UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Klient distribuované sítě

Michal Malec

Diplomová práce
2013

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2012/2013**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Michal Malec**
Osobní číslo:	**I10394**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Klient distribuované sítě**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

Cílem diplomové práce je analýza, návrh a implementace klienta distribuované sítě jednotek s operačním systémem Android.
V teoretické části bude zpracován stručný přehled architektur distribuovaných systémů. Dále v teoretické části bude analýza a návrh klienta distribuovaného systému pro jednotku s operačním systémem Android podle těchto systémových požadavků:
1 Klient bude zpracovávat údaje ze senzorů, které daná jednotka obsahuje, například polohu, akcelerometr, gyroskop, osvětlení nebo barometrický tlak.
2) Klient bude zpracované data ze senzorů zasílat ostatním klientům, kteří budou v dosahu.
3) Klient bude zobrazovat svá data ze senzorů i přijatá data od ostatních klientů v síti.
4) Klient bude rozesílat přijatá data od klientů dalším klientům podle algoritmu na odstraněním duplicit v šíření dat.
5) Analýza a návrh bude v UML.
V praktické části bude popis implementace a ověření funkčnosti klienta. Realizace aplikace klienta bude v jazyku Java pro operační systém Android. Doporučuje se použít vývojové prostředí Eclipse. Na paměťovém médiu, které bude přiloženo k diplomové práci, budou uloženy soubory s projektem klienta.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

[1] Arlow J., Neustad I., UML2 a unifikovaný proces vývoje aplikací, Computer Press 2007, 567s, ISBN 978-80-251-1503-9
[2] Klimeš, C. Distribuované systémy, Ostrava: Ostravská univerzita 2004,
[3] Penzeš, J. Distribuované systémy platformy JAVA, Pardubice, Univerzita Pardubice, 2011
[4] MURPHY, Mark L. Android 2: průvodce programováním mobilních aplikací. Vyd. 1. Brno: Computer Press, 2011, 375 s. ISBN 978-80-251-3194-7.

Vedoucí diplomové práce: **Ing. Karel Šimerda**
Katedra softwarových technologií

Datum zadání diplomové práce: **31. října 2012**
Termín odevzdání diplomové práce: **17. května 2013**

Univerzita Pardubice Česká republika

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2012

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 20. 08. 2013 Michal Malec

Poděkování

Děkuji především vedoucímu práce Ing. Karlu Šimerdovi, že se mnou měl trpělivost a ochotu konzultovat práci i v nezvyklých časech.

Anotace

Teoretická část práce shrnuje základní poznatky o distribuovaných systémech a sítích, včetně příkladů technologií, které se dají použít pro tvorbu distribuovaných systémů. Praktická část práce zpracovává analýzu, návrh a implementaci klienta distribuované sítě pro operační systém Android pomocí knihovny Wi-Fi Direct. V jednotlivých kapitolách jsou popsány nejdůležitější části klienta.

Klíčová slova

Distribuovaná síť, Android, Wi-Fi, Wi-Fi Direct

Title

Client of distributed network

Annotation

The theoretical part of the thesis summarizes the basic knowledge of distributed systems and networks, including examples of technologies that can be used for the development of distributed systems. The practical part of the work process analysis, design and implementation of distributed network client for the Android operating systém using Wi-Fi Direct library. In each chapter describes the most important part of the client.

Keywords

Distributed network, Android, Wi-Fi, Wi-Fi Direct

Obsah

Seznam zkratek ... **8**

Seznam obrázků ... **9**

1 Úvod ... **10**

1.1 Formátovací konvence ... 10

2 Distribuované systémy ... **10**

2.1 Základní požadavky ... 10

2.2 Příklady technologie distribuovaných systémů ... 11

2.2.1 CORBA ... 12

2.2.2 Java RMI ... 12

3 Analýza ... **13**

3.1 Model požadavků ... 13

3.1.1 Funkční požadavky ... 13

3.1.2 Nefunkční požadavky ... 14

3.2 Model případů užití ... 14

3.2.1 Retranslace ... 14

3.2.2 Vysílání údajů ... 16

3.2.3 Zobrazení a administrátorský zásah ... 17

3.2.4 Dispečerský zásah ... 19

3.3 Analytické třídy ... 19

3.4 Diagramy aktivit ... 21

3.4.1 Spuštění aplikace ... 21

3.4.2 Síťová komunikace ... 22

3.4.3 Příjem zprávy ... 24

3.5 Jak byla analýza ovlivněna implementací. ... 25

4 Návrh ... **25**

4.1 Návrhové třídy ... 25

4.1.1 Třídy stereotypu Model ... 25

4.1.2 Třídy stereotypu Controller ... 27

4.1.3 Třídy stereotypu View ... 29

4.2 Jak byl návrh ovlivněn použitou technologií ... 30

5 Realizace ... **30**

5.1 Použité nástroje 30
5.2 Debugování aplikace 32
6 Implementace 33
6.1 Wi-Fi Direct 34
6.2 Java NIO 34
6.3 Fragmenty 35
6.4 Implementační třídy 35
6.4.1 Retranslace 36
6.4.2 ServerSideAsyncTask 38
6.4.3 ClientSideAsyncTask 42
6.5 Sekvenční diagramy retranslace 44
6.6 Fragment monitoringu 45
6.7 Řízení zapínání a vypínání režimů 49
7 Instalace aplikace 50
7.1 Android a práva pro aplikace 50
7.2 Instalace 50
Závěr 52
Literatura 54
Příloha A – Diagram tříd celé aplikace 8

Seznam zkratek

API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
IDL	Interface Definition Language
JVM	Java Virtual Machine
LAN	Local Area Network
MVC	Model View Controller
PC	Personal Computer
RMI	Remote Method Invocation
USB	Universal Seril Bus
XML	Extensible Markup Language

Seznam obrázků

Obrázek 1- Model funkčních požadavků 13
Obrázek 2 - Model nefunkčních požadavků 14
Obrázek 3 - Případy užití retranslace 15
Obrázek 4 - Model případu užití vysílání údajů 17
Obrázek 5 - Případy užití zobrazování a administrátorského zásahu 18
Obrázek 6 - Diagram analytických tříd metodou CRC štítků 20
Obrázek 7 - Diagram aktivity spuštění aplikace 22
Obrázek 8 - Diagram startu síťové části klienta 23
Obrázek 9 - Diagram aktivity, algoritmus přijmutí a vyhodnocení zprávy 24
Obrázek 10 - Diagram návrhových tříd – Model 26
Obrázek 11 - Model návrhových tříd – Controller 28
Obrázek 12 - Model návrhových tříd – View 29
Obrázek 13 - Android SDK manager 31
Obrázek 14 - Nastavení vývojářských nástrojů v operačním systému Android 33
Obrázek 15 - Třída argumenty 36
Obrázek 16 - Implementační třídy - retranslace 37
Obrázek 17 - Sekvenční diagram retranslace, serverová část 44
Obrázek 18 - Sekvenční diagram retranslace, klientská část 45
Obrázek 19 - Diagram tříd monitoringu 46
Obrázek 20 - Nastavení povolení instalovat aplikace 51
Obrázek 21 - Instalace aplikace 52

1 Úvod

Cílem této práce je naprogramovat klienta distribuované sítě, který poběží na operačním systému Android. Každý z klientů by měl monitorovat své okolí pomocí bezdrátového připojení k síti a v případě, že se v jeho okolí vyskytne nějaký další klient, měli by se navzájem propojit do sítě a začít si vyměňovat mezi sebou zprávy o stavu svých senzorů. Zařízení s operačním systémem Android budu dále nazývat jednotka, zatímco samotný program bude klient. Název jednotka jsem vybral záměrně, jelikož operační systém Android, konkrétně ve verzi 4.0. a vyšší, může obsahovat širší škála zařízení. Mohou to být tedy jak telefony, tak třeba tablety. Z hlediska práce je ovšem jedno, jestli je to telefon nebo tablet.

Je-li tedy více jednotek vzájemně v dosahu, sestaví se do sítě a vzájemně pomocí retranslace začnou mezi sebou rozesílat zprávy o stavu svých senzorů. V ideálním případě by tedy všechny jednotky měly mít přehled o všech ostatních, připojených do sítě. Vznikne tedy síť na bázi obecného grafu, kdy jednotlivé jednotky jsou propojeny s okolními v závislosti na dosahu jejich bezdrátové sítě.

Každá z jednotek by měla mít několik základních režimů. Tím prvním je režim retranslace, který se stará o předávání zpráv ostatním jednotkám v okolí. Druhým režimem je aktivní rozesílání svých údajů okolním jednotkám. Třetím režimem je administrátorský zásah, kdy je možné z jednotky s administračními právy zapínat a vypínat režimy ostatních jednotek. Posledním režimem je prohlížení detailů ostatních jednotek.

Toto byla původní představa, jak by měla vypadat distribuovaná síť jednotek. V průběhu práce se však ukázalo, že je potřeba několik konceptů změnit, jelikož technologie neumožňuje libovolné propojování jednotek. Detaily těchto změn rozeberu níže.

1.1 Formátovací konvence

V textu budou uvedeny odkazy na názvy tříd, datových typů nebo metod pomocí fontu Courier New velikosti 12. Například `Trida` nebo datový typ `String`. Dále se pak v textu mohou vyskytovat ukázky zdrojového kódu. Ty jsou odlišeny pro lepší orientaci barevně a jsou formátovány písmem Consolas velikosti 10. Například ukázka zdrojového kódu metody `onCreate()` třídy `MainActivity`.

```
protected void onCreate(Bundle savedInstanceState) {}
```

2 Distribuované systémy

2.1 Základní požadavky

Distribuovaný systém je soubor počítačů, které jsou vzájemně propojené pomocí počítačové sítě, nejčastěji to bývá síť LAN, a distribuovaného software, který je propojuje,

tzv. middleware. Jednotlivé počítače tak spolupracují na společné úloze, stejně jako mohou vzájemně sdílet zdroje, které poskytuje třeba server v dané síti.

Aby takový systém mohl vůbec fungovat, musí splňovat určité požadavky. Těmito požadavky jsou různorodost, otevřenost, bezpečnost, rozšiřitelnost, odolnost vůči chybám a konkurenční prostředí a transparentnost.

Transparentnost můžeme dělit na několik dalších požadavků. Jsou jimi:

- Přístupová transparentnost – klient by neměl poznat rozdíl mezi lokálním a vzdáleným prostředkem (například souborem). Systém by měl poskytovat takové služby, aby se nemuselo rozlišovat mezi souborem na lokálním disku a souborem na vzdáleném serveru.
- Lokační transparentnost - klienti by měli vidět pouze jedno celistvé úložiště, nezávislé na skutečné lokaci souboru. Systém by měl zajišťovat takové prostředky, aby uživatelé nemuseli měnit cesty k souboru v závislosti na jeho místě uložení.
- Konkurenční transparentnost – uživatelé nebo aplikace v distribuovaném systému by měly být schopné přistupovat k prostředkům nebo souborům v systému ve stejný čas, aniž by docházelo k nežádoucí interferenci mezi jednotlivými požadavky.
- Replikační transparentnost – tento požadavek je kladen zejména na distribuované souborové systémy. Distribuovaný souborový systém by měl být schopný vytvářet více kopií stejného souboru tak, aby bylo možné ho poskytovat co nejsnadněji všem uživatelům, kteří ho požadují. Avšak uživatel by neměl být schopný určit, kolik kopií daného souboru existuje.
- Chybová transparentnost – uživatelé nebo programy běžící v distribuovaném systému by měli být schopni dokončit svoje úlohy bez ohledu na chyby hardwaru jednotlivých částí systému. Tzn., že systém by měl být schopný nahrazovat a vypořádat se s případnými výpadky určité části systému, aniž by se to na venek projevilo.
- Migrační transparentnost – uživatel by neměl poznat, že se zpracování procesu přesunulo na jiný počítač v důsledku třeba optimalizace výpočetního výkonu. Tento mechanizmus je důležitý právě pro lepší optimalizaci výpočetního výkonu a odlehčování zátěži počítačům, které jsou třeba vytěžovány více než ostatní.
- Transparentnost rozšiřitelnosti systému – systém by měl být snadno rozšiřitelný. Běh procesů by neměl být ovlivněn rozsáhlostí systému. Systém by tedy měl běžet stále stejně i v případě, že je rozšířen o další počítač.

2.2 Příklady technologie distribuovaných systémů

Uvedu zde dvě technologie, které se používají pro provoz distribuovaných systémů. Tou první je CORBA, což je standard definovaný Object Management Group, který umožňuje běh aplikací, napsaných v různých jazycích, jako jeden systém. Druhou technologií je Java RMI, což je technologie pouze pro Javu umožňující dálkovou správu objektů.

2.2.1 CORBA

CORBA dovoluje separovaným programům napsaným v různých programovacích jazycích a běžících na různých počítačích kooperovat, jako by byly jeden program nebo jedna sada služeb. Je to vlastně mechanizmus normalizování volání metod mezi aplikačními objekty sdílejícími stejný adresní prostor v rámci aplikace. Případně umožňuje to samé pro komunikaci přes počítačovou síť.

CORBA používá pro komunikaci mezi objekty různých jazyků takzvané IDL, což je rozhraní, které popisuje objekt nezávisle na jazyku, ve kterém je implementován. Načež CORBA definuje mapování z IDL na specifickou implantaci už v konkrétním jazyce jako je třeba Java nebo C++.

V základu je tedy nutné vytvořit IDL kód, který reprezentuje rozhraní pro vytvořené objekty v konkrétním programovacím jazyku. Poté nastupuje implementace CORBA, která obsahuje kompilátor pro IDL, který je schopný vytvořit mapování na objekty v už konkrétním programovacím jazyce.

2.2.2 Java RMI

RMI je zkratka pro Remote Method Invocation, což znamená vzdálené volání metod. Java RMI je technologie, která umožňuje vytvářet distribuované systémy založené na Javě. Technologie umožňuje volat metody objektů vytvořených pomocí RMI i když jsou jejich instance na jiném JVM, případně na jiném vzdáleném počítači. RMI dovoluje skutečné mapování a přenášení objektů, aniž by se ztrácely nebo byly ořezávány určité datové typy tak, jak se občas děje při obyčejné serializaci. Stejně tak umožňuje přenášení garbage collectingu mezi jednotlivými počítači spojenými touto technologií v jeden systém.

Technologie RMI funguje na principu klient – server, kdy server poskytuje klientovi údaje o objektech, které může klient používat. Jinak řečeno, server vytváří stub pro každý objekt a ukládá ho do RMI registru, do kterého se pak klient obrací se žádostmi o instance objektu.

Jako malý náhled do problematiky Java RMI jsem použil tutoriál podle webových stránek (Reilly, 2006), který ukazuje jednoduchý příklad implementace služby používající právě technologii RMI. Ve zkratce, aniž bych používal zdrojové kódy, shrnu funkčnost Java RMI v následujících odstavcích.

Funkce Java RMI je poměrně jednoduchá. V základu musíme udělat pro každý objekt rozhraní, které dědí od `java.rmi.Remote`. K tomuto rozhraní pak vytvoříme objekt, který ho implementuje a přitom může vyhodit `RemoteException`. Tím máme hotový objekt, který můžeme volat vzdáleně, z jiného JVM.

Posledním krokem je zaregistrování instance takto vytvořené služby pomocí `Naming.bind(„identifkator“,instance)`. Takto uložíme instanci služby pod identifikátorem, kterým ji pak následně můžeme zavolat ze vzdálené JVM.

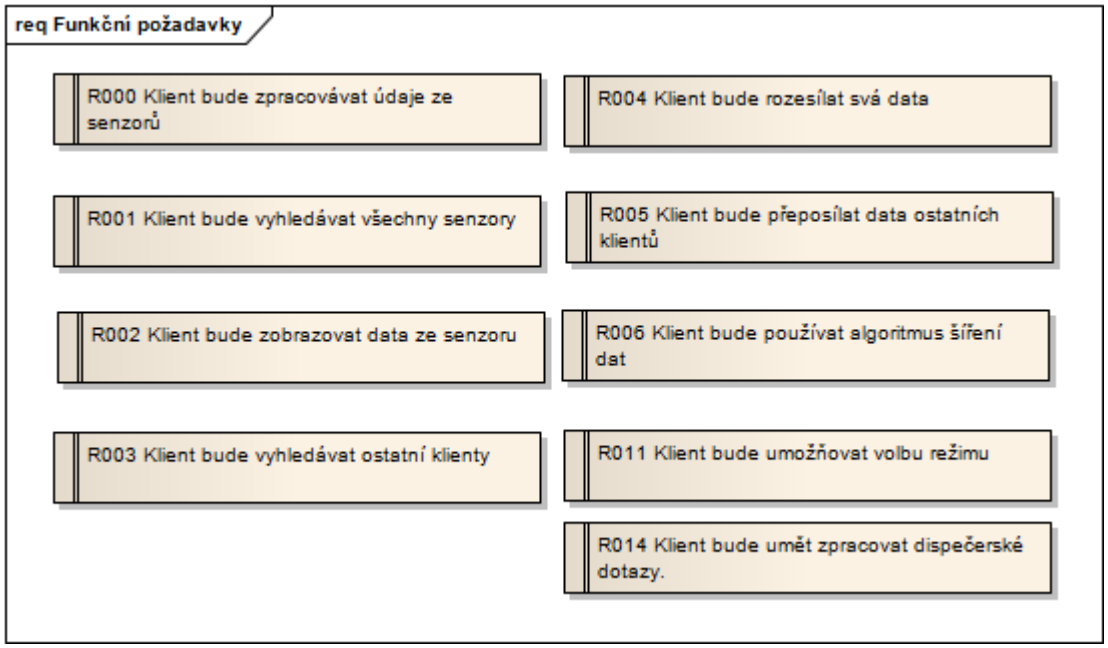
Instanci objektu pak můžeme v jiném programu, který umí používat RMI, získat tím, že zavoláme `Naming.lookup(„cesta/identifikátor“)`. RMI se podívá na zadanou cestu a hledá objekt, který má požadovaný identifikátor. V případě, že ho najde, vrátí jeho instanci.

3 Analýza

3.1 Model požadavků

3.1.1 Funkční požadavky

Model funkčních požadavků vznikl na základě analýzy zadání. Zahrnuje funkčnost klienta distribuované sítě tak, jak bylo stanoveno v zadání práce. Model funkčních požadavků si můžeme lépe prohlédnout na obrázku Obrázek 1, který zahrnuje všechny funkční požadavky, které vyplynuly z analýzy.

