architektury hybridního klienta* a popis jednotlivých komponent je součástí kapitoly *4.3.6.2 Knihovna pro peer-to-peer komunikaci*. Pro lepší pochopení fungování knihovny byl navržen workflow jejího vývoje, jenž je vyobrazen na *obrázku 12*.

5.4.4.1 Workflow vývoje



Obrázek 12 - workflow vývoje knihovny pro P2P komunikaci

Schéma na *obrázku 12* v podobě postupu vývoje knihovny pro P2P komunikaci zároveň reflektuje chování této knihovny v rámci hybridního klienta, a to jak v roli P2P klienta, tak v roli P2P serveru. Chování u obou rolí se liší pouze obsahem zasílaných a přijímaných zpráv, jež jsou dále předávány do prostředí hybridního klienta.

5.4.4.2 Vzorová implementace

Jak již bylo řečeno v kapitole *4.3.6.2 Knihovna pro peer-to-peer komunikaci*, stěžejní technologií pro implementaci služeb knihovny pro P2P komunikaci je *Wi-Fi Direct (P2P) API* (40). Samotný komunikační proces je zahájen inicializací klíčových objektů. Vzhledem k tomu, že komunikace mezi zařízeními je zprostředkována pomocí technologie Wi-Fi Direct, je nutné zajistit, aby dané zařízení mělo zapnutý Wi-Fi přijímač. Pro tento účel slouží instance třídy *WifiManager*, skrze níž je možné zkontrolovat stav Wi-Fi a pakliže je vypnuta, je možné ji aktivovat. Pro vyhledávání okolních zařízení a následnému připojování skrze Wi-Fi slouží *WifiP2pManager*, který tak představuje hlavní komunikační nástroj a umožňuje komunikaci realizovat. Pro připojení aplikace k nástrojům a službám, které poskytuje *WifiP2pManager*, slouží třída *Channel*. Nakonec je potřeba inicializovat *BroadcastReceiver* (více o zasílání a zachytávání *Broadcast* zpráv v kapitole *5.4.1.1 Komunikace (předávání dat) mezi*

klientem a knihovnami), který bude naslouchat vybraným událostem v P2P komunikaci. Stěžejní jsou především dvě tyto události – *WIFI_P2P_PEERS_CHANGED_ACTION*, která značí, že došlo ke změně stavu některého okolního zařízení, tedy byl změněn seznam dostupných zařízení (peerů) a *WIFI_P2P_CONNECTION_CHANGED_ACTION*, která již indikuje přímo změny v připojení zařízení (peerů). Pomocí těchto událostí je možné komunikaci mezi zařízeními ovládat. Jakmile je *BroadcastReceiver* vytvořen, je potřeba jej registrovat v hlavní aktivitě hybridního klienta – tato implementace vede k nutnosti přenést instanci hlavní aktivity ještě před zahájením provádění funkcí knihovny, což je vyřešeno inicializační metodou, kterou je nastavována mimo jiné také role, podle niž se má knihovna chovat (viz kapitola *5.4.2 Problémy spojené s externími knihovnami*).

Po úspěšné inicializaci komunikace je možné zahájit komunikaci vyhledáním dostupného zařízení v okolí (peeru) metodou *discoverPeers()*, jež je volána na instanci *WifiP2pManager*. Metoda přijímá dva parametry – instanci třídy *Channel* získanou inicializací *WifiP2pManageru* (používá se u všech metod pro P2P komunikaci) a implementaci listeneru s metodami, jež informují o úspěšném či neúspěšném nalezení okolních zařízení ke komunikaci. Ukázka inicializace připojení prostřednictvím metody *discoverPeers()* je součástí *kódu 13*.

Je důležité poznamenat, že tyto metody neslouží k získání seznamu dostupných zařízení a případnému pokusu o připojení k danému zařízení. Pro tyto účely je využívána událost (*WIFI_P2P_PEERS_CHANGED_ACTION*) odchytávána ve výše zmíněném *BroadcastReceiveru*, neboť inicializace vyhledání dostupných zařízení metodou *discoverPeers()* tuto událost vyvolává. Seznam dostupných zařízení je možné získat zavoláním metody *requestPeers()*. Ta (krom *Channel* objektu) v parametru obsahuje instanci rozhraní *PeerListListener*, prostřednictvím něhož je možné získat informace o zařízeních, jež byly úspěšně detekovány. Díky těmto informacím lze zjistit, zda se dané zařízení připojilo k P2P síti či naopak odpojilo a již není dále k dispozici ke komunikaci.

```
public class MainClass extends AppCompatActivity implements LibraryLoaderInterface {
  . . . .
  // metoda pro vyhledání zařízení (peers)
  public void discoverPeers() {
    // kontrola zažádání povolení k získání polohy
    if (permissionGranted) {
      p2pManager.discoverPeers(channel, new WifiP2pManager.ActionListener() {
        @Override
         public void onSuccess() {
          // podařilo se zahájit vyhledávání
          Log.d(TAG, "Vyhledávání peers zahájeno");
        }
        @Override
        public void onFailure(int reason) {
          // nepodařilo se zahájit vyhledávání
          Log.e(TAG, "Nepodařilo se vyhledat dostupné peers. Důvod:" + reason);
        }
      });
    }
  }
}
```

Kód 13 - knihovna pro P2P komunikaci – zahájení komunikace vyhledáním dostupných zařízení