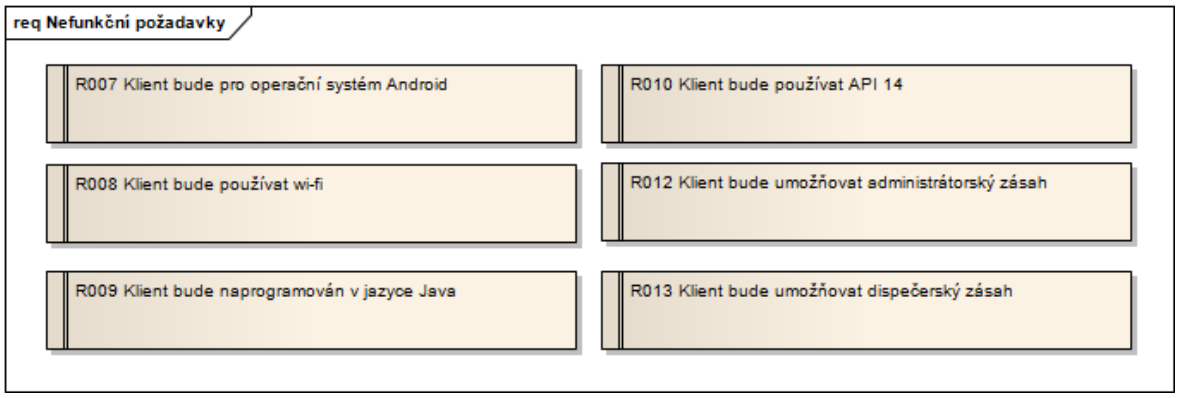


Obrázek 1- Model funkčních požadavků

Nejdůležitějšími požadavky jsou ty, které definují klienta jako součást distribuované sítě. Jde především o požadavek R005, který říká, že klient bude přeposílat data ostatních klientů a tím pádem bude docházet k distribuci dat mezi všemi klienty v síti. Dále pak R006, který definuje, že rozesílání zpráv nebude čistě záplavové, ale bude použit algoritmus, který zabezpečí nejmenší možnou režii na síti. Požadavky na síť pak definuje R003. Tento požadavek zajišťuje, že se klient bude pokoušet aktivně vyhledávat okolní klienty a tím dosáhnout pružnosti sítě a obnovování ztracených připojení mezi jednotlivými klienty. Požadavky R000, R001, R002 a R004 definují samostatnou funkci klienta. Každý klient v síti bude monitorovat stav svých senzorů a rozesílat je ostatním klientům.

3.1.2 Nefunkční požadavky

Model nefunkčních požadavků můžeme vidět na obrázku Obrázek 2. Nefunkční požadavky byly stanoveny na základě analýzy zadání a na základě potřebného vybavení jednotky[1] s operačním systémem Android tak, aby bylo možné zprovoznit síťové spojení mezi jednotlivými klienty.



Obrázek 2 - Model nefunkčních požadavků

Požadavek R008 zabezpečuje, že každá z jednotek bude mít možnost navázat síťové spojení s ostatními klienty. Pro tuto práci byla vybrána technologie Wi-Fi, jelikož Bluetooth nemá požadovaný dosah. Jednotky by totiž měly být schopné komunikovat na větší vzdálenosti. Požadavky R007 a R008 určují platformu, na které bude klient pracovat. Operační systém Android byl vybrán kvůli své rozšířenosti a otevřenosti, včetně toho, že většina telefonů, případně ostatních zařízení, má možnost přístupu k bezdrátovému připojení k síti. K tomuto se váže i požadavek R010, který určuje minimální úroveň API, které je nutné k provozování klienta. API 14 bylo vybráno, protože obsahuje knihovnu Wi-Fi Direct, která umožňuje komunikaci jednotek bez nutnosti připojení přes access point.

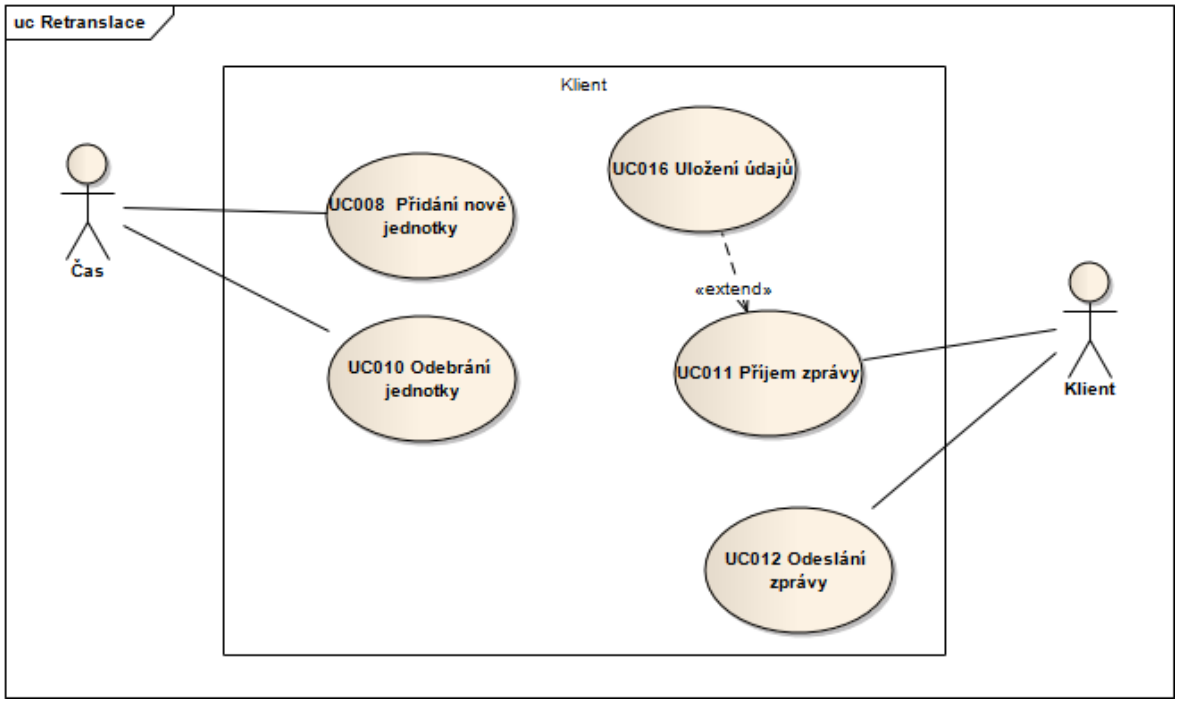
3.2 Model případů užití

Uvedu zde modely případů užití pro klíčové stránky klienta. Jedná se zejména o retranslaci zpráv mezi jednotkami, která je z mého pohledu nejzajímavější částí systému. Dále pak zobrazování jednotlivých údajů o okolních jednotkách včetně údajů z vlastních senzorů. Ve třetí části nastíním problematiku ostatních případů užití, které patří do systému, ale nepatří mezi klíčové vlastnosti nebo funkčnosti.

3.2.1 Retranslace

Na obrázku Obrázek 3 si můžeme prohlédnout případy užití retranslace zpráv mezi jednotkami. Retranslace je mechanizmus přeposílání zpráv mezi jednotkami, který zajišťuje, že se ke všem jednotkám v síti za určitý čas dostanou údaje o ostatních jednotkách v síti.

[1] Jednotkou je myšlen přístroj s operačním systémem Android a možností připojení k bezdrátové síti Wi-Fi.



Obrázek 3 - Případy užití retranslace

Aktéry v případě retranslace jsou čas a samotný klient nebo systém. Čas zde zastupuje události, které se mohou stát vně systému a je potřeba na ně reagovat. Jsou to případy, kdy je potřeba přidat nebo odebrat jednotku ze sítě. UC008 zabezpečuje, že v případě, že se v dosahu některé jednotky se spuštěným klientem sítě objeví nová jednotka, taktéž se zapnutým klientem sítě, která ještě není připojená do již fungující sítě, bude tato jednotka do sítě zapojena. Tento mechanizmus zajišťuje dynamické fungování sítě. Tohoto se týká i UC010. Tento případ užití zase zajišťuje možnost, že se některá z jednotek dostane z dosahu všech ostatních. V tomto případě by mělo dojít k odebrání jednotky ze seznamu jednotek zapojených do sítě a k zastavení rozesílání zpráv na její adresu.

Aktér klient je zde samotný systém. Systém zabezpečuje příjem a odesílání zpráv v rámci sítě. UC012 nejlépe shrnuje jeho scénář, na který se můžeme podívat níže.

1. Systém vybere první zprávu z fronty na odeslání.
2. Systém vytvoří dočasnou kopii seznamu všech sousedů. Z tohoto seznamu se budou odstraňovat jednotky, které tuto zprávu nedostanou.
3. Systém postupně prochází seznam retranslačních bodů.
4. Pokud najde retranslační bod v seznamu příjemců, tento bod z něj odstraní.
5. Systém postupně odešle tuto zprávu příjemcům v dočasném seznamu.
6. Systém smaže dočasný seznam příjemců.

Algoritmus popsaný výše zabezpečuje, že se zpráva nebude rozesílat záplavově, ale pouze těm jednotkám, které tuto zprávu ještě neobdržely. Tímto se značně sníží vytížení sítě.

Na algoritmu retranslace se podílí i UC011, který zachycuje příjem jedné příchozí zprávy. I tady nejlépe celý proces zachytí scénář:

1. Systém přijme zprávu
2. Systém zprávu zařadí do fronty zpráv
3. Systém vybere první zprávu z fronty
4. Systém zjistí typ zprávy. Pokud je to administrační zpráva, pokračuje se příslušným scénářem.
5. Pokud to je obyčejná zpráva, pokračuje se tímto scénářem.
6. Systém se podívá do seznamu retranslačních bodů zprávy.
7. Pokud je najde v seznamu, tuto zprávu zahodí.
8. Systém přidá svoje ID do seznamu retranslačních bodů.
9. Pokud je zařízení v režimu Zobrazování, pokračuje se alternativním scénářem.
10. Systém přemístí zprávu do fronty čekajících zpráv na odeslání.

Zde je obsažena další část algoritmu retranslace a snížení počtu zpráv, které je nutné odeslat přes síť. Jednotka zahazuje zprávy, které se přes ni už jednou šířily, tím se zamezí efektu, kdy by se jedna zpráva mohla „odrazit od konce sítě“ a začala by se vracet přes jednotlivé retranslační body zpátky. Dále je pak potřeba uvést ještě scénář pro příjem administrační zprávy.

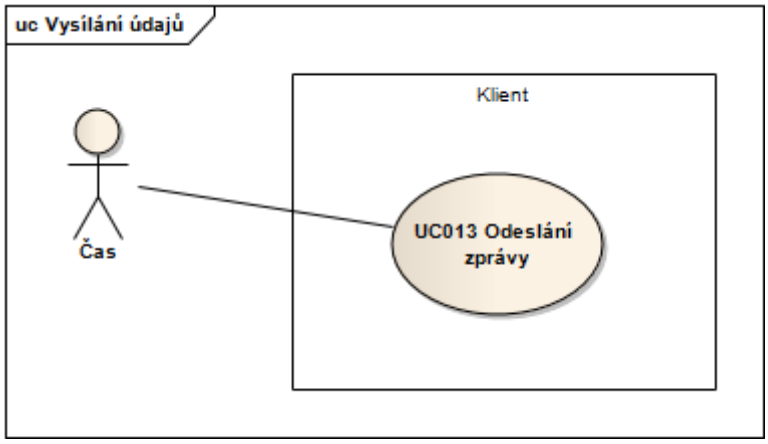
1. Systém zjistí, zda je příjemcem této zprávy.
2. Pokud je příjemcem této zprávy, přečte údaje o režimech.
3. Systém aktivuje nebo deaktivuje příslušné režimy.
4. Pokud není příjemcem této zprávy, pokračuje se základním scénářem v bodě 6.

K UC011 se ještě váže UC016, který zachycuje uložení a zobrazení údajů z okolních jednotek. Opět zde uvedu scénář, podle kterého se postupuje v případě, že je zařízení v režimu zobrazování.

1. Pokud neexistuje seznam prohlížených zařízení, systém vytvoří nový seznam.
2. Systém projde seznam prohlížených zařízení.
    a. Pokud zařízení ještě není v seznamu, systém vytvoří nový záznam.
    b. Systém vloží tento záznam do seznamu.
3. Pokud zařízení v seznamu již je:
    a. Systém zjistí časové razítko údajů.
    b. Pokud má zpráva novější časové razítko, tak systém aktualizuje údaje.
4. Jinak systém zprávu ignoruje.

3.2.2 Vysílání údajů

Dalším režimem jednotky je aktivní vysílání svých údajů do sítě. Tento režim může a nemusí být zapnutý. Jednotka tedy v případě, že režim zapnutý není, nevysílá aktivně své údaje ze senzorů, pouze zobrazuje své údaje a údaje z ostatních jednotek. Jestliže je režim zapnutý, tak jednotka naopak aktivně rozesílá údaje ze svých senzorů do sítě tak, aby ostatní jednotky mohly zobrazovat její údaje. Na tento velice jednoduchý diagram případu užití se můžeme podívat na obrázku Obrázek 4 níže.



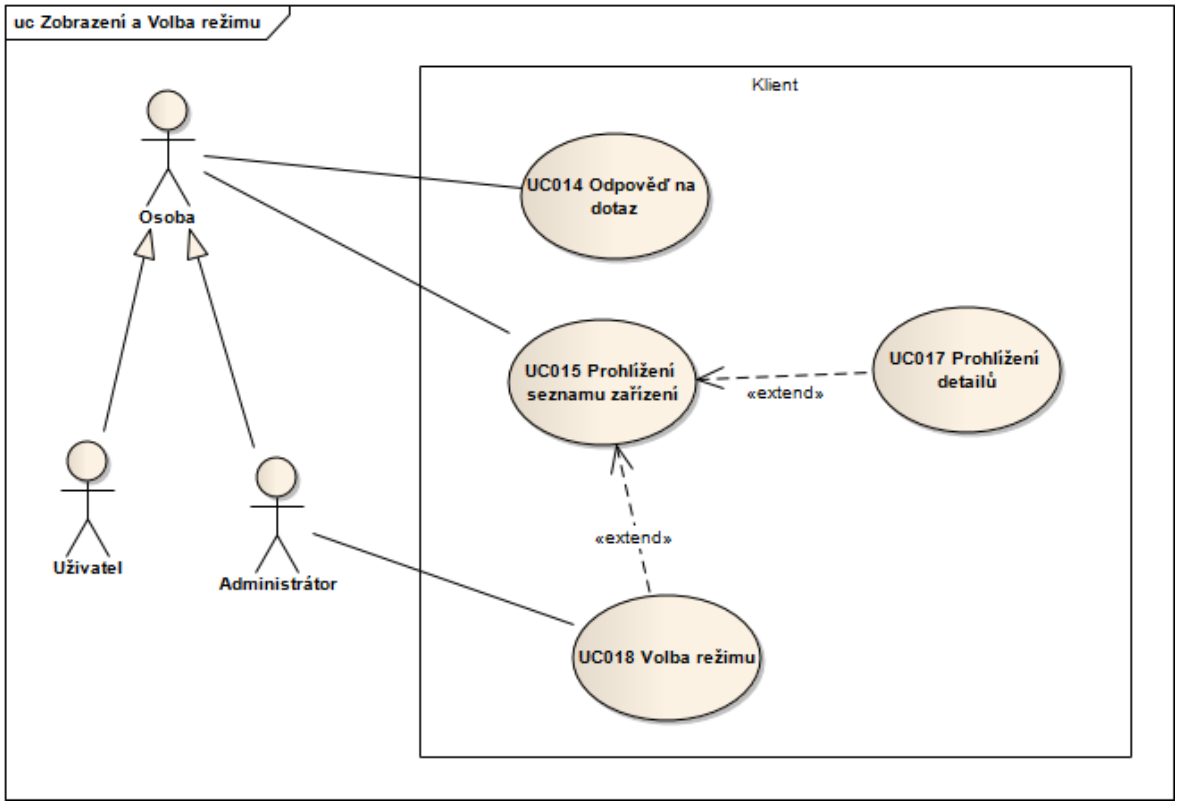
Obrázek 4 - Model případu užití vysílání údajů

Pro upřesnění toho, jak režim funguje, uvedu scénář pro UC013.

1. Systém zjistí aktuální data ze senzorů.
2. Systém vytvoří zprávu pro odeslání.
3. Systém zapíše svoji adresu do seznamu retranslačních bodů.
4. Systém zařadí zprávu do fronty zpráv, čekajících na odeslání.

3.2.3 Zobrazení a administrátorský zásah

Poslední skupinou případů užití je zobrazování údajů a administrace jednotek. Zobrazování slouží, jak již název vypovídá, k zobrazování údajů ze senzorů jednotky, tak údajů ostatních jednotek, které přišly retranslací. Diagram těchto případů užití si můžeme prohlédnout na obrázku Obrázek 5.



Obrázek 5 - Případy užití zobrazování a administrátorského zásahu

Samotné zobrazování probíhá ve dvou úrovních. První úrovní, která zároveň slouží i pro administrátorský zásah, je prohlížení seznamu zařízení. V tomto seznamu jsou uvedeny všechny jednotky, které jsou aktuálně připojené do sítě. Respektive je to seznam jednotek, které buď aktivně vysílají své údaje, nebo se podílejí na retranslaci, případně ostatní klienti ještě nerozhodli o tom, že se tato jednotka odpojila ze sítě. Tuto problematiku řeší UC015. Pro upřesnění uvedu scénář, kterým se řídí prohlížení seznamu zařízení.

1. Systém projde seznam okolních zařízení.
2. Pro každé zařízení vytvoří položku zobrazovaného seznamu obsahující ID zařízení.
3. Systém zobrazí seznam zařízení uživateli.

Prohlížení detailů o zařízení, tedy údajů z jeho senzorů, řeší UC017, který rozšiřuje případ užití UC015. Opět pro přesnější představu o funkci tohoto případu uvedu jeho scénář.

1. Uživatel vybere ze seznamu zařízení to, které ho zajímá.
2. Systém načte aktuální údaje o zařízení.
3. Systém zobrazí novou obrazovku s detaily o zařízení.

Při zobrazování údajů bude jednotka zobrazovat údaje z daného zařízení, které jsou poslední označené jako aktuální. Byly tedy přijaty v poslední zpracované zprávě. Údaje jsou aktualizovány pokaždé, když je zpracována další příchozí zpráva, která obsahuje údaje o právě zobrazovaném zařízení.

Dalším případem užití je administrátorský zásah. Administrátor, respektive jednotka, která má administrátorská práva, má možnost zasílat konkrétním zařízením administrační zprávy. Administrátor má možnost ze seznamu zařízení vybrat jedno konkrétní a tomu pak navolit režimy, které má daná jednotka zapnout nebo vypnout. Na základě tohoto výběru je pak sestavena administrační zpráva, která je zařazena do fronty odchozích zpráv. Zpráva je pak odeslána retranslací jako každá jiná. Z toho vyplývá, že není odeslána přímo danému zařízení, ale musí projít retranslací. Změna režimů se tedy v síti projeví až za nějaký čas.

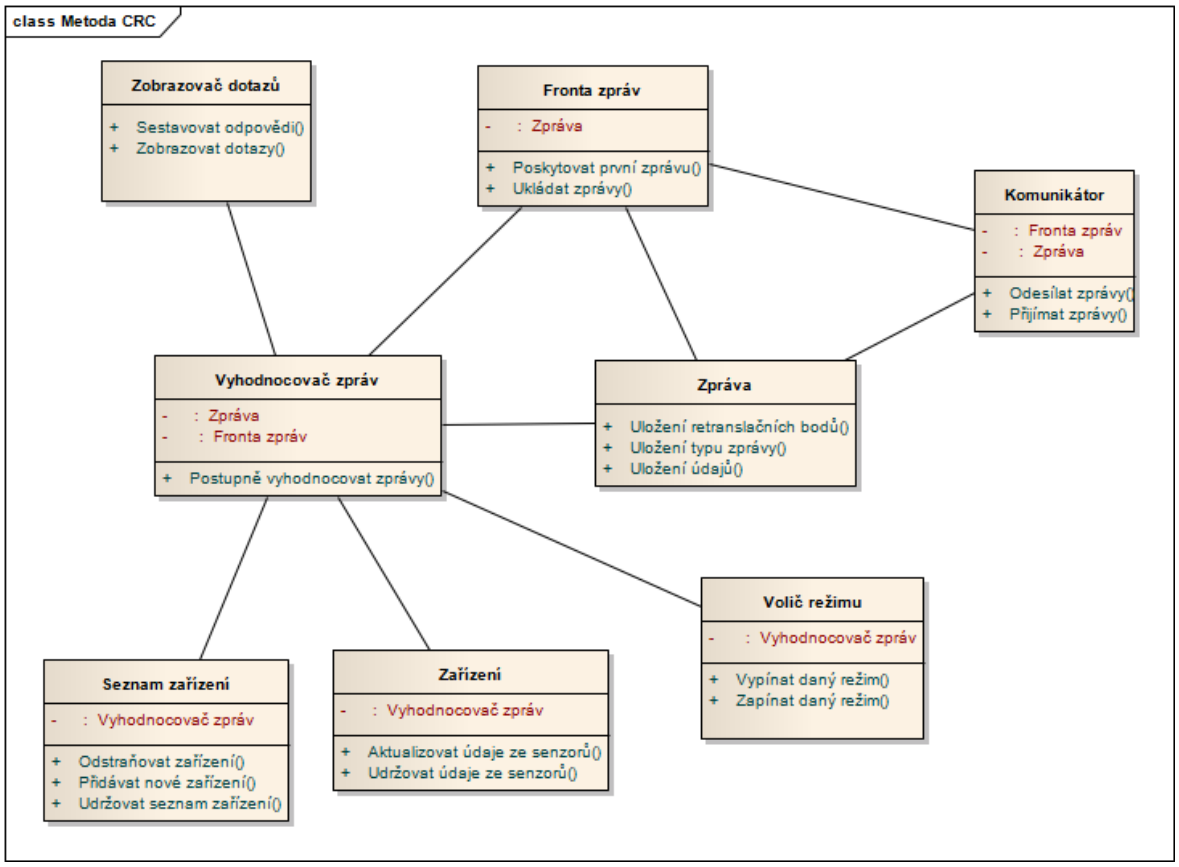
Posledním případem užití je UC014, který se váže k dispečerskému zásahu. Samotný dispečerský zásah bude popsán níže. Zde se budu věnovat pouze jeho části, která se zobrazuje. V případě přijmutí dispečerské zprávy je uživateli dané jednotky zobrazen text zprávy, načež uživatel může vybrat ze dvou možností odpovědi, a to buď možnost ano, nebo možnost ne. Po stisku vybraného tlačítka je vytvořena zpráva, která se zařadí do fronty odchozích zpráv a je stejně jako ostatní zprávy rozeslána přes síť retranslačním mechanizmem.

3.2.4 Dispečerský zásah.

Klient bude umožňovat dispečerský zásah. Tento zásah spočívá v tom, že síťové rozhraní klienta bude umožňovat připojení i jiných zařízení než klientů s operačním systémem Android. Dispečer bude mít možnost do sítě vyslat konkrétnímu klientovi zprávu, která se mu zobrazí na displeji jednotky. Uživatel následně bude mít možnost odpovědět na tuto zprávu stisknutím tlačítka s předdefinovanou odpovědí. Dispečerské zařízení by ideálně mělo být připojeno do sítě ostatních klientů stejným způsobem jako ostatní jednotky, a to z toho důvodu, aby se daly používat stejné zprávy jak pro rozesílání údajů mezi klienty, tak právě pro dispečerský zásah. Dispečerské zprávy se budou sítí šířit stejným způsobem jako ostatní zprávy a proto i dispečerské zařízení by se, v případě, že je připojené do sítě, mělo podílet na retranslaci zpráv.

3.3 Analytické třídy

Z modelu požadavků a z modelů případů užití jsem pomocí metody CRC štítků odvodil model analytických tříd. Tento model byl konzultován s vedoucím práce a upraven do konečné podoby. Diagram těchto tříd si můžeme prohlédnout na obrázku Obrázek 6 níže.



Obrázek 6 - Diagram analytických tříd metodou CRC štítků

Základem veškeré komunikace je třída Zpráva. Tato třída je zodpovědná za uložení všech potřebných údajů, které se musí přenášet sítí mezi jednotlivými klienty. Přes síť v jedné zprávě potřebujeme přenášet zejména seznam retranslačních bodů, který je využíván při zjišťování adresátů při odesílání zprávy. Dále je celkem zřejmé, že budeme muset rozlišovat alespoň tři druhy zpráv, proto tento údaj musíme také přenášet. Poslední zodpovědností Zprávy je přenášení údajů. Jednotlivé údaje se budou lišit podle typu zprávy.

Třída Vyhodnocovač zpráv má za úkol zpracovávat postupně jednotlivé zprávy. Po přečtení a vyhodnocení jedné zprávy má tato třída ještě jednu další důležitou zodpovědnost. Tou je řízení daného klienta. Jde zejména o sestavování seznamu dostupných zařízení (jednotek) a jeho aktualizaci, předávání instrukcí v případě, že se má zapnout nebo vypnout určitý režim klienta. Poslední zodpovědností této třídy je řazení zpráv do fronty odchozích zpráv a to jak aktuálně zpracovávaných, tak i zpráv, které vytvořil sám klient (zpráva o změně stavu senzorů).

Fronta zpráv má poměrně jednoduchou zodpovědnost. Udržuje frontu odchozích a příchozích zpráv a poskytuje ostatním třídám, které pracují se zprávami, přístup k první zprávě v obou frontách.

Zobrazovač dotazů je třída, která pouze reaguje na dispečerské zprávy v případě, že ji na to upozorní Vyhodnocovač zpráv. Tímto její zodpovědnosti končí.

Podobnou funkci má Volič režimu. Jeho zodpovědností je, že bude zapínat nebo vypínat jednotlivé režimy podle toho, jak bude Vyhodnocovač zpráv předávat instrukce k zapnutí nebo vypnutí jednotlivého režimu. Předpokladem ovšem je, že administrační zprávy budou spíše minoritní a jednotlivé režimy bude potřeba zapínat či vypínat ojediněle.

Třída Zařízení monitoruje senzory dané jednotky a reaguje na změnu jejich stavu. Tuto změnu zapisuje a po uplynutí předem daného časového intervalu sestaví z nejaktuálnějších údajů zprávu a předá ji ke zpracování Vyhodnocovači zpráv.

Poslední třídou je Komunikátor. Ten je zodpovědný za samotnou síťovou komunikaci s ostatními jednotkami, tedy vyhledávání okolních klientů, rozesílání a přijímání zpráv, stejně jako sestavování seznamu adresátů, kterým bude daná zpráva odeslána.

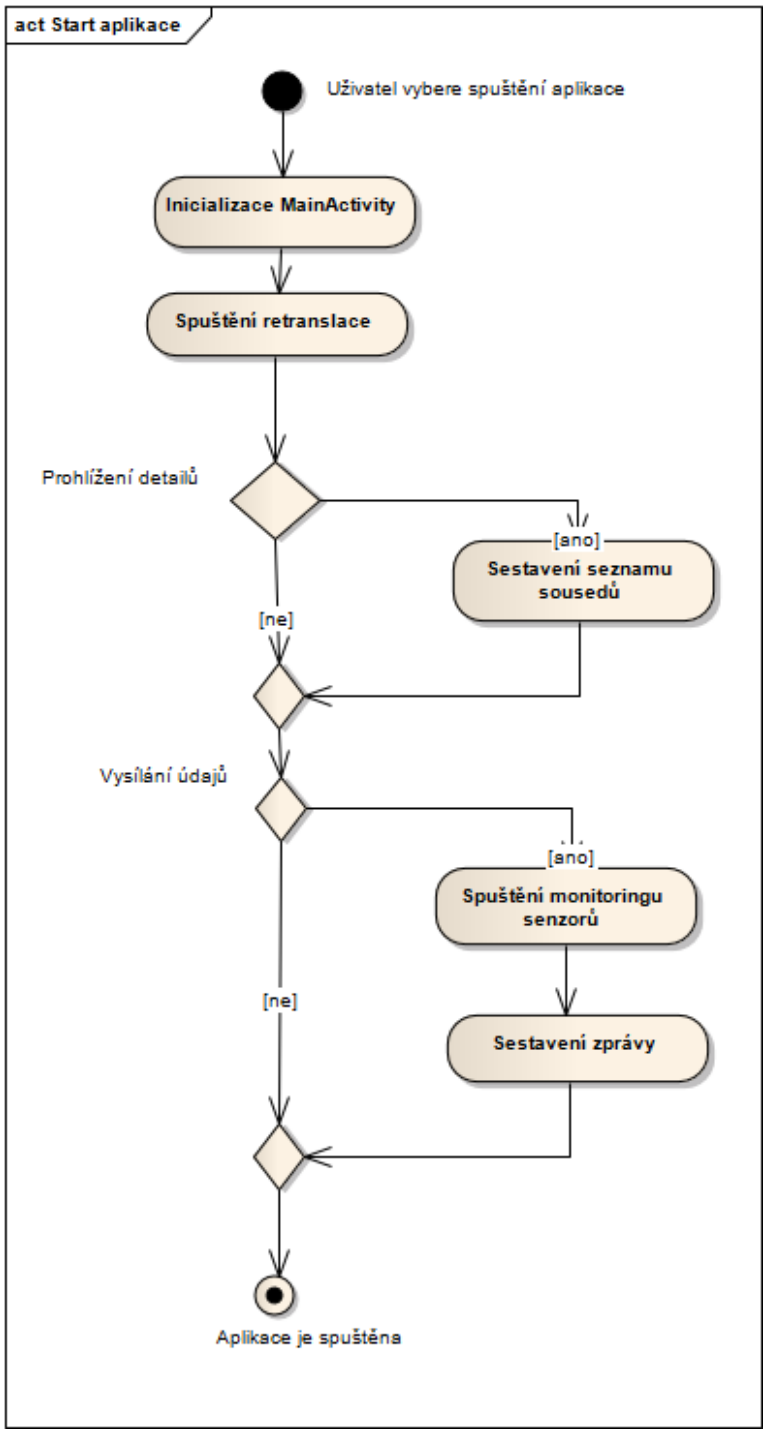
3.4 Diagramy aktivit

Uvedu zde několik diagramů aktivit. Diagramy by lépe měly vystihnout, jak bude vlastně aplikace v budoucnu fungovat a které kroky bude potřeba implementovat.

3.4.1 Spuštění aplikace

Na obrázku Obrázek 7 si můžeme prohlédnout diagram aktivity spouštění aplikace.

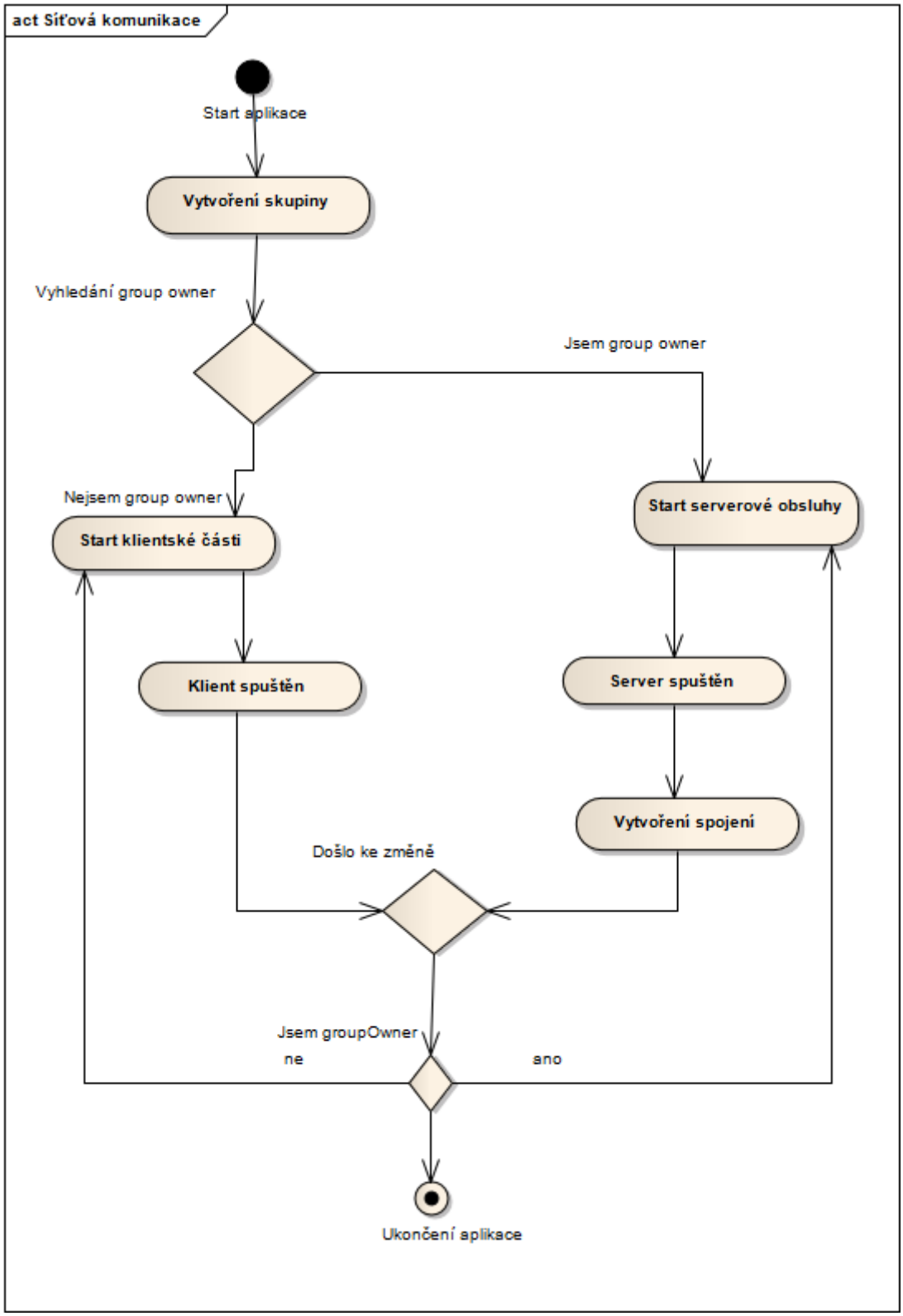
Můžeme vidět, že počet kroků nutných ke spuštění aplikace je rozdílný podle toho, kolik režimů se musí spustit. Přičemž vycházím z předpokladu, že retranslace, jakožto základní funkčnost klienta, se bude spouštět defaultně při každém startu aplikace a teprve až administrátorským zásahem v průběhu času bude možné retranslaci vypnout.



Obrázek 7 - Diagram aktivity spuštění aplikace

3.4.2 Síťová komunikace

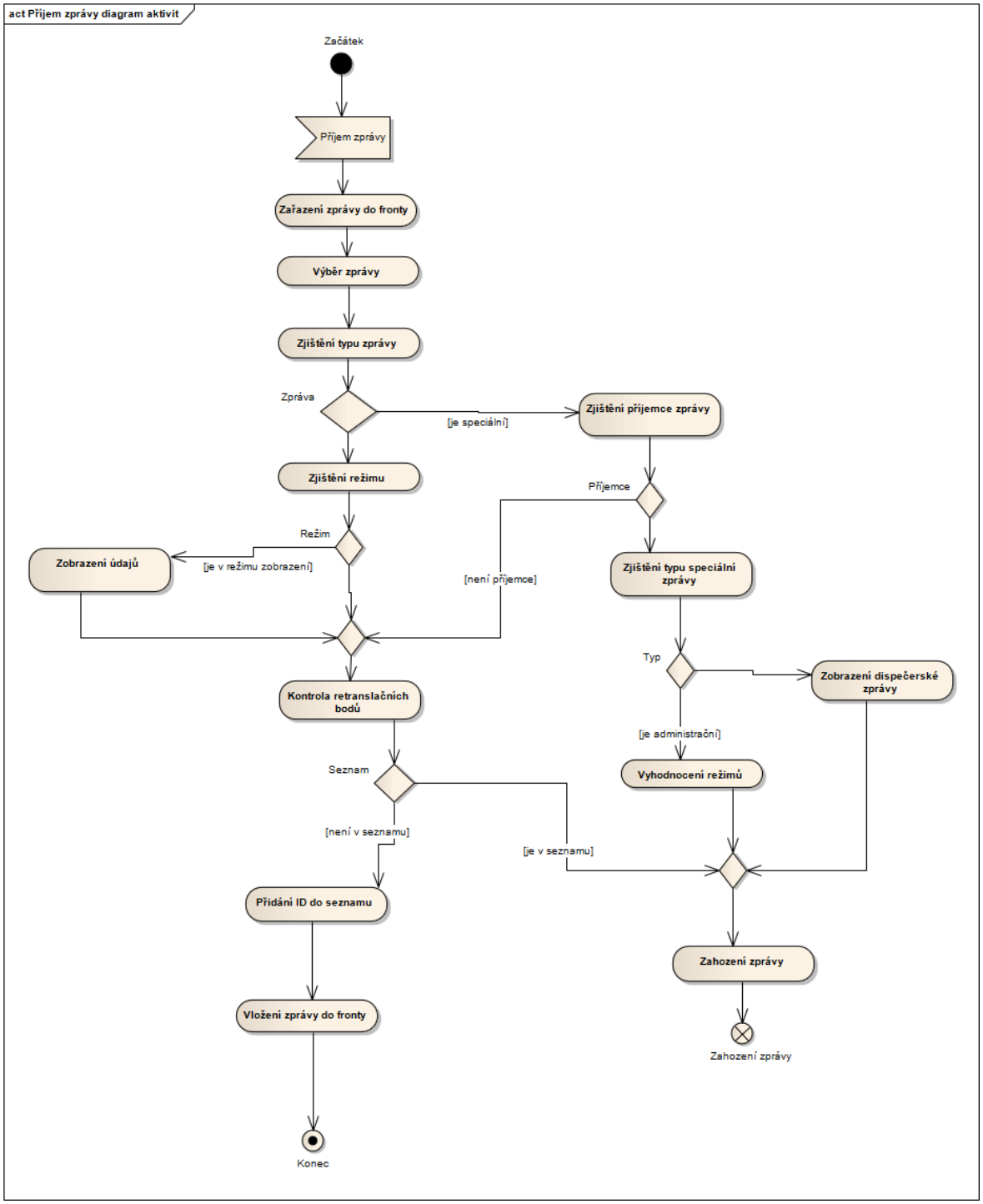
Na obrázku Obrázek 8 si můžeme prohlédnout diagram síťové komunikace. Tento diagram zachycuje představu komunikace na síti, z druhé iterace vývoje, jelikož zde už počítáme s klientskou a serverovou částí, která se právě ve druhé iteraci objevila. Tehdy jsem zjistil, jak bude popsáno níže, že knihovna Wi-Fi Direct neumožňuje náhodné propojování jednotek, jak bylo zamýšleno dříve, ale že se bude muset klient předělat na typ klient – server.



Obrázek 8 - Diagram startu síťové části klienta

3.4.3 Příjem zprávy

Posledním diagramem aktivity, který bych zde chtěl uvést, je diagram, který zobrazuje algoritmus zpracování příchozí zprávy. Ten si můžeme prohlédnout níže na obrázku Obrázek 9.



Obrázek 9 - Diagram aktivity, algoritmus přijmutí a vyhodnocení zprávy

Na tomto diagramu si můžeme v grafické podobě prohlédnout algoritmus zpracování zprávy. Přesněji řečeno algoritmus retranslace, který zabraňuje rozesílání příliš velkého

počtu duplicitních zpráv. Jak je patrné, zprávy se zahazují za určitých podmínek, zejména v případě, že je zařízení již uvedeno v seznamu retranslačních bodů nebo je adresátem jednoho ze servisních druhů zprávy (administrační a dispečerská). Tento algoritmus zůstal platný po celou dobu vývoje, jelikož i přes obtíže s technologií, které jsou popsány níže, se tento způsob zpracování nezměnil.