V *kódu 13* je rovněž patrné, že před samotným procesem vyhledávání dostupných zařízení v okolí je potřeba získat povolení pro získání aktuální polohy uživatelova zařízení. Tento požadavek je nutné implementovat přímo v kódu knihovny od verze Android OS vyšší než verze 4.4 KitKat (úroveň API 19). Pro všechny verze OS pak platí, že je nutné uvést oprávnění pro získání polohy do konfiguračního souboru *Android Manifest*. Součástí *kódu 14* je implementace rozhraní *PeerListListener*, respektive metody *onPeersAvailable()*, jež je součástí tohoto rozhraní. Jak již bylo zmíněno tato implementace slouží k získání seznamu dostupných zařízení.

```
private final List<WifiP2pDevice> peers = new ArrayList<>();
@Override
public void onPeersAvailable(WifiP2pDeviceList peerList) {
  if (!peerList.getDeviceList().equals(peers)) {
    // došlo k updatu seznamu dostupných zařízení (peers)
    peers.clear();
    peers.addAll(peerList.getDeviceList());
    peersArray = new WifiP2pDevice[peerList.getDeviceList().size()];
    int index = 0;
    for (WifiP2pDevice device : peerList.getDeviceList()) {
      peersArray[index] = device;
      index++;
    }
    // zahájení procesu připojení vždy při změně seznamu peers
    // a připojení iniciuje zařízení s rolí serveru
    if(peers.size() != 0 && mode == RUN_LIBRARY_AS_SERVER_VALUE) connectToPeers();
  }
}
```

Kód 14 - knihovna pro P2P komunikaci – získání seznamu dostupných zařízení

Implementace spočívá nejprve ve vytvoření pomocné datové struktury pro uložení aktuálního seznamu zařízení (v *kódu 14* proměnná *peers*). Tento seznam je při každém vyhledání dostupných zařízení porovnán se seznamem vyhledaných zařízení (proměnná *peerList*) a pakliže se seznamy neshodují, lze říci, že došlo ke změně, tudíž je nutné obnovit aktuální seznam dostupných zařízení a rovněž obnovit pole všech dostupných zařízení (proměnná *peersArray*), které slouží k následnému připojování v metodě *connectToPeers()*. Implementace této metody je v *kódu 15*.

```
private void connectToPeers() {
  // metoda connect() opět vyžaduje povolení k poloze -> v této fázi by již mělo být přiděleno
  if (permissionGranted) {
    // proces připojení k vyhledaným zařízením (pole vyhledaných peers)
    for (WifiP2pDevice device : peersArray) {
      WifiP2pConfig config = new WifiP2pConfig();
      config.deviceAddress = device.deviceAddress;
      // nastavení pravděpodobnosti, že pojící se zařízení bude group owner dle módu
      if (mode == RUN_LIBRARY_AS_CLIENT_VALUE) {
        config.groupOwnerIntent = 15;
      } else {
        config.groupOwnerIntent = 0;
      }
      // init procesu připojení k zařízení - metoda connect()
      p2pManager.connect(channel, config, new WifiP2pManager.ActionListener() {
        @Override
        public void onSuccess() {
          Log.d(TAG, "Připojení k zařízení " + device.deviceAddress + "bylo úspěšné");
        }
        @Override
        public void onFailure(int reason) {
          Log.e(TAG, "Připojení k zařízení" + device.deviceAddress + "se nezdařilo");
        }
      });
    }
  }
}
```

Kód 15 - knihovna pro P2P komunikaci – inicializace připojení k dostupným zařízením

Před samotným připojením k jednotlivým okolním zařízením je dobré určit v jaké roli budou zařízení při komunikaci vystupovat (ne vždy je to potřeba, vzhledem k odlišnostem v chování u rolí hybridního klienta se to však přímo nabízí). Pro tento účel slouží objekt *WifiP2pConfig*, respektive jeho klíčový atribut *groupOwnerIntent*. Tento atribut nabírá hodnot od 0 do 15 a platí, že čím vyšší je jeho hodnota, tím je větší šance, že zařízení s takovým konfiguračním nastavením bude v následné komunikaci představovat roli serveru (hosta). V terminologii *Wi-Fi Direct (P2P) API* je zařízení, jež představuje server či hosta označen jako tzv. *„groupOwner“*, tedy *„vlastník skupiny“*. Pokud se atribut vyhledaného zařízení naopak nastaví na nulu je víceméně jasné, že

takovéto zařízení se bude chovat jako klient. Po úspěšném nastavení konfigurace je možné provést samotné připojení a to prostřednictvím metody *connect()*. Ta v parametrech přijímá opět kanál pro přístup k službám *Wi-Fi Direct (P2P) API*, konfiguraci připojení a listener, jehož metody informují o úspěchu či neúspěchu daného připojení. Metoda navíc vyvolává událost *WIFI_P2P_CONNECTION_CHANGED_ACTION* odchytávanou *BroadcastReceiverem* a právě skrze tuto událost je možné získat informace o připojení a dále definovat roli či chování obou rolí, v nichž může být knihovna spuštěna. Získání informací je vyvoláváno metodou *requestConnectionInfo()* a zachytáváno pomocí listeneru - *ConnectionInfoListener*, jenž je součástí parametru metody. Implementace tohoto listeneru je obsahem *kódu 16*.

```
public WifiP2pManager.ConnectionInfoListener connectionInfoListener = info -> {
   final InetAddress groupOwnerAddress = info.groupOwnerAddress;
   // definice funkcí dle rolí v P2P připojení (groupOwner == server role)
   if (info.groupFormed && info.isGroupOwner) {
      // host - server role
      Log.d(TAG, "Zařízení připojeno v roli serveru");
      ServerRole serverRole = new ServerRole(this);
      serverRole.start();
   } else if (info.groupFormed) {
      // klient role
      Log.d(TAG, "Zařízení připojeno v roli klienta");
      clientRole = new ClientRole(groupOwnerAddress, this);
      clientRole.start();
   }
};
```

Kód 16 - knihovna pro P2P komunikaci – rozdělení chování zařízení v P2P komunikaci dle role

Veškeré informace o připojení jsou v *kódu 16* obsaženy v proměnné *info*. Nejprve je vytažena adresa vlastníka skupiny (*groupOwnerAddress*), tedy adresa zařízení, které vystupuje jako server. Tato adresa je v pozdější implementaci využita pro připojení zařízení v roli klienta. Jakmile je vytvořena komunikační skupina (*groupFormed*), tedy zařízení jsou ve spojení a mohou začít komunikovat, je třeba rozdělit chování dle role. Pro určení role zařízení je možné využít atribut *isGroupOwner*, který definuje, zda dané zařízení je vlastníkem skupiny, tedy server (host) či nikoliv a má představovat roli klienta.

Implementace obou rolí je rozdělena do samostatných tříd (*ServerRole* a *ClientRole*) a vzhledem k tomu, že následující P2P komunikace je implementována asynchronně, běží v novém vlákně na pozadí.

```
public class ServerRole extends Thread {
   private final MainClass mainClass;
   public static final int SOCKET_PORT_NUMBER = 8888;

   @Override
   public void run() {
      try {
         // vytvoření socketů pro komunikaci
         ServerSocket serverSocket = new ServerSocket(SOCKET_PORT_NUMBER);
         Socket socket = serverSocket.accept();
         // zahájení naslouchání komunikace
         SendReceive sendReceive = new SendReceive(socket, mainClass);
         sendReceive.start();
         // odeslání seznamu klientů na zařízení v roli klienta
         sendReceive.writeMessage(mainClass.getMessageToSend());
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Kód 17 - knihovna pro P2P komunikaci – implementace role serveru

V *kódu 17* je představena ukázka implementace role serveru. Lze vidět, že třída dědí od třídy *Thread*, neboť jak již bylo řečeno, implementace by měla být spuštěna na novém vlákně. Jakmile je vlákno spuštěno voláním metody *start()* na instanci třídy reprezentující roli serveru (*ServerRole*), proběhne vytvoření komunikačního soketu (třídy *java.net.ServerSocket* a *java.net.Socket*), následně může být zahájeno naslouchání komunikace skrze vlastní třídu *SendReceive* a zaslání či odeslání dat, opět skrze tuto pomocnou třídu. Úkolem třídy *SendReceive* je tedy samotná implementace procesu přijímání a odesílání zpráv (dat). Vzorová implementace této třídy je obsažena v následujícím *kódu 18*.

Implementace role klienta (třída *ClientRole*) je velmi podobná. Liší se pouze ve způsobu připojování a obsahu zasílaných zpráv. Definice soketu je v tomto případě nahrazena připojením k vytvořenému soketu klienta v podobě P2P serveru (metoda

connect() na instanci třídy *java.net.Socket*). K tomuto účelu je nutné znát adresu serveru (v *kódu 16* proměnná *groupOwnerAddress*). Proto je konstruktor třídy navíc rozšířen o tento atribut. Při připojování je rovněž důležité, aby se shodoval port soketu (v ukázkových kódech se jedná o proměnnou *SOCKET_PORT_NUMBER*). Čtení a zápis dat, tedy užívání třídy *SendReceive* je zcela totožné.

```
public class SendReceive extends Thread {
   private final Socket socket;
   private InputStream inputStream;
   private OutputStream outputStream;
   private final MainClass mainClass;
   private static final int BUFFER_SIZE = 1024;
   private static final int MESSAGE_READ = 1;

   @Override
   public void run() {
      byte[] buffer = new byte[BUFFER_SIZE];
      int bytes;
      while (socket != null) {
         try {
            bytes = inputStream.read(buffer);
            // pokud zpráva není prázdná
            if (bytes > 0) {
               mainClass.handler.obtainMessage(MESSAGE_READ, buffer)
                     .sendToTarget();
            }
         } catch (IOException e) {
            e.printStackTrace();
         }
      }
   }

   // metoda pro zaslání zprávy
   public void writeMessage(byte[] bytes) {
      try {
         outputStream.write(bytes);
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Kód 18 - knihovna pro P2P komunikaci – zasílání a obdržení zpráv

Nejprve jsou definovány struktury a proměnné pro čtení a zápis dat. V konstruktoru jsou poté tyto proměnné inicializovány – především se jedná o *InputStream* a *OutputStream* pro čtení, respektive zápis bajtů z či do komunikačního soketu, jenž byl inicializován při definování chování příslušné role v komunikaci (buď vytvořením soketu nového – role serveru, nebo připojením k existujícímu – role klienta). Čtení dat běží asynchronně v novém vlákně a ukázková implementace je obsažena v metodě *run()* v *kódu 18*. Pakliže zpráva není prázdná, to znamená, že počet bajtů přečtených metodou *read()* a uložených do pole, jež bude následně předané do prostředí hybridního klienta (v *kódu 18* je pole pojmenováno *„buffer“*) je větší než nula, potom je pole předáno do hlavní třídy skrze *Handler* (více v (54)). Princip odesílání a zachytávání zpráv je podobný jako v případě již zmiňovaných *Broadcast* zpráv a listeneru v podobě třídy *BroadcastReceiver*. Zpráva je reprezentována objektem *Message*, jenž je identifikovatelný pomocí jednoznačného kódu (v tomto případě celočíselná konstanta *MESSAGE_READ*, která slouží k identifikace zprávy ke čtení). Zpráva dále obsahuje objekt předávaných dat (nyní pole bajtů), potažmo může obsahovat další dvě celočíselné hodnoty (zde není potřeba). Odchytávání zprávy pomocí *Handleru* je znázorněno v *kódu 19*. Posledním krokem je poté již pouze předání dat (obdržené zprávy) do prostředí hybridního klienta, respektive do jeho hlavní aktivity. Tento proces již byl popsán v kapitole *5.4.1.1 Komunikace (předávání dat) mezi klientem a knihovnami*.

```
public Handler handler = new Handler(new Handler.Callback() {
  @Override
  public boolean handleMessage(@NonNull Message msg) {
    if (msg.what == MESSAGE_READ) {
      // předání dat do prostředí klienta dle role - viz kód 11
      ....
    }
    stopListening(); // zprávy odeslány -> můžeme zastavit vykonávání knihovny
    return true;
  });
```