3.5 Jak byla analýza ovlivněna implementací.

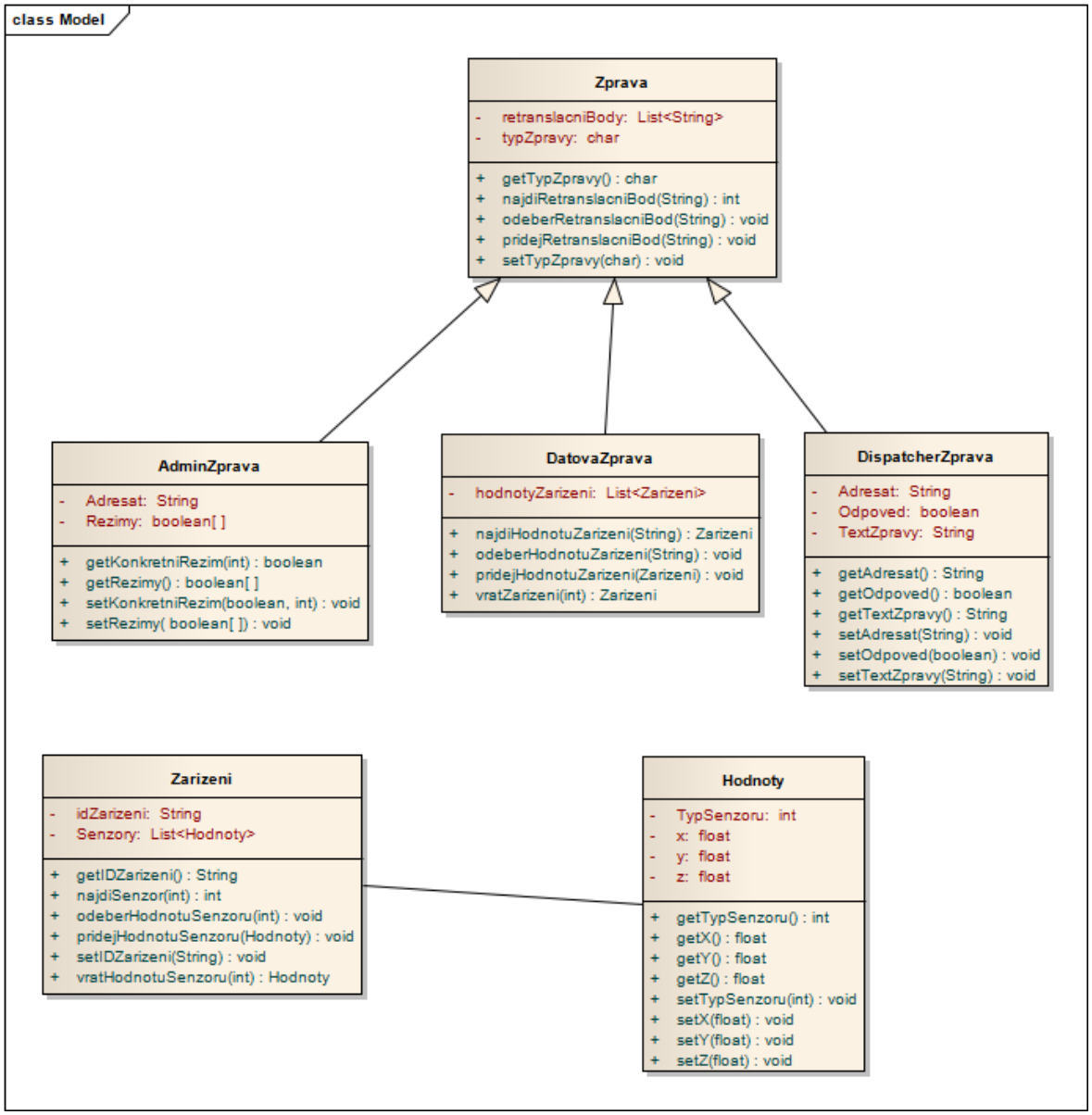
V průběhu implementace se ukázalo, že představa distribuované sítě tak, jak je popsána v zadání, je pomocí dostupné technologie nerealizovatelná. Síť nemůže být sestavena čistě náhodně, musí být vždy typu hvězda. Tím se vyskytlo omezení zejména na její dosah. Avšak také se ukázalo, že algoritmus retranslace je použitelný i s omezeními technologií a tudíž ho nebylo potřeba měnit. Stejně se ukázalo, že i ostatní scénáře je možné aplikovat i přes omezení technologií (omezení daná technologií budou popsána v kapitole Implementace) a nebylo je potřeba měnit. Zásadní změny tedy bylo potřeba provést zejména v implementaci.

4 Návrh

4.1 Návrhové třídy

4.1.1 Třídy stereotypu Model

Na obrázku Obrázek 10 je vidět model návrhových tříd. Z analýzy vyplynulo, že bude potřeba tří druhů zpráv pro rozesílání datových zpráv, administračních pokynů a dispečerských zpráv. Vzhledem k tomu, že zprávy mají jednotící prvek, a to seznam retranslačních bodů, vznikl jeden společný předek všem zprávám, třída `Zprava`. Od ní dědí ostatní druhy zpráv. Jak vyplývá z diagramu na obrázku Obrázek 10, jednotlivé třídy si přidávají nezbytné atributy pro předání potřebných údajů.



Obrázek 10 - Diagram návrhových tříd – Model

Administrační zpráva předává seznam režimů, které se mají zapnout a které vypnout. K tomu stačí obyčejné pole typu `boolean`, jelikož režim může být zapnutý nebo vypnutý. Jediná další informace, kterou administrační zpráva potřebuje, je cílové zařízení, které je identifikováno IP adresou uloženou v proměnné typu `String`, jejíž formát je v klasické tečkové notaci.

Datová zpráva přenáší seznam zařízení a jejich hodnoty senzorů. Tento typ zprávy by měl být nejčastějším přenášeným údajem v síti.

Posledním typem zprávy je dispečerská zpráva. Tato zpráva je stejně jako administrační zpráva adresována konkrétní jednotce, proto si přenáší informaci o IP adrese cíle. Příznak Odpověď značí, zda jde o odpověď na dotaz dispečera. Nabývá hodnoty `false`, pokud se jedná o dotaz, a `true`, pokud se jedná o odpověď klienta na příchozí dispečerský dotaz.

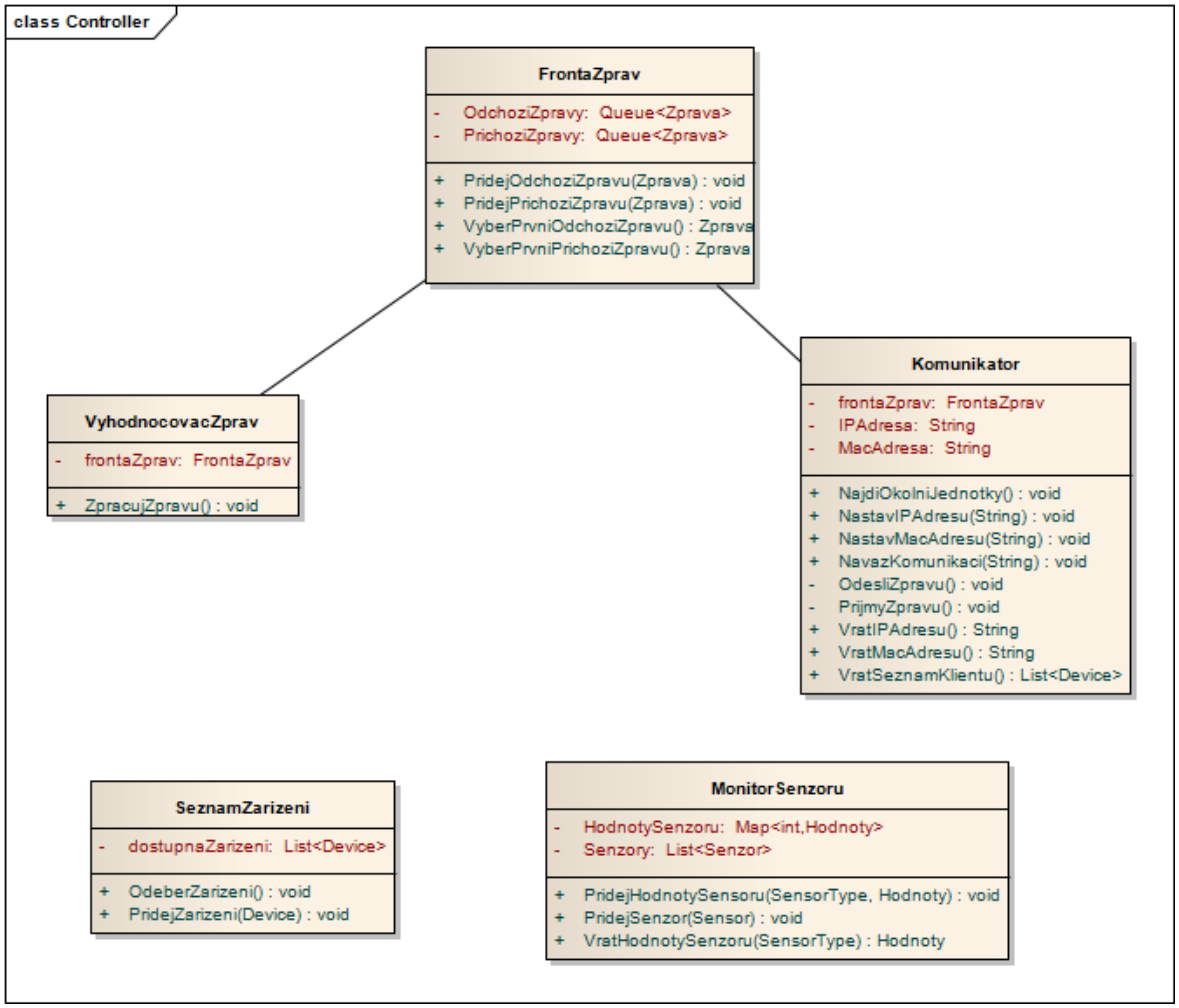
Text zprávy obsahuje samotný dotaz v případě, že se jedná o dotaz na klienta, nebo textovou reprezentaci uživatelem zvolené možnosti (typicky Ano nebo Ne), v případě, že se jedná o klientem vytvořenou odpověď.

Přenášené údaje o senzorech jsou reprezentovány třídou `Hodnoty`. Tato třída reprezentuje jeden senzor jednotky a ukládá údaje o něm. Hodnoty jsou reprezentovány třemi hodnotami `float`, jelikož hodnoty senzorů jsou uloženy v poli datového typu `float` a záleží na typu senzorů, kolik hodnot je v tomto poli uloženo. Příkladem budiž akcelerometr, který používá tři složky, jednu pro každou ze směrových os. Tři složky byly stanoveny jako dostačující, protože žádný ze senzorů neposkytuje více než 3 údaje.

Třída `Zarizeni` představuje jednu jednotku a údaje z jejích senzorů. Každá jednotka je identifikována svojí IP adresou, která je zde uložená v `idZarizeni`. Dále je zde seznam hodnot ze senzorů daného zařízení. Tento seznam je proměnlivý, jelikož každé zařízení může mít rozdílnou sadu senzorů. Jednotlivé senzory je možné vyhledávat podle jejich identifikátorů, což je proměnná typu `integer`. Tato identifikace vychází ze samotné platformy operačního systému Android, jelikož zde jsou jednotlivé senzory identifikovány výčtem typu `integer`.

4.1.2 Třídy stereotypu Controller

V modelu tříd stereotypu Controller z analýzy zůstaly třídy: `FrontaZprav`, `VyhodnocovacZprav`, `SeznamZarizeni` a `Komunikator`. Zůstaly i jejich zodpovědnosti, jelikož zde se mnoho změn udělat nemuselo. Na diagram tříd kontrolérů se můžeme podívat na obrázku Obrázek 11 níže.



Obrázek 11 - Model návrhových tříd – Controller

Třída `FrontaZprav` udržuje dvě fronty. Jednu pro odchozí zprávy a druhou pro příchozí. Stejně tak poskytuje přístup k frontám, konkrétně k prvním zprávám v obou frontách. Taktéž umožňuje vložení nových zpráv na konec fronty.

`VyhodnocovacZprav` má stále stejnou funkci. Postupně vybírá zprávy z fronty příchozích zpráv a zpracovává je. Algoritmus zpracování zpráv nebylo potřeba měnit, jelikož v průběhu návrhu se ukázalo, že i přes některé nutné změny v konceptu klienta se algoritmy, jak již bylo řečeno i v analýze, nezměnily.

Stejně tak se v návrhu příliš nezměnila funkce třídy komunikátoru. Jeho úkolem je stále vyhledávat nové klienty a navazovat s nimi síťové spojení. A samozřejmě rozesílat a přijímat zprávy od ostatních klientů.

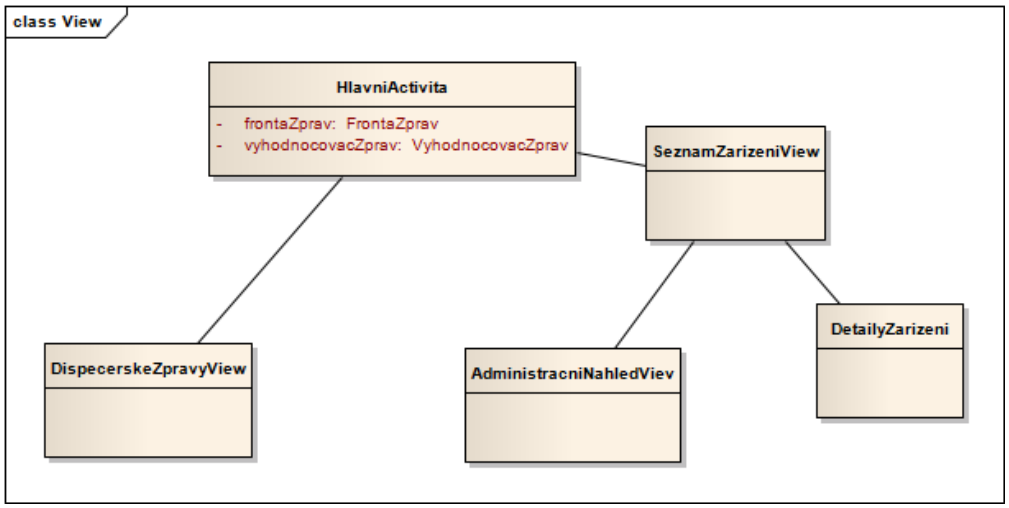
`SeznamZarizeni` se stará o udržování co nejvíce aktuálního seznamu okolních zařízení a připravování podkladů pro jejich zobrazování. Jeho úkolem je také udržovat seznam hodnot ze senzorů daného zařízení, tedy aktualizovat je pokaždé, když jsou k dispozici novější údaje. Tyto údaje pak transformuje do srozumitelné podoby pro zobrazení.

Poslední třídou je `MonitorSenzoru`. Jeho odpovědnost je téměř shodná s třídou `Zarizeni` z analýzy. Tato třída monitoruje senzory svého zařízení a ukládá údaje z nich do podoby, ve které jsou pak přenášeny datovou zprávou. Jde tedy o uložení jednotlivých hodnot ze senzorů do instance třídy `Hodnoty` a následně sestavení a zapouzdření těchto hodnot z jednotlivých senzorů do třídy `Zarizeni`.

Jedinou třídou, která se ukázala jako nová oproti analýze v návrhu, je `Volič režimu`. Tato třída měla za úkol zapínat a vypínat dané režimy v případě, že přišla administrační zpráva s pokynem přepnutí režimů. Tato funkce nakonec připadla samotné hlavní aktivitě, která na to má poměrně efektivní mechanizmus přidávání a odebírání fragmentů v průběhu běhu aplikace. Tento mechanizmus bude podrobně rozebrán v kapitole Implementace

4.1.3 Třídy stereotypu View

Model návrhových tříd byl nejvíce ovlivněn použitou technologií. Operační systém Android totiž nabízí možnosti fragmentů, což jsou samostatně fungující „obrazovky“ aplikace, které se mohou, v závislosti na velikosti displeje zařízení, zobrazovat najednou nebo jeden po druhém. Jednotlivé třídy stereotypu view si můžeme prohlédnout na obrázku Obrázek 12.



Obrázek 12 - Model návrhových tříd – View

Základní třídou pro model View je `HlavniActivita`. Tato třída působí tak trochu jako kontrolér pro částečné náhledy na jednotlivé funkčnosti klienta. Jejím úkolem je spravovat jednotlivé náhledy. V podstatě převzala funkčnost třídy Volič režimu z analytických tříd, jelikož zobrazováním nebo vypínáním jednotlivých náhledů může řídit jednotlivé režimy klienta. Sama o sobě nic nezobrazuje.

O funkčnost dispečerského zásahu se stará třída `DispecerskeZpravyView`. Zobrazuje dotazy od dispečera a nabízí uživateli možnost výběru odpovědi nastavením předem definovaných odpovědí pomocí tlačítek. Jak již bylo řečeno v analýze, uživatel bude mít většinou možnost odpovídat možnostmi Ano a Ne stisknutím příslušného tlačítka. View pak zařadí zprávu s odpovědí do fronty odchozích zpráv.

`SeznamZarizeniView` je jednoduchý seznam zařízení, zobrazených pomocí svých identifikátorů. Seznam je průběžně aktualizován v závislosti na tom, jak kvalitní je připojení mezi jednotkami a jak často dochází ke ztrátě spojení mezi jednotlivými jednotkami sítě. Tato třída reaguje na dvě události. První z událostí je uživatelský výběr zařízení, pro které se mají zobrazit detaily. Uživatel provádí výběr krátkým stisknutím položky vybraného zařízení. V tomto případě se zobrazí nový náhled detailů daného zařízení, kde se zobrazuje seznam dostupných senzorů a posledních hodnot, které byly označeny jako aktuální. Tento náhled má na starosti třída `DetailyZarizeni`. Seznamem senzorů je možné volně procházet, avšak seznam již nereaguje na další výběr senzoru.

Druhou událostí, na kterou reaguje seznam zařízení je, v případě, že jednotka má administrační práva, kontextová nabídka při výběru zvoleného zařízení. Kontextová nabídka obsahuje seznam režimů, které se mohou zapnout nebo vypnout zaškrtnutím zaškrtávacího políčka. V okamžiku, kdy je výběr režimů potvrzen, je sestavena administrační zpráva a zařazena do fronty odchozích zpráv.

4.2 Jak byl návrh ovlivněn použitou technologií

V průběhu návrhu jsem se začínal setkávat s omezeními, která začala měnit zamýšlený směr směřování vývoje. Původně jsem zamýšlel stavět aplikaci klienta klasickým MVC návrhovým vzorem, jenže se ukázalo, že operační systém Android, ač se snaží tento návrhový vzor dodržovat, občas spojuje stereotyp view se stereotypem controller. Toto se promítlo hlavně v tom, že řízení zapínání a vypínání režimů se dá udělat poměrně efektivní, ale relativně těžkopádnou technikou přidáváním a odebíráním jednotlivých fragmentů - obrazovek. Proto v části modelu stereotypu view jsou pro každý z režimů zařízení samostatné třídy typu view. Hlavní obrazovka aplikace slouží jen jako kontrolér, který kontroluje přidávání a odebírání jednotlivých obrazovek. Přesný mechanizmus fungování fragmentů bude popsán v kapitole Implementace. Každá z tříd view pak slouží jako vstupní obrazovka pro danou funkčnost klienta, i když některé používají stejné kontroléry, především frontu zpráv, která je stejná pro celou konkrétní instanci klienta.