Kód 19 - knihovna pro P2P komunikaci – odchytávání zpráv pomocí třídy Handler

Zápis dat (metoda *writeMessage()* v *kódu 18*) spočívá pouze v zápisu pole bajtů předaného od hybridního klienta do komunikačního soketu skrze *OutputStream*. Obsah pole je dán rolí klienta.

Je důležité zmínit, že komunikační proces je zahájen (po inicializaci komunikace v metodě *start()*) vždy na základě vyvolání událostí v *BroadcastReceiveru*, kde tyto události jsou vyvolány pouze v případě, že došlo buď ke změně v dostupnosti okolních zařízení či ke změně v připojení s dostupnými zařízeními. Není tedy nutné explicitně znovu volat metody pro vyhledání zařízení (metoda *disoverPeers()*) či pro připojení k danému zařízení (metoda *connect()*), neboť tyto funkce knihovny probíhají automaticky a to do doby, dokud je v hlavní aktivitě hybridního klienta registrovaný *BroadcastReceiver*. Jakmile je odregistrován, chod knihovny je zastaven (jedná se o zásadní implementační krok v řídících metodách pro pozastavení chodu knihovny – metoda *stop()* a pro ukončení běhu knihovny – metoda *exit()*).

Závěrem je dobré říci, že fungování procesu peer-to-peer komunikace značně ovlivnilo použití *Wi-Fi Direct (P2P) API*. Ačkoliv se jedná o nativní API, ne vždy proces proběhl dle předpokladů. Při testování vzorové aplikace se problémy objevily především ve fázi připojování, kdy se poměrně často stávalo, že zařízení se nepřipojila, respektive připojení se podařilo až na několikátý pokus.

6 Shrnutí výsledků

Předmětem diplomové práce byl proces rekonfigurace mobilního klienta na základě změn v kontextu uživatele. Tyto změny se netýkají pouze kontextu vlastního zařízení, nýbrž i okolních zařízení, na což je rovněž brán zřetel. Bylo nutné nadefinovat klienta, jenž je takového chování schopen – tzv. hybridní klient. Dále definovat jeho architekturu či vnitřní strukturu a také navrhnout jakýsi centrální server, jenž umožňuje celou ideu konceptu dále rozšířit o řadu nových možností. Navržený koncept vychází z teoretického návrhu tzv. *„architektury m klient m klient server“* v práci (5). Právě *„architektura m klient m klient server“* a s tím spojený *„m klient“* jsou základní stavební kameny pro koncept hybridního klienta představeného v této diplomové práci. Všechny tyto myšlenky mají bezpochyby velký potenciál využití, nicméně k tomu, aby byly opravdu použitelné, je potřeba je otestovat v reálném prostředí za využití dostupných a ideálně běžně používaných technologií a nástrojů. Za tímto účelem byla vybrána platforma Android, jakožto nejpoužívanější mobilní operační systém současnosti a technologie s ní spojené. Je vhodné podotknout, že součástí teoretického návrhu architektury hybridního klienta již jsou zmíněné technologie, jež se vážou s vývojem aplikací pro Android OS a architektura je tak této platformě částečně přizpůsobena.

Pro správnou funkci hybridního klienta byly vytyčeny následující klíčové aplikační moduly: Modul pro dynamické zavádění externích knihoven, modul pro vyhodnocení datové propustnosti, knihovna pro offline režim a knihovna pro peer-to-peer komunikaci. Součinnost všech modulů má na starost centrální modul a ke komunikaci mezi klientem a centrálním serverem slouží komunikační modul. Klíčovým implementačním nástrojem se ukázala být *Java reflexe*, která umožnila řešit modularitu, tedy zavádění a následné řízení chodu externích knihoven. Zároveň vyřešila většinu problémů s předáváním dat mezi kontextem hybridního klienta a dynamicky zavedenými knihovnami (zbytek problémů vyřešilo zasílání a odchytávání událostí – *BroadcastReceiver API*).

Nejproblematičtějším modulem z hlediska vývoje funkcí byl modul pro dynamické zavádění externích knihoven a obecně problematika týkající se zavádění externích knihoven při zachování jednotného způsobu řízení prostřednictvím metod společného rozhraní pro řízení chodu externích knihoven. Ukázalo se, že spousta externích knihoven (obě testované knihovny v této práci) může ke svému správnému fungování potřebovat různá data z prostředí aplikace čili hybridního klienta, potažmo dodatečnou implementaci

před spuštěním chodu knihovny (např. inicializace stavu, ve kterém má být knihovna zavedena a spuštěna – viz knihovna pro P2P komunikaci) či po jejím spuštění (např. předání seznamu klientů z prostředí aplikace do kontextu knihovny pro offline režim). Toto zjištění může způsobovat značné problémy v budoucím užití konceptu hybridního klienta, tak jak byl navržen v této práci, neboť při větším množství externích knihoven, se dá předpokládat, že nároky na implementaci modulu pro dynamické zavádění externích knihoven budou narůstat, z čehož plyne, krom složitějšího vývoje modulu, i nárůst vytížení zdrojů mobilních zařízení, což není vzhledem k omezeným zdrojům těchto (především starších) zařízení žádoucí.

Nepatrně problematickým se stal rovněž vývoj, respektive následný chod knihovny pro P2P komunikaci. V tomto případě nebyl problém s vývojem či naplněním veškerých funkcí, nýbrž se samotným fungováním nativního *Wi-Fi Direct (P2P) API* pro P2P komunikaci na platformě Android, neboť komunikace mezi klienty, ve smyslu předání požadovaných zpráv, probíhala často se zpožděním či dokonce v některých případech neproběhla vůbec. Řešením tohoto problému by mohlo být nahrazení nativního API jiným externím API či knihovnou.

Nejdůležitějším zjištěním však bylo, že se v prostředí Android OS podařilo implementovat všechny navržené funkce či služby teoretického konceptu hybridního klienta, a to většinou bez jakýchkoliv zásahů do navržené architektury či původního plánu vývoje. Vzorová aplikace reprezentující hybridního klienta byla otestována na více mobilních zařízeních s různými verzemi Android OS, což potvrdilo, že architektura a její implementace je velmi variabilní ve smyslu použití na různých zařízeních (včetně starších zařízení s horšími technickými předpoklady a možnostmi). Platforma Android se v některých případech dokonce ukázala jako velmi vhodná varianta pro vývoj hybridního klienta, což dokazuje kupříkladu značná spojitost ve významu mezi řídícími metodami stavu aktivit, jakožto základní implementační komponenty Android OS, a řídícími metodami společného rozhraní pro externí knihovny, jež ovládají chod dynamicky zavedených knihoven. Teoretický koncept hybridního klienta, jenž umožňuje dynamickou rekonfiguraci mobilní aplikace na základě kontextu uživatele, lze tedy považovat na platformě Android za realizovatelný.

7 Závěry a doporučení

Cíle práce byly splněny. V teoretické části byly představeny základní pojmy důležité pro pochopení této diplomové práce. Jednalo se o pojmy z oblasti síťových architektur a typů klientů vyskytujících se v těchto architekturách, kde byl vytyčen klíčový pojem celé práce a tím je koncept hybridního klienta. Další klíčovou oblastí, jež byla představena byla modularita a s tím spojené řešení modularity v jazyce Java, respektive v souvislosti s Android OS, neboť právě modularita umožňuje rekonfiguraci hybridního klienta jakožto základní princip jeho chování. Nakonec byla představena architektura samotného Android OS coby cílové platformy pro otestování vzorové aplikace představující praktickou podobu teoretického konceptu hybridního klienta. Prakticko-teoretická část se zaměřovala především na bližší představení konceptu hybridního klienta. Byla specifikována architektura a možné podoby či role, jež může klient zastávat. Zároveň byla uvedena role a význam centrálního serveru, který slouží jako koordinátor celého systému a komunikační prostředník pro spolupráci mezi vícero hybridními klienty za účelem dosažení jejich cílů. Čtenář byl rovněž seznámen se vzorovým IS, který slouží k demonstraci možností hybridního klienta v praxi. Závěrem této části je vizuální ukázka vzorové aplikace s patřičným popisem funkcí. Praktická část se poté zabývala již samotnou implementací vzorové aplikace či hybridního klienta a rovněž problémy, které se při vývoji objevily. Součástí jsou i navrhovaná řešení těchto vzniklých problémů.

Hlavním cílem práce bylo odpovědět na otázku, zdali je možné dynamicky za běhu rekonfigurovat mobilní aplikaci na základě změn v kontextu uživatele. Z čistě teoretického pohledu je odpovědí na tuto otázku navržený koncept hybridního klienta z prakticko-teoretické části. Nejdůležitější je však zjištění z ryze praktické roviny, neboť touto diplomovou prací bylo ověřeno, že hybridního klienta je možné skutečně použít v reálném prostředí prakticky v plném rozsahu bez větších omezení tak, jak byl představen a navržen v této práci. Důležité je však poznamenat, že vzorový systém byl otestován pouze v rámci platformy Android, která se zároveň ukázala jako velmi dobrá varianta pro implementaci a následné nasazení hybridního klienta, kdy funkčnost byla ověřena i v širším rozsahu zařízení a verzí OS. Do budoucna se nabízí možnost otestování konceptu i na dalších platformách, především pak rovněž na velmi používaném iOS. Dále je třeba poznamenat, že výsledky práce se odvíjí od složitosti vzorového IS, tedy funkcí, jež by měl hybridní klient být schopen zastávat. Je zřejmé, že složitost IS je úzce svázána se

složitostí vývoje hybridního klienta, a to především ve spojitosti se schopností klienta zastoupit v případě nedostatečné datové propustnosti sítě server, respektive být schopen vykonat alespoň stěžejní služby serveru (je jasné, že se zde musí počítat s určitými omezeními). Tato spojitost se pak může kupříkladu projevit při návrhu a vývoji modulu pro dynamické zavádění externích knihoven, neboť zavádění a spouštění chodu knihoven se může značně zkomplikovat. Problematika složitosti nejen vývoje, ale i samotné aplikace, může také způsobovat výkonnostní problémy především u starších mobilních zařízení, které mají značně omezené systémové zdroje. Je tedy vždy vhodné posoudit, na jaká zařízení je aplikace cílena, a především vymezit funkce, které by měl být schopen hybridní klient vykonávat u složitějších IS.