Jediná funkčnost, která nemá svůj view, je retranslace. Je to z důvodu, že retranslace nic nezobrazuje, ale je možné ji vypnout. Proto i pro ni bylo zvoleno fungování přes fragmenty, aby v případě, že by bylo potřeba vypnout retranslaci zpráv, bylo možné tuto funkčnost vypnout stejným mechanizmem, jako se zapínají a vypínají ostatní režimy klienta.

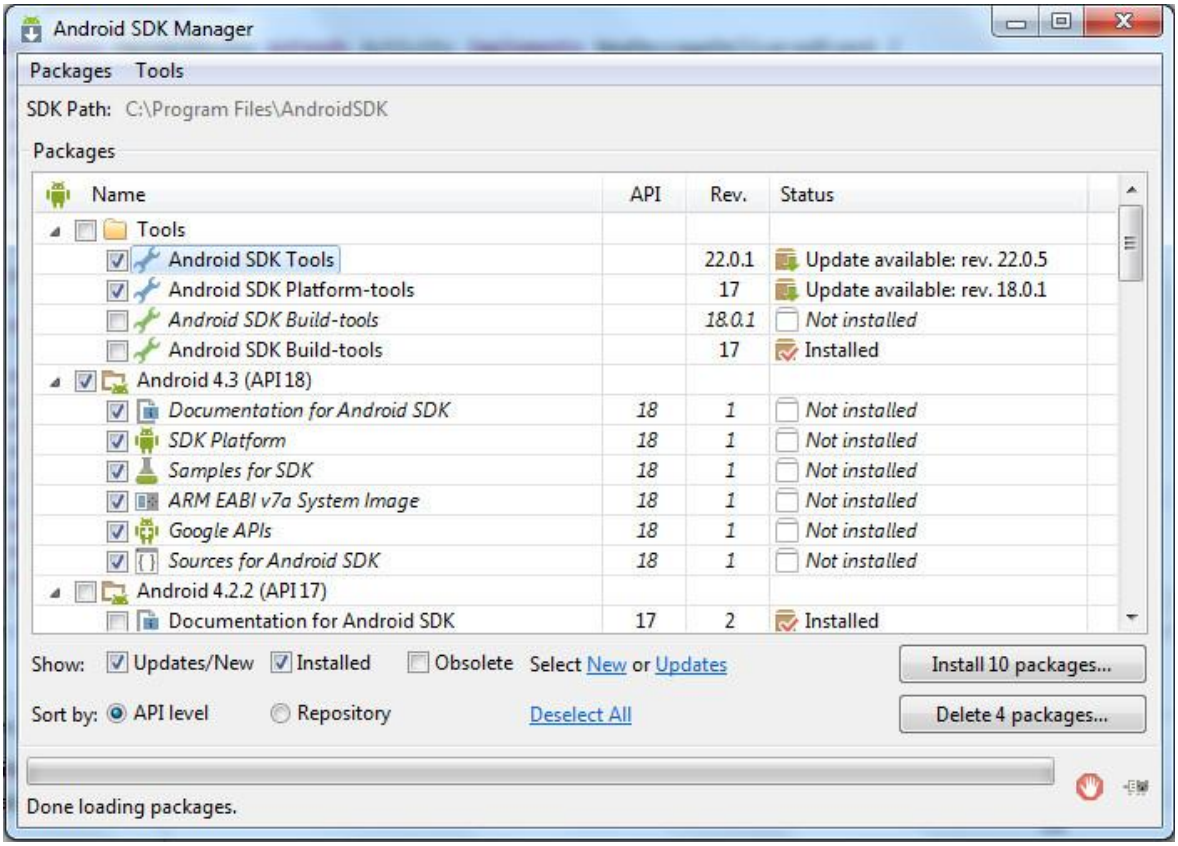
5 Realizace

5.1 Použité nástroje

Pro realizaci vývoje klienta jsem zvolil vývojové prostředí Eclipse ve verzi Juno pod operačním systémem Microsoft Windows 7. Prostředí jsem zvolil záměrně, jelikož je

společností Google prosazováno jako hlavní vývojové prostředí pro operační Android. Důležitým faktorem pro výběr je také to, že pro Eclipse existuje oficiální zásuvný modul, který do tohoto prostředí integruje nezbytné funkčnosti pro pohodlné vyvíjení aplikací. Jde zejména o přidání podpory pro vytváření projektů aplikací, které už mají přednastavenou adresářovou strukturu a vygenerované některé základní soubory nezbytné pro každou aplikaci na operační systém Android. Vývojové prostředí Eclipse je dostupné zdarma, takže se používá velice často.

Pro vývoj aplikace je potřeba mít nainstalován balíček Android SDK, který obsahuje nezbytné nástroje a knihovny pro vývoj aplikací. Prostředí Eclipse s nainstalovaným zásuvným modulem pro Android umožňuje velice pohodlnou práci s balíčkem SDK. Zásuvný modul také přidá do prostředí pohodlnější spouštění Android SDK Manažera, což je nástroj pro spravování knihoven. Používá se zejména pro stahování nových knihoven pro nové verze operačního systému Android. Tento nástroj se v průběhu vývoje ukázal jako velice šikovný. Vývoj aplikace jsem totiž začínal na verzi operačního systému Android 4.2.1, ale dokončoval na poslední vydané verzi, což byla verze 4.3.



Obrázek 13 - Android SDK manager

Na obrázku Obrázek 13 si můžeme prohlédnout okno Android SDK manažera s možností updatu čerstvě vydaného balíčku SDK pro Android 4.3. včetně updatu nástrojů.

Vzhledem k tomu, že emulátor zařízení s operačním systémem Android je velice pomalý a neumožňuje testovat a debugovat určité funkčnosti, bylo potřeba debugovat aplikaci na

konkrétním zařízení. Emulátor totiž neumožňuje testovat senzory, nehledě na to, že se velice špatně emuluje komunikace po bezdrátové síti. Jako testovací hardware jsem proto zvolil Google Nexus 7, což je tablet, který má jak dostatečný počet senzorů, tak připojení k bezdrátové síti. Navíc je to referenční přístroj, takže se na něj průběžně dostávají nejnovější aktualizace operačního systému. Přístroj byl zakoupen s operačním systémem ve verzi 4.2.1 a průběžně aktualizován až na poslední verzi 4.3.

Jako poslední nástroj, který jsem používal pro potřeby práce, je Enterprise Architect ve verzi 7.5. Tento nástroj sloužil k modelování všech diagramů v částech Analýza a Návrh. Také jsem ve značné míře využíval možnost reverzního modelování, tedy tvorby diagramů z již hotového kódu. Tuto funkčnost jsem používal nejvíce pro kapitolu Implementace, kde jsem pomocí ní modeloval diagramy tříd, jelikož bylo možné importovat již hotové zdrojové kódy.

Jako neocenitelnou se tato jeho funkčnost ukázala v průběhu analýzy, kdy jsem měl k dispozici zdrojové kódy k volně dostupné demo aplikaci používající knihovnu Wi-Fi Direct. Pomocí reverzního modelování jsem tak mohl sestavit velice složitý sekvenční diagram celé této demo aplikace a tím aspoň částečně získat přehled o tom, jak vůbec daná knihovna funguje.

5.2 Debugování aplikace

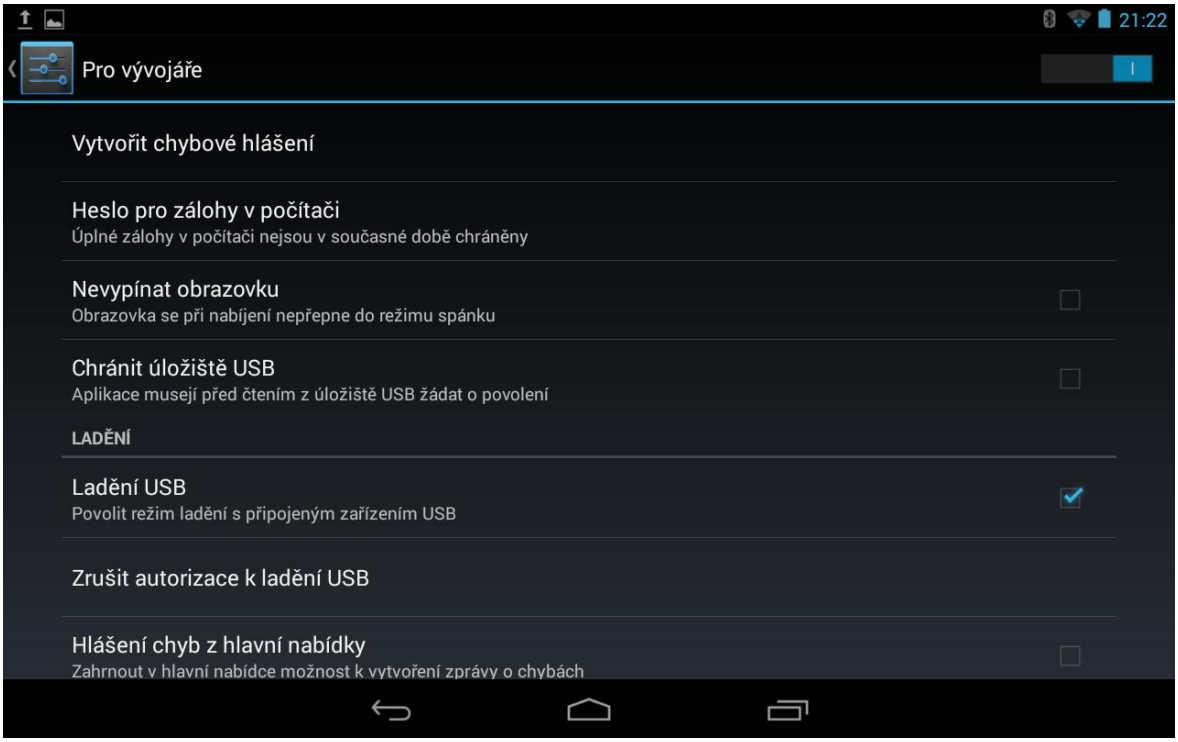
K debugování aplikace na konkrétním zařízení je potřeba nainstalovat poslední nástroj a tím je Android Debug Bridge, který umožňuje připojení zařízení k počítači a následně spuštění aplikace na něm, přičemž je ve vývojovém prostředí možné sledovat běh programu, jako bychom ho debugovali přímo v něm. Samozřejmostí je, že na rozdíl od emulátoru, je debugování na zařízení mnohonásobně rychlejší a tím pádem pohodlnější. Aplikace navíc má přístup k reálným zdrojům zařízení, takže je možné sledovat třeba i komunikaci po síti.

Dalším krokem, který potřebujeme k debugování aplikace přímo na zařízení, je mít ho nastavené ve vývojářském nastavení. To se v případě Androidu verze 4.2. a vyšší provede poklepáním sedmkrát na číslo sestavení. V tomto menu musíme mít zaškrtnutou možnost Ladění. Lépe si to můžeme prohlédnout na obrázku Obrázek 14.

Posledním krokem k debugování aplikace na zařízení je nastavení aplikace na debugování. To se provede vepsáním parametru debuggable do manifestačního XML souboru. Příznak se vepisuje do tagu `application` následujícím způsobem:

```
android:debuggable="true"
```

Tím je vše připravené na debugování aplikace na reálném zařízení.



Obrázek 14 - Nastavení vývojářských nástrojů v operačním systému Android

6 Implementace

Implementace klienta byla složitá, protože se ukázalo, že knihovna Wi-Fi Direct operačního systému Android sice poskytuje funkčnost pro propojování jednotek mezi sebou bez nutnosti access pointu, ale zdaleka nefunguje tak, aby se pomocí ní dala sestavit plnohodnotně fungující distribuovaná síť. Jak bylo řečeno na začátku v kapitole Úvod a Analýza, původně se mělo jednat o distribuovanou síť, kterou by bylo možné sestavovat z klientů, kteří jsou v dosahu alespoň jednoho dalšího klienta již zapojeného v síti. Právě tento bod se ukázal jako nerealizovatelný z důvodu, že pro propojení dvou jednotek přes Wi-Fi Direct je potřeba uživatelský zásah, který je pro potřeby distribuované sítě nevhodný.

Pomocí knihovny Wi-Fi Direct je totiž možné propojit dvě zařízení mezi sebou tak, aby mohla komunikovat obousměrně. V tomto případě se zařízení dohodnou, které z nich bude nahrazovat access point. Komunikace pak probíhá bezproblémově v obou směrech. Jako neřešitelný problém se ovšem ukázalo připojení třetího klienta do této již existující sítě. Jednotka totiž nemá způsob, jak se dozvědět, který z těchto dvou již propojených klientů na sebe vzal roli přípojného bodu. Pokud se pokusíme připojit tuto novou jednotku k zařízení, které není přípojným bodem, dojde k selhání připojení. Jediná možnost je připojit se k zařízení, které na sebe vzalo roli přípojného bodu. Ovšem bez uživatelského zásahu, kdy uživatel sám vybere zařízení, které je již přípojným bodem, nemáme způsob, jak určit, které zařízení na sebe tuto roli vzalo a ke kterému se tedy máme připojovat.

Z toho plyne problém, že jakmile je jednou zařízení propojeno s druhým a je určena role přípojného bodu, nejde se připojit k zařízení, které není přípojným bodem. A opět, dokud není ustanoven přípojný bod pro první dvě zařízení, neexistuje způsob, jak mohou zařízení mezi sebou komunikovat.

Tento problém jsem vyřešil tím, že zařízení, které má nastavená administrátorská práva na sebe, bude vždy brát roli přípojného bodu. V tomto případě každé ze zařízení, které nemá administrátorská práva, je automaticky klientem a pokouší se připojit k serveru, tedy k jednotce s administrátorskými právy.

Drobným omezením je, že síť může být sestavena pouze z klientů, kteří jsou v momentě iniciace sítě v dosahu administrační jednotky – serveru. Vznikne nám tak síť typu hvězda s komunikací server – klient. Komunikaci samotnou zajišťuje technologie Java NIO. Nicméně i přes toto omezení se podařilo docílit alespoň funkce distribuce zpráv mezi všemi jednotkami připojenými do sítě, kdy server zajišťuje celý mechanizmus retranslace, přičemž si sám aktualizuje a zobrazuje údaje o ostatních jednotkách v síti. Klienti pak zasílají a dostávají zprávy od serveru a taktéž si aktualizují údaje o ostatních klientech v síti. Jediným omezením je tedy dosah určený dosahem bezdrátové sítě serveru.

Nyní se podíváme trochu podrobněji na technologie, které jsem použil pro realizaci klienta distribuované sítě.

6.1 Wi-Fi Direct

Wi-Fi Direct je knihovna obsažená v operačním systému Android verze 4.0 a vyšší (API úroveň 14). Tato knihovna umožňuje propojení zařízení podobným mechanizmem jako například Bluetooth, ale spojení je realizováno bezdrátovou sítí, tudíž dosah spojení je několikanásobně vyšší než v případě zmiňovaného Bluetooth. Hlavní třídou v tomto API je WifiP2pManager. Tato třída umožňuje vyhledávat okolní jednotky, sestavovat jejich seznam a navazovat spojení s jednotlivými jednotkami, a to bez použití přípojného bodu. Třída umožňuje spojit vzájemně dvě jednotky pomocí metody connect(). Toto spojení je ovšem použitelné pouze pro dvojici zařízení. Pro potřeby práce jsem tedy musel použít metodu createGroup(), která z dostupných zařízení vytvoří síť typu hvězda, kdy jedna z jednotek přijme roli serveru.

Knihovna dále poskytuje posluchače, pomocí nichž můžeme reagovat na vzniknuvší události jako je třeba dostupný seznam zařízení nebo ztráta spojení či objevení se nové jednotky v dosahu.

6.2 Java NIO

Java NIO je knihovna klasické Javy. Pomocí této knihovny jsme schopni naprogramovat neblokující síťové sokety, které jsou pro tuto práci přímo zásadní. Klasické sokety jsou blokující a tedy program nebo vlákno, ve kterém se čeká na příjem dat, je zablokováno, dokud nejsou k dispozici nějaká data. Pomocí Java NIO jsme tuto blokaci schopni odstranit a vlákno programu, které čeká na data, tedy může naslouchat všem příchozím spojením.

Java NIO dále umožňuje multiplexování příchozích připojení. Všichni klienti se tedy připojují na stejný setkávací port, ale po navázání spojení je komunikace převedena na jiný port a je tedy možné, aby se další klient připojil na stejný port.

Server je pak schopný všechna tato spojení uchovávat a komunikovat tak s každým připojeným klientem.

6.3 Fragmenty

Fragment je svým chováním podobný Activitě. Jedná se o „obrazovku“ aplikace, respektive o její část. Každý fragment musí být součástí nějaké Activity, s jejímž životním cyklem je spjat. Filozofie fragmentů je jednoduchá. Máme-li zařízení s malou obrazovkou, pak je fragment zobrazen jako celá obrazovka daného zařízení. Jedná-li se ovšem o větší obrazovku, na kterou se vejde více fragmentů, pak jsou zobrazeny vedle sebe. Každý z fragmentů představuje samostatnou funkčnost aplikace a může tedy zobrazovat jiná data.

Fragmenty je možné buď definovat napevno pomocí XML dokumentu, ve kterém je uveden design Activity, nebo je možné je přidávat dynamicky za běhu aplikace pomocí FragmentManageru. Právě díky tomuto jsem fragmenty zvolil pro implementaci zapínání režimů, kdy každý fragment představuje jeden režim zařízení.

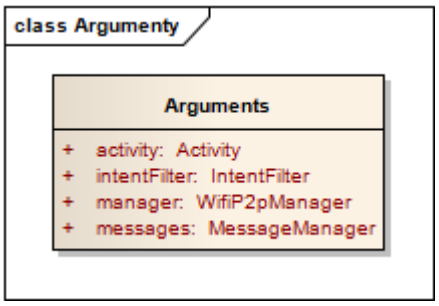
6.4 Implementační třídy.

Použité technologie jsem již rozebral. Teď se podíváme, jak jsou realizovány jednotlivé funkčnosti klienta, respektive jeho nejzajímavější části. První z nich je samozřejmě retranslace, která tvoří základ celého klienta distribuované sítě a je tedy nejdůležitější součástí systému. Jako další rozeberu podrobněji fungování monitoringu senzorů. Myslím, že by tento systém zde měl být rozebrán z toho důvodu, že se sítí rozesílají právě údaje ze senzorů. Třetím a posledním mechanizmem, který je z hlediska práce zajímavým a zaslouží si podle mne podrobnější rozebrání, je systém zapínání a vypínání jednotlivých režimů klienta.

6.4.1 Retranslace

Nejprve se podíváme na diagram tříd, které se přímo podílejí na retranslaci zpráv v síti. Tento diagram můžeme vidět na obrázku Obrázek 16.

Základem retranslace je třída – fragment `RetranslateFragment`. Tento fragment zastřešuje veškerou funkčnost, co se týče síťové komunikace. Celý proces komunikace je zahájen voláním veřejné metody `configureRetranslation(Arguments args)`. Třída `Arguments` obsahuje veškeré potřebné údaje pro retranslaci. Lépe si ji můžeme prohlédnout na obrázku Obrázek 15. Jsou to zejména odkazy na instance `Activity`, `WifiP2pManagera`. A odkaz na `MessageManagera`, který implementuje funkčnost třídy `FrontaZprav`.

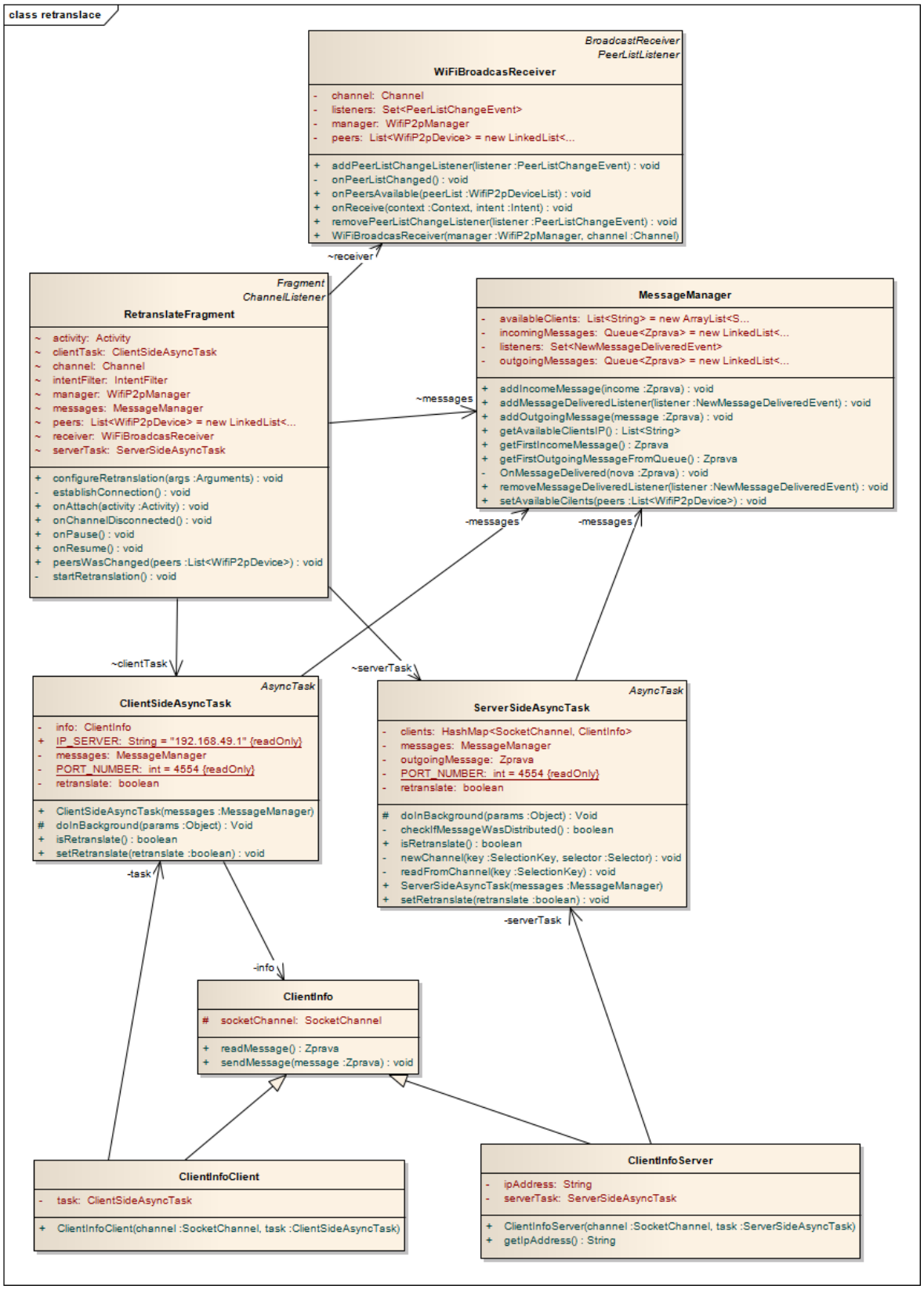


Obrázek 15 - Třída argumenty

Metoda `configureRetranslation(Arguments args)` obsahuje mimo samotné uložení potřebných odkazů ještě volání soukromé metody `startRetranslation()`, kterou můžeme vidět níže:

```
private void startRetranslation() {
      manager.discoverPeers(channel, new ToastActionListener());
}
```

a která provádí pouze jednu jedinou věc: iniciuje voláním `discoverPeers()` vyhledání okolních zařízení.



Obrázek 16 - Implementační třídy - retranslace

V momentě, kdy je k dispozici seznam okolních zařízení, který poskytuje třída `WiFiBroadcastReceiver` v reakci na systémovou událost `onReceive()`, ve které

je `RetranslateFragment` upozorněn na změnu seznamu okolních zařízení, je volána soukromá metoda

```
private void establishConnection() {
	if (((MainActivity) activity).isAdministration()) {
		//We are server for all devices
		manager.createGroup(channel, null);
		//now, we should have in ARP all IP addresses
		messages.setAvailableCilents(peers);
		//Start server AsyncTask
		serverTask=new ServerSideAsyncTask(messages);
		serverTask.execute();
	}else {
		//We aren't server for devices
		//Start client AsyncTask
		clientTask=new ClientSideAsyncTask(messages);
		clientTask.execute();
	}
}
```

Tato metoda má za úkol rozhodnout, zda je jednotka serverem pro ostatní, nebo zda bude jednotka v roli klienta. Rozhodovacím parametrem je příznak `administration` v hlavní aktivitě. Jestliže má jednotka administrátorská práva, bude vytvořena instance `ServerSideAsyncTask`, v opačném případě `ClientSideAsyncTask`.

Obě dvě třídy jsou potomky `AsyncTask`, což je třída operačního systému Android. V podstatě se jedná o odlehčenou verzi vláken. Jediný markantní rozdíl od vlákna je ten, že se nedá synchronizovat s hlavní aktivitou, ale běží, dokud nevykoná svoji činnost.

6.4.2 ServerSideAsyncTask

Nejprve se budu zabývat serverovou částí retranslace, která je obsažena v `ServerSideAsyncTask` třídě. Při tvorbě instance této třídy se do konstruktoru předá odkaz na instanci fronty zpráv, aby tato třída mohla manipulovat s příchozími a odchozími zprávami, jelikož server se snaží rozesílat všechny zprávy, které ještě má smysl dál distribuovat (tedy v jejich seznamu retranslačních bodů ještě není daná jednotka uvedena). V momentě, kdy se zavolá nad instancí `ServerSideAsyncTask` metoda `execute()`, se začne vykonávat metoda `doInBackground()`. Na její signaturu se můžeme podívat níže.

```
@Override
protected Void doInBackground(Object... params)
```

Parametrem této metody je libovolný počet objektů typu `Object`. (Typ parametru se dá upravit při vytváření třídy, jelikož `AsyncTask` je generická.) Dá se tedy nastavit typ předávaných parametrů metodě `doInBackground()` a stejně tak i návratový typ této metody.

V této metodě nejprve vytvoříme instanci `ServerSocketChannel`, což je třída z knihovny Java NIO, která umožňuje vytvářet neblokující serverové sokety. Toto je pro

klienta velice důležitá funkčnost, jelikož my dopředu nevíme, kdy bude klientská jednotka vysílat své údaje a hlavně která to bude. Jak již bylo řečeno v popisu knihovny Java NIO, je možné pomocí této knihovny mulitplexovat příchozí spojení, takže se všichni klienti sice připojují na jeden konkrétní port, ale po navázání spojení je komunikace převedena na jiný, takže se na tento port mohou připojit i ostatní. Toto zrealizujeme pomocí kódu níže.

```
ServerSocketChannel serverChannel=ServerSocketChannel.open();
serverChannel.configureBlocking(false);
ServerSocket serveSocket=serverChannel.socket();
serveSocket.bind(new InetSocketAddress(PORT_NUMBER));
Selector selector=Selector.open();
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

Nejdříve tedy otevřeme `ServerSocketChannel` a nakonfigurujeme ho jako neblokující (příznak `false`). Dále pak získáme odkaz na soket, přes který se bude komunikovat, a přiřadíme mu adresu a číslo portu, přes který se bude komunikovat.

Třída `Selector` slouží k multiplexování připojení.

Řádkem `serverChannel.register(selector, SelectionKey.OP_ACCEPT);`

říkáme `SocketChannelu`, že budeme očekávat klíče typu `accept`, tedy příchozí spojení.

Následně provádíme již samotný příjem připojení a to tak dlouho, dokud je zapnutý režim retranslace. Algoritmus příjmu připojení a rozesílání zpráv si můžeme prohlédnout níže.

```
selector.select();
Iterator<SelectionKey> it=selector.selectedKeys().iterator();
SelectionKey key;
if (outgoingMessage==null) {
	//This message is need to send all available clients
	outgoingMessage = messages.getFirstOutgoingMessageFromQueue();
}
while (it.hasNext()) {
	key=(SelectionKey) it.next();
	it.remove();
	if (key.isAcceptable()) {
		newChannel(key,selector);
	}
	if (key.isReadable()) {
		readFromChannel(key);
		}
}
```

Prvně vybereme sadu příchozích klíčů a vytvoříme si iterátor, přes který budeme sadu klíčů procházet. Pak se podíváme, zda máme nějakou odchozí zprávu. Jestliže `outgoingMessage` je `null`, znamená to, že tato odchozí zpráva již byla odeslána všem klientům, kteří jsou k serveru připojeni, a musíme tedy z fronty zpráv vybrat novou zprávu, která bude rozesílána připojeným klientům.

Jako další krok je již samotné procházení sady klíčů, které máme k dispozici (byly zaslány klienty). Vybereme tedy klíč, odstraníme ho ze sady klíčů, jelikož jsme tento požadavek právě začali řešit a následně vyhodnotíme, o jaký požadavek se jedná. Jestliže jde o nový požadavek, tedy je zaslán klíč `accept`, vytvoříme nové spojení v metodě `newChannel()`, které předáme klíč a `selector`, podle nichž pak určíme klienta, který požadavek zaslal. Jestliže však má klient nastaven klíč na `readable`, tedy má nějaká data, která chce odeslat, voláme metodu `readFromChannel()`, které předáme klíč.

Nyní se podrobněji podíváme na obě metody. Nejprve začnu metodou `newChannel()`, která, jak jsem již zmínil výše, zpracovává nově příchozí požadavek klienta na připojení do distribuované sítě.

```
private void newChannel(SelectionKey key, Selector selector) {
	try {
		ServerSocketChannel serverChannel
			=(ServerSocketChannel)key.channel();
		SocketChannel channel=serverChannel.accept();
		channel.configureBlocking(false);
		channel.register(selector,SelectionKey.OP_READ);
		clients.put(channel,new ClientInfoServer(channel, this));
	} catch (IOException e) {
		Toast.makeText(null, "Can't register income connection",
			Toast.LENGTH_SHORT).show();
		}
	}
```

V této metodě nejprve pomocí klíče vytvoříme nový kanál, po kterém se bude s daným klientem komunikovat a nastavíme ho jako neblokující, čímž zabráníme tomu, aby server čekal až do okamžiku, kdy budou nějaká data k dispozici, ale v momentě, kdy žádná data ještě nejsou k dispozici, pokračoval ve své činnosti. Tomuto kanálu pak nastavíme hodnotu klíče na `read` a tím dáme najevo, že klient bude odesílat nějaká data. Nakonec zaregistrujeme, přidáním do mapy, tento požadavek, kdy klíčem k záznamu je kanál, na kterém se komunikuje a hodnotou je nová instance tříd `ClientInfoServer`. (Tuto třídu si rozebereme hned vzápětí.) Tím je dokončena registrace nového klientského požadavku.

Nyní se podrobněji podíváme, jak to vypadá v případě, že se klient již se serverem spojil dříve a v tomto okamžiku má nějaká data k odeslání. Jak již bylo řečeno, k tomu slouží metoda `readFromChannel()`. Opět ji zde nejdříve uvedu a pak vysvětlím její funkčnost.

```
private void readFromChannel(SelectionKey key) {
	SocketChannel channel=(SocketChannel)key.channel();
	ClientInfoServer info=(ClientInfoServer)clients.get(channel);
	if (info!=null) {
		Zprava message=info.readMessage();
		if (message!=null) {
			messages.addIncomeMessage(message);
		}
		if (outgoingMessage!=null) {
```

```
                    info.sendMessage(outgoingMessage);
                }
        }
}
```

Nejprve získáme správný kanál pro tohoto klienta a to pomocí jeho klíče. Následně si ze seznamu klientů vybereme správné info o klientovi. Třídu `ClientInfo` popíšu později. V tuto chvíli stačí vědět, že obsahuje metody pro čtení a zapisování dat do soketů.

Tato metoda je volána v okamžiku, kdy klient pro nás má připravenou zprávu k odeslání. Je tedy logické, že si ji hned v dalším kroku přečteme a pokud nám nějaká zpráva skutečně přišla, tak ji zařadíme do fronty příchozích zpráv.

Jelikož je soket obousměrný, máme-li k dispozici nějakou odchozí zprávu, pokusíme se ji odeslat přes metodu `info.sendMessage()`. Tímto způsobem jsem využil serverový `AsyncTask`. Nyní se podrobněji ještě podíváme na třídy `ClientInfo`, které zapisují z a do soketů.

`ClientInfo` je třída, která je předkem pro dvě třídy: `ClientInfoServer`, která se používá na straně serveru pro uchování údajů o připojeném klientovi a `ClientInfoClient`, která na straně klienta uchovává informace o připojení k serveru. Samotná `ClientInfo` pak poskytuje metody pro zápis a čtení ze soketů.

Server si tedy uchovává pomocí těchto tříd údaje o připojených klientech. Dělá to pomocí proměnné `clients`, kterou si můžeme prohlédnout níže.

```
private HashMap<SocketChannel, ClientInfo> clients;
```

Je to HashMapa, jejímž klíčem je `SocketChannel`, který slouží k identifikaci spojení, a jejíž hodnotou je instance potomka `ClientInfo`, která uchovává potřebné údaje.

Čtení a zápis dat z a do soketu probíhá v metodách `readMessage()` a `sendMessage()` třídy `ClientInfo`. Celý proces je založen na serializaci instance třídy `Zprava`. Odeslání zprávy tedy probíhá následovně:

```
public void sendMessage(Zprava message) {
        try {
                ByteArrayOutputStream byteStream=new ByteArrayOutputStream();
                ObjectOutputStream oOutput=new ObjectOutputStream(byteStream);
                oOutput.writeObject(message);
                oOutput.flush();
                socketChannel.write(ByteBuffer.wrap(byteStream.toByteArray()));
        } catch (IOException e) {
                e.printStackTrace();
        }
}
```

Metoda v základu serializuje objekt typu `Zprava`, jenže jelikož používáme neblokující sokety, nemůžeme použít klasickou serializaci přes `ObjectOutputStream`, jelikož ten používá blokující vstupní a výstupní operace. Proto musíme serializační objektový proud

nejdříve obalit instancí `ByteArrayOutputStream`, která je schopná převést serializovaný objekt do bytového pole, které můžeme poslat metodou `write()` na `SocketChannelu`. Tímto se zbavíme blokující vstupně – výstupní operace a celé odeslání zprávy tak může proběhnout jako neblokující operace.

Na velice podobném principu funguje čtení zprávy:

```
public Zprava readMessage() {
	Zprava message=null;
	try {
		ByteBuffer data=null;
		socketChannel.read(data);
		ByteArrayInputStream byteStrean=new
ByteArrayInputStream(data.array());
		ObjectInputStream oInput=new ObjectInputStream(byteStrean);
		message=(Zprava)oInput.readObject(); }
```

Uvádím zde jen tu nejdůležitější část kódu. Jak můžeme vidět, je nejdříve potřeba ze soketu přečíst data do bytového pole. Z bytového pole si můžeme vytvořit `ByteArrayInputStream` a ten obalit `ObjectInputStreamem`. Tento proces je opět důležitý, jelikož se potřebujeme zbavit blokujících vstupně – výstupních operací. Teprve nakonec můžeme provést deserializaci objektu a získat tak zprávu, kterou vrátíme.

6.4.3 ClientSideAsyncTask

Tato třída zastává síťovou komunikaci na straně klienta. Stejně jako serverová část i tato třída dědí od `AsyncTask`, jelikož ani klient si nemůže dovolit být blokován v průběhu odesílání zpráv. V případě, že by klient byl blokován odesíláním zpráv na server, uživatel aplikace by mohl zjistit, že aplikace v určitých chvílích neodpovídá na jeho povely. Tento stav samozřejmě nechceme a tak odesílání zpráv probíhá v samostatném vlákně. Stejně jako serverová část je i klientská část realizovaná pomocí knihovny Java NIO. Její implementace je ovšem značně jednodušší, jelikož nemusíme spravovat příchozí připojení od ostatních klientů, ale pouze udržujeme spojení se serverem.

Opět zde uvedu výňatek z metody `doInBackground()`, která obsahuje hlavní část komunikace se serverem.

```
@Override
protected Void doInBackground(Object... params) {
	retranslate=true;
	try {
		SocketChannel channel=SocketChannel.open();
		channel.configureBlocking(false);
		channel.connect(new InetSocketAddress(IP_SERVER, PORT_NUMBER));
		while (!channel.finishConnect()) {
			//Probably do nothing, we just wait for connection to server
			}
		info=new ClientInfoClient(channel, this);
		while (retranslate) {
			Zprava message=messages.getFirstOutgoingMessageFromQueue();
			if (message!=null) {
				//send message
```

```
                info.sendMessage(message);
            }
            //Try to read incoming message if any is available
            Zprava incomeMessage=info.readMessage();
            if (incomeMessage!=null) {
                messages.addIncomeMessage(incomeMessage);
                }
    }
```

Na rozdíl od serverové části, kdy otevíráme `ServerSocketChannel`, klient používá pouze `SocketChannel`. Nicméně stejně jako u serveru i zde ho konfigurujeme jako neblokující. V následujícím kroku se pokusíme otevřít síťovou komunikaci na IP adresu serveru. IP adresa serveru je pokaždé stejná a je nastavená na 192.168.49.1. Tato hodnota je výchozí a přiděluje ji knihovna Wi-Fi Direct zařízení, které se stává vlastníkem skupiny. V našem případě je to vždy jednotka, která má administrátorská práva a je tedy serverem. Port jsem zvolil náhodně a má hodnotu 4554.

Pak následuje cyklus, který pouze čeká na to, až bude plně navázané spojení se serverem. Klientské vlákno komunikace nemusí dělat nic jiného, a proto může čekat až do okamžiku, kdy bude navázáno spojení se serverem.

V okamžiku, kdy je spojení navázáno, si vytvoříme instanci `ClientInfo`, její mutaci pro klientské jednotky. `ClientInfoClient` obsahuje metody pro odesílání zpráv na server. Respektive jelikož se jedná opět jen o serializaci a deserializaci objektů, jsou tyto metody obsažené v samotné `ClientInfo`, a `ClientInfoClient` pouze obsahuje dodatečné identifikátory pro klientskou část.

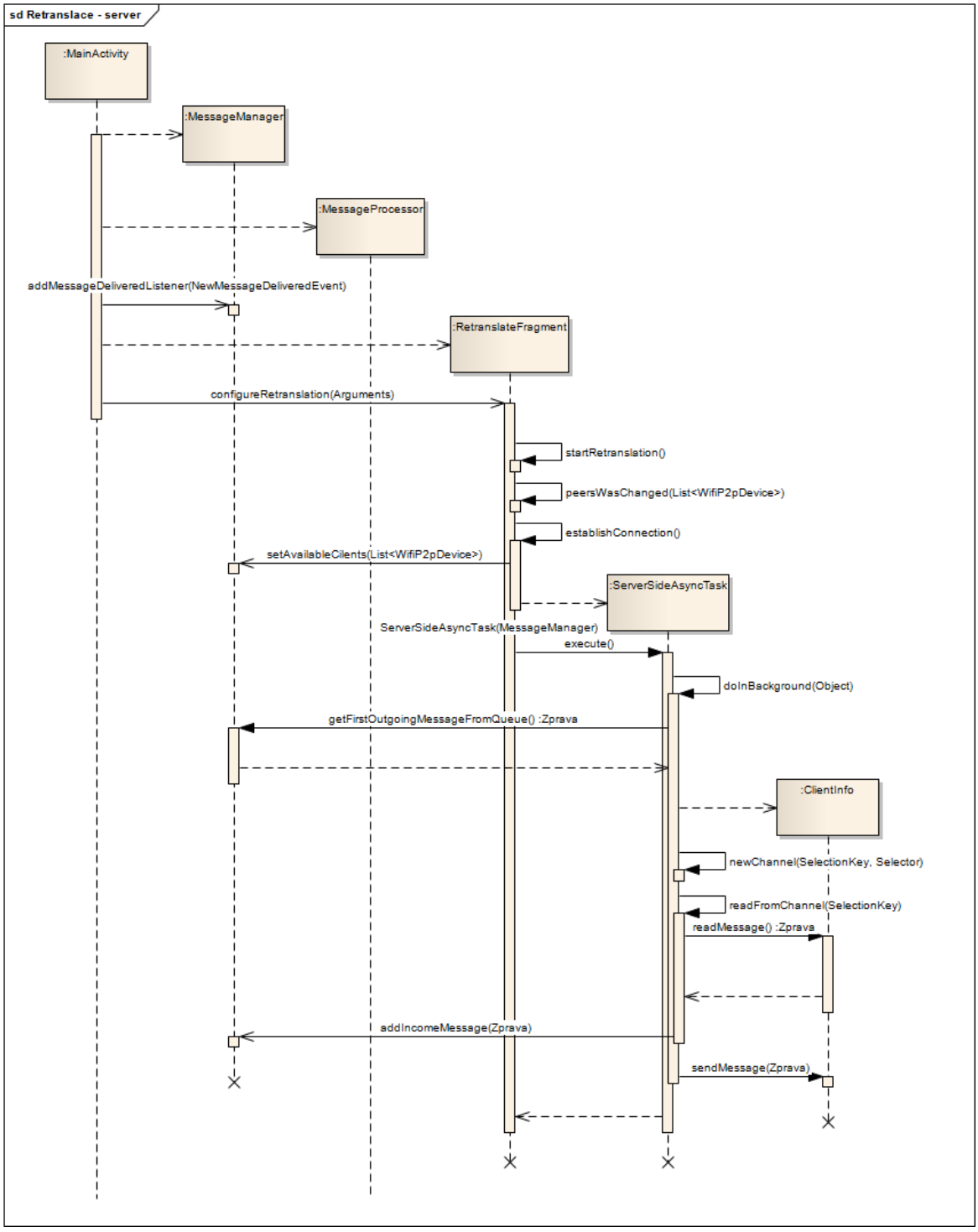
Zasílání zpráv je velice jednoduché. Vezmeme první zprávu z fronty odchozích zpráv a odešleme ji na server. Tentokrát nemusíme kontrolovat, jestli byla zpráva odeslána všem jednotkám, jelikož klient může komunikovat stejně pouze se serverem a tudíž není potřeba kontrolovat seznam adresátů.

Aby se zabránilo rozesílání zpráv, které už jednotka jednou zpracovala, stále tady funguje mechanizmus vyhodnocování příchozích zpráv, o němž se ještě zmíním. Tento algoritmus by měl zaručit, že se do fronty odchozích zpráv dostanou zprávy skutečně určené k odeslání na server.

Opět proběhne odeslání zprávy pomocí metody `sendMessage()`, která je totožná s tou na serverové straně. Stejně pak probíhá čtení zprávy, protože v okamžiku, kdy se připojujeme k serveru s tím, že odesíláme zprávu, server na tento požadavek reaguje tak, že si naši zprávu přečte a pokusí se nám odeslat zprávu novou, ze svojí fronty odchozích zpráv. Tím dojde k plnému zužitkování právě probíhající komunikace v jednom soketu. V případě, že máme nějakou příchozí zprávu, zařadí ji do fronty příchozích zpráv.

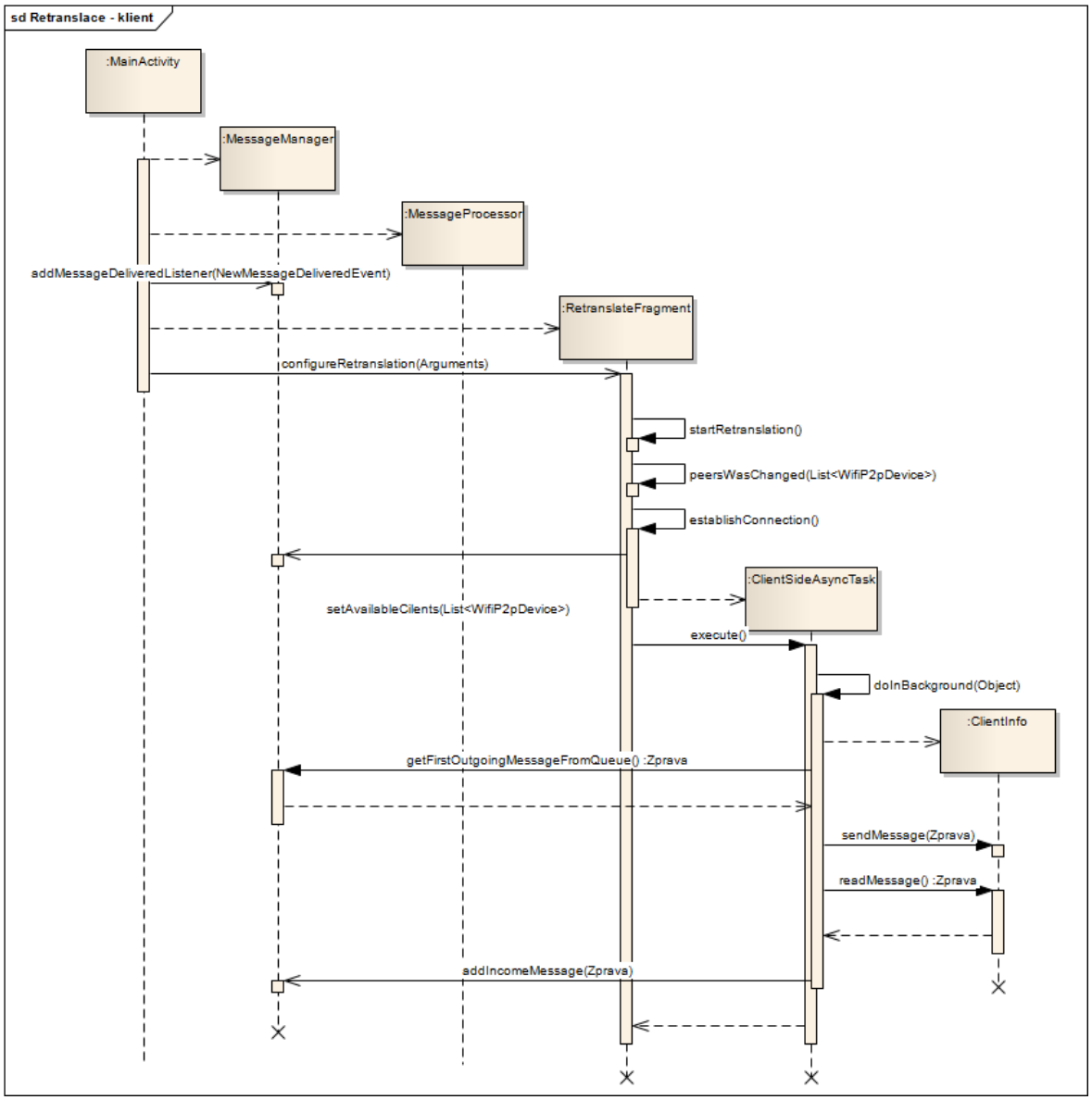
6.5 Sekvenční diagramy retranslace

Nejlépe si celou problematiku komunikace prohlédneme na sekvenčních diagramech. Na obrázku Obrázek 177 se můžeme podívat na diagram retranslace, od vytvoření hlavní aktivity až po komunikaci na síti pomocí serverového asynchronního úkolu.



Obrázek 17 - Sekvenční diagram retranslace, serverová část

Na obrázku Obrázek 18 je pak sekvenční diagram klientské části retranslace. Už na první pohled je vidět, že tato část je značně jednodušší, jelikož klient pouze odesílá a přijímá zprávy, nemusí se nijak výrazně starat o samotnou síťovou komunikaci. Také si můžeme povšimnout, že je prohozené odesílání a přijímání zpráv. Klient odesílá zprávu jako první a tím tedy vzniká požadavek na server, který jej tak může přijmout a následně nám zaslat nějakou zprávu zpátky.

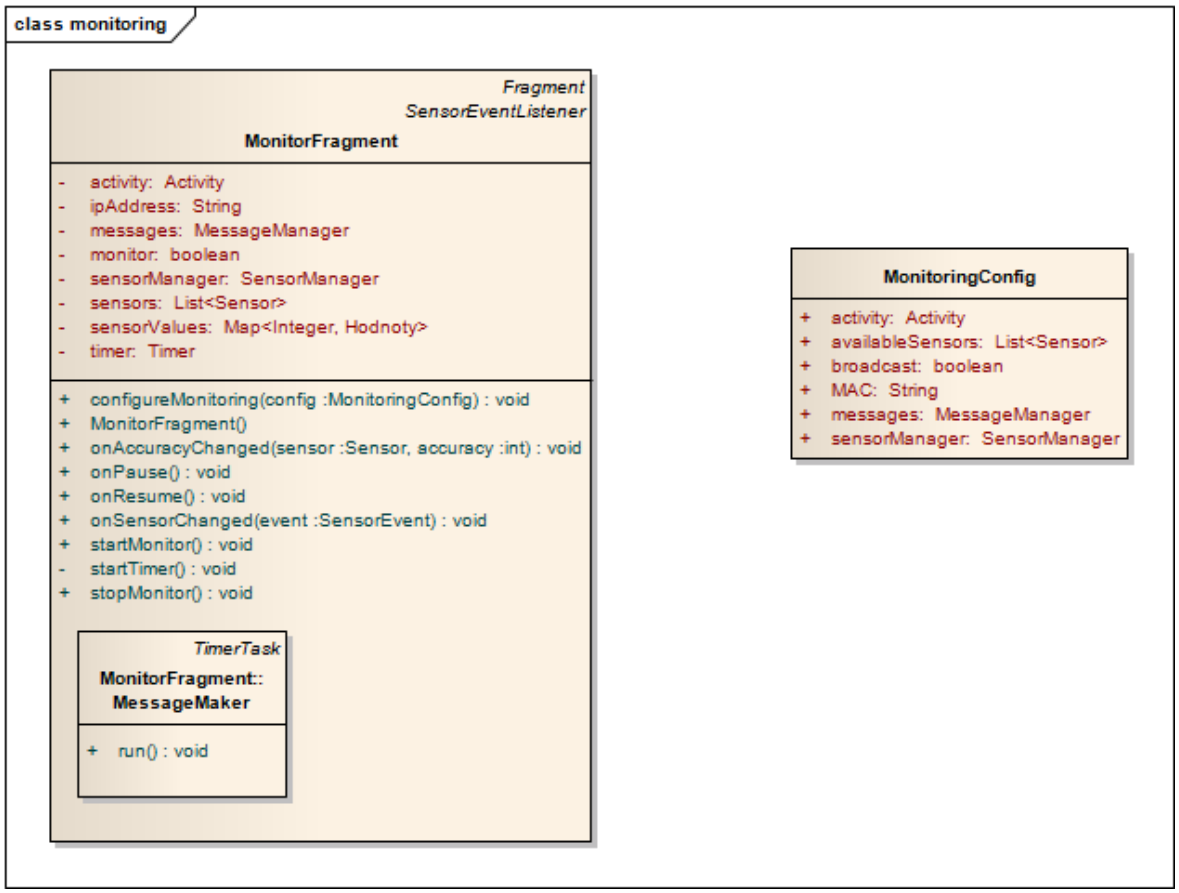


Obrázek 18 - Sekvenční diagram retranslace, klientská část

6.6 Fragment monitoringu

Další částí, kterou bych chtěl podrobněji rozebrat, je fragment monitoringu. Tento fragment se stará o sběr dat ze senzorů daného zařízení. Jelikož celá síť je založená na posílání právě těchto údajů, připadá mi důležité zde tuto funkčnost podrobněji rozebrat.

Tento fragment si můžeme prohlédnout na obrázku Obrázek 199, na kterém je vidět diagram tříd celé sekce monitoringu.



Obrázek 19 - Diagram tříd monitoringu

Třída `MonitoringConfig` slouží jako přenašeč parametrů důležitých pro chod monitoringu. Je to zejména odkaz na instanci aktivity, seznam dostupných senzorů, MAC adresa jednotky, manažera zpráv a systémovou třídu `SensorManagera`. Manažer zpráv je pro třídu důležitý, jelikož právě tento fragment se stará o vytváření datových zpráv a jejich přidávání do fronty odchozích zpráv.

Fragment implementuje posluchač `SensorEventListener`, který slouží k zachytávání událostí vzniklých změnou hodnot na senzoru. K tomu slouží dvě metody, které je potřeba překrýt. První je `onAccuarryChanged()`, která slouží k obsluze změn přesnosti snímání senzorů. Tato metoda pro jednotku není nijak důležitá, jelikož nás zajímají údaje ze senzorů jako takové a ne s jakou přesností byly údaje změřeny.

Druhou metodou je `onSensorChanged()`, která je pro nás důležitá, jelikož se volá pokaždé, když se změní hodnota senzoru. Právě v této metodě získáváme nové údaje ze senzorů. Parametrem této metody je událost, která nese údaje o typu senzoru, který danou událost vyvolal, a nové hodnoty z tohoto senzoru. Metodu si můžeme prohlédnout níže.

```
@Override
public void onSensorChanged(SensorEvent event) {
	float[] values=event.values;
	if (sensorValues.containsKey(event.sensor.getType())) {

		if (values.length>=1) {
			sensorValues.get(event.sensor.getType()).setX(values[0]);
		}
		if (values.length>=2) {
			sensorValues.get(event.sensor.getType()).setY(values[1]);
		}
		if (values.length>=3) {
			sensorValues.get(event.sensor.getType()).setZ(values[2]);
		}
	}else {
		Hodnoty newHodnoty=new Hodnoty();
		newHodnoty.setTypSenzoru(event.sensor.getType());
		if (values.length>=1) {
			newHodnoty.setX(values[0]);
		}
		if (values.length>=2) {
			newHodnoty.setY(values[1]);
		}
		if (values.length>=3) {
			newHodnoty.setZ(values[2]);
		}
		sensorValues.put(newHodnoty.getTypSenzoru(), newHodnoty);
		}
	}
```

Nejdříve si tedy uložíme do pole hodnoty z daného senzoru. Následně zjistíme, zda nejde o případ, kdy ještě v mapě senzorů tento typ senzoru nemáme uložený. Mapa uchovává údaje o senzorech, klíčem jí je typ senzoru a hodnotou je instance třídy Hodnoty, která, jak bylo popsáno výše, uchovává hodnoty ze senzoru. Jestliže senzor je již jednou uložen v mapě senzorů s příslušnými hodnotami, tak pouze aktualizujeme údaje o něm. Jelikož dopředu nevíme, o který senzor se jedná (to se dozvíme až v okamžiku, kdy dojde ke změně údajů a systém nás upozorní voláním tohoto callbacku), nevíme ani dopředu, kolik údajů bude přenášeno v parametru. Proto musíme postupně zjistit, jestli pole ještě obsahuje nějakou další hodnotu a tu následně uložit do příslušných atributů instance třídy Hodnoty.

Jestliže mapa hodnot senzorů ještě neobsahuje informace o daném senzoru, tak podobným principem vytvoříme instanci Hodnoty a tuto pak vložíme do mapy.

Abychom vůbec mohli senzorům „naslouchat“, musíme nejdříve zaregistrovat posluchače pro každý senzor zvlášť. Proto fragmentu monitoringu předáváme list dostupných senzorů a samotnou registraci senzorů provádíme v metodě `onResume()`, která se volá při vzniku fragmentu a pak pokaždé, když je aplikace přenesena do popředí. Metodu `onResume()` si můžeme prohlédnout níže.

```
@Override
public void onResume() {
	super.onResume();
	for (Sensor sensor:sensors) {
```

```
            sensorManager.registerListener(this, sensor,
            SensorManager.SENSOR_DELAY_NORMAL);
      }
            timer.schedule(new MessageMaker(), 3000);
      }
```

Nejprve tedy zaregistrujeme fragment monitoringu jako posluchače pro každý senzor, který má daná jednotka k dispozici. Pro potřeby klienta stačí, když budeme monitorovat senzory s normálním zpožděním. Senzory jde monitorovat i rychleji, ale pro potřeby klienta je to zbytečné. Nakonec naplánujeme spuštění úlohy, která má za úkol vytvořit novou datovou zprávu.

Ke změnám senzorů dochází velice často, obzvláště když se jednotka pohybuje. Takovou frekvenci změn senzorů nepotřebujeme rozesílat po síti. Takovým množstvím datových zpráv by byla síť za chvíli zahlcena. Proto klient používá časovač, který každé 3 sekundy vezme aktuální uložené údaje ze senzorů a vytvoří z nich datovou zprávu.

Plánování úlohy je vidět výše, v metodě `onResume()`. Zde časovači řekneme, která třída obslouží událost v momentě, kdy uběhne potřebný časový interval, po kterém se mají úlohy spouštět. V našem případě je to vnitřní třída `MessageMaker` a interval tří sekund. Tento interval je dle mého názoru dostatečně krátký, aby síť dokázala zachycovat důležité změny na senzorech a dostatečně dlouhý na to, aby se síť příliš nezahlcovala velkým množstvím datových zpráv.

Třídu si můžeme prohlédnout níže.

```
class MessageMaker extends TimerTask{

      @Override
      public void run() {
            DatovaZprava message = new DatovaZprava();
            Zarizeni device = new Zarizeni(ipAddress);
            for (Entry<Integer, Hodnoty> pair : sensorValues.entrySet()) {
                  device.pridejHodnotuSenzoru(pair.getValue());
            }
            message.pridejZarizeni(device);
            messages.addOutgoingMessage(message);
      }
}
```

Třída dědí z `TimerTask`, což nám říká, že bude obsahovat jednu metodu `run()`, kterou budeme muset překrýt. Tato metoda se spouští pokaždé, když nám časovač řekne, že vypršel daný časový limit, nebo nastal čas, na který byla daná událost naplánována.

Samotná metoda je velice jednoduchá. Vytvoříme novou instanci datové zprávy a připravíme si instanci zařízení, které identifikujeme vlastní IP adresou. Pak pro všechny hodnoty senzorů, které máme aktuálně uložené v mapě hodnot senzorů, přidáme záznam do instance `Zarizeni`. Nakonec instanci `Zarizeni` zabalíme do zprávy a přidáme ji do fronty odchozích zpráv. Tímto způsobem funguje monitoring senzorů.

6.7 Řízení zapínání a vypínání režimů

Posledním mechanizmem, který bych rád rozebral podrobněji a je podle mne zajímavý pro ukázku, je řízení zapínání a vypínání jednotlivých režimů. Jak bylo předesláno dříve, každý režim má svůj vlastní fragment, který zastřešuje funkčnost daného režimu. Nejjednodušším prostředkem pro řízení takové funkčnosti je právě možnost operačního systému Android přidávat a odebírat fragmenty za běhu aplikace. K tomu nám slouží dvě třídy. Tou první je `FragmentManager`, která spravuje jednotlivé fragmenty, druhou je třída `FragmentTransaction`, která umí přidávat fragmenty za běhu aplikace. Opět bude nejlepší, když uvedu příklad na části zdrojového kódu. První ukázkou je metoda, která přidá fragment retranslace.

```
private void addRetranslationFragment() {
	transaction=fragmenter.beginTransaction();
	transaction.add(new RetranslateFragment(), RETRANSLATION_FRAGMENT_TAG);
	transaction.commit();
	fragmenter.executePendingTransactions();
	retranslatorFragment=(RetranslateFragment)fragmenter.findFragmentByTag(RE
	TRANSLATION_FRAGMENT_TAG);
	retranslatorFragment.configureRetranslation(getArguments());
	}
```

V první řadě je potřeba zahájit transakci. Samotná transakce je svým způsobem podobná klasické databázové transakci. Je tedy potřeba ji nejdříve zahájit voláním `beginTransaction()` nad instancí `FragmentManageru`. Tím dostaneme instanci transakce. Následně můžeme přidat samotný fragment metodou `add()`, které řekneme, jaký fragment chceme přidat (v našem případě to je fragment retranslace) a tag, pod kterým daný fragment můžeme vyhledat.

Přidávaných fragmentů může být více v rámci jedné transakce, ale vzhledem k tomu, že vždy zapínáme jen určitý režim, bude to jen konkrétní jeden fragment. Transakci potvrdíme voláním `commit()`. Zavoláním `executePendingTransactions()` třídy `FragmentManager` donutíme vykonat všechny zamýšlené změny ihned. Tento krok je důležitý, jelikož samotné přidání fragmentu by také mohlo nějakou dobu trvat. Provedením těchto změn pak hned můžeme uložit odkaz na právě přidaný fragment retranslace tím, že ho vyhledáme pomocí tagu, jenž jsme mu přiřadili v momentu přidání přes transakci. Tím je celý proces přidání fragmentu za běhu aplikace hotový.

Obdobně probíhá jejich odebírání. Opět zde uvedu příklad zdrojového kódu.

```
if (config.retranslateShutDown) {
	retranslate=false;
	transaction=fragmenter.beginTransaction();
	transaction.remove(retranslatorFragment);
	transaction.commit();
	fragmenter.executePendingTransactions();
	retranslatorFragment=null;
```

Jedná se o výňatek z metody `manageModes`, která přijímá konfiguraci `config`, podle které vypíná nebo zapíná dané režimy. Jestliže je nastaven příznak `retranslateShutDown` na `true`, znamená to, že se má vypnout režim retranslace. Nastavíme tedy globální příznak retranslace na `false` a opět začneme transakci pro odebrání fragmentu. Odebrání se provede metodou `remove()`, které předáme instanci fragmentu, který chceme odstranit. Transakce potvrdíme a vynutíme si provedení nastalých změn stejným způsobem, jako když jsme fragment přidávali.

7 Instalace aplikace

Jako kapitolu na závěr jsem zvolil návod, jak samotnou aplikaci klienta nainstalovat do zařízení. Postup je to poměrně jednoduchý, ale nejdříve k tomu potřebujeme znát něco o právech aplikace v operačním systému Android.

7.1 Android a práva pro aplikace

Celý koncept práv operačního systému Android vychází z jeho linuxového jádra, kde jsou aplikace odizolované navzájem od sebe, a od toho, aby mohly ovlivňovat samotný operační systém. Což v praxi znamená, že každá aplikace běží se svojí identitou oddělená od ostatních, aby svojí činností nemohla poškozovat nebo zneužívat okolní aplikace, nebo zasahovat do uživatelských dat. Ve výchozím stavu tak aplikace nemá žádná práva k tomu, aby mohla zasahovat do svého okolí. Jenže aplikace potřebují čas od času ke své funkci sdílet zdroje daného zařízení. V našem případě klienta distribuované sítě to je přístup k bezdrátové síti. K povolení přístupu aplikace k danému systémovému zdroji slouží práva. Aplikace tedy stanoví, která práva požaduje po systému, aby jí byla udělena. Systém pak při instalaci aplikace vyzve uživatele, aby aplikaci tato práva udělil nebo je zamítl. V případě, že uživatel schválí požadovaná práva aplikaci, je aplikace nainstalována a jsou jí přidělena všechna práva. V současnosti nejde vybrat, která práva mají být schválena, a proto zamítneme-li přijmout všechna práva, aplikace nebude nainstalována.

Jednotlivá práva, která bude aplikace vyžadovat, se zapisují do manifestačního XML souboru aplikace. Klient sítě bude potřebovat přístup k Internetu, přistupovat ke stavu bezdrátové sítě a možnost tento stav měnit. Tato práva uvedeme do již zmíněného manifestačního XML následovně:

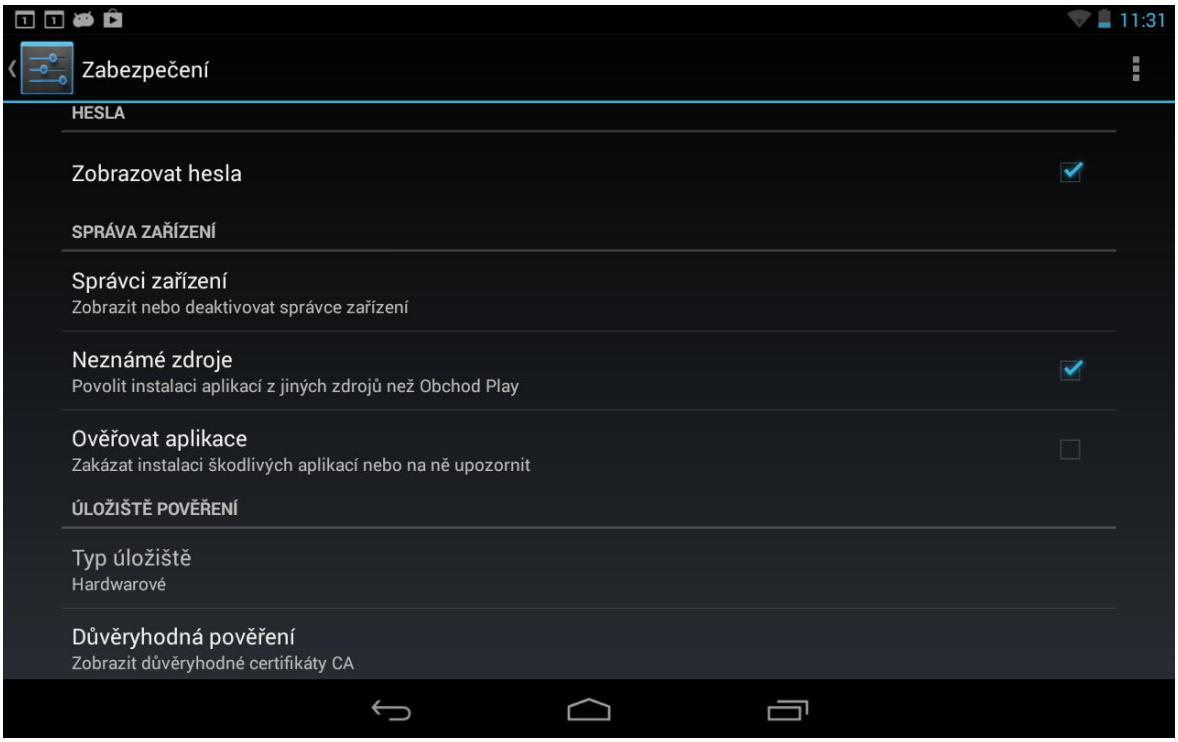
```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
```

7.2 Instalace

Instalace aplikace do zařízení je jednoduchá, ale nejprve musíme zajistit pár předpokladů. Jelikož aplikace není instalovaná z obchodu aplikací Google Play™, musíme povolit instalování aplikací z neznámých zdrojů. Toto nám umožní spustit instalační balíček .apk a nainstalovat tak aplikaci. To provedeme následujícím způsobem.

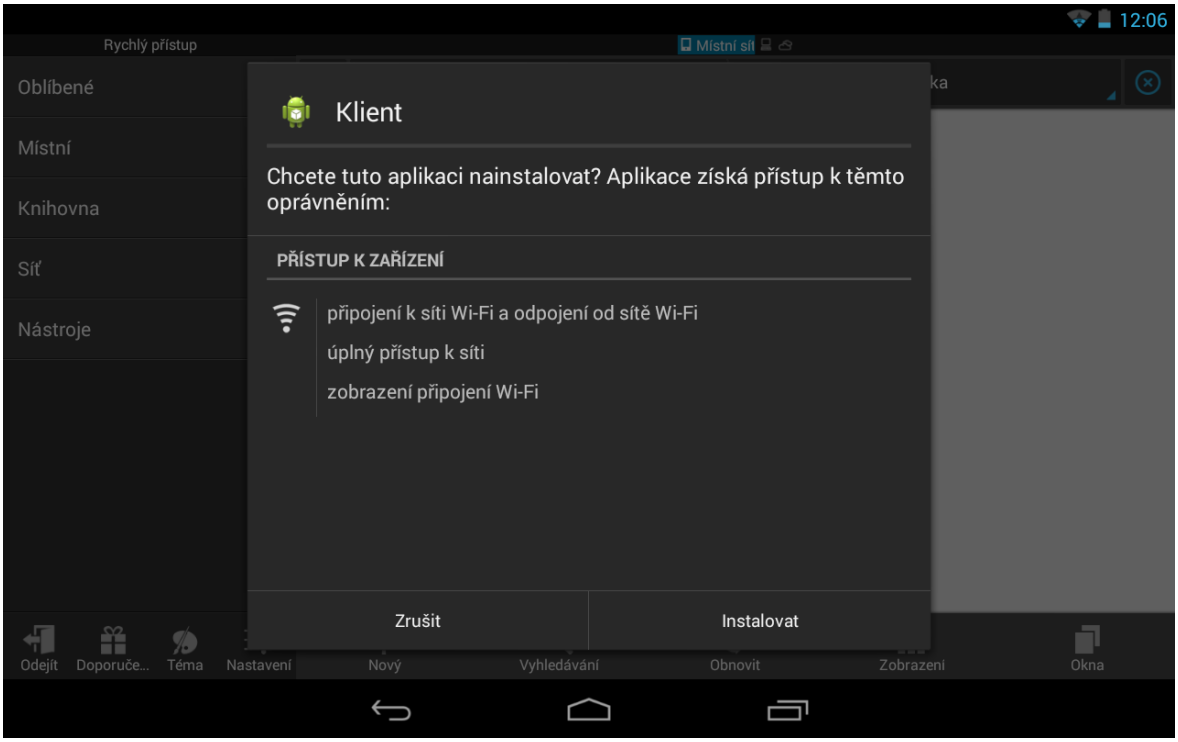
1. Otevřeme Nastavení
2. Zvolíme Zabezpečení
3. Zaškrtneme Neznámé zdroje

Výsledný stav si můžeme prohlédnout na obrázku Obrázek 20, který je snímkem obrazovky zařízení, na které aplikaci budeme instalovat.



Obrázek 20 - Nastavení povolení instalovat aplikace

Jako další krok přeneseme do zařízení instalační balíček. Postup záleží na tom, jaké máme možnosti připojení zařízení k PC, případně jinému zařízení, kde máme uložený instalační balíček. Nejjednodušší cesta vede přes připojení zařízení k PC pomocí datového kabelu USB. V momentu, kdy máme instalační balíček přenesený v zařízení, tak pomocí správce souborů otevřeme složku, kde je uložený. Výběrem balíčku spustíme jeho instalaci. Systém zobrazí hlášení o instalaci aplikace s přehledem práv, která aplikace vyžaduje. Toto si můžeme prohlédnout na obrázku Obrázek 21 níže. Jestliže souhlasíme s požadovanými právy, stiskneme tlačítko instalovat. Systém spustí instalaci, o jejímž průběhu nás bude informovat. Vzhledem k tomu, že aplikace Klienta je malá, proběhne samotná instalace velice rychle. Po dokončení instalace budeme dotázáni, zda chceme aplikaci spustit nebo ne. Tím je proces instalace dokončen.



Obrázek 21 - Instalace aplikace

Závěr

Úkolem práce bylo ověřit, zda je možné naprogramovat klienta distribuované sítě pro operační systém Android pomocí knihovny Wi-Fi Direct, která umožňuje přímé spojení dvou a více zařízení přes bezdrátovou síť bez nutnosti propojení přes access point. Jednotlivé jednotky měly být schopné navazovat spojení se všemi okolními jednotkami a distribuovat si mezi sebou zprávy o údajích ze senzorů.

V průběhu návrhu jsem však začal zjišťovat, že tato představa je nereálná, jelikož knihovna Wi-Fi Direct tak, jak je poskytovaná, je programově spíše náhradou technologie Bluetooth než účinný nástroj pro komunikaci mezi jednotkami. Jak bylo řečeno výše, není možné programově, bez zásahu člověka, spojit více než dvě zařízení. Myšlenku, že by se jednotky mezi sebou propojovaly a pak zase odpojovaly po odeslání zprávy, jsem velice záhy opustil, jelikož testování propojování jednotek na reálných zařízeních ukázalo, že toto spojení není příliš kvalitní a dochází ke značnému množství selhání. Proto jsem v diplomové práci použil možnosti vytvoření skupiny zařízení, kdy jedna jednotka je serverem pro ostatní a stará se o propojení všech jednotek z okolí. Ve výsledku se tak dá říci, že server distribuuje všem jednotkám zprávy z celé sítě. Značně se tak ovšem zkrátil dosah sítě a zůstala pouze síť typu hvězda modelu klient – server.

Implementace samotného klienta byla velice náročná, protože kromě oficiální dokumentace, kde je knihovna velice zevrubně popsaná, neexistuje příliš mnoho materiálů, ze kterých by se daly vyčíst odpovědi na určité otázky. Ze začátku jsem tedy spoustu důležitých věcí pro implementaci musel odhadovat nebo dedukovat z těch pár komentářů a

dotazů, které se mi podařilo na Internetu k této problematice najít. Některé otázky implementace jsem tedy musel řešit testováním na reálných zařízeních a na demo aplikaci, která je volně dostupná na stránkách Android Developers (Google, 2013) a která slouží jako demonstrace základní funkčnosti knihovny Wi-Fi Direct. Nicméně i přes tuto možnost bylo testování časově náročné a vedlo pouze do slepé uličky, jelikož jsem tím pouze zjistil, že se propojování zařízení bez lidského zásahu realizuje jen těžko. Doslova dedukcí jsem usoudil, že by mohlo fungovat vytváření skupin a pomocí klasické serverové obsluhy spojení tak síť mezi jednotkami nakonec sestavit.

K tomu je použitá technologie Java NIO, která nakonec umožnila distribuci zpráv mezi jednotkami. I když je nutno říci, že zde se do role dostává opět knihovna Wi-Fi Direct. Spojení mezi jednotkami je občas značně nestálé, jelikož úplně přesně nefunguje detekce událostí na bezdrátové síti, a tak občas dochází k tomu, že se jednotky odpojují. A jelikož detekce událostí na bezdrátové síti se děje v jiném vlákně aplikace, než běží a musí běžet samotná serverová nebo klientská část aplikace, dochází tak k přerušování komunikace, kterou si server v některých případech nedokáže sám ohlídat.

Další nevýhodou knihovny Wi-Fi Direct je, že její komunikace je založená více na MAC adresách než na IP adresách, zatímco Java NIO používá klasicky IP adresy a čísla portů, po kterých se komunikuje. Tento problém je však řešen propagací několika proměnných napříč aplikací.

Abych to tedy shrnul. V práci se mi nakonec podařilo rozchodit funkční mechanizmus retranslace zpráv, které obsahují údaje z jednotlivých senzorů daného zařízení. Dosah sítě je však omezený dosahem jednotky, která vystupuje v roli serveru. Není tak možné do sítě připojit jednotku mimo dosah bezdrátové sítě serveru. Základní funkce distribuované sítě jsou funkční a bylo ověřeno, že knihovnu Wi-Fi Direct je k tomuto účelu možné použít, i když to rozhodně není snadná cesta.

Aplikace je teoreticky připravená na administrátorský zásah, respektive na zapínání a vypínání jednotlivých režimů, funkce nicméně není plně implementovaná, zbývá vyřešit grafické rozhraní pro tuto funkčnost. Stejným způsobem by bylo potřeba dořešit dispečerský zásah, který spočívá v přidání fragmentu, který by zobrazoval text a po stisku tlačítka generoval zprávu.

Literatura

Arlow, Jim a Neustadt, Ila. 2007. *UML 2 a unifikovaný proces vývoje aplikací.* Brno : Computer press, 2007. ISBN 978-80-251-1503-9.

Darwin, Ian. 2006. *Java Kuchařka programátora.* Brno : Computer Press, 2006. ISBN 80-251-0944-5.

Davison, Andrew. 2006. *Programování dokonalých her v Javě.* Brno : Computer Press, 2006. ISBN 80-7226-944-5.

Georgiana, Macariu. 2011. Parallel Algorithms Lab. *Google Sites.* [Online] Google, 2011. [Citace: 18. Srpen 2013.] https://sites.google.com/site/parallelalgorithmslab/mpi.

Google. 2013. Training. *Android Developer.* [Online] Google, 2013. [Citace: 18. Srpen 2013.] http://developer.android.com/training/index.html.

Karp, Michal. 2007. A Survey of Parallel Algorithms for Shared-Memory Machines. *EECS Technical Reports.* [Online] University of Illinois, 2007. [Citace: 18. Srpen 2013.] http://techreports.lib.berkeley.edu/accessPages/CSD-88-408.html.

Klimeš, Cyril. *Distribuované systémy.* Ostrava : Ostravská univerzita, Přírodovědecká fakulta, Katedra informatiky a počítačů.
http://www1.osu.cz/~prochazka/ds/SkriptaKlimes.pdf.

Mantena, Sudheer. 2006. Transparency in Distributed Systems. *crystal.uta.edu.* [Online] 2006. [Citace: 18. Srpen 2013.] http://crystal.uta.edu/~kumar/cse6306/papers/mantena.pdf.

Murphy, Mark. 2011. *Android 2: Průvodce programováním mobilních aplikací.* Brno : Computer Press, 2011. ISBN: 978-80-251-3194-7.

Object Management Group. 2013. OMG. *CORBA® BASICS.* [Online] 14. Duben 2013. [Citace: 18. Srpen 2013.] http://www.omg.org/gettingstarted/corbafaq.htm.

Oracle. 2013. Remote Method Invocation Home. *Oracle Technology Network.* [Online] Oracle, 2013. [Citace: 18. Srpen 2013.]
http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html.

Reilly, David. 2006. Introduction to Java RMI. *Java Coffee Break.* [Online] 5. Červenec 2006. [Citace: 18. Srpen 2013.]
http://www.javacoffeebreak.com/articles/javarmi/javarmi.html.

Příloha A – Diagram tříd celé aplikace