Koncept hybridního klienta vyžaduje bez pochyby značné otestování na vícero IS, a především na více platformách. Nicméně to, že se podařilo ověřit funkčnost a realizovatelnost konceptu hybridního klienta jako takového lze považovat nejen za splnění cíle této diplomové práce, ale rovněž za značný úspěch v souvislosti s vývojem mobilních klientů. A to zejména těch mobilních klientů, kteří jsou závislý na službách určitého IS skrze internetovou síť a chtějí umět rychle a efektivně reagovat na špatnou datovou propustnost sítě a fungovat tak i v jakémsi offline režimu, tedy nebýt vždy plně závislý na komunikaci se serverem. Především pro takovéto klienty může být koncept hybridního klienta možnou inspirací či návrhovým vzorem. Možnosti rekonfigurace však nejsou omezeny pouze na datovou propustnost sítě, ale mohou se týkat celého uživatelského kontextu. Je možné reagovat například na změny v poloze zařízení či na změny v okolí zařízení prostřednictvím dat z mnoha vestavěných senzorů. Speciálním případem a motivací využití tohoto konceptu pak může být lokalizace ztraceného mobilního zařízení, které není připojeno k internetu. A to za pomoci okolních klientů, kteří mohou komunikovat se ztraceným zařízením na bázi peer-to-peer komunikace, jež je rovněž součástí implementace klienta, a předat údaje o poloze ztraceného telefonu do prostředí internetu. Zásadním přínosem konceptu hybridního klienta tedy je především možnost efektivně reagovat na všechny možné změny v kontextu uživatele daného zařízení dynamicky, tedy za běhu aplikace, a to takovým způsobem, aby změny vnitřní struktury aplikace či jejího chování nikterak neovlivnily koncového uživatele, respektive jeho interakce s aplikací v podobě hybridního klienta.

8 Seznam použité literatury

1. Statista Research Department. Biggest app stores in the world 2020. *Statista* [online]. 2021 [vid. 2021-02-25]. Dostupné z: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

2. JOHNSON, Joseph. Internet users in the world 2021. *Statista* [online]. 2021 [vid. 2021-03-05]. Dostupné z: https://www.statista.com/statistics/617136/digital-population-worldwide/

3. MindSea Team. 28 Mobile App Usage & Revenue Statistics To Know In 2021. *MindSea* [online]. [vid. 2021-03-05]. Dostupné z: https://mindsea.com/app-stats/

4. YAN, Zhiwei a Jong-Hyouk LEE. Mobility capability negotiation for IPv6 based ubiquitous mobile Internet. *Computer Networks* [online]. 2019, **157**, 24–28. ISSN 1389-1286. Dostupné z: doi:10.1016/j.comnet.2019.03.012

5. MALÝ, Filip. *Architektura m klient m klient server*. Hradec Králové, 2013. text Habilitační práce. Univerzita Hradec Králové, Fakulta informatiky a managementu.

6. Mobile Operating System Market Share Worldwide | StatCounter Global Stats. *StatCounter* [online]. 2021 [vid. 2021-03-05]. Dostupné z: https://gs.statcounter.com/os-market-share/mobile/worldwide

7. CONRAD, Eric, Seth MISENAR a Joshua FELDMAN. Chapter 4 - Domain 3: Security Engineering (Engineering and Management of Security). In: Eric CONRAD, Seth MISENAR a Joshua FELDMAN, ed. *CISSP Study Guide (Third Edition)* [online]. Boston: Syngress, 2016 [vid. 2021-03-05], s. 103–217. ISBN 978-0-12-802437-9. Dostupné z: doi:10.1016/B978-0-12-802437-9.00004-7

8. BUYYA, Rajkumar, Christian VECCHIOLA a S. Thamarai SELVI. Chapter 2 - Principles of Parallel and Distributed Computing. In: Rajkumar BUYYA, Christian VECCHIOLA a S. Thamarai SELVI, ed. *Mastering Cloud Computing* [online]. Boston: Morgan Kaufmann, 2013 [vid. 2021-03-05], s. 29–70. ISBN 978-0-12-411454-8. Dostupné z: doi:10.1016/B978-0-12-411454-8.00002-4

9. MOUNA, mallipeddi. Thin Client Vs Thick Client. *Medium* [online]. 2018 [vid. 2021-03-05]. Dostupné z: https://medium.com/@mouna.mallipeddi/thin-client-vs-thick-client-69d90c13d02d

10. CHEN, Hung-Ming a Chuan-Chien HOU. Asynchronous online collaboration in BIM generation using hybrid client-server and P2P network. *Automation in Construction* [online]. 2014, **45**, 72–85. ISSN 0926-5805. Dostupné z: doi:10.1016/j.autcon.2014.05.007

11. IVANKOV, Alex. Thick Client vs. Thin Client: Advantages and Disadvantages. *Profolus* [online]. 2018 [vid. 2021-03-06].

Dostupné z: https://www.profolus.com/topics/thick-client-vs-thin-client-advantages-and-disadvantages/

12. ADHIKARI, Binayak. Types of Network Architecture. *Medium* [online]. 2018 [vid. 2021-03-06]. Dostupné z: https://medium.com/@blazebnayak/types-of-network-architecture-393e4ef5530e

13. GREENLAW, Raymond a Y. Daniel LIANG. Object-Oriented Programming. In: Hossein BIDGOLI, ed. *Encyclopedia of Information Systems* [online]. New York: Elsevier, 2003 [vid. 2021-03-09], s. 347–361. ISBN 978-0-12-227240-0. Dostupné z: doi:10.1016/B0-12-227240-4/00124-6

14. SURYANARAYANA, Girish, Ganesh SAMARTHYAM a Tushar SHARMA. Chapter 5 - Modularization Smells. In: Girish SURYANARAYANA, Ganesh SAMARTHYAM a Tushar SHARMA, ed. *Refactoring for Software Design Smells* [online]. Boston: Morgan Kaufmann, 2015 [vid. 2021-03-08], s. 93–122. ISBN 978-0-12-801397-7. Dostupné z: doi:10.1016/B978-0-12-801397-7.00005-9

15. MUSTAFIC, Asmir. Modular Application Architecture - Intro. *goetas* [online]. 2017 [vid. 2021-03-09]. Dostupné z: https://www.goetas.com/blog/modular-application-architecture-intro

16. VAHDAT-NEJAD, Hamed, Azam RAMAZANI, Tahereh MOHAMMADI a Wathiq MANSOOR. A survey on context-aware vehicular network applications. *Vehicular Communications* [online]. 2016, **3**, 43–57. ISSN 2214-2096. Dostupné z: doi:10.1016/j.vehcom.2016.01.002

17. OTEBOLAKU, Abayomi Moradeyo a Maria Teresa ANDRADE. User context recognition using smartphone sensors and classification models. *Journal of Network and Computer Applications* [online]. 2016, **66**, 33–51. ISSN 1084-8045. Dostupné z: doi:10.1016/j.jnca.2016.03.013

18. VADILLO MORENO, Laura, María Luisa MARTÍN RUIZ, Javier MALAGÓN HERNÁNDEZ, Miguel Ángel VALERO DUBOY a María LINDÉN. Chapter 14 - The Role of Smart Homes in Intelligent Homecare and Healthcare Environments. In: Ciprian DOBRE, Constandinos MAVROMOUSTAKIS, Nuno GARCIA, Rossitza GOLEVA a George MASTORAKIS, ed. *Ambient Assisted Living and Enhanced Living Environments* [online]. B.m.: Butterworth-Heinemann, 2017 [vid. 2021-03-09], s. 345–394. ISBN 978-0-12-805195-5.
Dostupné z: doi:10.1016/B978-0-12-805195-5.00014-4

19. GOOGLE LLC. Platform Architecture. *Android Developers* [online]. 2021 [vid. 2021-03-13]. Dostupné z: https://developer.android.com/guide/platform

20. Android Runtime (ART) and Dalvik. *Android Open Source Project* [online]. 2020 [vid. 2021-03-13]. Dostupné z: https://source.android.com/devices/tech/dalvik?hl=cs

21. GOOGLE LLC. App resources overview. *Android Developers* [online]. 2020 [vid. 2021-03-13].

Dostupné z: https://developer.android.com/guide/topics/resources/providing-resources

22. GOOGLE LLC. Notifications Overview. *Android Developers* [online]. 2020 [vid. 2021-03-13]. Dostupné z: https://developer.android.com/guide/topics/ui/notifiers/notifications

23. GOOGLE LLC. Introduction to Activities. *Android Developers* [online]. 2019 [vid. 2021-03-13]. Dostupné z: https://developer.android.com/guide/components/activities/intro-activities

24. GOOGLE LLC. Package Index. *Android Developers* [online]. 2021 [vid. 2021-03-14]. Dostupné z: https://developer.android.com/reference/packages

25. GOOGLE LLC. android.app. *Android Developers* [online]. 2021 [vid. 2021-03-14]. Dostupné z: https://developer.android.com/reference/android/app/package-summary

26. GOOGLE LLC. Android Jetpack. *Android Developers* [online]. [vid. 2021-03-14]. Dostupné z: https://developer.android.com/jetpack

27. DUTT, Parth. Compilation and Execution of a Java Program. *GeeksforGeeks* [online]. 2018 [vid. 2021-03-15]. Dostupné z: https://www.geeksforgeeks.org/compilation-execution-java-program/

28. ORACLE. Essentials, Part 1, Lesson 1: Compiling & Running a Simple Program. *Oracle* [online]. [vid. 2021-03-15]. Dostupné z: https://www.oracle.com/java/technologies/compile.html

29. BHATIA, Krishna. Differences between JDK, JRE and JVM. *GeeksforGeeks* [online]. 2017 [vid. 2021-03-15]. Dostupné z: https://www.geeksforgeeks.org/differences-jdk-jre-jvm/

30. MANOJ, Dannana. ClassLoader in Java. *GeeksforGeeks* [online]. 2019 [vid. 2021-03-15]. Dostupné z: https://www.geeksforgeeks.org/classloader-in-java/

31. MCCLUSKEY, Glen. Using Java Reflection. *Oracle* [online]. 1998 [vid. 2021-03-15]. Dostupné z: https://www.oracle.com/technical-resources/articles/java/javareflection.html

32. BAELDUNG. Guide to Java Reflection. *Baeldung* [online]. 2016 [vid. 2021-03-15]. Dostupné z: https://www.baeldung.com/java-reflection

33. MARKOVIC, Ban. Process of compiling Android app with Java/Kotlin code. *Medium* [online]. 2020 [vid. 2021-03-15]. Dostupné z: https://medium.com/@banmarkovic/process-of-compiling-android-app-with-java-kotlin-code-27edcfcce616

34. GOOGLE LLC. DexClassLoader. *Android Developers* [online]. 2019 [vid. 2021-03-15].
Dostupné z: https://developer.android.com/reference/dalvik/system/DexClassLoader

35. STEGNER, Ben. What Is an APK File and What Does It Do? *MUO* [online]. 2017 [vid. 2021-03-15]. Dostupné z: https://www.makeuseof.com/tag/what-is-apk-file/

36. GOOGLE LLC. Sensors Overview. *Android Developers* [online]. 2019 [vid. 2021-03-23].
Dostupné z: https://developer.android.com/guide/topics/sensors/sensors_overview

37. GEER, Zulaikha. What is REST API? — A Comprehensive Guide To RESTful APIs. *Medium* [online]. 2019 [vid. 2021-04-02]. Dostupné z: https://medium.com/edureka/what-is-rest-api-d26ea9000ee6

38. CHRISTENSSON, P. IMEI (International Mobile Equipment Identity) Definition. *TechTerms* [online]. 20. červenec 2018 [vid. 2021-03-29]. Dostupné z: https://techterms.com/definition/imei

39. Wi-Fi Alliance. Wi-Fi Direct. *Wi-Fi Alliance* [online]. [vid. 2021-04-02]. Dostupné z: https://www.wi-fi.org/discover-wi-fi/wi-fi-direct

40. GOOGLE LLC. Wi-Fi Direct (peer-to-peer or P2P) overview. *Android Developers* [online]. 2021 [vid. 2021-04-02].
Dostupné z: https://developer.android.com/guide/topics/connectivity/wifip2p

41. GOOGLE LLC. App Manifest Overview. *Android Developers* [online]. 2021 [vid. 2021-04-02].
Dostupné z: https://developer.android.com/guide/topics/manifest/manifest-intro

42. PICHL, Daniel. *Modulární aplikace na platformě Android* [online]. Hradec Králové, 2015 [vid. 2021-04-02]. Bakalářská práce. Univerzita Hradec Králové, Fakulta informatiky a managementu. Vedoucí práce doc. Ing. Filip Malý, Ph.D. Dostupné z: https://theses.cz/id/nndyb0/

43. VMware, Inc. Spring Boot. *spring* [online]. [vid. 2021-04-02]. Dostupné z: https://spring.io/projects/spring-boot

44. DUTTA, Prashant. Quick Guide to Spring Controllers. *Baeldung* [online]. 2016 [vid. 2021-04-03]. Dostupné z: https://www.baeldung.com/spring-controllers

45. BALASUBRAMANIAM, Vivek. Defining JPA Entities. *Baeldung* [online]. 2019 [vid. 2021-04-03]. Dostupné z: https://www.baeldung.com/jpa-entities

46. GIERKE, Oliver, Thomas DARIMONT, Christoph STROBL, Mark PALUCH a Jay BRYANT. Spring Data JPA - Reference Documentation. *docs.spring* [online]. 2021 [vid. 2021-04-03].
Dostupné z: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories

47. BAELDUNG. @Component vs @Repository and @Service in Spring. *Baeldung* [online]. 2018 [vid. 2021-04-03]. Dostupné z: https://www.baeldung.com/spring-component-repository-service

48. Huawei Y6 - Full phone specifications. *gsmarena* [online]. [vid. 2021-05-29]. Dostupné z: https://www.gsmarena.com/huawei_y6-7440.php

49. Huawei P smart - Full phone specifications. *gsmarena* [online]. [vid. 2021-05-29]. Dostupné z: https://www.gsmarena.com/huawei_p_smart-8961.php

50. ORACLE. Socket (Java Platform SE 7). *Java™ Platform, Standard Edition 7 API Specification* [online]. [vid. 2021-06-08].
Dostupné z: https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html

51. GOOGLE LLC. NetworkOnMainThreadException. *Android Developers* [online]. 2021 [vid. 2021-06-11].
Dostupné z: https://developer.android.com/reference/android/os/NetworkOnMainThreadException

52. GOOGLE LLC. SharedPreferences.Editor. *Android Developers* [online]. 2021 [vid. 2021-06-21].
Dostupné z: https://developer.android.com/reference/android/content/SharedPreferences.Editor?hl=cs

53. GOOGLE LLC. Broadcasts overview. *Android Developers* [online]. 2021 [vid. 2021-06-21].
Dostupné z: https://developer.android.com/guide/components/broadcasts?hl=cs

54. GOOGLE LLC. Handler. *Android Developers* [online]. 2021 [vid. 2021-08-06].
Dostupné z: https://developer.android.com/reference/android/os/Handler

9 Seznam obrázků

10 Seznam kódů

11 Přílohy

Příloha 1 - zadání diplomové práce

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Akademický rok: 2019/2020

Studijní program: Aplikovaná informatika
Forma studia: Prezenční
Obor/kombinace: Aplikovaná informatika (ai2-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

Jméno a příjmení:	**Bc. Ondřej Schneider**
Osobní číslo:	**I1900806**
Adresa:	Pražská 1552, Náchod – Staré Město nad Metují, 54701 Náchod 1, Česká republika
Téma práce:	Rekonfigurace mobilní aplikace na základě kontextu uživatele
Téma práce anglicky:	Configuring mobile application based on the user's context
Vedoucí práce:	doc. Ing. Filip Malý, Ph.D. Katedra informatiky a kvantitativních metod

Zásady pro vypracování:

Osnova: 1) Úvod 2) Síťové architektury a typy klientů 3) Architektura Android OS 4) Návrh architektury a vzorových aplikací 5) Workflow vývoje 6) Implementace klienta a serveru 7) Zhodnocení výsledků a závěr 8) Literatura
Cíl práce: Navrhnout řešení a otestovat možnosti rekonfigurace mobilní aplikace za běhu na platformě Android v závislosti na kontextu uživatele, především na datové propustnosti sítě.

Seznam doporučené literatury:

Podpis studenta: Datum:

Podpis vedoucího práce: Datum: