VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

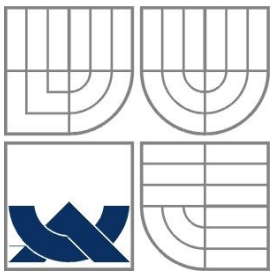
INTERPRET JAZYKA ALLL PRO OPERAČNÍ SYSTÉM ANDROID

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

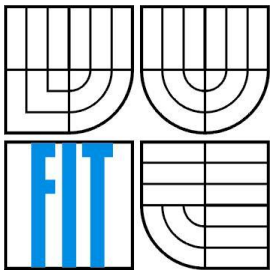
AUTOR PRÁCE
AUTHOR

Bc. DAN SKÁCEL

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

INTERPRET JAZYKA ALLL PRO OPERAČNÍ SYSTÉM ANDROID

INTERPRETER OF ALLL LANGUAGE FOR ANDROID

DIPLOMOVÁ PRÁCE
MASTERS'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. DAN SKÁCEL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN HORÁČEK

BRNO 2012

ZADANI

Abstrakt

Diplomová práce pojednává o tvorbě aplikace na mobilní zařízení s operačním systémem Android. Tato aplikace slouží jako interpret jazyka ALLL, to znamená, že vytváří z mobilního zařízení uzel schopný provozu v senzorové síti. V práci jsou nejprve vysvětleny obecné principy bezdrátových senzorových sítí, agenti a jazyk ALLL, které byly vyžity pro tvorbu této aplikace. Následně se věnuje postupu implementace a ukázkovým agentům, jejichž akce aplikace interpretuje.

Abstract

This master's thesis deals with creating an application for mobile devices with an Android operating system. The main task of this application is interpreting ALLL language commands. This allows any mobile device running this application to be a node of a wireless sensor network. First, well-known principles of wireless sensor networks, agents and ALLL language, which describes the agents in wireless sensor network, are explained in this project. The method for building the application on these bases follows. There are also some examples of agents interpreted by this application at the end.

Klíčová slova

Android, jazyk ALLL, agent, WSageNt, interpret, bezdrátová senzorová síť, ANTLR

Keywords

Android, ALLL language, agent, WSageNt, interpreter, wireless sensor network, ANTLR

Citace

Skácel Dan: Interpret jazyka ALLL pro operační systém Android, diplomová práce, Brno, FIT VUT v Brně, 2012.

Interpret jazyka ALLL pro operační systém Android

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Jana Horáčka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Dan Skácel
17. května 2012

Poděkování

Rád bych poděkoval panu Ing. Janu Horáčkovi za cenné rady a poskytnutou podporu při tvorbě této diplomové práce.

Obsah

1 Úvod

V dnešní době má již téměř každý člověk mobilní telefon. Neustále rozvíjející se doba však postupně nahrazuje klasické mobilní telefony vybavenějšími přístroji, tzv. chytrými telefony (*smartphone*). Tyto telefony jsou nejen lépe hardwarově vybaveny, ale mají také pokročilý operační systém a aplikační rozhraní, díky kterému můžeme do telefonu instalovat další aplikace.

Zlepšování hardwarové výbavy mobilních zařízení není již pouze o zvyšování taktu procesoru, ale také o přidávání nových technologií. Původně to byly Bluetooth a WiFi moduly pro bezdrátovou komunikaci, poté se začal stávat běžným modul GPS a nyní jsou to různé typy senzorů. Senzory v chytrých telefonech zatím plní pouze podpůrné funkce pro pohodlné ovládání telefonu. Ve skutečnosti však mají daleko větší možnosti.

V průmyslu se senzory, jako součásti mikrokontrolérů, používají již řadu let. Slouží především k získávání informací o okolním prostředí. Mohou měřit různé fyzikální veličiny, například teplotu, tlak, pohyb, vzdálenost apod. Skutečná síla senzorů není pouze v jediném uzlu, nýbrž v jejich vzájemném propojení a komunikaci. Tak vznikají senzorové sítě. Komunikace mezi uzly sítě může být buď drátová, nebo bezdrátová. Drátová komunikace se stává čím dál tím méně oblíbenou, neboť je nutné řešit vedení pevného spojení mezi uzly sítě. V této práci jsem se zabýval senzorovou sítí, ve které jsou jednotlivé uzly sítě tvořeny mobilními telefony, tudíž nás zajímá pouze bezdrátová komunikace. Uzly mezi sebou mohou komunikovat pomocí prostředků, které nabízí mobilní telefon. Jsou to například mobilní internet nebo již zmíněné Bluetooth či WiFi moduly.

Pokročilé operační systémy nám umožňují s telefony zacházet jako s klasickými počítači, musíme však počítat s menším výpočetním výkonem. Velký nárůst nejen na trhu, ale také na popularitě, získává operační systém Android. Chytré telefony s tímto operačním systémem se stávají čím dál tím víc běžnějším zařízením, a proto se tato práce zabývá tvorbou aplikace pro tento operační systém. Tato aplikace představuje interpret jazyka ALLL, který slouží k popisu agenta, jenž se vyskytuje na senzorovém uzlu v síti. Co je to agent a jak takový agent vypadá, si vysvětlíme později v tomto textu.

V následujících kapitolách si postupně rozebereme, co je Android, a jelikož se budeme zabývat senzorovou sítí, přiblížíme si taky funkce a způsoby, které nám dovolí se senzory v mobilních telefonech pracovat. Následovat bude kapitola o bezdrátových senzorových sítích, jejich topologiích a způsobu směrování paketů uvnitř sítě. Poté si vysvětlíme, co je agent a jak se takový agent chová v senzorových sítích. Dále si přiblížíme platformu *WSageNt* a podíváme se na strukturu jazyka ALLL. Před samotným popisem implementace se budeme věnovat knihovně ANTLR a prostředí ANTLRWorks pro definici gramatiky jazyka. V popisu implementace se zaměříme na bezdrátovou komunikaci přes Bluetooth rozhraní a datové struktury pro uložení jednotlivých částí agenta. Poté si představíme vzhled celkové aplikace pro operační systém Android a její nezbytné části. Na to navážeme popisem serveru, se kterým tato aplikace komunikuje. V závěru se budeme zabývat ukázkami agentů a testováním aplikace.

2 Platforma Android

Android není pouze operační systém, nýbrž rozsáhlá *open source* platforma, která vznikla pro mobilní zařízení, jako jsou chytré mobilní telefony, tablety apod. Na jeho vývoji se podílí konsorcium Open Handset Alliance vedené společností Google. Samotná platforma Android dává k dispozici nejen operační systém s uživatelským rozhraním, založeném na Linuxovém jádře, ale také kompletní řešení včetně ovladačů pro nasazení operačního systému na odlišná hardwarová zařízení různých výrobců. Jelikož je Android *open source* platforma, nabízí vývojářům efektivní vývojové prostředky pro vývoj nových aplikací prostřednictvím vývojového balíku SDK (Software Development Kit).

2.1 Vývoj pro Android

Vývoj aplikací pro Android není omezený pouze na jedinou platformu a můžeme tedy použít jak operační systém Windows, Linux, tak i Mac OS. K vývoji aplikací pro Android se využívá vývojový balík Android SDK, který poskytuje nezbytné nástroje a API, za použití programovacího jazyka Java. Pro úspěšné nainstalování balíku Android SDK jsou potřebné další dva softwarové systémy: Java Development Kit (JDK) a vývojové prostředí Eclipse (IDK). Tyto dva systémy nejsou součástí balíku Android SDK, protože mohou být používány také samostatně. Zatímco JDK je nezbytnou součástí instalace Android SDK, vývojové prostředí Eclipse je pouze doporučené a může být nahrazeno jiným vývojovým prostředím [1].

Abychom byli schopni vyvíjet aplikace pro Android, musíme si vybrat verzi Androidu, pro kterou se bude naše aplikace překládat. To může způsobit značné komplikace. Čím nižší verzi si vybereme, tím bude naše aplikace více kompatibilní se staršími telefony, ale budeme mít k dispozici nižší verzi API, ve které mohou chybět potřebné funkce. O konkrétnějších problémech se postupně zmíním v dalších kapitolách.

2.2 Senzory

Při tvorbě této kapitoly, popisující práci se senzory na platformě Android, jsem čerpal z [1]. Moderní chytré telefony umožňují uživateli víc, než jen komunikovat s ostatními uživateli pomocí hovorů, zpráv a sociálních sítí. Přidání externích senzorů, které mohou referovat o prostředí, ve kterém se telefon nachází, vytváří z telefonu výkonnější a užitečnější zařízení jak pro uživatele, tak i pro vývojáře. SW podpora pro práci se základní množinou senzorů je dostupná od verze Androidu 1.5 (API verze 3). Mezi základní podporované senzory můžeme zařadit akcelerometr, který měří akceleraci v jednotlivých souřadnicových osách, gyroskop, který měří rotační změnu zařízení kolem těchto os. Dále pak senzor, měřící sílu magnetického pole, světelný senzor, který zjišťuje intenzitu okolního osvětlení, proximity senzor, který udává vzdálenost nejbližšího objektu od zařízení, tepelný senzor, který měří okolní teplotu a také tlakový senzor, který se chová jako barometr. Hodnoty naměřené na těchto senzorech jsou fyzikální síly, které na senzory působí a pro jejich následné použití je potřeba výsledky dále zpracovat. Zpracování může probíhat kombinováním získaných dat z různých senzorů nebo provedením potřebného výpočtu. Jako příklad si uveďme hodnoty získané z gyroskopu. Tyto hodnoty jsou uvedené v jednotkách úhlové rychlosti, ale nás zajímá pouze úhel, o který byl telefon pootočen. Získané hodnoty ze senzoru tak musíme integrovat přes časový úsek.

Pro přístup k senzorům poskytuje Android vhodnou systémovou službu nazvanou SensorManager, kterou získáme voláním metody `getSystemService()` s argumentem `Context.SENSOR_SERVICE` pro senzorové služby. Pro přístup k jednotlivým senzorům jsou dostupné metody `getSensorList()` a `getDefaultSensor()`. Také zde je potřeba argumentem `Sensor.TYPE_AAA` specifikovat typ senzoru[1]. Ve chvíli, kdy máme vybraný daný senzor, je potřeba u něho povolit sběr dat. Sběr dat povolíme zaregistrováním *listeneru*. Data ze senzoru budeme dostávat v intervalu, který si zvolíme v argumentu. Můžeme si zvolit některý z následujících intervalů:

- `SENSOR_DELAY_NORMAL`
- `SENSOR_DELAY_UI` - interval vhodný pro běžnou práci s uživatelským rozhraním.
- `SENSOR_DELAY_GAME` - velice krátký interval dostačující pro ovládání her.
- `SENSOR_DELAY_FASTEST` - posílej data, jak nejrychleji můžeš.
- vlastní interval zadaný v jednotkách milisekund.

```
SensorEventListener listener = new SensorEventListener()
{
      @Override
      public void onAccuracyChanged(Sensor sensor, int Accuracy) { }

      @Overide
      public void onSensorChanged(SensorEvent event) { }
}
```

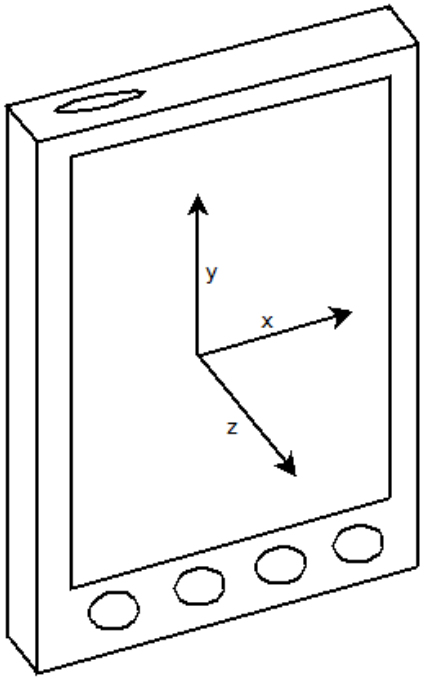
Metody `onAccuracyChanged()` a `onSensorChanged()` jsou volány ve chvíli, kdy jsou data z dotyčného senzoru dostupná. První z nich je volána, kdykoliv se na senzoru vyskytne změna stupně chyby nebo přesnosti. Druhá metoda je zajímavější, neboť nám poskytuje data ze senzoru zabalená do objektu typu `SensorEvent`.

Jelikož se pořád bavíme o mobilních zařízeních, která jsou napájená baterií, je velice důležité šetřit její energii. Proto je potřebné vypnout senzor (odregistrovat *listener*) ve chvíli, kdy už jej nepotřebujeme. Například v době, kdy je pozastavena aktivita. Jinak bude docházet k neustálému odběru energie z napájecí baterie. Žádný uživatel mobilního telefonu by nebyl spokojený, pokud by se mu baterie jeho přístroje vybila během dne, aniž by přístroj využíval. Systém se o vypnutí senzoru nepostará ani v případě vypnutí displeje zařízení a je to tedy celé v režii programátora, který musí vše zajistit ručně pomocí metody `unregisterListener(listener)`.

2.2.1 Pozice

Jak jsem se zmínil již dříve, některé senzory vrací hodnoty naměřené v jednotlivých osách. Je tedy potřeba definovat souřadný systém. Souřadný systém vychází ze základní orientace telefonu a jeho displeje. Osy x, y a z jsou zobrazeny na obrázku 2.1. Když uživatel hýbe telefonem, osy následují jeho pohyb a nevyměňují si pozice. Jejich orientace je následující:

[1] `AAA` je zastoupeno reálným názvem senzoru.



Obrázek 2.1: Souřadný systém telefonu.

Osa x

Horizontální osa, kde kladné hodnoty jdou doprava a záporné doleva.

Osa y

Vertikální osa. Kladné hodnoty se pohybují směrem nahoru a záporné směrem opačným.

Osa z

U této osy kladné hodnoty vystupují směrem z telefonu dopředu a záporné směřují dozadu za telefon.

Přesnost různých senzorů, a tedy odchylka od skutečných hodnot, záleží na kvalitě hardware. V mnoha případech je měření výrazně ovlivněno šumem, který musí být odstraněn. Jako příklad pro eliminaci šumu si můžeme uvést filtr typu dolní propust, ale záleží to čistě na vývojáři, jaký postup si zvolí.

Akcelerometr

Akcelerometr měří zrychlení zařízení a vrací výsledky v trojrozměrném vektoru `value`, ve třech výše zmíněných osách (`value[0]` pro osu x, `value[1]` pro osu y a `value[2]` pro osu z). Hodnoty jsou udávány v jednotkách SI m/s^2. Je důležité upozornit, že z hodnot získaných ze senzoru není odstraněna gravitační síla. Leží-li zařízení na stole displejem směrem nahoru, bude hodnota vyčtená z `value[2]` rovna gravitační konstantě 9,81 m/s^2.

V dnešní době je běžná potřeba eliminovat gravitační sílu ve všech osách, proto byla do operačního systému Android od verze 2.3 (API 9) přidána SW podpora pro lineární akcelerační senzor a gravitační senzor. Lineární akcelerační senzor je senzor, který zjednodušuje výpočty při práci s akcelerometrem. Vrací výsledky v trojrozměrném vektoru. Každá složka vektoru udává zrychlení zařízení v příslušné ose, ale bez gravitační síly. To znamená, že hodnoty udávají zrychlení v každé ze tří os, ale bez efektu gravitace. Tento způsob velmi zjednodušuje odstranění gravitační konstanty a tím pádem zjištění skutečného zrychlení při používání mobilního telefonu na Zemi.

Naproti tomu hodnoty vrácené v trojrozměrném vektoru z gravitačního senzoru jsou ovlivněné směrem a velikostí gravitace.

Gyroskop

Gyroskop měří úhlovou rychlost nebo míru rotace okolo tří os. Všechny hodnoty jsou v radiánech za sekundu. Rotace je pozitivní proti směru hodinových ručiček a negativní ve směru hodinových ručiček. Abychom zjistili úhel, který má pro nás větší informační hodnotu, je potřeba hodnoty integrovat přes časový interval.

Je to podobný problém jako u akcelerometru, proto i zde byla od verze Androidu 2.3. přidána SW podpora pro senzor rotačního vektoru. Rotační vektor reprezentuje orientaci zařízení jako kombinaci úhlu a osy otáčení, ve které bylo zařízení rotováno o úhel θ v prostoru $< x, y, z >$. I když mohl být rotační vektor počítán pomocí hodnot z gyroskopu, mnoho vývojářů ho potřebovalo a počítalo tak často, že se společnost Google rozhodla pro usnadnění přidat jeho podporu.

Tři složky rotačního vektoru jsou $< x * sin(\theta/2), y * \sin(\theta/2), z * \sin(\theta/2) >$. Rozsah rotačního vektoru je roven $\sin(\theta/2)$ a jeho směr je určen směrem osy rotace. Složky rotačního vektoru odpovídají posledním třem složkám kvaternionu a jsou bezrozměrné. Ačkoli není použití kvaternionu moc známé, je to nejjednodušší způsob, jak vypočítat rotaci okolo osy v trojrozměrném prostoru. Především díky tomu, že skládání rotací odpovídá násobení příslušných kvaternionů. Mnohem známější reprezentace pomocí Eulerových úhlů je v tomto případě podstatně složitější.

2.2.2 Světelný senzor

Tento senzor vrací hodnotu pouze v jednorozměrném poli (`value[0]`), která vypovídá o intenzitě okolního osvětlení. Tato hodnota je uváděna v jednotkách SI lux. Informace o okolním osvětlení se u mobilních telefonů využívá především k upravení jasu podsvícení displeje dle prostředí, ve kterém se nacházíme.

2.2.3 Magnetický senzor

Magnetický senzor měří úroveň okolního magnetického pole ve všech třech osách. Naměřené hodnoty jsou uváděny v jednotkách mikroTesla.

2.2.4 Tlakový senzor

Mnoho zařízení tlakový senzor vůbec nemá, ale Android jej podporuje. Pro daná zařízení, která jej obsahují, jsou hodnoty udávány v jednotkách kiloPascal.

2.2.5 Proximity senzor

Je to senzor, který měří vzdálenost objektu od senzoru v centimetrech a vrací ji v jednorozměrném poli (`value[0]`). Tento senzor se v mobilních telefonech využívá především pro zhasnutí displeje při přiblížení telefonu k uchu. Některé senzory proto vrací místo vzdálenosti v centimetrech pouze dvě hodnoty "blízko" (0) a "daleko" (1).

2.2.6 Teplotní senzor

Teplotní senzor, stejně jako tlakový, se vyskytuje pouze v malém počtu zařízení. Naměřené hodnoty jsou udávány ve stupních Celsia.

2.3 Lokalizace mobilního telefonu

Možnost lokalizovat pozici telefonu umožňuje vývojářům vyvíjet "chytřejší" aplikace, které mohou uživatelům poskytovat lepší informace. Do bezdrátových senzorových sítí může vnést přesná lokalizace uzlu velmi užitečné informace. Zjišťování pozice uzlů sítě podle síly signálu není zdaleka tak přesné, jako využití GPS senzoru. Ačkoli je zeměpisná pozice získaná pomocí GPS senzoru ta nejpřesnější, ne vždy je však dostupná, protože GPS signál funguje pouze venku. Proto nám platforma Android poskytuje také jiné možnosti lokalizace telefonu, například pomocí připojení k GSM či WiFi síti.

Pro přístup k GPS senzoru použijeme systémovou službu LocationManager, kterou získáme voláním metody `getSystemService()` s argumentem `Context.LOCATION_SERVICE` pro lokalizační služby. Sběr dat z GPS senzoru povolíme zaregistrováním *listeneru*, pomocí funkce `requestLocationUpdates()`, která má 4 argumenty:

- `String provider` - představuje název poskytovatele lokalizace. `GPS_PROVIDER` slouží pro lokalizaci pomocí senzoru GPS. Pro lokalizaci pomocí připojení k síti použijeme `NETWORK_PROVIDER`.
- `Long minTime` - tímto argumentem můžeme nastavit dobu v milisekundách, kterou LocationManager počká před každou aktualizací pozice. Pomocí delších intervalů pro aktualizaci tak můžeme šetřit energii baterie.
- `Float minDistance` - představuje minimální změnu vzdálenosti v metrech potřebnou pro aktualizování pozice.
- `LocationListener locationListener` - je název metody *listeneru*, která je zavolána při každé aktualizaci pozice. V těle *listeneru* musí být implementovány 4 metody, které jsou uvedeny v následujícím zdrojovém kódu.

```
LocationListener locationListener = new LocationListener()
{
   public void onLocationChanged(Location location) {}

   public onStatusChanged(Sting provider,int status,Bundle extra){}

   public void onProviderEnabled(String provider) {}

   public void onProviderDisabled(String provider) {}
} ;
```

Nejdůležitější je metoda `onLocationChanged()`, která vrátí aktualizovanou pozici telefonu. Další tři metody se týkají změny nastavení poskytovatele lokalizace. Stejně jako u ostatních senzorů, i v tomto případě musíme kvůli šetření energie baterie vypnout GPS senzor (odregistrovat *listener*) ve chvíli, když už jej nepotřebujeme.

Abychom mohli v naší aplikaci použít výše zmíněný postup pro lokalizaci telefonu, je potřeba povolit lokalizaci ve speciálním souboru Manifest příkazem:

```
<uses-permision android:name =
"android.permission.ACCESS_FINE_LOCATION" />
```

3 Bezdrátové senzorové sítě

Senzory spojují fyzický svět s digitálním tak, že zachycují jevy reálného světa a převádějí je do formy, ve které mohou být zpracovány a uloženy. Senzory se v dnešní době integrují do mnoha prostředí a zařízení, a tak přinášejí společnosti obrovskou výhodu. Mohou nám pomoci zabránit katastrofickým poruchám infrastruktury, zachovat přírodní zdroje, zvýšit produktivitu, zdokonalit bezpečnost nebo pomoci vytvořit "chytrá prostředí" [11].

Chytrá prostředí můžeme chápat jako další technologický krok, využívající se například v budovách, průmyslových centrech, domácnostech, na lodích, v transportních systémech apod. Tak, jako každý živý organismus, i chytrá prostředí spoléhají především na data z reálného světa získaná pomocí smyslů - senzorů. Data nepřicházejí jen z jednoho senzoru, nýbrž z mnoha rozmístěných v prostředí, které zkoumáme. Chytrá prostředí potřebují informace o okolí stejně tak, jako informace o vnitřním chodu. Můžeme to přirovnat k rozdílu mezi exteroreceptory a proprioreceptory v biologickém systému. Informace, které chytrá prostředí potřebují, jsou poskytovány bezdrátovými senzorovými sítěmi (WSN)[2] [12].

3.1 Metriky směrování

Bezdrátové senzorové sítě a jejich aplikace se značně liší ve své charakteristice a omezeních, která musí být brána v úvahu při vytváření směrovacího protokolu. Většina WSN je omezena příkonem energie, výpočetním výkonem a úložnou kapacitou. Senzorové sítě se využívají v různých lokalitách, které se od sebe značně liší. Globální adresovací schéma, jak je známe například z internetu pomocí IP adres, je v tomto případě nevhodné, ba dokonce nemožné, zvláště pak v sítích s mobilními uzly. Z aplikačního hlediska mohou být data ze senzorů měřena pomocí různých přístupů:

- Schéma řízené časem - uzly sítě posílají data ze senzorů na základní stanici periodicky v časových intervalech. Například monitorování prostředí.
- Schéma řízené událostmi - uzly podávají hlášení o nasbíraných datech pouze ve chvíli, když nastane příslušná událost.
- Schéma řízené dotazem - uzly poskytují naměřená data pouze na dotaz od základní stanice.

Bez ohledu na použité schéma v síti, návrh směrovacího protokolu záleží na dostupných zdrojích a na účelu, který od sítě požadujeme. Nyní si některé směrovací mechanismy představme [11].

3.1.1 Metrika Minimum Hop[3]

Nejznámější metrika používaná ve směrování je hledání nejkratší cesty. Směrovací protokol se snaží nalézt cestu v síti přes nejmenší možný počet směrovacích uzlů. V této jednoduché technice mají všechny přechody stejnou cenu a protokol vybere tu cestu, která má nejmenší celkovou cenu od zdroje k cíli. Základní myšlenkou tohoto protokolu je při použití nejkratší cesty docílení nejmenšího možného zpoždění a minimalizace spotřeby zdrojů. Bohužel tato technika směrování nebere na vědomí dostupnost zdrojů v jednotlivých uzlech, což může často vyústit v zahlcení sítě. Zvolená cesta pak není optimální z pohledu zpoždění, ani z pohledu spotřeby energie. Přesto je tato metrika ve

[2] WSN - Wireless Senzor Network

[3] hledání nejkratší cesty

směrovacích protokolech používána, především díky své jednoduchosti a schopnosti udržet pořadí už oceněných cest, které jsou prodloužené o společnou část cesty.

3.1.2 Metriky dle využití energie

Využití energie je při směrování jeden ze stěžejních aspektů. Neexistuje však jednotná metrika pro nejefektivnější řízení spotřeby energie. Místo toho jsou používány různé interpretace:

1. Minimální spotřebovaná energie na paket: Toto je asi nejpřirozenější koncept pro efektivní využití energie. Cílem je minimalizovat množství vynaložené energie na přenesení jednoho paketu od zdroje k cíli. Výsledná energie je suma vynaložené energie na každém uzlu cesty při přijímání a přeposílání tohoto paketu.
2. Nejdelší doba k rozdělení sítě: Když poslednímu uzlu, který spojoval dvě části sítě, dojde energie nebo selže, je síť rozdělena na dvě podsítě. Snahou tohoto protokolu je omezit spotřebu energie v klíčových uzlech, které spojují části sítě do jednotného celku. Je totiž potřeba, abychom byli schopni dosáhnout každého uzlu alespoň jednou cestou.
3. Minimální odchylka uzlů v kapacitě energie: V tomto případě jsou všechny uzly považovány za stejně důležité a snahou je rozdělit spotřebu energie všech uzlů rovnoměrně. Cílem je prodloužení životnosti celé sítě místo jejího postupného zmenšování kvůli vybití baterie jednotlivých uzlů. V ideálním případě, ale prakticky nemožném, dojde energie všem uzlům zároveň.
4. Maximální (průměrná) kapacita energie: Tento přístup se nezajímá o množství spotřebované energie při přenosu paketu, ale o zbývající kapacitu baterií jednotlivých uzlů. Vybere tu cestu, jejíž součet kapacity baterií má nejvyšší dostupnou energii.
5. Maximální z minimálních kapacit energie: Poslední přístup, který si vysvětlíme, vybírá tu z cest, jejíž uzel s minimální kapacitou baterie má největší kapacitu vzhledem k ostatním minimálním kapacitám na uzlech ostatních cest.

3.1.3 Kvalita služeb

Kvalita služeb (QoS[4]) vychází z měření výkonu v sítích. Výkon posuzujeme především podle zpoždění přenášených paketů, propustnosti sítě, jitteru[5] a ztráty paketů. Pro různé senzorové sítě se nastavuje různá kvalita služeb. Například v sítích, které se zaměřují na detekcí cíle a jeho sledování, bude požadováno zpoždění přenosu co nejmenší. Oproti tomu v multimediálních sítích budeme zase požadovat vysokou propustnost. Jako společná metrika pro vyjádření zpoždění paketů při přenosu se používá ETT[6]:

$$ETT = ETT \times \frac{S}{B}$$

kde S je průměrná velikost paketu a B je šířka pásma. Většinou se nepoužívají jen samostatné QoS metriky, ale jejich kombinace (např. zpoždění a ztráta paketů). Jaké metriky jsou nakonec vybrané, záleží na typu WSN. Vždy je však potřeba brát zřetel nejen na kvalitu služeb, ale také na spotřebu energie [11].

[4] Quality of Service - QoS
[5] kolísaní zpoždění paketů
[6] Expected Transmission Time - očekávaná doba přenosu

3.1.4 Robustnost

Mnoho senzorových sítí si přeje komunikovat přes dlouhodobě stabilní a spolehlivé cesty. K dosažení tohoto požadavku může uzel měřit či odhadovat kvalitu spojení ke každému svému sousedovi. Ke směrování paketu si poté vybere ten uzel, který má největší pravděpodobnost úspěšného přenosu. Tato metrika se ale málokdy používá ke směrování samostatně. Většinou se může vyskytovat ve spojení s metrikou nejkratší cesty, kde se nejprve určí více "nejkratších" cest a z nich se vybere ta nejspolehlivější.

3.2 Topologie WSN

V předchozí podkapitole jsme si vysvětlili některé principy směrování v bezdrátových senzorových sítích. Zmínili jsme se také o QoS. Nyní se podívejme na základní topologie WSN, volených v závislosti na požadovaných QoS, prostředí, ve kterém síť instalujeme, ceně a účelu použití sítě. Každý uzel v síti obsahuje senzory, výpočetní jednotky a rádio, pomocí kterého přijímá a odesílá zprávy.

Jako první topologii si uveďme plně propojenou síť. Plně propojená síť však zdaleka není ideální pro používání. S každým přidaným uzlem do sítě roste exponenciálně počet propojení jednotlivých uzlů. Ve velmi rozsáhlých sítích pak může být směrování výpočetně neřešitelné, i při dostupnosti vysoké výpočetní síly.

Další používanou topologií jsou *mesh* sítě. *Mesh* sítě jsou rovnoměrně rozložené sítě, které umožňují posílat zprávy jen nejbližším sousedním uzlům. Uzly v těchto sítích jsou většinou identické, takže bývají označovány také jako *peer-to-peer* sítě. Díky tomu, že v těchto sítích existuje více různých cest pro směrování paketů, jsou tyto sítě obecně odolnější proti výpadku jednotlivých uzlů. Další výhodou je fakt, že ačkoliv jsou všechny uzly sítě identické (mají stejnou výpočetní sílu i schopnost posílat zprávy), vybrané uzly mohou být určené jako vůdčí a dostávají tak dodatečnou funkcionalitu. Pokud dojde k výpadku vůdčího uzlu, může jeho funkcionalitu převzít jiný uzel.

Topologie, ve které jsou všechny uzly připojené k jednomu vůdčímu uzlu[7], se nazývá hvězdicová. Hub je oproti ostatním uzlům vybaven lepšími schopnostmi pro zpracování zpráv, směrování a rozhodování. Pokud je přerušena komunikační cesta, ovlivní to pouze jediný uzel. Pokud ale dojde k chybě na hubu, je tím ovlivněna celá síť. Podobnou topologií je topologie kruhová, ve které mají uzly stejnou funkcionalitu, ale chybí zde vůdčí uzel. Zprávy zaslané mezi uzly kolují dokola v kruhu, ale pouze v jednom směru. Pokud dojde k přerušení cesty mezi dvěma uzly, je veškerá komunikace ztracena. Jako vylepšení této topologie vznikla samoléčící se kruhová topologie, která obsahuje dva komunikační okruhy.

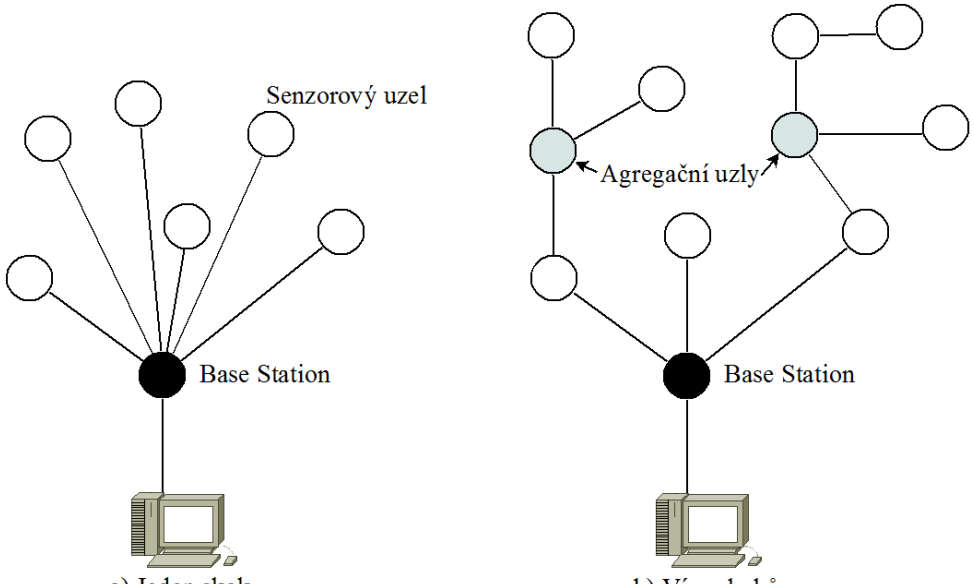
Poslední topologii, kterou si uvedeme je topologie sběrnicová. Zprávy jsou zasílány metodou *broadcast* přes společnou sběrnici všem uzlům. Každý uzel zkontroluje hlavičku příchozí zprávy. Pokud je zpráva určena jemu, zpracuje ji, jinak se o ni nezajímá. Topologie sítě je oproti ostatním topologiím pasivní v tom smyslu, že každý uzel pouze poslouchá na sběrnici a není zodpovědný za přesměrování zprávy [12].

[7] nazývá se hub

3.3 Komunikace uzlů

Nejznámější rodina standardů IEEE 802.11 pro bezdrátovou komunikaci, známá jako WiFi, se využívala spíše v počátcích WSN. Dnes se s ní také můžeme setkat, ale pouze u sítí s vysokými nároky na přenosové pásmo, například multimediálních. Od podstatné většiny WSN se však očekává vysoká výdrž jejich uzlů, které jsou napájené malými bateriemi. Použiti IEEE 802.11 je energeticky náročné, takže je v těchto sítích prakticky nepoužitelné. Navíc požadavky na data v senzorových sítích nejsou ani zdaleka tak velké, jaké nám standardy 802.11 poskytují. To vše vedlo k vývoji standardů nových a pro použití v bezdrátových senzorových sítích vhodnějších. Především díky menší spotřebě energie. Standard, který byl navržen především pro komunikaci na krátké vzdálenosti a s nízkou spotřebou energie, nese označení IEEE 802.15.4 a je dnes podporován většinou uzlů ve WSN.

Pokud je přenosový dosah rádií ve všech senzorových uzlech sítě dostatečně velký a každý uzel může přenášet data rovnou do *base station*[8], může být síť formována do hvězdicové topologie. Komunikace nemusí být směrována přes sousední uzly a každý uzel komunikuje pouze s *base station* (jeden skok), viz. obrázek 3.1.a. Většinou však senzorové sítě pokrývají rozlehlá území a síla rádiového signálu je omezena kvůli prodloužení energetické výdrže uzlu napájeného baterií. Poté je potřeba komunikaci vzdálenějších uzlů od *base station* směrovat přes ostatní uzly v síti (více skoků). Jako příklad si můžeme uvést *mesh* topologii, ve které se uzly kromě snímání a šíření svých dat podílejí také na přenosu dat z okolních uzlů. Ukázka na obrázku 3.1.b.



Obrázek 3.1: Ukázka senzorových sítí. Jedno skoková vs. více skoková komunikace. Čerpáno z [11].

[8] základní stanice, nebo také hub. Jedná se o uzel, který je připojen k PC

4 Agenti pro bezdrátové senzorové sítě

V této kapitole si nejprve vysvětlíme, co je to agent a jaké má vlastnosti. Poté si popíšeme existující agentní platformu *WSageNt* a nakonec si rozebereme jazyk ALLL, který slouží pro popis a komunikaci s agenty na uzlech WSN.

4.1 Agenti

Slovo agent pochází z latinského agere - jednat. I když se pojem agent vyskytuje v mnoha oblastech, například v umělé inteligenci, robotice, agent CIA apod., vždy pod tímto pojmem můžeme rozumět nějakou entitu, která jedná ve prospěch někoho dalšího. Agent může být cokoliv, o čem můžeme říct, že vnímá svoje okolí pomocí senzorů a reaguje na něj pomocí aktivátorů. Od počítačových agentů se očekává, že budou mít další vlastnosti, které je rozliší od pouhých počítačových programů, jako například schopnost jednat autonomně, vnímat prostředí, vydržet v daném prostředí delší časové období, přizpůsobovat se změnám a mít schopnost začít pracovat na jiném cíli. Racionální je takový agent, který umí v daném okamžiku jednat tak, aby dosáhl svých cílů [2]. Jako základní vlastnosti agentů podle [3] si uveďme:

- **Autonomnost** - Jednají bez přímého zásahu člověka.
- **Sociální schopnosti** - Agenti jsou schopni komunikovat s ostatními agenty pomocí určitého agentního jazyka.
- **Reaktivita** - Schopnost vnímat prostředí a reagovat na jeho podněty včas.
- **Proaktivita** - Agenti nejednají pouze v reakci na prostředí, ale jsou schopní převzít iniciativu za účelem dosažení naplánovaných cílů.

4.1.1 BDI Agenti

BDI je softwarový model, vytvořený pro programování inteligentních agentů. BDI agent je speciální případ deliberativního[9] agenta, jehož chování vychází z jeho představ (Beliefs), přání (Desires) a záměrů (Intentions). Důležitou součástí BDI agentů je plánovač. Ten na základě přání a záměrů sestavuje plán, jak daných cílů dosáhnout. Pro konkrétní záměry však mohou být plány sestaveny již dopředu a v průběhu jednání agenta jsou už pouze vybírány z databáze. BDI agenti jsou schopni vhodně rozvrhnout svůj čas mezi volbou plánu a vykonáváním již vybraného plánu. Architektura modelu BDI je rozdělena do tří částí [4]:

- **Beliefs** - získané představy agenta o reálném světě, nemusí být pravdivé a mohou být proměnlivé.
- **Desires** - přání a touhy, kterých by agent chtěl dosáhnout. Mohou být krátkodobé nebo dlouhodobé, ty dlouhodobé jsou označované jako cíle (goals). Ne všech cílů je agent vždy schopen dosáhnout.
- **Intentions** - záměry, které se agent rozhodl vykonávat, neboli vykonávání některého z plánů.

[9] deliberativní agenti odstraňují některé nedostatky nejjednodušších, pouze reaktivních agentů

4.2 Platforma WSageNt

O tom, co je to agent, jsem se zmínil již v kapitole 4.1 a bezdrátové senzorové sítě jsou popsané v kapitole 3. My se ale snažíme o přítomnost agenta přímo v uzlech senzorové sítě. Agentem rozumíme program společně s jeho architekturou. Je však nutné řešit problém, jak přizpůsobit uzel v senzorové síti tak, aby byl schopen hostit běh agenta. Přivedení agentů do světa WSN znamená, že uzly musí být programovatelné systémy, které jsou schopny interpretovat agentní program s ohledem na architekturu senzorového uzlu. Z tohoto důvodu bylo potřeba vytvořit platformu, která se nachází mezi agentem a hardwarem senzorového uzlu. Řešení, vytvořené na VUT FIT jako téma, bakalářských, diplomových a dizertačních prací, se jmenuje *WSageNt* [8].

Již z názvu je tedy patrné, že *WSageNt* je agentní platforma, která běží na uzlech senzorové sítě. Podporované uzly této platformy, nazývané *mote*, jsou v současné době zařízení od firmy CrossBow: MicaZ a Iris *motes*. V případě nepřítomnosti hardwarových zařízení lze platformu *WSageNt* také spustit v simulátorech TOSSIM nebo T-Mass [9]. Pro popis agenta je použit jazyk ALLL, který je popsaný v kapitole 4.3. Proto je na této platformě přítomný interpret tohoto jazyka.

Při vývoji aplikací pro bezdrátové senzorové sítě se využívá na zdroje nenáročný operační systém TinyOS. Stejně tak tomu bylo i v případě platformy *WSageNt*. Používaný programovací jazyk v TinyOS je NesC, který je odvozený od známého jazyka C. Hlavní odlišností je komponentní návrh, kde každá komponenta představuje samostatnou funkční jednotku, která má své rozhraní, jehož pomocí komunikuje s ostatními komponentami [5].

Jednou ze základních utilit, ale pro vytváření WSN podstatných, je *base station*, která se chová jako most mezi sériovým portem a bezdrátovou sítí. *Base station* může být libovolný *mote*, který je připojen k počítači různými způsoby (COM, USB, bezdrátově apod.). Přes *base station* tak probíhá veškerá komunikace počítač - bezdrátová síť a naopak. Každá zpráva z PC je nejprve zaslána na *base station*, která ji poté přepošle na daný uzel v síti.

Pro představu si uveďme základní technické parametry uzlu MicaZ. Obsahuje procesor o frekvenci 8 MHz. Pro komunikaci využívá standardu IEEE 802.15.4/ Zigbee, který komunikuje na frekvenci 2.4 - 2.4835 GHz. Dále obsahuje sériové rozhraní UART a 10 bitový ADC převodník. Každý *mote* lze pomocí 51 pinového rozhraní rozšířit například o senzorovou desku.

4.2.1 Struktura a služby platformy

Celý systém *WSageNt* obsahuje senzorový uzel, agenta, ALLL jazyk pro popis agenta a případně další nezbytné komponenty pro správnou funkčnost v systémech WSN. Platforma jako taková je složená z několika modulů, které zajišťují správný běh agenta na jednotlivém uzlu. Pojďme se nyní na některé nezbytné moduly podívat.

Modul pro přenos zpráv je zodpovědný za správné doručení zpráv vybranému agentovi. Základní rozhraní operačního systému TinyOS poskytuje rozhraní pro přenos zpráv o maximální délce 28 bytů. Často je však potřeba posílat delší zprávy. Zpráva je potom rozdělena do jednotlivých paketů, které jsou odeslány na uzel příjemce, kde jsou složeny zpět do původní zprávy.

Jednotka pro mobilitu agentů se stará o pohyb agenta mezi jednotlivými uzly senzorové sítě. Jedná se o kopírování nebo přemístění agenta z uzlu na uzel. Kód popisující agenta je často delší než 28 bytů, takže je princip podobný jako u posílání zpráv.

Dále je zde modul pro měření dat ze senzorů a jejich ukládání do logu. Je možné aktivovat asynchronní proces, který v určeném intervalu periodicky čte data z vybraného senzoru, aniž by to ovlivňovalo interpretování kódu agenta.

O interpretování kódu agenta se stará interpret jazyka ALLL. Interpretování kódu agenta běží ve smyčce tak dlouho, dokud jsou v zásobníku vykonávaného plánu přítomné nějaké akce.

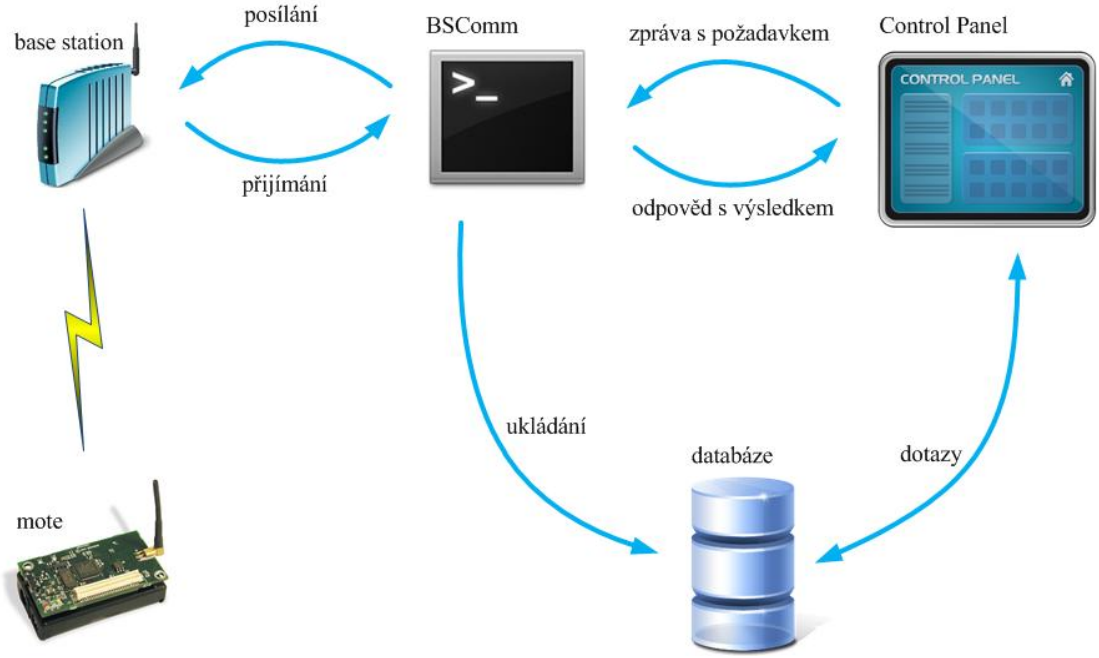
Jako poslední si popišme jednotku poskytující služby agentovi. Agent využívá služeb platformy tak, že je zavolá během vykonávání svého plánu. Jako příklad si můžeme uvést výstup na některou ze tří diod, aritmetické nebo logické operace, požadavky na migraci agenta na jiný uzel, ukládání dat ze senzorů do logů apod. [8].

4.2.2 Uživatelské webové rozhraní Control Panel

Postupem času bylo k této platformě vytvořeno grafické uživatelské rozhraní pro pohodlnější ovládání a zjišťování informací ze senzorové sítě. Toto GUI se nazývá *Control Panel* a bylo vytvořeno Bc. Martinem Gáborem jako téma diplomové práce [10]. Hlavní činností tohoto rozhraní je monitorování provozu a spravování topologie sítě.

Pomocí rozhraní *Control Panel* může uživatel definovat agenta a odeslat jej na vybraný uzel. V rámci komunikace může agentům posílat zprávy nebo vyzvedávat zprávy přicházející na *base station*. Pomocí speciálních zpráv zasílaných na uzly lze zjistit sílu signálu, na jejímž základě lze zobrazit topologii sítě.

Celý systém je složen ze dvou samostatně spustitelných částí: *Control Panel* a *BSComm*. Aplikace *Control Panel* je vytvořena jako webové rozhraní v programovacím jazyce Java EE s pomocí *open source frameworku* Struts 2. Jelikož se zde jedná o informační systém, je potřeba ukládat perzistentní data. K tomuto účelu je použita knihovna Hibernate, která poskytuje *framework* umožňující do relační databáze ukládat přímo instance vytvořených objektů. Druhá část *BSComm* je konzolová aplikace pro komunikaci s *base station*. Její činností je příjímání zpráv z webového rozhraní a jejich přeposílání do *base station*. Když *BSComm* přijme zprávu od *base station*, uloží ji do databáze, odkud si ji *Control Panel* vyzvedne. Ukázka vzájemné komunikace na platformě *WSageNt* je zobrazena na obrázku 4.1, který vychází z [10].



Obrázek 4.1: Ukázka vzájemné komunikace na platformě *WSageNt*.

K propojení *Control Panelu* s *BSComm* byl navržen komunikační protokol. Ten umožňuje *Control Panelu* zasílat do *BSComm* agentní nebo jednoduché zprávy obsahující data. Formát agentní zprávy je následující:

```
typ_zprávy #[10] adresa_uzlu # planBase # plán # beliefBase # inputBase
```

Formát jednoduché zprávy je velmi podobný:

```
typ_zprávy # adresa_uzlu # planBase
```

Příkladem zprávy může být:

```
M;.,WS,-;3;.,WS,-;(100)
```

Vzhledem k obsahu přenášených zpráv není nutné komunikaci šifrovat. Nedochází tak k zbytečnému zatěžování a zdržování průběhu komunikace. V reakci na každou zprávu je odeslána odpověď ve formě číselné konstanty. Její významy jsou uvedeny v tabulce 4.1 [10].

Číslo	Konstanta	Význam
0	SENDING_SUCCESS	Úspěch operace.
1	SENDING_ERROR	Neurčitá chyba.
2	SENDING_EXPIRED	Čas odesílaní vypršel.
3	SENDING_DEV_BUSY	Rozhraní už se zařízením komunikuje.

Tabulka 4.1: Odpovědi na zprávy. Čerpáno z [10].

4.3 Jazyk ALLL

Jazyk ALLL (Agent Low Level Language) je agentní jazyk pro popis agentů, jejich komunikace a zasílání na jednotlivé uzly senzorové sítě vytvořený na VUT FIT. Při vytváření této kapitoly jsem čerpal především z těchto tří publikací [5], [6], a [7], kde je již jazyk ALLL popsán. Jelikož se v této práci zabývám interpretem tohoto jazyka pro operační systém Android, nemůže být jeho vysvětlení v tomto textu opomenuto.

4.3.1 Struktura jazyka

Tak jako každý jazyk, je i jazyk ALLL složený z vět. V tomto případě věty reprezentují jednotlivé plány agenta. Plány jsou taktéž věty, které jsou složené z jednotlivých akcí. Jazyk tedy umožňuje hierarchické zanořování plánů. Nepodaří-li se některý podplán vykonat, je smazán a provádění akce pokračuje na vyšší úrovni. Mezi jednotlivé akce jazyka můžeme zařadit spouštění plánů či podplánů, komunikaci přes rádio, operace pro práci s bází znalostí a volání služeb platformy. Výčet všech akcí jazyka ALLL je zobrazen v tabulce 4.2. Jejich význam bude v následujících kapitolách podrobněji vysvětlen. U každé akce si uvedeme příklady převzaté z [6].

Plány lze spouštět buď přímo, anebo nepřímo. Přímé spuštění znamená, že plán je přímo parametrem volané akce. U nepřímého spouštění jsou pojmenované plány vyhledávané v bázi plánů, do které byly dříve uloženy. Pod komunikací si můžeme představit vyzvedávání přijatých zpráv ze vstupní báze, a také odesílání zpráv na vybraný uzel. Mezi operace pro práci s bází znalostí řadíme:

[10] # představuje oddělovač zprávy. Pokud by výchozí znak nevyhovoval, může byt nahrazen jiným znakem.

přidávaní, odebírání a test na přítomnost znalostí v bázi. Při volání služby platformy zadáme kód služby a její parametry. Platforma poté provede tuto akci za agenta.

4.3.2 Datové struktury jazyka

Pro zachování rysů BDI agentů a jejich vhodné ukládání byl agent rozdělen do následujících 7 částí:

- BeliefBase - báze znalostí.
- InputBase - báze vstupů z rádia, či naměřené hodnoty ze senzorů.
- PlanBase - báze plánů.
- Plan - aktuální záměr.
- Univerzální registr číslo 1.
- Univerzální registr číslo 2.
- Univerzální registr číslo 3.

Jako hlavní datové struktury v interpretu jazyka ALLL slouží tabulky, seznamy, n-tice, zásobník.

Tabulky jsou tvořeny seznamy, jejichž pořadí není důležité. Seznam představuje posloupnost akcí. Jejich pořadí už důležité je, neboť jsou prováděny právě v tomto pořadí. Posloupnost akcí v seznamu je uzavřená do kulatých závorek. Jednotlivé akce od sebe nemusí být nijak speciálně oddělené, protože každá akce má svůj speciální počáteční znak. Tabulek se v interpretu jazyka ALLL vyskytuje několik. Jsou to PlanBase, ve které jsou uložené pojmenované plány, InputBase, která obsahuje zprávy přijaté z rádia nebo hodnoty naměřené ze senzorů a BeliefBase, do které si agent ukládá znalosti nabyté během své činnosti.

Jako další datovou strukturu používanou v interpretu jazyka ALLL jsme si uvedli n-tici. Její syntaxe je podobná seznamu jen s tím rozdílem, že jednotlivé položky jsou od sebe odděleny znakem ",". N-tice se využívají především pro uchování znalostí v BeliefBase nebo slouží pro předávání parametrů při volání akcí platformy.

Pro práci s aktuálním plánem se využívá zásobník. Jednotlivé akce, které se mají provést, jsou postupně vkládány na vrchol zásobníku. Po provedení akce je tato akce z vrcholu zásobníku odstraněna.

4.3.3 Pomocné registry

Interpret má k dispozici tři, především pomocné, registry. Používají se pro dočasné uložení seznamů (n-tic) nebo výsledků volaných služeb platformy. Aktivní však v danou chvíli může být pouze jeden. Po vykonání akce je její výsledek vždy vrácen do aktivního registru. Změna aktivního registru se provádí pomocí symbolu & a čísla registru (&(1), &(2), &(3)).

Při sestavování plánu můžeme v každé akci na místo konkrétní hodnoty použít zástupný znak registru (&1, &2, &3), v němž je hodnota uložena. Před provedením akce pak dojde k nahrazení registru jeho obsahem.

Při vkládání do báze znalostí se substituce neprovádí. N-tice je uložena ve svém přesném tvaru. Provedení substituce nastane až při výběru n-tice z báze znalostí.

4.3.4 Unifikace

V jazyce ALLL se využívá operace unifikace pro prohledávání tabulek. Při unifikaci se vyhledává přítomnost zadané n-tice v tabulce. Pro tyto účely je možnost v jazyce definovat symbolem "_" tzv. anonymní proměnnou. Anonymní proměnnou lze unifikovat na libovolný prvek jazyka. Výsledkem unifikace je buď nahrazení anonymní proměnné za konkrétní hodnotu, čímž se obě n-tice stanou identickými, nebo neúspěch. Jazyk ALLL neobsahuje klasické proměnné, které známe z jiných jazyků. Operace unifikace je tímto podstatně zjednodušena, protože dochází pouze k hledání dvou stejných seznamů, popřípadě n-tic.

4.3.5 Vkládání do BeliefBase

Požadovanou vlastností báze znalostí je jedinečnost uchovávaných údajů. Proto je jednou z podstatných vlastností při vkládání do báze znalostí kontrola duplicity. Jestliže není vkládaná akce v tabulce přítomná, je vložena na její začátek. Při vkládání do báze znalostí na platformě *WSageNt* může dojít k přetečení pole, vložení není provedeno a akce končí chybou. Zařízení používající Android jako operační systém mají podstatně větší RAM paměti. Navíc je tato paměť spravována manažerem, který v případě nedostatku místa odstraní z paměti nepoužívané aplikace. K přetečení by tedy nemělo docházet.

Příklady:

První registr obsahuje jednoprvkovou n-tici `(456).`

`+(123,456)` Akce vloží na začátek báze znalostí n-tici `(123,456).` Opětovné volání této akce nebude mít žádný význam, protože jsme si vysvětlili, že se v bázi znalostí nemohou vyskytovat duplicitní položky.

`+(123,&1)` Do báze znalostí bude vložena n-tice `(123,&1).` Při výběru z báze znalostí je registr 1 nahrazený svým obsahem a n-tice vypadá následovně `(123,456).`

Kód akce	Parametry	Význam
+	n-tice | registr	Vkládání do BeliefBase, duplicitní vkládání se kontroluje unifikací.
-	n-tice | registr	Odebrání položek z BeliefBase pomocí unifikace.
!	číslo | registr n-tice | registr	Odeslání zprávy zadané n-ticí nebo registrem na mote[11] se zadanou adresou.
?	n-tice | registr	Test InputBase na zprávu od mote / senzoru se zadanou adresou.
@	seznam akcí	Přímé spuštění, akce se vloží na zásobník se zarážkou.
^	jméno | registr	Nepřímé spuštění, hledá se plán v PlanBase se stejným jménem.
&	Číslo	Změna aktivního registru.
*	n-tice | registr	Test BeliefBase na zadanou n-tici nebo registr, výsledek se uloží do aktivního registru.
$	písmeno [n-tice | registr]	Volání služeb platformy, první parametr (písmeno) je kód operace. Druhý parametr jsou parametry služby, nemusí být u všech služeb.
#	žádné	Zarážka za plánem, sémanticky tato akce nemá žádný význam.

Tabulka 4.2: Přehled dostupných akcí v jazyce ALLL. Čerpáno z [6].

[11] mote - používáme pro označení senzorového uzlu na platformě WSageNt. Budeme se tohoto názvosloví držet i v případě platformy Android, kde jsou uzly tvořeny mobilními telefony.

4.3.6 Odebírání z BeliefBase

Zadaná n-tice je postupně vyhledávána v celé bázi znalostí a všechny položky vyhovující unifikaci jsou odstraněny. Při odebírání z báze znalostí lze vhodně využít anonymní proměnné, viz následující příklady.

Příklady:

Báze znalostí obsahuje dvě jednoprvkové `(123),(456)` a jednu dvouprvkovou `(123,456)` n-tici.

`-(123)` Akce vyhledá v bázi znalostí tuto n-tici. Pokud je hledání úspěšné, je zadaná n-tice z tabulky odstraněna. V našem případě bude z tabulky odstraněna první n-tice.

`-(_)` Na odstranění je vybraná jednoprvková n-tice zadaná anonymní proměnnou. Ta se úspěšně unifikuje na první dvě n-tice v tabulce, které smaže. V tabulce zůstane třetí n-tice.

`-(456,_)` I když je druhá hodnota n-tice také anonymní proměnná, nelze zadanou n-tici unifikovat s žádnou n-ticí v tabulce, neboť se neshoduje první hodnota. Nebude tedy smazána žádná hodnota.

4.3.7 Odeslání zprávy

Akce odeslání zprávy přes rádio musí obsahovat dva parametry. První parametr je adresa cílového uzlu a druhý je obsah zprávy k odeslání zadaný n-ticí. Oba parametry mohou být zadány pomocí registru. Nejprve je do výstupního bufferu zkopírována odesílaná zpráva. Je-li potřeba, registry jsou nahrazeny svými obsahy a zpráva je odeslána.

Příklady:

V prvním registru je uložena n-tice `(1)`, ve druhém `(123,456)`.

`!(1,(abc))` Uzlu s adresou 1 je odeslána zpráva `(abc)`.

`!(&1,&2)` Z prvního registru se zjistí adresa uzlu, kterému se má zpráva poslat. Z druhého registru se zjistí obsah zprávy. Výsledkem této akce je odeslání zprávy `(123, 456)` uzlu s adresou 1.

4.3.8 Testování InputBase

Přijaté zprávy z rádia jsou automaticky ukládány do vstupní báze. Tato akce slouží k testování tabulky na přítomnost zprávy od uzlu se zadanou adresou. Adresa uzlu je jediný parametr této akce. Prvky ve vstupní bázi jsou ve tvaru (adresa, n-tice). Při nalezení zprávy od požadovaného uzlu je obsah zprávy přesunut do aktivního registru a zpráva je ze vstupní báze odstraněna.

Příklady:

InputBase obsahuje dvě n-tice `(1,(abc))` a `(2,(123))`. V prvním registru je uložena jednoprvková n-tice (2). Druhý, aktivní registr, je prázdný.

`?(1)` Ve vstupní bázi se hledá zpráva od prvního uzlu. V našem případě je nalezena a přesunuta do aktivního registru. Druhý registr tak obsahuje n-tici `(abc)`. Zároveň je první údaj smazán z tabulky, ve které zůstane po provedení akce už jen údaj `(2,(123))`.

`?&1` Stejný případ, jako v předchozím příkladu, pouze adresa uzlu je zadána registrem. Ve vstupní bázi se vyhledá zpráva od uzlu určeného obsahem 1. registru. Druhý registr tak bude obsahovat údaj `(123)` a ve vstupní bázi zůstane už jen údaj `(1,(abc))`.

`?(_)` Jako parametr je zadána anonymní proměnná. I když je schopná se unifikovat na více n-tic, bude v tomto případě ze vstupní báze vybrána první n-tice.

4.3.9 Přímé spuštění plánu

Jako parametr této akce zadáváme seznam akcí, jež se mají provést. Na zásobník se vloží zarážka a před ní se postupně vkládají jednotlivé akce plánu. Přímé spuštění může být použito u plánů s rizikem selhání některé akce a u kterých není žádoucí, aby selhal celý plán na vyšší úrovni.

Příklad:

Na zásobníku je kód `+(123).`

`@(+(abc)!(2,(456)))` Na zásobník je vložena zarážka a kód akce, zásobník tedy bude vypadat následovně: `+(abc)!(2,(456))#+(123).`

4.3.10 Nepřímé spuštění plánu

Nepřímé spuštění plánu se od toho přímého liší v parametru volání akce. Namísto seznamů akcí je zde jako parametr název plánu, který je uložený v bázi plánů. Po úspěšném vyhledání plánu jsou jeho akce postupně přidávány před zarážku na zásobník jako v předchozím případě. V opačném případě končí akce chybou.

Příklady:

V bázi plánů je uložen jeden plán `(plan,(+(def)!(3,(789)))`. Na zásobníku je kód `+(123)` a první registr obsahuje jedinou jednoprvkovou n-tici (`plan`).

`^(plan)` V bázi plánu je vyhledán plán se jménem `plan`. V našem případě dojde k úspěšnému nalezení a akce plánu jsou vloženy na zásobník, jenž bude vypadat následovně: `plan,(+(def)!(3,(789))#+(123).`

`^&1` Parametrem volání akce je registr, ve kterém je stejný název plánu jako v předchozím případě, takže obsah zásobníku bude po vykonání akce shodný.

4.3.11 Změna aktivního registru

Tato triviální akce slouží k přepnutí aktivního registru. Při přepnutí registru dojde k jeho vynulování. Do aktivního registru jsou ukládány výsledky některých volaných akcí nebo služeb platformy. Podle návrhu jazyka ALLL pro platformu *WSageNt* má interpret k dispozici tři registry. Při programování interpretu pro platformu Android bude přítomnost registrů simulována pro udržení kompatibility jazyka.

Příklad:

`&(1)` Nastavení prvního registru jako aktivního a vymazaní jeho obsahu.

4.3.12 Testování BeliefBase

Při testování báze znalostí dochází k vyhledání, zda se zadaná n-tice vyskytuje v tabulce. Všechny unifikovatelné n-tice jsou vráceny ve formě seznamu do aktivního registru. Pokud se daná n-tice v tabulce nevyskytuje, končí akce chybou.

Příklady:

BeliefBase obsahuje tyto dvě jednoprvkové n-tice `(123)`, `(456)` a dvouprvkovou n-tici `(abc,def)`. Aktivní je v tomto případě první registr.

`*(123)` Tato n-tice je v bázi znalostí nalezena a do prvního registru je uložen jednoprvkový seznam. První registr bude tedy vypadat následovně: `((123))`.

`*(_)` N-tice je jednoprvková zadaná anonymní proměnou. V tabulce jsou dvě jednoprvkové n-tice. Obě tedy vyhovují unifikaci a budou přidány do seznamu. Jelikož jak unifikace, tak přidávání do seznamu probíhá od počátku, budou n-tice v aktivním registru v opačném pořadí: `((456),(123))`.

`*(abc)` V bázi znalostí se zadaná n-tice nevyskytuje. Akce skončí s chybou a aktivní registr zůstane prázdný.

Kód	Parametry	Popis
a	žádné	Při běžícím interpretu se pomocí této služby aktivuje sledování příchozích zpráv.
f	seznam | registr	Služba vloží do aktivního registru první prvek seznamu jako jednoprvkovou n-tici nebo první ze seznamů, je-li jich více.
k	žádné	Voláním této služby zastavíme činnost interpretu.
l	seznam | registr	Pomocí této služby se ovládají LED diody na *mote*, parametr musí povinně obsahovat kód barvy LED (r, g, y) a volitelně nastavení stavu LED diody (0 - nesvítí, 1 - svítí), není-li stav explicitně zadán, dojde k jeho přepnutí.
m	seznam | registr	Parametr této služby obsahuje adresu (číslo) platformy, kam se má kód agenta přesunout. Celý kód se zkopíruje do cílové platformy a provádění pokračuje na obou platformách dál. Je-li cílová adresa stejná jako ta, na které agent momentálně běží, provedením této služby se nic nestane. Za adresou může být volitelně parametr "s", který značí zastavení vykonávání kódu na zdrojové platformě.
r	seznam | registr	Služba vloží do aktivního registru zbytek seznamu bez prvního prvku (prvky mohou být také seznamy). Když je seznam pouze jednoprvkový, vloží se do aktuálního registru prázdná n-tice.
s	žádné	Zastavení provádění kódu, dokud nepřijde zpráva z rádia. Bylo-li aktivováno sledování a zpráva byla už přijata, provádění pokračuje dále.
w	seznam | registr	Pomocí této služby je možné pozastavit vykonávání kódu na určitou dobu zadanou v milisekundách.
d	seznam | registr	Tato služba slouží pro změření aktuální hodnoty ze senzoru, pokud je bez parametrů. Parametry *m, M* resp. *a* určují, zda požadujeme minimální, maximální nebo průměrnou hodnotu. Musí být následován číslem, které určuje, z kolika hodnot se má min., max., resp. průměr vypočítat.
o	n-tice	Volání matematických operací. Jejich výčet je zobrazen v tabulce 4.3.4 a jejich volání je vysvětleno v kapitole 4.3.15.

Tabulka 4.3: Služby platformy. Čerpáno z [6].

4.3.13 Volání služeb platformy

Pro složitější akce, jako je získávání dat ze senzorů, ovládání LED diod, pozastavení interpretu, volání matematických funkcí apod. slouží služby platformy. V tabulce 4.3 je uveden výčet všech služeb, jejich kódových označení, včetně případných parametrů. Volání služby platformy vypadá následovně: `$(<kod služby> [<parametry>])`.

4.3.14 Změření hodnot ze senzorů

V kapitole 2.2 jsou popsány veškeré senzory podporované platformou Android. Zdaleka ne všechny senzory jsou ovšem na všech telefonech dostupné. Zařízení mně dostupná jsou telefony firmy HTC podporující gyroskop, GPS senzor, digitální kompas, proximity senzor a senzor okolního osvětlení. Dále tablet firmy Asus podporující senzor pohybu, gyroskop, GPS senzor, digitální kompas a světelný senzor. Jako implicitní senzor pro interpretaci této služby byl zvolen senzor okolního osvětlení. V tabulce 4.4 jsou uvedeny kódy dostupných senzorů, které slouží jako námět pro rozšíření aplikace.

Příklad:

`$(d)` Do aktivního registru je vložena naměřená hodnota ze senzoru. Není-li senzor na zařízení dostupný, služba končí chybou.

Kód seznoru	Význam
Gyr	Změří hodnotu z gyroskopu.
Acc	Zaznamená hodnotu z akcelerometru.
Prx	Změří vzdálenost objektu od zařízení.
Amb	Vrátí intenzitu okolního osvětlení.
Com	Změří hodnoty magnetického pole v daném místě na zemi.
Gps	Zjistí aktuální pozici.

Tabulka 4.4: Kódy pro změření hodnoty ze senzoru.

4.3.15 Matematické operace

Jako jedna ze služeb poskytovaná platformou je volání jednoduchých matematických operací. Ty jsou rozdělené na unární a binární. Jejich tvar volání je následující:
`$(o,<typ>,<operand1>,[<operand2<],<název1>,[<název2>],<výsledek>)`[12],
kde:

- `o` je kód služby platformy pro matematické operace.
- `<typ>` značí typ operace. Jejich výčet je uveden v tabulce 4.5.
- `<operand>` určuje operandy matematické operace. Může být zadaný seznamem nebo registrem, ve kterém je seznam operandů uložen. Pokud je operand pouze jeden, je seznam jednoprvkový, operandů však v seznamu může být i více. Potom, v případě unární operace, je operace aplikována na každý operand ze seznamu. V případě binární operace dostaneme výsledek jako kartézský součin jednotlivých operandů mezi sebou. Tvar operandu v seznamu je dvouprvková n-tice `(<název_operandu>,<hodnota>)`.

[12] Údaje v hranatých závorkách jsou použity jen při binárních operacích, při unárních jsou vypuštěné.

- `<název>` je n-tice, která reprezentuje název operandu. Tento prvek je zkontrolován s každým prvkem `<název_operandu>` v operandu. Při výpočtu se nebere v úvahu. Operand je také možné zadat bez názvu. V tom případě je tento prvek prázdná n-tice.
- `<výsledek>` zastupuje libovolný název identifikátoru, který slouží k pojmenování výsledků při uložení.

Příklady:

`$(o,mul,&1,&2,(x),(y),(z))`

Operace binárního součinu. První registr obsahuje seznam dvou n-tic `(((x),1)((x),3))`. Ve druhém registru je pouze jednoprvkový seznam `(((y),2))`. Třetí registr je nastavený jako aktivní, tím pádem prázdný. Operace násobení uloží do třetího registru výsledky této operace v podobě seznamu `(((z),2)((z),6))`.

`$(o,not,&1,(x),(notx))`

Operace negace operandu. V prvním registru je seznam obsahující dva operandy `(((x),5)((x),-6))`. Druhý registr je nastavený jako aktivní. Výsledkem operace negace bude vložení seznamu `(((notx),-5)((notx),6))` do druhého registru.

Kód operace	Význam	Příklad
Mul	násobení dvou čísel	operand 1 * operand 2
Div	celočíselné dělení	operand 1 ÷ operand 2
Mod	zbytek po celočíselném dělení	operand 1 % operand 2
Add	součet dvou čísel	operand 1 + operand 2
Sub	rozdíl dvou čísel	operand 1 - operand 2
Les	menší než	operand 1 < operand 2
Leq	menší nebo rovno než	operand 1 ≤ operand 2
Mor	větší než	operand 1 > operand 2
Meq	větší nebo rovno než	operand 1 ≥ operand 2
Equ	rovno	operand 1 == operand 2
Neq	nerovno	operand 1 ≠ operand 2
And	logický součin dvou čísel	operand 1 & operand 2
Orr	logický součet dvou čísel	operand 1 | operand 2
Min	unární minus	- operand 1
Not	negace	¬ operand 1
Cpy	kopírování obsahu proměnných	operand 1 = operand 2

Tabulka 4.5: Kódy jednotlivých matematických operací. Čerpáno z [7].

4.3.16 Zarážka

Sémanticky nemá tato akce žádný význam. Symbol zarážky slouží pouze pro jednoznačné oddělení plánů na zásobníku. V případě selhání některé akce v plánu jsou ze zásobníku odstraněny všechny akce tohoto plánu až po zarážku.

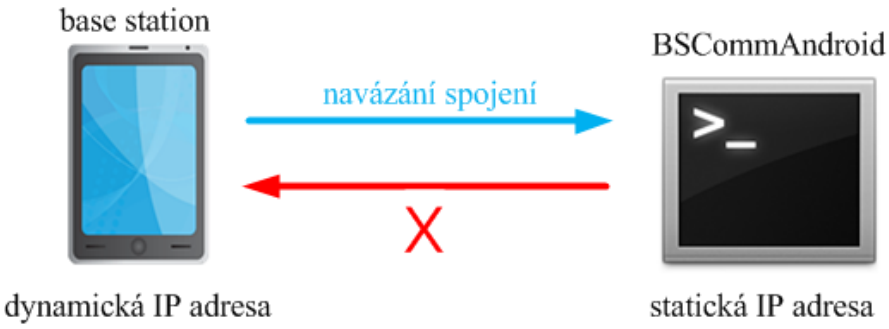
5 Návrh interpretu pro Android

V předchozích kapitolách jsme si představili bezdrátové senzorové sítě, platformu *WSageNt*, řekli jsme si, jak se chová agent a jak vypadá jazyk, kterým agenta popisujeme. Také jsme zjistili, že senzory jsou již přítomné i v mobilních telefonech a operační systém Android poskytuje programátorům funkce pro jejich využití ve svých aplikacích. V této kapitole se tedy budeme zabývat tvorbou bezdrátové senzorové sítě vytvořené z mobilních telefonů.

Síť bude tvořena pomocí mobilních zařízení s operačním systémem Android. Na otestování mám k dispozici telefony značky HTC a tablet firmy Asus.

5.1 Komunikace base station s počítačem

Na platformě *WSageNt* je *base station* připojena k počítači pomocí USB kabelu. *Base station* na platformě Android je libovolné zařízení, které s počítačem komunikuje pomocí mobilního internetu[13]. Jelikož je IP adresa mobilního telefonu dynamická, je potřeba, aby server na počítači používal statickou a hlavně veřejnou IP adresu. Telefon, představující *base station*, po zadání správné IP adresy a portu vytvoří *socket*, pomocí kterého se připojí k počítači. Naopak to kvůli dynamickým adresám mobilních telefonů nebude možné. Tuto situaci vystihuje obrázek 5.1. Jakmile je vytvořen *socket* a navázáno ustálené spojení, komunikace již probíhá oběma směry.



Obrázek 5.1: Připojení *base station* k *BSCommAndroid*.

5.2 Komunikace mezi uzly sítě

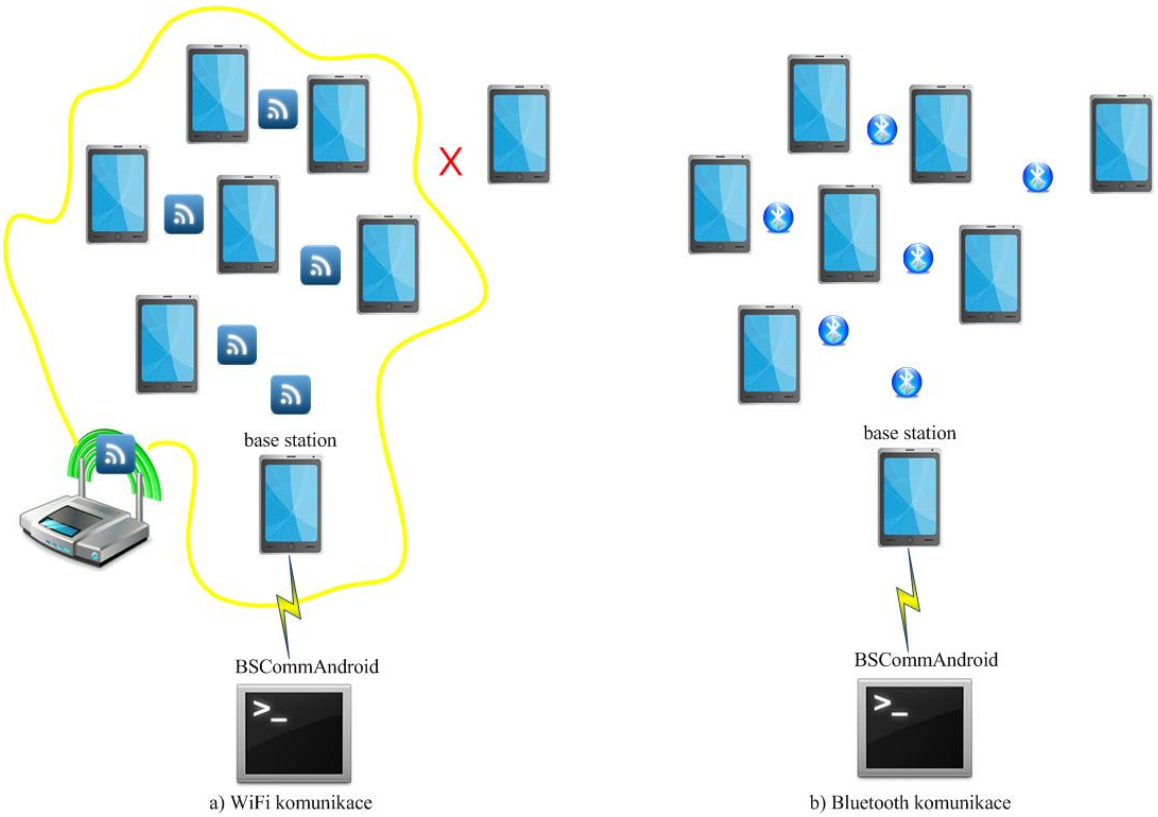
Jednotlivé uzly sítě na platformě *WSageNt* komunikují s *base station* pomocí standardu IEEE 802.15.4/Zigbee. Uzly, tvořené mobilními telefony na platformě Android, tento standard nepodporují. Nabízí však dvě jiné varianty. První je komunikace uzlů pomocí modulu WiFi. Druhá varianta je komunikace přes rozhraní Bluetooth. Nejprve jsem se rozhodl využít komunikaci přes WiFi rozhraní, vytvořením ad-hoc sítě, ale při její implementaci jsem narazil na problém, který spočívá v přímé komunikaci dvou uzlů bez přítomnosti AP[14]. Podrobněji se na něj podíváme v následujících odstavcích.

[13] pomocí GPRS, edge nebo 3G připojení.
[14] access point - přístupový bod.

Jak jsem se již v tomto textu jednou zmínil, existuje mnoho verzí platformy Android. S každou novou verzí Androidu přichází také nová verze API s novými funkcemi. V době psaní této práce již byla sice dávno vydána verze Android 4.0 (API 14 - ...), která sjednocuje vývoj tohoto systému pro telefony i tablety, nicméně počet zařízení, na kterých byl tento operační systém podporován, byl stále mizivý. Nejrozšířenější verze Androidu na telefony byla 2.3.x (API 9, 10) a na tablety 3.2.x[15] (API 11 - 13).

Důvod, proč uvádím dostupné verze platformy Android je ten, že podpora přímé komunikace dvou zařízení s operačním systémem Android přes WiFi rozhraní, nazývána WiFi Direct, je dostupná až od API verze 14. Možná řešení problému komunikace jsou následující:

- Centralizovaná komunikace přes WiFi rozhraní - tato varianta náš problém sice vyřeší, ale není ideální, protože všechny uzly sítě musí být v dosahu signálu AP. Uzel, který není v dosahu signálu AP, nedostane přidělenou IP adresu, tudíž nepatří do lokální sítě a nemůže s ostatními uzly komunikovat. Situaci vystihuje obrázek 5.2 a).
- Komunikace přes Bluetooth rozhraní - v momentálním případě je to nejlepší varianta. Není potřeba dodatečného hardware v podobě AP. Uzly mohou mezi sebou komunikovat prostřednictvím ad-hoc sítě. Tato varianta je zobrazena na obrázku 5.2 b).
- Pro využití technologie WiFi Direct ke komunikaci uzlů je vhodné počkat od té doby, než bude operační systém Android ve verzi 4.0 rozšířenější.



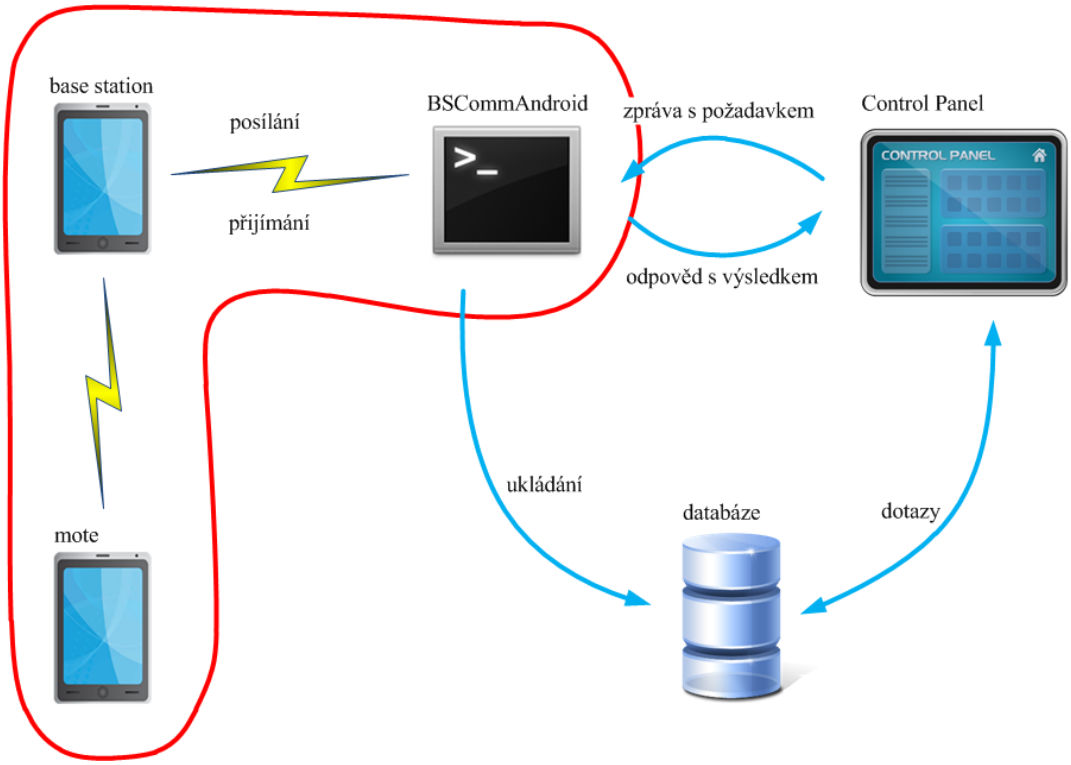
Obrázek 5.2: Rozdíl v komunikaci přes WiFi a Bluetooth rozhraní.

I když jsem si na telefon nainstaloval neoficiální ROM s verzí OS Android 4.0, nebyla technologie WiFi Direct podporována. Na můj tablet dokonce vyšla nová verze operačního systému Android oficiálně, podpora výše zmíněné technologie však stejně chyběla. Z výše uvedených důvodů je pro komunikaci jednotlivých uzlů sítě s *base station* využito rozhraní Bluetooth. Využití technologie WiFi Direct tak zůstává jako námět k pozdějšímu rozšíření aplikace.

[15] x značí číslo pod-verze, na různé modely telefonu jsou dostupné různé pod-verze.

5.3 Uživatelské webové rozhraní Control Panel

Pro platformu *WSageNt* vzniklo webové rozhraní *Control Panel*, viz kapitola 4.2.2. Toto rozhraní bude použito pro správu agentů a odesílání zpráv uzlům senzorové sítě i na platformě Android. Bude však nahrazena nižší vrstva *BSComm*, která slouží pro propojení webového rozhraní *Control Panel* s *base station*. Komunikace bude probíhat odlišným způsobem, popsaným v kapitole 5.1. *BSCommAndroid* bude také konzolová aplikace, která bude z velké většiny vycházet z aplikace *BSComm* a zachová z ní tu část aplikace, která komunikuje s webovým uživatelským rozhraním. Pro komunikaci s *base station* budou vytvořené nové třídy, které budou zajišťovat komunikaci prostřednictvím TCP/IP socketů. Obrázek 5.3 zobrazuje celkovou komunikaci jednotlivých komponent na platformě Android. Červeně orámované jsou komponenty, které byly v porovnání s platformou *WSageNt* odebrány a nahrazeny novými.



Obrázek 5.3: Ukázka vzájemné komunikace na platformě Android.

5.4 Interpret jazyka ALLL

Interpret jazyka ALLL bude aplikace na mobilním zařízení s operačním systémem Android. Aplikace bude podporovat dva typy chování. Buď bude představovat běžný uzel senzorové sítě, nebo se připojí ke konzolové aplikaci *BSCommAndroid* a bude se chovat jako *base station*. Aplikace bude rozdělena do několika částí. Bude zde vlákno, které se bude starat o samotný běh aplikace a grafické uživatelské rozhraní. Dále vlákno, které bude zajišťovat komunikaci prostřednictvím Bluetooth modulu. Samozřejmostí bude vlákno, které bude interpretovat příkazy jazyka ALLL. Jelikož aplikace poběží na uzlu senzorové sítě, nesmí chybět vlákno pro měření dat ze senzoru.

6 Framework ANTLR

ANTLR je zkratka pro *Another Tool for Language Recognition*. ANTLR představuje víceúčelový nástroj, který poskytuje *framework* pro práci s formálními jazyky. Pomáhá při tvorbě překladačů a interpretů jazyka. Vychází z popisu libovolné gramatiky jazyka, která bude interpretována anebo překládána. Výstupem jsou zdrojové třídy v jednom z podporovaných programovacích jazyků, mezi které patří například Java, C, C++, C#, PHP, JavaScript a další.

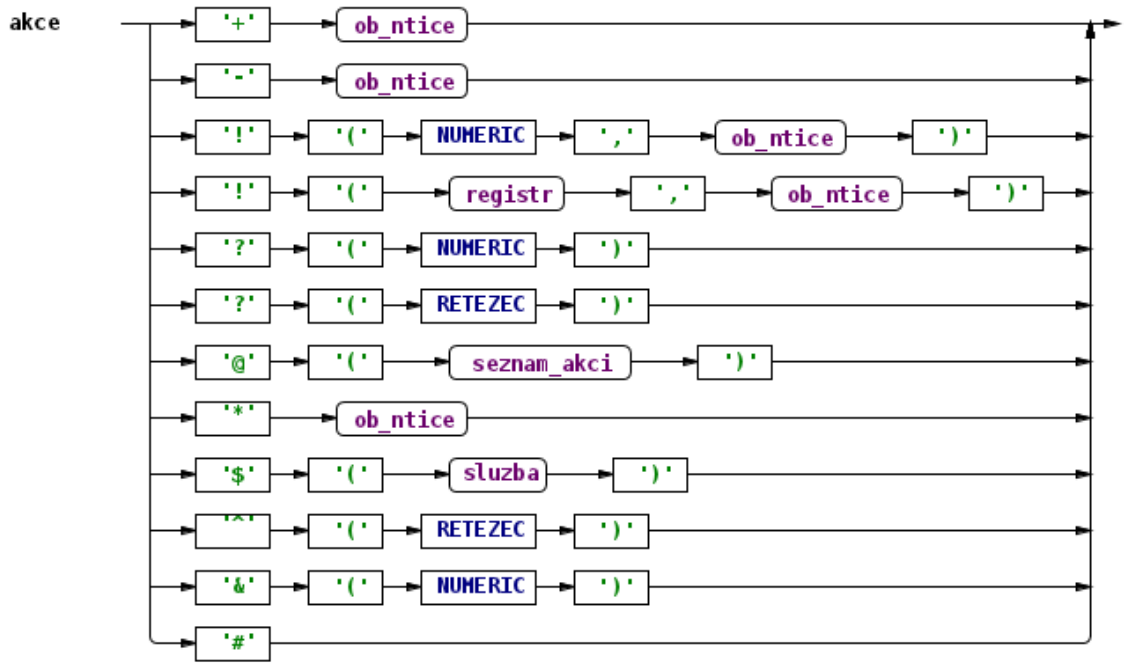
Vstupní gramatiku lze vytvořit v libovolném textovém editoru, jejíž zdrojový soubor bude mít příponu .g. Tuto gramatiku lze poté přeložit v konzoli pomocí příkazu:

```
java -cp antlr-X.X.jar org.antlr.Tool Gramatika.g,
```

kde `X.X` označuje číslo použité verze ANTLR a `Gramatika.g` je soubor s nadefinovanou gramatikou. Alternativní přístup pro tvorbu gramatiky je využití grafického vývojového prostředí ANTLRWorks.

6.1 Vývojové prostředí ANTLRWorks

ANTLRWorks je archivní balík typu .jar, který lze použít pro vývoj gramatiky jazyka na platformách Windows, Linux i MacOS. V tomto vývojovém prostředí je zahrnut nejen kvalitní editor gramatiky s interpretem pro rychlé prototypování, ale také debugger pro rychlé odhalování chyb a pohodlné ladění pravidel gramatiky. V editoru je také možnost zobrazení přehledného diagramu pro syntaxi pravidel gramatiky, který umožňuje odstranit nedeterminismus. Obrázek 6.1 ukazuje diagram syntaxe pro pravidlo akce, které popisuje syntaxe všech akcí, které mohou v jazyce ALLL nastat. Fialovou



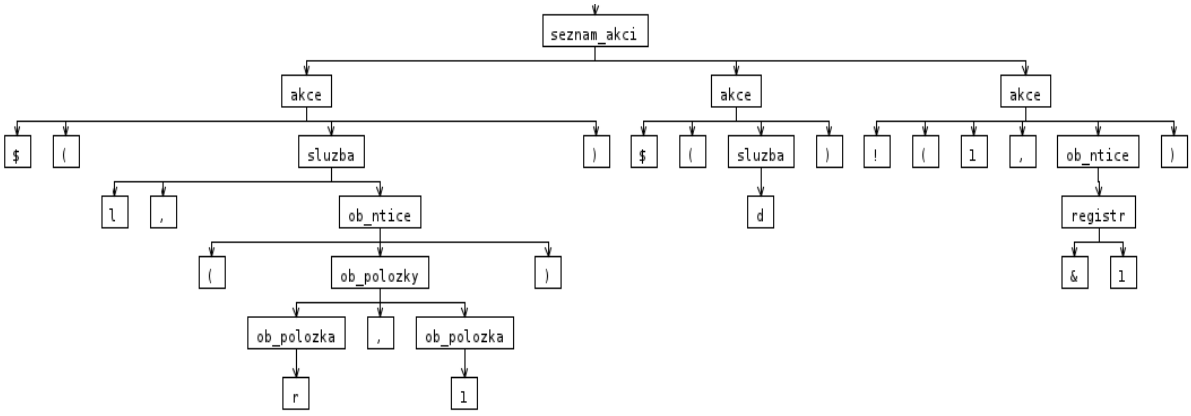
Obrázek 6.1: Syntaktický diagram pravidla pro akci jazyka ALLL.

barvou jsou zvýrazněné nonterminální symboly jazyka. Modrá barva představuje označení nadefinovaných složených terminálních symbolů, jako jsou řetězec a číslo. Poslední barvou, zelenou,

jsou označeny terminální symboly, které již nejsou nikde nadefinovány a je tak brána přímo jejich hodnota.

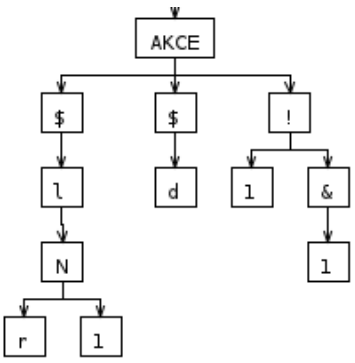
Jako první krok při vývoji interpretu jazyka ALLL jsem použil prostředí ANTLRWorks pro definici pravidel gramatiky tohoto jazyka. Pravidla gramatiky jsem v tomto prostředí odladil pomoci výše zmíněného debuggeru. Debugger umožňuje zobrazit jak derivační strom, který zobrazuje ve stromové struktuře terminální a nonterminální symboly jazyka, tak i abstraktní syntaktický strom. Abstraktní syntaktický strom se od derivačního stromu liší tím, že abstraktní syntaktický strom nemusí zobrazovat každý detail, který se vyskytuje v reálné syntaxi. Abstraktní syntaktický strom získáme doplněním pravidel pro generování tohoto stromu v gramatice jazyka. Jako příklad si uveďme příkaz jazyka ALLL:

```
$(l,(r,1))$(d)!(1,&1).
```



Obrázek 6.2: Derivační strom.

Zatímco na obrázku 6.2 je zobrazen celý rozsáhlý derivační strom, obrázek 6.3 ukazuje již zjednodušený abstraktní syntaktický strom téhož příkazu, ve kterém jsou odstraněny například seskupující závorky, které jsou ve stromové struktuře implicitní a další oddělovací symboly. Abstraktní syntaktický strom je nejen přehlednější pro programátora, ale je také vhodný pro řízení programu zejména pro just-in-time kompilátory.



Obrázek 6.3: Abstraktní syntaktický strom.

6.2 Lexikální a syntaktická analýza

Po vytvoření gramatiky jazyka (viz. příloha A) vygeneruje vývojové prostředí ANTLRWorks zdrojové třídy pro lexikální a syntaktickou analýzu pro zvolený programovací jazyk. V našem případě jsou to třídy programovacího jazyka Java, ve kterém probíhá vývoj pro mobilní platformu Android. Konkrétně se jedná o třídy `AlllLexer.java` a `AlllParser.java`, které jsou přidány k ostatním zdrojovým třídám, pomocí kterých vytvářím interpret jazyka ALLL.

Prvně zmíněná třída, `AlllLexer.java`, slouží jako lexikální analyzátor a provede tak lexikální analýzu. Lexikální analyzátor je vstupní částí každého interpretu, či překladače. Jeho hlavní a jedinou činností je rozdělení vstupní posloupnosti znaků, které dostane ve formě řetězce, na lexémy[16]. Proud lexémů je poté předán na vstup syntaktického analyzátoru k dalšímu zpracování.

Syntaktický analyzátor[17] reprezentuje druhá třída, kterou vygeneruje vývojové prostředí ANTLRWorks, `AlllParser.java`. Syntaktická analýza je proces analýzy vstupní posloupnosti tokenů, s cílem určit, zda odpovídá pravidlům nadefinované formální gramatiky. Syntaktická analýza může probíhat dvěma směry. Buď směrem shora dolů anebo směrem opačným zdola nahoru. ANTLR používá k syntaktické analýze LL parser, který analyzuje proud vstupních tokenů shora dolů, směrem zleva doprava a konstruuje nejlevější derivaci věty. Výstupem syntaktické analýzy parseru `AlllParser.java` je abstraktní syntaktický strom. Procházení tohoto stromu, uložení jeho uzlů do vnitřních objektů a interpretace těchto uzlů bude popsána v následujících kapitolách.

V následujícím příkladu je ukázka funkce v programovacím jazyce Java, ve které jsou použity třídy `AlllLexer.java` a `AlllParser.java` pro vygenerování abstraktního syntaktického stromu ze vstupního řetězce, který obsahuje příkaz v jazyce ALLL.

```
public CommonTree makeTree(String input)
{
   CharStream cs = new ANTLRStringStream(input);
   AlllLexer lexer = new AlllLexer(cs);
   CommonTokenStream tokens = new CommonTokenStream(lexer);
   AlllParser parser = new AlllParser(tokens);

   AlllParser.agent_return retVal = null;
   try
   {
      retVal = parser.agent();
   }
   catch (RecognitionException e1)
   {
      e1.printStackTrace();
   }
   CommonTree tree = (CommonTree) retVal.getTree();

   return tree;
}
```

Vstupem této funkce je řetězec `input`, který obsahuje příkaz jazyka ALLL a výstupem je abstraktní syntaktický strom uložený v objektu `tree`.

[16] Lexém je lexikální jednotka, pod kterou si lze představit prvky daného jazyka. Například řetězce, konstanty, operátory, klíčová slova atd. Lexémy bývají také označovaný jako tokeny.
[17] Syntaktický analyzátor bývá také označován jako parser.

6.3 Knihovna ANTLR v prostředí Android

Abychom mohli provést lexikální a syntaktickou analýzu příkazu v jazyce ALLL, jak jsme si ukázali v předchozí kapitole, musíme do našeho projektu přidat vygenerované třídy z vývojového prostředí ANTLRWorks, `AlllLexer.java` a `AlllParser.java`. Tyto vygenerované třídy používají funkce z knihovny ANTLR, konkrétně z archivního balíku typu .jar: `antlr-X.X.jar`, kde `X.X` představuje číslo verze použité knihovny. Poslední dostupná verze knihovny ANTLR v době tvorby této diplomové práce byla verze 3.4. Bylo tedy nezbytné přidat tuto knihovnu do našeho projektu pro doplnění chybějících referencí. Jelikož knihovna ANTLR podporuje programovací jazyk Java a vývoj aplikací pro operační systém Android probíhá právě v tomto jazyce, vývojové prostředí Eclipse se po přidání této knihovny do našeho projektu tváří, že je vše v pořádku a naše aplikace bude bez problému fungovat. Problém však nastane v době překladu aplikace. I když se pořád bavíme o stejném programovacím jazyce, jeho kompilace pro běh na operačním systému Android je jiná, než kompilace pro běh v Java Virtual Machine na osobním počítači. Při překladu aplikace jsem se tak setkal s chybovou hláškou, že knihovna ANTLR byla sestavena jiným překladačem. Aby tedy bylo možné využít již zmíněné třídy `AlllLexer.java` a `AlllParser.java`, je nutné místo knihovny ANTLR v podobě archivního balíku typu .jar, přidat do naší aplikace přímo zdrojové soubory této knihovny. Tyto soubory jsou pak při překladu naší aplikace přeloženy společně s ní stejným překladačem a vše funguje tak, jak má. Kompletní popis knihovny ANTLR a vývojového prostředí ANTLRWorks, včetně možností jejich stažení, je dostupný z [13].

7 Implementace komunikace uzlů a datových struktur

V kapitole 5.2 jsme rozebrali různé možnosti komunikace mezi uzly senzorové sítě, které jsou tvořené zařízeními s operačním systémem Android. Uvedli jsme si dvě možnosti: buď pomocí modulu WiFi, anebo pomocí modulu Bluetooth. I když jsem se dočkal v průběhu mojí práce nové verze operačního systému Android (verze 4.0) pro některá má zařízení, podpora technologie WiFi Direct stejně nebyla na všech zařízeních dostupná. Z důvodu malého počtu zařízení, která komunikaci prostřednictvím WiFi Direct podporují, byl pro komunikaci uzlů zvolen modul Bluetooth. Podpora pro modul Bluetooth je na platformě Android dostupná již od Android API verze 5, tudíž bude tato aplikace podporována na větším počtu zařízení s operačním systémem Android.

Můžeme říct, že datové struktury tvoří velkou a nezbytnou část pro zajištění funkčnosti naší aplikace. Veškeré zprávy přenášené mezi uzly sítě jsou uložené v datových strukturách. Navíc, jak si později ukážeme, pracujeme s datovými strukturami téměř při každé akci interpretace agenta (příkazu jazyka ALLL). V kapitole 7.2 se podíváme na to, jakým způsobem byly tyto struktury implementované v naší aplikaci.

7.1 Implementace Bluetooth komunikace

Bluetooth komunikace se skládá ze čtyř hlavních částí: zapnutí a nastavení Bluetooth modulu, vyhledání již spárovaných anebo dostupných zařízení v dosahu, připojení se k zařízení a přenos dat mezi zařízeními. Nyní se podíváme na to, jak jsou tyto části implementované v naší aplikaci. Abychom vůbec mohli v naší aplikaci využívat Bluetooth modul, je nutné jako první a nezbytný krok udělit aplikaci potřebná oprávnění, která se zadávají ve speciálním souboru `AndroidManifest.xml`. Tato oprávnění vypadají následovně:

```
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"
/>.
```

První z nich dovoluje poslat žádost o připojení, přijmout žádost o připojení a výměnu dat. Druhé oprávnění je potřebné pro možnost zapnutí samotného Bluetooth modulu. Pro veškeré operace s Bluetooth modulem jsem si vytvořil třídu `Bluetooth.java`, která bude poskytovat nezbytné metody a vlákna pro zajištění Bluetooth komunikace.

Jelikož uzly senzorové sítě mezi sebou komunikují přes Bluetooth modul, první krok po startu aplikace je tedy kontrola na jeho přítomnost na daném zařízení a jeho bezprostřední zapnutí. K tomuto účelu poskytuje Bluetooth API třídu `BluetoothAdapter`, která obstarává veškerou interakci s Bluetooth modulem. V následující ukázce je vidět, jakým způsobem můžeme získat instanci třídy `BluetoothAdapter`.

```
BluetoothAdapter mBluetoothAdapter =
            BluetoothAdapter.getDeafaulAdapter();
if (mBluetoothAdapter == null)
{
    // Zarizeni nepodporuje Bluetooth
}
```

Další krok v Bluetooth komunikaci je vyhledání a spárování dostupných zařízení. Tato aplikace již žádné další zařízení nevyhledává, nýbrž pracuje pouze se zařízeními, která byla spárována před zapnutím aplikace za využití prostředků operačního systému Android. Chceme-li do naší senzorové sítě přidat další uzel, je potřeba jej spárovat s ostatními zařízeními. Jelikož využíváme technologii Bluetooth na našich tabletech a telefonech s operačním systémem Android také pro jiné účely, mohou být v seznamu spárovaných zařízení také jiná zařízení, která nepatří do naší senzorové sítě, jako například *hands-free* zařízení apod. Z tohoto důvodu bylo zavedeno jednotné pojmenování pro zařízení, která představují uzly naší senzorové sítě. Název každého uzlu vypadá následovně: `"uzelX"`, kde `X` představuje pořadové číslo uzlu. Uzel s pořadovým číslem 1 bude představovat *base station*. Díky tomuto jednoduchému a hlavně jednotnému pojmenování je možné v aplikaci implementovat filtr, který při vyhledání všech již spárovaných zařízení vrátí seznam, ve kterém jsou zapsána pouze ta zařízení, která tvoří uzly naší senzorové sítě. Prvky tohoto seznamu jsou dvojice klíč a hodnota. Klíč je číslo daného uzlu, a proto musí být pro každý uzel unikátní. Hodnota je objekt třídy `BluetoothDevice`. Třída `BluetoothDevice` se využívá k reprezentaci vzdáleného Bluetooth zařízení a obsahuje veškeré jeho informace, jako jsou název, MAC adresa, třída Bluetooth zařízení a další. Také slouží pro vytvoření Bluetooth socketu pro připojení k danému zařízení. Seznam těchto dvojic uložíme do kontejneru typu `LinkedHashMap<Integer, BluetoothDevice>` s názvem `_devices`.

Samotná komunikace pak probíhá dle klasického schématu klient - server. Pro vytvoření spojení mezi dvěma zařízeními v naší aplikaci je potřeba implementovat obě dvě strany. Jedno zařízení vytvoří Bluetooth server socket a čeká na připojení. Druhé pak inicializuje spojení a snaží se připojit k serveru pomocí jeho MAC adresy. Klient a server jsou navzájem propojeni, pokud oba dva mají připojen Bluetooth socket ke stejnému RFCOMM kanálu. Podívejme se teď na vytvoření serveru a klienta podrobněji.

7.1.1 Vytvoření serveru

V kapitole 7.1 jsem uvedl, že první krok po startu aplikace je zapnutí Bluetooth modulu. Ihned po zapnutí tohoto modulu je vytvořeno nové vlákno `AcceptThread`, které vytvoří Bluetooth server socket a čeká na připojení klienta. Nové vlákno se vytvoří proto, neboť čekání na připojení klienta je blokující operace, avšak my potřebujeme, aby aplikace dále vykonávala svou původní činnost. Po připojení klienta je vytvořen nový Bluetooth socket a původní socket, Bluetooth server socket, je uzavřen. Nebude již dále potřebný, protože komunikace bude probíhat skrz nově vytvořený socket. Základní funkce pro vytvoření vlákna serveru jsou vysvětleny v několika následujících bodech:

- Pro vytvoření Bluetooth server socketu poskytuje třída `BluetoothAdapter` funkci: `listenUsingRfcommWithServiceRecord(`*`NAME`*`, `*`UUID`*`)`. První parametr funkce, `NAME`, je libovolný řetězec zastupující jméno Bluetooth služby. Většinou se používá jméno aplikace. Druhý parametr, `UUID`, je již podstatnější. `UUID` představuje 128 bitový řetězcový identifikátor Bluetooth služby. Při vytváření spojení mezi serverem a klientem je potřeba, aby se identifikátory obou stran shodovaly.
- Po vytvoření Bluetooth server socketu přejde vlákno do fáze čekání na připojení klienta pomocí funkce `accept()`. Jak bylo zmíněno již dříve, tato funkce je blokující operace. Po úspěšném připojení klienta vytvoří funkce `accept()` Bluetooth socket, přes který poté obě strany komunikují.

- Na rozdíl od spojení TCP/IP, spojení přes RFCOMM kanál dovoluje pouze jednoho připojeného klienta na daném kanále. Proto je po připojení klienta Bluetooth server socket uzavřen pomocí funkce `close()`.

Po úspěšném připojení klienta původní vlákno `AcceptThread` skončí a je vytvořeno vlákno nové, které slouží pro přenos zpráv mezi klientem a serverem. Na tyto přenosy zpráv se podíváme později. Po ukončení vlákna s přenosem zpráv je znovu vytvořeno vlákno `AcceptThread`, což znamená, že uzel je znova připraven přijímat žádosti o připojení i od jiných uzlů. Poznamenejme tedy, že každý uzel se tváří jako server a čeká na žádost o připojení od ostatních uzlů až do doby, než sám potřebuje odeslat zprávu a stává se tak klientem. Na vytvoření klientského vlákna se podíváme v následující kapitole.

7.1.2 Vytvoření klienta

Tak jako pro vytvoření serveru, tak i pro vytvoření klienta je vytvořeno samostatné vlákno s názvem `ConnectThread`. Nevytváří se však ihned po startu aplikace, nýbrž jen v případech, kdy uzel potřebuje odeslat zprávu jinému uzlu. Aby se mohl klient připojit k serveru, musí nejprve získat objekt třídy `BluetoothDevice`, který jej reprezentuje. Jak jsem se zmínil již výše, veškeré uzly naší senzorové sítě jsou uloženy v kontejneru `_devices`. Pokud při interpretaci agenta, zapsaného v jazyce ALLL, narazí interpret na akci odeslání zprávy, vždy musí obsahovat v parametru číslo uzlu, na který se má daná zpráva odeslat. Na základě toho čísla je v kontejneru `_devices` vyhledán příslušný objekt třídy `BluetoothDevice`, který reprezentuje server, ke kterému se chystá klient připojit. Pomocí tohoto objektu je vytvořen Bluetooth socket a ustanoveno spojení se serverem. V následujících několika bodech si představíme nejdůležitější funkce vlákna `ConnectThread`:

- Při vytváření serveru jsme použili funkci pro vytvoření Bluetooth server socketu z třídy `BluetoothAdapter`. Při vytváření klienta použijeme podobnou funkci, která nám vrátí Bluetooth socket, avšak z objektu (serveru) třídy `BluetoothDevice`, ke kterému se chystáme připojit: `createRfcommSocketToServiceRecord(`*`UUID`*`)`. Parametr `UUID` jsme si již vysvětlili při vytváření serveru. Připomeňme však, že hodnoty parametru `UUID` obou komunikujících stran musí být stejné.
- Když získá klient Bluetooth socket, pokusí se připojit k serveru voláním funkce `connect()`. Po zavolání této funkce systém zkontroluje hodnoty `UUID` a pokud server přijme spojení od klienta, sdílejí spolu obě dvě strany RFCOMM kanál. Volání funkce `connect()` je blokující operace, proto je pro klienta vytvořeno samostatné vlákno.

Stejně jako u serveru, tak i u klienta po úspěšném připojení vlákno `ConnectThread` skončí a je vytvořené vlákno nové, které se stará o přenos zpráv.

7.1.3 Vlákno pro přenos zpráv

Výše v tomto textu jsem se zmínil, že jak po dokončení činnosti klientského vlákna, tak i po dokončení činnosti vlákna serveru, přichází na řadu vlákno nové, které se stará o přenos zpráv mezi oběma zúčastněnými stranami komunikace. Ve skutečnosti se však jedná o dvě vlákna `ConnectedAccepterThread` a `ConnectedSenderThread` odvozené od abstraktního vlákna `ConnectedThread`. V následujícím odstavci se na ně podíváme podrobněji.

V momentu, kdy máme obě komunikující strany propojené Bluetooth socketem, vytvoříme datové proudy `InputStream` a `OutputStream` pro příjem, respektive odesílání zpráv. Tyto datové proudy dědí od abstraktního vlákna `ConnectedThread` obě dvě odvozená vlákna. Pro skutečný přenos zprávy přes tyto vytvořené datové proudy existují funkce `read(byte[])` a `write(byte[])`. První zmíněná, funkce `read()`, je blokující operace. Druhá funkce, `write()`, může být blokující v případě přeplnění vyrovnávacích bufferů při nedostatečně rychlém volání funkce `read()`. Z tohoto důvodu jsou v našem projektu implementována dvě odvozená vlákna. Vlákno `ConnectedAccepterThread` se stará o příjem zpráv, tzn. volá funkci `read()`, zatímco vlákno `ConnectedSenderThread` se stará o odesílání zpráv, tzn. volá funkci `write()`.

7.1.4 Protokol přenosu zpráv přes Bluetooth rozhraní

V předchozí kapitole jsme si popsali vlákno, které se stará o přenos zpráv mezi uzly senzorové sítě přes Bluetooth rozhraní. Neřekli jsme si však, jak bude vypadat formát přenášených zpráv. Pro přenos zpráv byl navržen textový protokol, jenž bude předmětem této kapitoly. Mezi uzly je možné přenášet dva typy zpráv:

- agentní zprávy
- jednoduché zprávy

Agentní zpráva je složená ze sedmi částí, které jsou od sebe navzájem odděleny symbolem ";". Tyto části tvoří obsahy datových struktur[18], popsaných v kapitole 4.3.2. Uzel, který odesílá agentní zprávu, převede postupně obsahy všech svých datových struktur z vnitřní reprezentace na textové řetězce, které zřetězí do jedné zprávy, kterou následně odešle. Uzel, který agentní zprávu přijme, ji nejprve převede z textového řetězce zpět do vnitřní reprezentace a uloží ji do svých datových struktur. Formát agentní zprávy vypadá následovně:

```
PlanBase ; Plan ; BeliefBase ; InputBase ; Registr_1 ; Registr_2 ;
Registr_3;
```

Jednoduchá zpráva je pro zachování jednotnosti také tvořena sedmi částmi, které jsou od sebe oddělené stejným symbolem jako v předchozím případě. Na rozdíl od agentní zprávy jsou však všechny části, kromě části s obsahem datové struktury InputBase, prázdné. V části, určené pro obsah datové struktury InputBase, není uložen skutečný obsah datové struktury InputBase, nýbrž obsah zprávy, který se bude odesílat. Formát jednoduché zprávy má tedy tvar:

```
; ; ; InputBase ; ; ; ;
```

Jednoduchá zpráva se posílá v části určené pro obsah datové struktury InputBase z jednoduchého důvodu. Do datové struktury InputBase jsou totiž mimo jiné ukládány také zprávy, které uzel obdržel přes rozhraní Bluetooth. Zprávy se do ní ukládají ve formátu dvojice klíč a hodnota. Klíč představuje číslo uzlu, který zprávu zaslal a hodnota je skutečný obsah zprávy. Při odesílání jednoduché zprávy je tedy vytvořena zpráva ve formátu dvojice: číslo odesilatele a obsah zprávy. Následně je tato zpráva převedena na textový řetězec a odeslána v části určené pro obsah datové struktury InputBase. Uzel, který přes Bluetooth rozhraní přijme jednoduchou zprávu, ji po převedení zpět z textového řetězce do

[18] Podrobná implementace datových struktur je popsána v kapitole 7.2.

vnitřní reprezentace může rovnou uložit do své datové struktury InputBase, protože tato jednoduchá zpráva již byla zaslána ve správném formátu pro ukládání zpráv do datové struktury InputBase.

Aby přijímající uzel věděl, kterou z výše popsaných zpráv bude přijímat, je před agentní respektive jednoduchou zprávou, zaslána kratičká zpráva, která pouze určuje typ následující zprávy. Typ zprávy je přenášen samostatně, aby agentní popřípadě jednoduchá zpráva nemusela být po příjmu parsována kvůli zjištění svého typu a mohla být rovnou odeslána ke zpracování. Zpracování zpráv bude vysvětleno v kapitole 8.3

7.2 Implementace datových struktur

V kapitole 4.3.2 byl uveden návrh datových struktur pro zachování rysů BDI agentů a jejich vhodné ukládání. Dále jsme se s těmito datovými strukturami setkali v kapitole 7.1.4, ve které jsme se zabývali protokolem přenášených zpráv. Tam jsme si uvedli, že se protokol pro přenášení zpráv skládá z obsahů právě těchto struktur.

Veškerá komunikace uzlů senzorové sítě probíhá ve formě textových řetězců, ve výše zmíněných formátech. Po úspěšném obdržení zprávy uzlem je tato zpráva předána na vstup lexikální analýzy. Výstup této analýzy je vstupem pro syntaktickou analýzu, ze které získáme abstraktní syntaktický strom. Tento strom je poté postupně procházen a jednotlivé uzly stromu jsou ukládány do datových struktur.

Jednou z nejzákladnějších datových struktur je n-tice. Vnitřní interpretaci n-tice v této aplikaci implementuje třída `Ntuple.java`. N-tice může mít neomezený počet heterogenních prvků, mezi které mohou patřit čísla, textové řetězce, registry nebo i vnořené n-tice. Z tohoto důvodu je n-tice implementována pomocí kolekce `ArrayList<Object>`. Při čtení prvků n-tice pak musíme zjistit o jaký prvek z výše zmíněných se jedná. K tomu účelu velmi dobře poslouží operátor `instanceof`.

N-tice sama o sobě není v této aplikaci nikdy uložena samostatně, tvoří však základní stavební kámen ostatních datových struktur. Nyní se podívejme na jednu ze zmíněných sedmi částí, jež tvoří agenta, a to na bázi znalostí. Bázi znalostí implementuje v této aplikaci třída `BeliefBase.java`. Báze znalostí je tabulka tvořena homogenním seznamem, jehož prvky jsou pouze typu n-tice. Pro implementaci báze znalostí tak byla použita kolekce `ArrayList<Ntuple>`. V této třídě jsou obsaženy také nezbytné metody pro odebírání, vyhledávání a vkládání nových n-tic.

Další částí agenta, která je tvořená seznamem n-tic, je vstupní báze (InputBase), která je implementována třídou `InputBase.java`. Tak jako v bázi znalostí, je i ve vstupní bázi kolekce `ArrayList<Ntuple>` pro uložení n-tic. N-tice ukládané do této báze jsou buď zprávy obdržené z Bluetooth rozhraní, nebo hodnoty naměřené ze senzorů. Potřebujeme tedy rozlišit, od kterého uzlu nám zpráva přišla, respektive jakou hodnotu ze senzoru jsme změřili. Z tohoto důvodu obsahuje vstupní báze ještě jednu kolekci `ArrayList<String>`. Do této kolekce jsou ukládány čísla uzlů (převedených na řetězec), od kterých přišla zpráva nebo znaky, které reprezentují typ hodnoty naměřené ze senzoru[19]. Položky v obou kolekcích na stejných pozicích jsou logicky provázány, a tudíž tvoří dvojici klíč - hodnota. Metody pro mazání, přidávání nových prvků a vyhledávání podle klíče tak musí vždy pracovat s oběma kolekcemi současně.

Pro vytvoření dalších tabulek reprezentujících jednotlivé části agenta si musíme nadefinovat další stavební kámen, jako tomu bylo v případě n-tice a báze znalostí. Tímto stavebním kamenem bude akce, kterou budeme reprezentovat objektem třídy `Action.java`. V tomto objektu je uložen kód akce, která se má vykonat, a k ní potřebné parametry ve formě n-tice či registru. Jedná - li se o

[19] *s* pro aktuální hodnotu, *m* pro minimální hodnotu, *M* pro maximálního hodnotu, *a* pro průměrnou hodnotu

akci odeslání zprávy, je zde uložená také adresa uzlu, kterému se bude při interpretaci uložených akcí odesílat zpráva. Každá akce, kterou má agent vykonat, je reprezentována právě touto třídou. Pro uložení posloupnosti těchto akcí v pořadí, v jakém se budou postupně vykonávat, si nadefinujeme poslední stavební kámen a tím je třída `Plan.java`. Tato třída slouží pro uchování seznamu akcí. Plány jsou v této třídě uchovány v kolekci `Arraylist<Action>`. Kromě této kolekce obsahuje tato třída řetězec reprezentující název plánu. Název plánu slouží pro vyhledání konkrétního plánu v tabulce plánů a jeho následného spuštění, čímž spustíme interpretaci jeho akcí.

Jak je z předchozího odstavce patrné, tabulka plánů je tvořena seznamem pojmenovaných plánů uložených v kolekci `Arraylist<Plan>`. O implementaci této tabulky se v naší aplikaci stará třída `PlanBase.java`.

Aktuální záměr agenta neboli akce, které agent bezprostředně po jejich obdržení provádí, jsou uloženy na zásobníku. Pro implementaci zásobníku jsem vytvořil třídu `MyStack.java`, která vznikla odvozením z již existující třídy v jazyce Java `Stack`.

Poslední částí datových struktur jsou tři pomocné univerzální registry, které jsou reprezentované třemi instancemi třídy `Register.java`. Objekt, reprezentující registr, obsahuje 3 podstatné vlastnosti: číslo daného registru, stav zda je registr aktivní a obsah daného registru. V danou chvíli může být aktivní vždy pouze jen jeden registr. To je hlídáno synchronizační funkcí, která kromě přepnutí aktivity registru také vymaže jeho obsah.

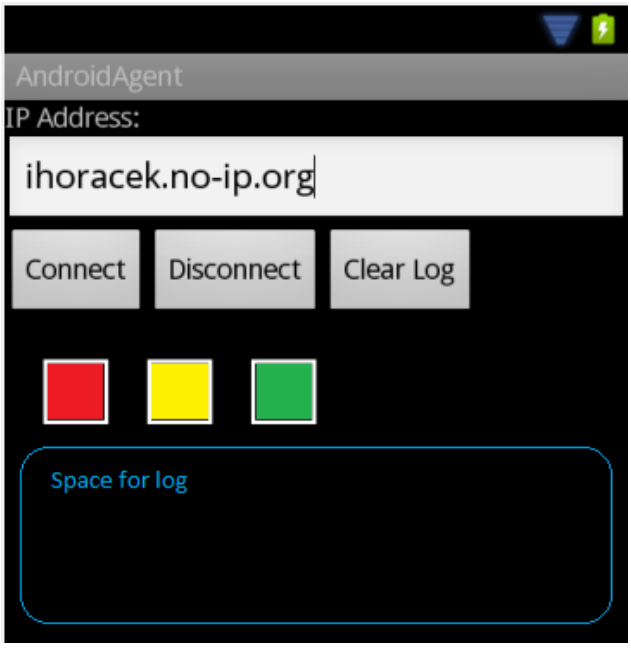
Všechny třídy, zmíněné v této kapitole, mají kromě svých již uvedených podstatných vlastností metodu `toString()`, která slouží pro převedení jejich obsahu do textového řetězce. Tato metoda je využívána při posílání agentních zpráv, kdy je potřeba dodržet formát zprávy představený v kapitole 7.1.4.

8 Aplikace na mobilním zařízení

Interpret jazyka ALLL je aplikace pro mobilní zařízení s názvem AndroidAgent. Aplikace je stejná pro všechny uzly senzorové sítě. Umožňuje však dva různé typy chování. První typ chování je chování klasického senzorového uzlu, který čeká na příjem zprávy přes Bluetooth rozhraní. Po příjmu zprávy začne interpretovat přijaté příkazy. Každé mobilní zařízení bezprostředně po startu této aplikace se chová tímto způsobem. Ten uzel, který se připojí k serveru[20], ztrácí vlastnosti klasického senzorového uzlu a přechází do druhého typu chování. Tento uzel je poté nazýván *base station*. Chování *base station* je jednodušší. Sám neinterpretuje žádné příkazy. Pouze přeposílá zprávy přijaté od serveru na uzly v senzorové sítí a naopak. V následujících kapitolách si představíme podstatné části aplikace podrobněji.

8.1 Vzhled a ovládání

Většina standardních aplikací v prostředí operačního systému Android, včetně naší aplikace, po svém startu zobrazí svou úvodní obrazovku. Jelikož naše aplikace simuluje uzel v senzorové síti, který nemá kromě LED diod žádný viditelný výstup, je tato obrazovka velmi jednoduchá a obsahuje pouze nezbytné elementy. Ukázka vzhledu obrazovky aplikace AndroidAgent je na obrázku 8.1.



Obrázek 8.1: Obrazovka aplikace.

První element je textové pole pro zadávání adresy serveru, na kterém je spuštěná konzolová aplikace *BSCommAndroid*. Na obrázku je předvyplněná adresa, která byla používána při vývoji a testování této aplikace. Po kliknutí na tlačítko Connect se aplikace připojí k této konzolové aplikaci a uzel se stává *base station*. Tlačítko Disconnect má funkčnost přesně opačnou. Po kliknutí na toto

[20] server je v tomto případě *BSCommAndroid*

tlačítko se aplikace odpojí od konzolové aplikace *BSCommAndroid*, uzel přestává být *base station* a stává se běžným uzlem senzorové sítě.

Jelikož mají uzly senzorové sítě na platformě *WSageNt* tři diody, bylo potřeba simulovat tento fakt i v této mobilní aplikaci. Většina mobilních telefonů má notifikační diodu, která slouží pro oznamovací účely o akcích, jako jsou přijatá zpráva v režimu spánku, zmeškaný hovor nebo stav nabíjení. Dioda je však pouze jedna. Aby tato aplikace věrně simulovala uzel z platformy *WSageNt*, jsou všechny tři diody nakreslené na displeji mobilního zařízení. Na obrázku 8.1 je zobrazen stav, kdy jsou všechny tři rozsvícené. Chová - li se aplikace jako *base station*, tzn., že pouze přeposílá zprávy, diody jsou na displeji zobrazeny také, ale jsou zhasnuté.

Poslední element zobrazený na displeji aplikace je prostor pro textový výpis[21], ve kterém jsou zobrazovány informativní zprávy. Například, že na *base station* přišel agent, že na uzel přišla zpráva, popřípadě naměřená hodnota ze senzoru. Pro vymazání tohoto logu slouží poslední prvek, o kterém jsme se ještě nezmínili, a to tlačítko Clear Log.

Již z úvodu kapitoly 8 víme, že se aplikace může chovat jako klasický uzel senzorové sítě nebo jako *base station*. Nyní se podrobněji podíváme na to, jak je každé z obou těchto chování implementováno, a ukážeme si, jak spolu komunikují vlákna, které dané chování umožňují.

8.2 Chování uzlu jako base station

Každá aplikace pro operační systém Android musí obsahovat třídu, která je odvozená od třídy `Activity`. V naší aplikaci je to třída `AndroidAgentActivity.java`. Je to třída, která je vytvořená při startu aplikace a slouží k interakci uživatele s aplikací. Jelikož uživatel provádí interakci se zařízením skrz uživatelské rozhraní, stará se tato třída také o vykreslení uživatelského rozhraní na displeji zařízení. Pro každou operaci, která bude trvat delší časový okamžik, je potřeba vytvořit nové vlákno, aby nedošlo k "zamrznutí" této třídy, což by znemožnilo ovládat aplikaci.

Má - li se aplikace chovat jako *base station*, musí být připojena ke konzolové aplikaci *BSCommAndroid*. Vysvětlení jak tato konzolová aplikace funguje ve spolupráci s *base station* se věnuje kapitola 9. Pro připojení mobilní aplikace *AndroidAgent* ke konzolové aplikaci *BSCommAndroid* je potřeba zadat správnou adresu serveru, na kterém je konzolová aplikace spuštěna. Port je nastaven na hodnotu 6789 a v dosavadní verzi ho není možné změnit. O samotné připojení se stará objekt třídy `Connector.java`, který je volán ze samostatného vlákna, které vytvoří třída `AndoidAgentActivity`, aby nedošlo k "zamrznutí" hlavní třídy. Ten vytvoří TCP/IP socket, který se pomocí metody `connect(InetSocketAddress)` připojí na zadanou adresu. Po navázání socketu jsou vytvořeny komunikační datové proudy (`InputStream a OutputStream`) pro příjem respektive odesílání zpráv. Nyní aplikace čeká pomocí funkce `readLine()` na zprávu od konzolové aplikace, aby ji mohla přeposlat na některý z uzlů senzorové sítě. Tato operace je blokující, proto pro ni také musí existovat samostatné vlákno. V tomto případě však nevytváříme samostatné vlákno sami, ale využijeme prostředků Android API, konkrétně třídy `AsyncTask`. Třída `AsyncTask` nám umožňuje řádně a jednoduše pracovat s vláknem uživatelského rozhraní. Umožňuje vykonávat operace na pozadí a jejích výsledky publikovat ve vlákně, které se stará o uživatelské rozhraní (`AndroidAgetnActivity`). Tuto vlastnost využijeme pro zobrazení příchozí zprávy do logu. Abychom mohli využít přívětivé vlastnosti třídy `AsyncTask`, vytvoříme si vlastní odvozenou třídu `BackgroundAsyncTask`, která bude implementovat její tři metody:

[21] log - výpis informativních zpráv, hlavně pro účely testování

- `doInBackground()` - tělo této metody provádí aktivitu na pozadí, proto je výše zmíněná metoda `readLine()` volána z tohoto místa. Abychom po přijetí zprávy mohli tuto zprávu zpracovávat, a zároveň znova čekat na příjem zprávy další, je vytvořeno nové vlákno pro zpracování příchozích zpráv `HandleinputThread`, které si popíšeme za chvíli.
- `onProgresUpdate()` - provádění této metody začne po zavolání metody publishProgres() v těle předchozí metody. V těle této metody zapisujeme příchozí zprávu z konzolové aplikace *BSCommAndroid* do logu, který je zobrazený na displeji *base station*.
- `onPostExecute()` - tělo této metody se provede po ukončení asynchronní třídy *BackgroundAsyncTask*, která skončí po ukončení navázaného spojení s konzolovou aplikací *BSCommAndroid*. Proto je v této metodě provedeno uzavření socketu i obou datových proudů.

V předchozím odstavci jsme si ukázali, jak *base station* přijímá zprávy od konzolové aplikace *BSCommAndroid*. Nyní se podíváme, co se s přijatou zprávou děje dále. Formát, v jakém aplikace *BSCommAndroid* odesílá zprávu, bude vysvětlen v kapitole 9.3. Nyní si vystačíme pouze s tím, že na začátku zprávy je číselný identifikátor, který určuje, zda se jedná o agentní či jednoduchou zprávu. Za ním následuje číslo uzlu, na který se odešle zbytek zprávy. Postup odeslání zprávy na daný uzel v senzorové síti přes Bluetooth rozhraní je vysvětlen v kapitole **Chyba! Nenalezen zdroj odkazů.**.

Prozatím jsme si vysvětlili způsob, jakým *base station* přeposílá zprávy z počítače do uzlů v senzorové síti. Teď se podívejme na proces opačný, a to zasílání zpráv z uzlů senzorové sítě na počítač. Tento proces už je velmi jednoduchý, neboť již máme vytvořené všechny nezbytné datové proudy i vlákna, přes které probíhá veškerá komunikace. V kapitole 7.1.1 jsme si uvedli, že ihned po startu aplikace a následném zapnutí Bluetooth modulu se vytvoří vlákno, které čeká na příjem zprávy přes Bluetooth rozhraní. Jakmile toto vlákno obdrží zprávu, oznámí tuto skutečnost hlavní třídě `AndroidAgentActivity`. Tato třída má vazbu na objekt třídy `Connector`, který drží otevřené datové proudy pro komunikaci s konzolovou aplikací *BSCommAndroid*. Zbývá tak pouze vložit zprávu do datového proudu pomocí metody `println(String)` a odeslat pomocí metody `flush()`.

8.3 Chování klasického uzlu senzorové sítě

Implementaci chování *base station* jsme si ukázali v minulé kapitole. Nyní se zaměříme na chování aplikace na všech ostatních uzlech senzorové sítě. Aby mohl senzorový uzel v síti reálně fungovat, musí být schopen přijímat příkazy, v našem případě agentní a jednoduché zprávy. Proto prvním krokem po startu aplikace je vytvoření vlákna `AcceptThread`, které čeká na příjem zprávy přes Bluetooth rozhraní. Jakým způsobem toto vlákno funguje, vysvětluje kapitola 7.1.1. Po přijetí zprávy na senzorovém uzlu mohou nastat čtyři následující situace:

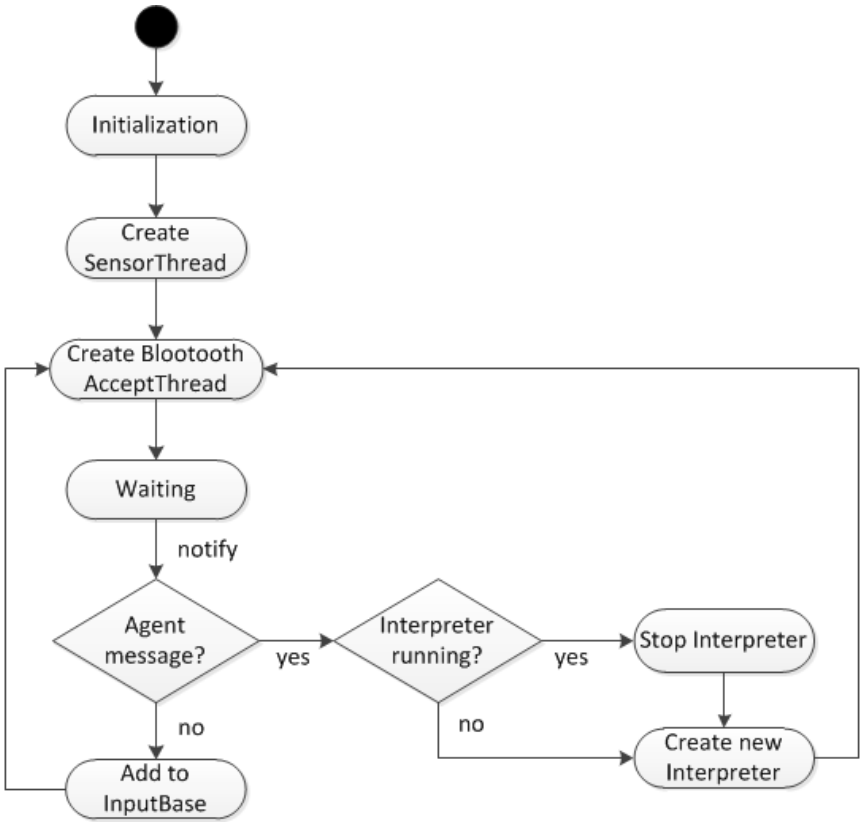
- přišla jednoduchá zpráva a na uzlu neběží interpret
- přišla jednoduchá zpráva a na uzlu běží interpret
- přišla agentní zpráva a na uzlu neběží interpret
- přišla agentní zpráva a na uzlu běží interpret

Jednoduchá zpráva je vždy vložena do tabulky InputBase, bez ohledu na to, zda běží interpret či ne. Co se děje s agentní zprávou popisuje kapitola 8.3.1.

Kromě čekání na příchozí zprávu přes Bluetooth rozhraní měří aplikace periodicky hodnotu ze senzoru. O tuto vlastnost se stará vlákno `SensorsThread`. Toto vlákno je také vytvořeno ihned po startu aplikace. Hodnoty se ze senzoru získávají zaregistrováním *listeneru* způsobem, který je popsán v kapitole 2.2. Jelikož se toto vlákno stará pouze o měření hodnot ze senzoru, je po dobu své nečinnosti uspáno. Délkou uspání vlákna se nastavuje frekvence měření hodnot. Při uspání vlákna je také odregistrovaný *listener*, aby nedocházelo k tak rychlému vybíjení baterie zařízení. Hodnoty, získané tímto periodickým měřením, jsou ukládány do bufferu, který implementuje třída `CircularBuffer.java`. Data v tomto bufferu slouží k získání minimální a maximální hodnoty a k výpočtu průměrné hodnoty z několika posledních měření. Třída `CircularBuffer` obsahuje kromě kruhového bufferu pro 4000 hodnot všechny nezbytné metody pro zjištění těchto agregovaných hodnot.

8.3.1 Interpretační smyčka

V této kapitole se konečně dostáváme k tomu, jakým způsobem jsou interpretovány příkazy jazyka ALLL. Tyto příkazy představují agentní akce, jež se mají na daném uzlu senzorové sítě vykonat. Po příjmu agentní zprávy zašle objekt třídy `Bluetooth` přijatou zprávu hlavní třídě naší aplikace `AndroidAgentActivity`. Ta se podívá, zda již existuje vlákno třídy `Interpreter`. Pokud ano, znamená to, že již aplikace interpretuje akce agentní zprávy, která byla doručena dříve. V tomto případě se interpretační vlákno ukončí. Následně je vytvořeno nové vlákno, které bude interpretovat akce nově doručené agentní zprávy. V jakých stavech se hlavní třída `AndroidAgentActivity` postupně nachází názorně ukazuje obrázek 8.2.

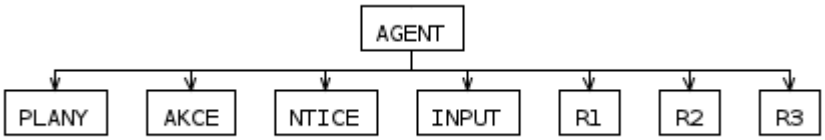


Obrázek 8.2: Stavový diagram hlavního vlákna aplikace.

Vlákno `Interpreter` dostane na vstupu agentní zprávu ve formě textového řetězce ve formátu, který je definovaný v kapitole 7.1.4. Tento řetězec je vstupem pro lexikální analýzu. Ihned po dokončení lexikální analýzy přichází na řadu syntaktická analýza, která zpracuje proud tokenů, který je výstupem z lexikální analýzy a vytvoří abstraktní syntaktický strom. Tyto analýzy provádí postupně třídy `AlllLexer.java` a `AlllParser.java`, které byly popsány v kapitole 6.2.

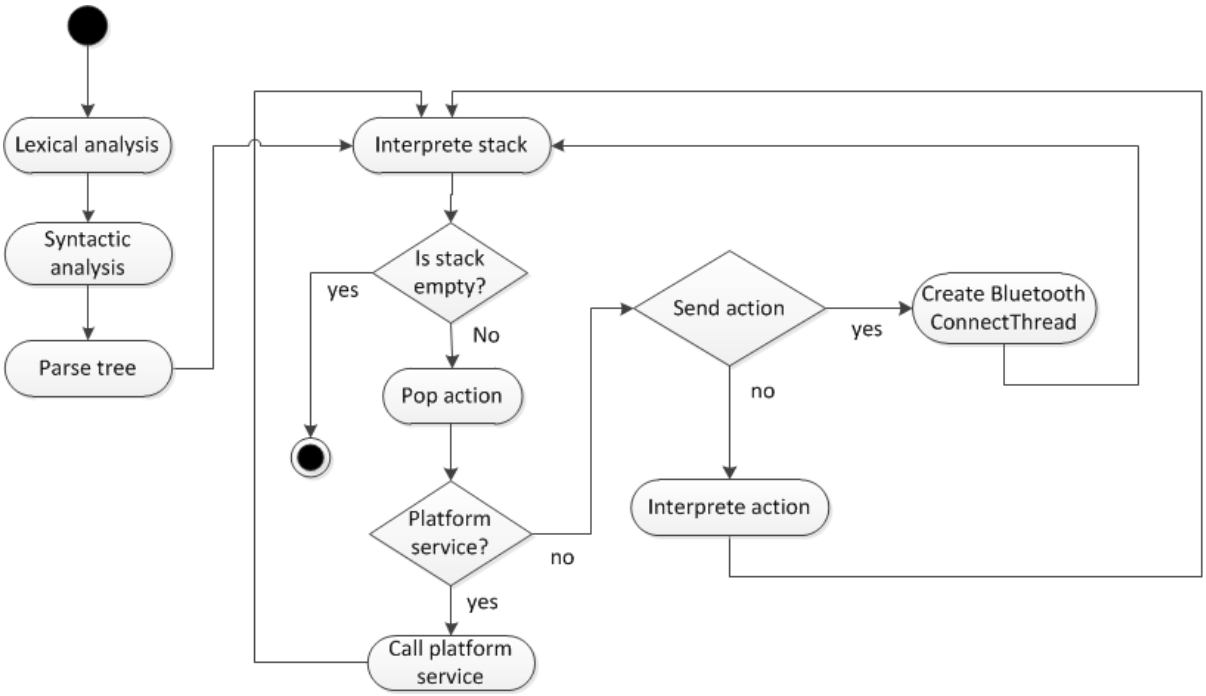
Další nezbytný krok před interpretací jednotlivých akcí představuje průchod abstraktním syntaktickým stromem, o který se stará metoda `interpreteTree(CommonTree)`. Tato metoda postupně volá pomocné metody pro průchod každého ze sedmi podstromů. Na obrázku 8.3 je zobrazen abstraktní syntaktický strom agenta, ve kterém všechny jeho bezprostřední synovské uzly představují kořenové uzly jednotlivých podstromů. Výstupem průchodu každého podstromu je naplnění odpovídající datové struktury. To znamená, že průchodem podstromu s kořenovým uzlem:

- PLANY naplní aplikace tabulku PlanBase,
- AKCE naplní aplikace zásobník,
- NTICE naplní aplikace tabulku BeliefBase,
- INPUT naplní aplikace tabulku InputBase a
- R1 - R3 naplní aplikace odpovídající registr.



Obrázek 8.3: Abstraktní syntaktický strom agentní zprávy.

Po naplnění všech datových struktur můžeme přejít k interpretaci akcí, které jsou uložené na zásobníku. Samotný interpretační cyklus probíhá tak dlouho, dokud jsou na zásobníku dostupné nějaké akce. Pokud je zásobník prázdný, nemá interpret co vykonávat a interpretační vlákno končí. Interpretační cyklus může být také ukončen interpretací speciální ukončovací akce nebo příchodem nové agentní zprávy. Po příchodu nové agentní zprávy je zásobník uměle vyprázdněn a interpretační vlákno tím pádem skončí. O interpretace akcí ze zásobníku se stará metoda `interpreteStack()`. Ta vyjme akci na vrcholu zásobníku a předá ji ke zpracování metodě `interpreteAction(Action)`. Pokud při interpretaci akce dojde k chybě, je vyvolána výjimka. Jsou taktéž odstraněny všechny akce ze zásobníku až po zarážku, od které může interpret pokračovat v činnosti. Není - li na zásobníku žádná zarážka vložena, jsou z něj odebrány všechny akce a interpretační vlákno končí. K reprezentaci výjimky byla v aplikaci implementována třída `InterpreteActionException.java`. Protože všechny akce, kromě dvou, pracují převážně s tabulkami a datovými strukturami, zvládne jejich interpretaci vlákno `Interpreter` samostatně. Při interpretaci akce pro posílání zprávy spolupracuje interpretační vlákno s objektem třídy `Bluetooth`, který se postará o odeslání zprávy do zvoleného uzlu. Interpretace akce volání služby platformy není v této aplikaci tak striktně odděleno, jak je tomu v interpretu jazyka ALLL na platformě *WSageNt* a některé služby zvládne vykonat přímo interpretační vlákno. Nicméně vykonání některých služeb platformy již vlákno `Interpreter` samostatně nezvládne a proto spolupracuje s jinými vlákny. V následující kapitole se podíváme, které služby zvládne vykonat interpretační vlákno samostatně a pro vykonání kterých služeb komunikuje s ostatními vlákny. Stavy, kterými interpretační vlákno prochází při interpretaci agentní zprávy, znázorňuje obrázek 8.4.



Obrázek 9.4: Stavový diagram interpretačního vlákna.

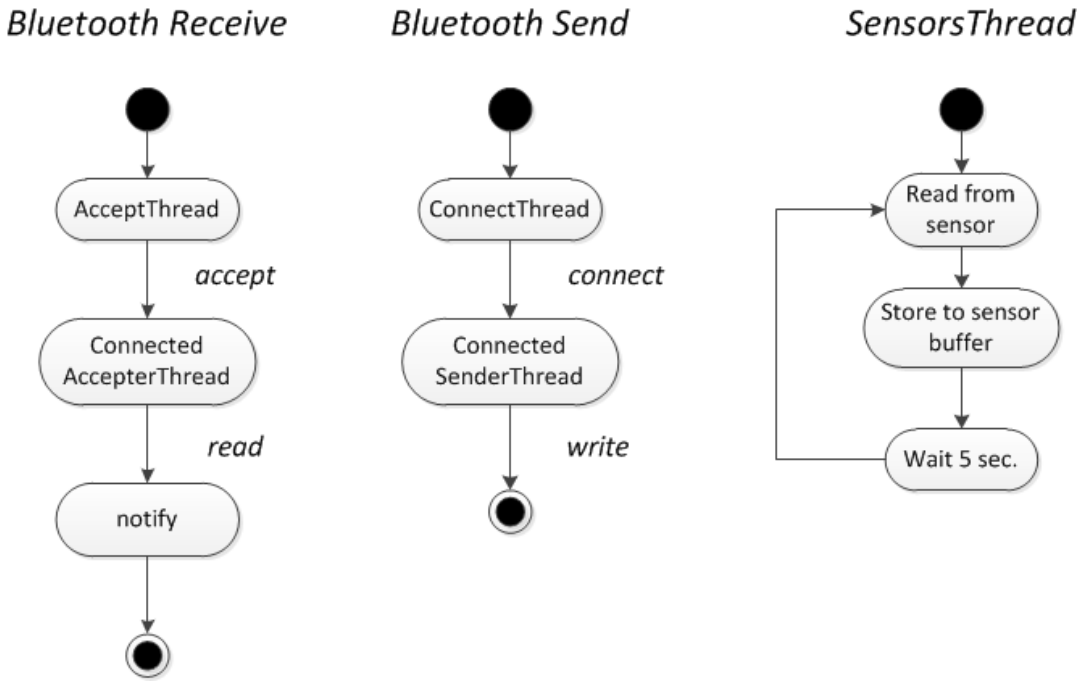
8.3.2 Implementace služeb platformy

V této kapitole se nejprve zmíníme o těch službách, které je interpretační vlákno schopné vykonávat samostatně a poté se podíváme, jak interpretační vlákno spolupracuje s ostatními vlákny pro vykonání složitějších služeb. Výpis všech podporovaných služeb platformy *WSageNt* je uveden v tabulce 4.3. Tato aplikace podporuje všechny služby kromě matematických operací, které tak zůstávají možným námětem pro další vývoj aplikace.

Programovací jazyk Java nám poskytuje vyspělé implementační prostředky pro zajištění synchronizace a výlučného přístupu. Služba pro uspání interpretu tak může být vykonaná přímo v interpretačním vlákně voláním metody `wait(Integer)`. Tato metoda uspí interpretační vlákno `Interpreter` na zvolenou dobu a tím pádem pozastaví činnost interpretu. Implementace služby pro ukončení interpretu je také velmi jednoduchá, stačí pouze odstranit všechny zbývající akce ze zásobníku. Tím pádem je ukončen interpretační cyklus a vlákno `Interpreter` skončí. Další služby, o které se postará interpretační vlákno samostatně, jsou dvě služby pro práci se seznamy. Tyto služby vloží do aktivního registru první prvek seznamu respektive zbytek seznamu kromě prvního prvku. Jedná se o služby, které vznikly na základě inspirace operátorů car a cdr z programovacího jazyka LISP. Služba pro sledování příchozích zpráv má význam pouze v případě, že někdy po ní bude následovat služba pro uspání interpretu do té doby, než přijde zpráva přes Bluetooth rozhraní. Prvně zmíněná služba pouze nastavuje příznak, že se od této chvíle bude sledovat, zda přišla nějaká jednoduchá zpráva. Po příjmu jednoduché zprávy a jejím vložení do tabulky InputBase je v interpretačním vlákně nastaven další příznak, který indikuje doručení jednoduché zprávy od doby sledování.

Služba, která uspí interpretující vlákno do doby, než přijde zpráva přes Bluetooth rozhraní, je také implementována pomocí funkce `wait()`. Po příjmu zprávy je interpretační vlákno vzbuzeno pomocí metody `notify()`. Je - li před touto službou zapnuté sledování příchozích zpráv a zároveň je nastaven příznak indikující její doručení, může interpretační vlákno tuto službu přeskočit a pokračovat, aniž by bylo nutné jej uspat a následně budit. Tato služba již vyžaduje spolupráci s

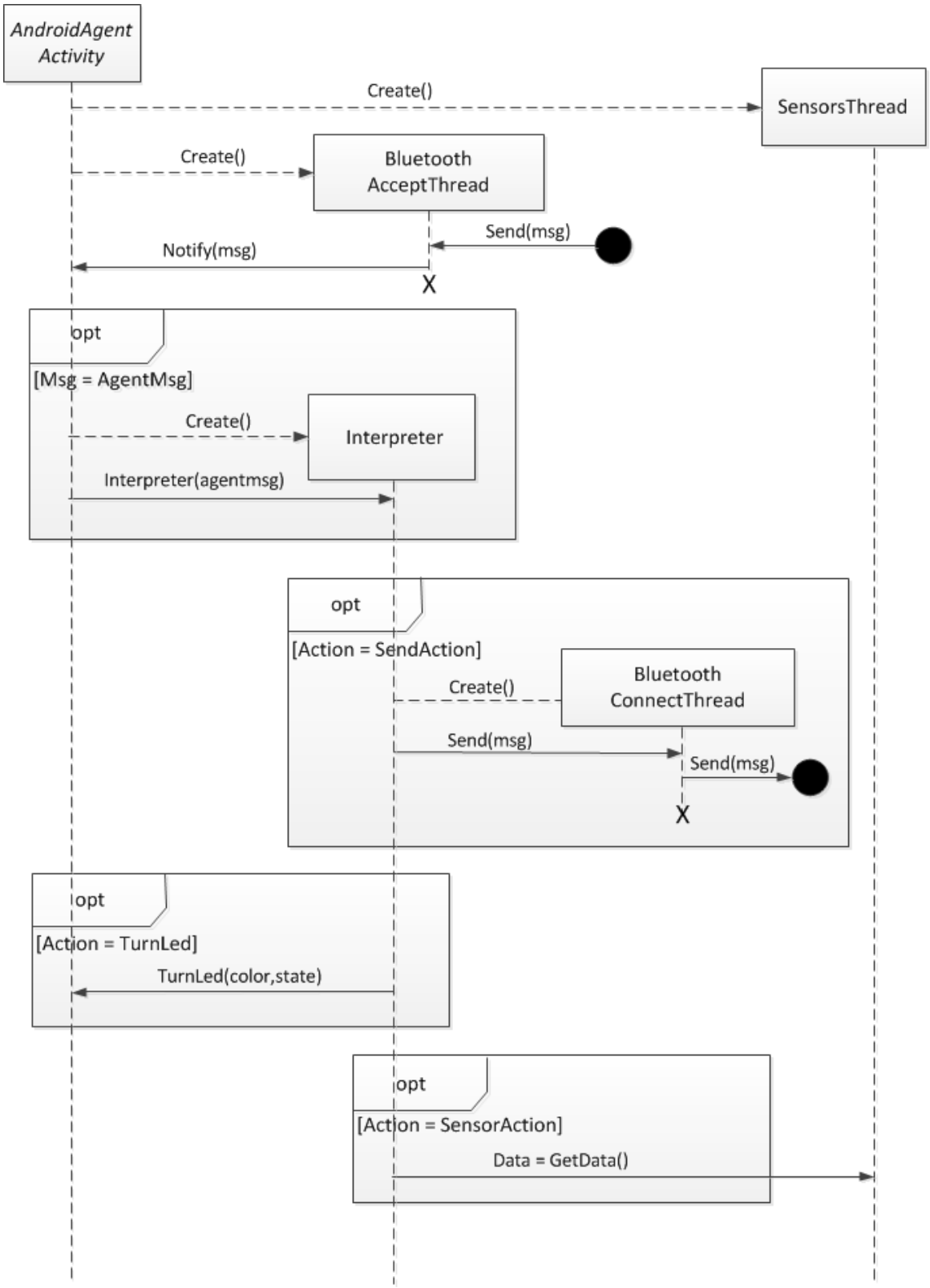
vláknem pro příjem zprávy přes Bluetooth rozhraní, které interpretační vlákno vzbudí, aby mohlo pokračovat. Další služba, která spolupracuje s vláknem objektu třídy `Bluetooth`, ale nyní s vláknem pro odesílání zpráv přes Bluetooth rozhraní, je migrace agenta. Pro vykonání služby, která rozsvěcí a zhasíná LED diody, komunikuje interpret s vláknem `AndroidAgentActivity`, které se stará o vykreslování grafického uživatelského rozhraní, ve kterém jsou LED diody nakresleny. Pro přístup k senzorům pro změření aktuální hodnoty volá interpretační vlákno `SensorsThread`, které změřenou hodnotu uloží do tabulky InputBase. Jelikož nám v tomto momentě běží dvě vlákna zároveň, musíme zajistit jejich vzájemnou synchronizaci. Mohlo by se totiž stát, že následná akce ze zásobníku, která by hledala hodnotu ze senzoru uloženou v Inputbase, by byla provedena dříve, než by ji tam vlákno `SensorsThread` stihlo vložit. Tato akce by potom skončila chybou. Jestliže tedy interpretační vlákno žádá vlákno `SensorsThread` o změření hodnoty, je ihned po odeslání této žádosti uspáno. Po vložení změřené hodnoty do InputBase je interpretační vlákno probuzeno a může pokračovat v činnosti. Na obrázku 8.5 jsou zobrazeny zbylá 3 vlákna, o kterých jsme v této kapitole hovořili, a stavy, ve kterých se v průběhu svého života nachází.



Obrázek 8.5: Zleva jsou zobrazeny stavy vláken objektu třídy `Bluetooth` pro příjem respektive odesílání zpráv. Vpravo je poté vlákno, které měří hodnoty ze senzoru.

8.3.3 Komunikace vláken

V předchozích kapitolách jsme se zmínili o některých vláknech, která zajišťují chod celé aplikace AndroidAgent a ukázali jsme si stavy, ve kterých se tato vlákna při vykonávání své činnosti nacházejí. Pojďme se nyní podívat na to, jakým způsobem mezi sebou tato vlákna komunikují. Tuto skutečnost nejlépe vystihuje obrázek 8.6. Tento obrázek je však pro přehlednost zjednodušený a zobrazuje jen ty podstatné zprávy, které si vlákna zasílají. Zároveň zachycuje situaci doručení a interpretování pouze jedné agentní zprávy. Vlákno ve třídě `Bluetooth` po přijetí zprávy informuje hlavní vlákno a skočí. Ihned je však vytvořeno nové vlákno pro příjem další zprávy a celá situace interpretace příchozí zprávy se tak může opakovat.



Obrázek 8.6: Sekvenční diagram komunikujících vláken.

9 Konzolová aplikace BSCommAndroid

Jak již bylo řečeno výše, *base station* je ten uzel senzorové sítě, který je připojený pomocí mobilního internetu k počítači. Tvoří tak most pro posílání zpráv mezi počítačem a uzly senzorové sítě. Pro vytvoření a odeslání agenta do senzorové sítě je na počítači potřebná spolupráce dvou aplikací. První z nich je uživatelské webové rozhraní *Control Panel*, které bylo vytvořeno už dříve pro posílání agentů na uzly senzorové sítě platformy *WSageNt*. Toto webové rozhraní je beze změny využito také pro zasílání agentů na uzly senzorové sítě na platformě Android. Druhá aplikace, která běží na počítači, je konzolová aplikace *BSCommAndroid*, která komunikuje jak s webovým rozhraním *Control Panel*, tak s *base station*. Jelikož budeme využívat webové rozhranní *Control Panel* beze změny, zůstane i ta část konzolové aplikace *BSCommAndroid*, která komunikuje s tímto uživatelským rozhraním shodná s původní konzolovou aplikací *BSComm* s platformy *WSageNt*. Druhá část této konzolové aplikace, která komunikuje s *base station*, však musí být nahrazena, protože komunikace již nebude probíhat pomocí USB kabelu, jak tomu bylo na platformě *WSageNt*, nýbrž pomocí mobilního internetu, tudíž prostřednictvím TCP/IP socketů. Konzolová aplikace *BSCommAndroid* se spouští se dvěma argumenty. Oba dva argumenty jsou čísla portů pro komunikaci prostřednictvím TCP/IP socketů. První z nich je využit pro komunikaci mezi *base station* a touto konzolovou aplikací. Druhý port pak slouží pro komunikaci s webovým uživatelským rozhraním *Control Panel*. V následujících kapitolách se podíváme na ty části konzolové aplikace *BSCommAndroid*, které se liší od původní verze *BSComm* pro platfromu *WSageNt*.

9.1 Nezbytné funkce pro TCP/IP komunikaci

Funkce, které se starají o komunikaci mezi konzolovou aplikací *BSCommAndroid* a *base station*, jsou implementovány ve třídě `AndroidComm.java`. Jedná se především o funkci, která čeká na připojení od *base station*, funkci, která toto připojení ukončí, funkci pro odesílání agentních a jednoduchých zpráv na *base station* a funkci, která zpracuje přijatou zprávu od *base station*. Pro samotný příjem zpráv je pak vytvořené samostatné vlákno, které si popíšeme v následující kapitole.

Ihned po startu konzolové aplikace *BSCommAndroid* je vytvořen objekt třídy `AndroidComm`, který už ve svém konstruktoru volá funkci `connect()`. Znova se zde jedná o již známé schéma klient - server. Funkce `connect()` tak vytvoří TCP/IP server socket a čeká na připojení od klienta (*base station*) na portu, který je zadaný argumentem při spouštění této konzolové aplikace. Při vývoji této aplikace bylo číslo portu nastaveno na 6789. Po úspěšném připojení klienta je TCP/IP server socket uzavřen a spojení s *base station* je navázáno přes nově vytvořený TCP/IP socket. Dále jsou vytvořeny datové proudy (`InputStream a OutputStream`) pro přenos dat a nové vlákno `WaitForMsgThread`, které se stará o příjem zpráv zaslaných z *base station*.

Další zmíněná funkce je funkce pro odeslání agentní zprávy `sendAgent(int, AgentMessage)`. Jelikož aplikace *BSCommAndroid* z velké části vychází z původní aplikace *BSComm*, zachovává tato funkce z důvodu kompatibility s ostatními třídami stejnou signaturu, jakou měla funkce v původní konzolové aplikaci *BSComm* v prostředí *WSageNt*, nicméně její tělo se liší. Funkce má dva parametry. První parametr je číslo uzlu v senzorové síti, na který bude agentní zpráva pomocí *base station* přeposlána. Druhý parametr je objekt, ve kterém je agent uložen tak, jak byl nadefinovaný ve webovém uživatelském rozhraní *Control Panel*. Hlavním úkolem funkce `sendAgent()` je vytvoření správného formátu agentní zprávy, jenž je nadefinovaný v kapitole 9.3, a následné odeslání této agentní zprávy na *base station*. Po úspěšném či neúspěšném odeslání agentní

zprávy je také odeslána notifikace na webové uživatelské rozhraní, aby uživatel, který nadefinoval a odeslal agentní zprávu, byl informován o výsledku této akce.

Tělo funkce `sendMsg(int, String)` pro odeslání jednoduché zprávy se od funkce pro odeslání agentní zprávy nijak významně neliší. Jediným rozdílem je vytváření formátu jednoduché zprávy. Tato funkce také zachovává stejnou signaturu s původní funkcí pro odesílání jednoduchých zpráv na platformu *WSageNt*. První parametr této funkce je, stejně jako v předchozím případě, číslo uzlu senzorové sítě, na který bude jednoduchá práva odeslána. Druhý parametr je pak samotný obsah zprávy ve formě textového řetězce.

Funkce `messageReceived(String,Integer)`, která slouží pro zpracování přijaté zprávy od *base station*, se od původní funkce z aplikace *BSComm* liší už i svou signaturou. Tuto funkci volá vlákno `WaitForMsgThread` vždy, když obdrží novou zprávu od *base station*. Jediný parametr této funkce je zpráva přijatá od *base station* uložená v textovém řetězci ve formátu, který je popsán v kapitole 9.3. Funkce `messageReceived()` přijatou zprávu dekóduje, uloží její obsah do objektu třídy `TransPacket` a oznámí úspěšné přijetí zprávy. Třída `TransPacket`, je třída už z původní aplikace *BSComm*. Její obsah je poté uložen do databáze, z které je následně načten a zobrazen v uživatelském rozhraní *Control Panel*.

9.2 Vlákno pro příjem zpráv

Vlákno pro příjem zpráv `WaitForMsgThread` je vytvořeno ihned po navázání spojení přes TCP/IP socket a vytvoření datového proudu pro čtení příchozích dat (`InputStream`). Slouží pouze pro příjem zpráv od klienta (*base station*) pomocí funkce `readLine()`. Tato funkce je opět blokující operace, proto musí probíhat v novém vlákně, aby neblokovala chod aplikace. Po úspěšném přijetí zprávy od klienta volá vlákno funkci `messageReceived()` pro její zpracování, viz předchozí kapitola. Vlákno běží tak dlouho, dokud neskončí aplikace *BSCommAndroid*, nebo klient neukončí spojení. Pokud klient ukončí spojení, zavolá vlákno před svým ukončením dvě metody. Nejprve metodu `disconnect()`, která korektně uzavře otevřený socket i oba dva otevřené datové proudy. Poté zavolá metodu `connect()`, která čeká na připojení nového klienta, jak to bylo popsáno v kapitole 9.1.

9.3 Formát přenosu zpráv přes TCP/IP rozhraní

Z předchozích kapitol již víme, jak vypadají základní funkce a vlákno, které se starají o přenos zpráv mezi konzolovou aplikací *BSCommAndroid* a *base station*. Nyní se podíváme na formát přenášených agentních i jednoduchých zpráv.

Agentní zpráva je složená z devíti částí, které jsou od sebe navzájem oddělené symbolem ";". První část je číselná konstanta, která určuje, že se jedná o agentní zprávu. Za ní následuje číslo uzlu, na který bude *base station* agentní zprávu přeposílat. Další čtyři části jsou obsahy jednotlivých tabulek agenta (PlanBase, Plan, BeliefBase, InputBase), definovaných v uživatelském webovém rozhraní Control Panel. Zbylé tři části jsou prázdné. Jsou zde pouze z důvodu zachování kompatibility s přenosem agentních zpráv mezi uzly senzorové sítě. Formát agentní zprávy vypadá následovně:

```
typ_zprávy ; adresa_uzlu ; planBase ; plan ; beliefBase ; InputBase;
; ; ;
```

Při akci migrace agenta z uzlu na jiný uzel senzorové sítě dochází k odeslání nejen výše zmíněných tabulek, ale také všech tří registrů. Po přijetí této agentní zprávy na *base station*, odřízne *base station* první dvě části. Z první části zjistí, zda se jedná o zprávu agentní či jednoduchou. Z druhé části převezme číslo uzlu, na který bude zbytek již nezměněné zprávy přeposílat.

Formát jednoduché zprávy je také textový řetězec složený z devíti na sebe navazujících částí, které jsou rovněž oddělené znakem ";". První část je opět číselná konstanta, která nyní identifikuje zprávu jako jednoduchou. Druhá část je číslo senzorového uzlu, kterému je jednoduchá zpráva adresována. Další tři části, určené pro uložení bází (BeliefBase, PlanBase) a plánu, jsou prázdné. Stejně, jako tomu bylo v případě přenosu jednoduché zprávy mezi uzly senzorové sítě přes Bluetooth rozhraní, tak i při přenosu přes TCP/IP socket je jednoduchá zpráva uložena v části určené pro vstupní bázi InputBase. Formát jednoduché zprávy vypadá následovně:

```
typ_zprávy ; adresa_uzlu ; ; ; ; InputBase ; ; ; ;
```

Z předchozích kapitol už víme, že v části pro InputBase jsou přenášeny dvojice ve tvaru klíč - hodnota. Klíč je adresa, odkud byla zpráva poslána. V tomto případě je jednoduchá zpráva poslána z webového uživatelského rozhraní, je tak nastavena na číslo uzlu, které má *base station*. Ve většině případů to bude číslo jedna. Hodnota je pak obsah zprávy, neboli druhý argument funkce `sendMsg()` popsané v kapitole 9.1.

10 Testování aplikace

Tato aplikace vytváří z mobilního zařízení s operačním systémem Android senzorový uzel, který interpretuje příkazy jazyka ALLL stejně jako uzly senzorové sítě na platformě *WSageNt*. To znamená, že po příchodu agentní zprávy se musí uzly tvořené těmito mobilními zařízeními chovat stejně, jako uzly na platformě *WSageNt*. Pro ověření shodného chování jsem využil služeb simulátoru, který interpretuje akce zadané v jazyce ALLL. K otestování správné funkčnosti této aplikace byly využity některé ukázkové příklady agentů, které se používají na platformě *WSageNt*. V této kapitole si projdeme několik agentních zpráv jazyka ALLL, na kterých byla aplikace testována.

10.1 Sběr dat ze senzoru

Podívejme se nyní na ukázkové agentní zprávy, které testují sběr dat ze senzoru. Přestože mají mobilní zařízení s operačním systémem Android k dispozici více senzorů, v této aplikaci byl pro prezentaci této služby vybrán senzor okolního osvětlení[22]. Ty jsou uvedeny v tabulkách 10.1 - 10.4. Tyto zprávy obsahují pouze část pro aktuální plán, ostatní části nejsou využity. Všechny čtyři uvedené zprávy nejprve rozsvítí zelenou LED diodu. Další akce se už u všech agentů liší. První agent žádá o aktuální hodnotu, další agenti postupně žádají o hodnotu minimální, maximální a průměrnou za zvolený počet měření. Třetí akce všech plánů je přepnutí aktivního registru na registr číslo 1. Následná akce hledá v tabulce InputBase požadovanou hodnotu, kterou posílá na *base station*. Poslední akce pouze zhasne rozsvícenou LED diodu.

Tabulka	**Kód v jazyce ALLL**
Plan	`$(l,(g,1))$(d)&(1)?(s)!(1,&1)$(l,(g,0))`

Tabulka 10.1: Agent měřící aktuální data ze senzoru [5].

Tabulka	**Kód v jazyce ALLL**
Plan	`$(l,(g,1))$(d,(m,15))&(1)?(m)!(1,&1)$(l,(g,0))`

Tabulka 10.2: Agent, který vrátí minimální naměřenou hodnotu.

Tabulka	**Kód v jazyce ALLL**
Plan	`$(l,(g,1))$(d,(M,15))&(1)?(M)!(1,&1)$(l,(g,0))`

Tabulka 10.3: Agent, který vrátí maximální naměřenou hodnotu.

Tabulka	**Kód v jazyce ALLL**
Plan	`$(l,(g,1))$(d,(a,15))&(1)?(a)!(1,&1)$(l,(g,0))`

Tabulka 10.4: Agent, který vrátí průměrnou naměřenou hodnotu.

[22] Ambient light sensor

10.2 Práce s plány a bází znalostí

Následující příklad komplexně otestuje nejen práci s bází znalostí, ale také práci s uloženými plány, registry a také význam zarážky. Zároveň při tom otestuje také služby pro ovládání LED diod, a práci se seznamy. Agentní zpráva již kromě samotného plánu bude obsahovat v tabulce PlanBase dva pojmenované plány, které obsahují akce pro periodické blikání LED diod. Tato agentní zpráva je uvedena v tabulce 10.5.

Nejprve se vykonají akce, které představují aktuální plán. Tyto akce rozsvítí a zhasnou všechny LED diody. Poslední akce v plánu hledá plán se jménem "napln" v tabulce PlanBase a jeho akce přidá na zásobník. Ty již pracují s tabulkou BeliefBase, do které postupně vloží tři n-tice, které obsahují informace o barvě a době, jak dlouho bude dioda svítit. Následně je v tabulce PlanBase vyhledán druhý plán s názvem "blik". Ten postupně vybírá z tabulky BeliefBase n-tice, které obsahují informace o LED diodách. Pomocí služeb pro práci se seznamy vytáhne z n-tic potřebné informace o barvě a času. Poté rozsvítí danou diodu na zadanou dobu. Na konci plán "blik" zavolá sám sebe. Když plán "blik" proběhne třikrát, tzn. že probliknou všechny tři diody, je už tabulka BeliefBase prázdná. Při hledání n-tice v této tabulce tak nastane chyba, neboť se v ní už žádná nevyskytuje. Proto jsou odstraněny všechny akce vykonávaného plánu ze zásobníku až po zarážku, kterou na zásobník vložil prvně volaný plán "napln". Za zarážkou se vyskytuje poslední akce, která znova zavolá plán "napln". Ten opět naplní tabulku BeliefBase a celý proces se opakuje.

Tabulka	**Kód v jazyce ALLL**

PlanBase	`(blik,(&(1)*(led,_,_)&(2)$(f,&1)-&2&(1)$(r,&2)&(3)$(f,&1)`
	`$(l,&3)&(2)$(r,&1)&(1)$(f,&2)$(w,&1)$(l,&3)^(blik)))`
	`(napln,(+(led,r,600)+(led,g,700)+(led,y,800)^(blik)#`
	`^(napln)))`

Plan	`$(l,(r,1))$(l,(g,1))$(l,(y,1))$(w,(300))$(l,(r,0))` `$(l,(g,0))$(l,(y,0))^(napln)`

Tabulka 10.5: Agent, který způsobí blikání LED diod na displeji zařízení [6].

10.3 Průchod agenta sítí

V této kapitole se podíváme na to, jakým způsobem může agent putovat sítí. Průchod agenta sítí si ukážeme na triviálních příkladech, které jsou uvedeny v tabulkách 10.6 a 10.7. V prvním příkladě se agentní zpráva vykoná na dvou uzlech. Na tom, kterému byla původně adresována, ale také na uzlu číslo 3, na který agent migruje. Červená dioda se tedy rozsvítí na obou uzlech. V příkladě druhém je situace odlišná. Agent také migruje na uzel číslo 3, ale vykoná se pouze na něm, protože po migraci agenta je vykonání na původním uzlu zastaveno. Pro příklad složitějšího agenta můžeme vložit akci `$(m,(3))` na počátek plánu z předchozího příkladu. Blikání pak bude probíhat na dvou uzlech.

Tabulka	**Kód v jazyce ALLL**
Plan	`$(m,(3))$(l,(r,1))`

Tabulka 10.6: Agent, který po migraci běží na dvou uzlech.

Tabulka	**Kód v jazyce ALLL**
Plan	`$(m,(3,s))$(l,(r,1))`

Tabulka 10.7: Agent, který po migraci zastaví svou činnost na původním uzlu.

10.4 Vzájemná komunikace uzlů mezi sebou

Pro otestování vzájemné komunikace mezi dvěma uzly pošleme na každý uzel jiného agenta. Ti jsou uvedeni v tabulkách 10.8 a 10.9. První z nich pouze čeká na zprávu od druhého agenta o tom, jakou LED diodu má rozsvítit. Po přijetí zprávy informuje druhého agenta. Druhý agent je tedy řídící. Podobně jako agent uvedený v tabulce 10.7 obsahuje plán pro naplnění tabulky znalostí a plán pro blikání LED diod. Navíc však obsahuje plán "odesli", ve kterém posílá zprávy prvnímu agentovi s informací o barvě diody a času, jak dlouho má svítit.

Tabulka	**Kód v jazyce ALLL**

PlanBase	`(prijem,($(s)&(2)?(3)$(a)!(3,&2)^(blik)))`
	`(blik,(&(1)$(r,&2)&(3)$(f,&1)$(l,&3)&(2)$(r,&1)&(1)`
	`$(f,&2)$(w,&1)$(l,&3)^(prijem)))`

Plan	`^(prijem)`

Tabulka 10.8: Kód agenta, který čeká na příjem zpráv [5].

Tabulka	**Kód v jazyce ALLL**

PlanBase	`(blik,(&(1)*(led,_,_)&(2)$(f,&1)-&2^(odesli)&(1)$(r,&2)`
	`&(3)$(f,&1)$(l,&3)&(2)$(r,&1)&(1)$(f,&2)$(w,&1)$(l,&3)`
	`^(blik)))`
	`(napln,(+(led,r,600)+(led,g,700)+(led,y,800)^(blik)#`
	`^(napln)))`
	`(odesli,($(a)!(2,&2)&(3)$(s)?(2)))`

Plan	`$(l,(r,1))$(l,(g,1))$(l,(y,1))$(w,(300))$(l,(r,0))` `$(l,(g,0))$(l,(y,0))^(napln)`

Tabulka 10.8: Kód agenta, který řídí komunikaci [5].

10.5 Zasílání agentů do sítě

Pro zasílání agentů do uzlů senzorové sítě jsem využíval webové uživatelské rozhraní *Control Panel*, které běží na serveru s veřejnou adresou ihoracek.no-ip.org. Na stejný server jsem umístil také nižší vrstvu webového rozhraní aplikaci *BSCommAndroid*. Jelikož je tato aplikace spuštěna na serveru s veřejnou adresou, může se k této aplikaci připojit uzel představující *base station*. Poté je již možné zasílat jak agentní, tak jednoduché zprávy do uzlů senzorové sítě tvořené zařízeními s OS Android.

11 Současný stav a závěr

V rámci této diplomové práce jsme se seznámili se základními principy bezdrátových senzorových sítí. Prozkoumali jsme platformu *WSageNt* a uzly, které ke své činnosti používá. Seznámili jsme se také s webovým rozhraním *Control Panel*, které slouží k ovládání agentů na uzlech v senzorové síti. Na základě těchto poznatků vznikla aplikace pro mobilní zařízení s operačním systémem Android, která interpretuje příkazy jazyka ALLL. Zůstala tak zachována kompatibilita agentů, kteří fungují na platformě *WSageNt*. Z tohoto důvodu je možné použít webové rozhraní *Control Panel* pro zasílání agentů a zpráv jednotlivým uzlům sítě. Mobilní telefony již mají v dnešní době v základní výbavě senzory a téměř každý člověk má mobilní telefon pořád u sebe. Může být tedy přínosné vytvoření senzorové sítě pomocí těchto zařízení, ať už pro lokalizaci osob, nebo jen pro zjištění stavu okolního prostředí. Na rozdíl od platformy *WSageNt* není uzel představující *base station* připojen k uživatelskému webovému rozhraní *Control Panel* na počítači pomocí USB kabelu, ale pomocí mobilního internetu. Z tohoto důvodu vznikla také konzolová aplikace *BSCommAndroid*, která upravuje tuto stranu komunikace původní aplikace *BSComm*.

Ke komunikaci mezi uzly sítě využívá aplikace modul Bluetooth. Jako námět pro rozšíření aplikace tak zůstává využití technologie WiFi Direct. V současné době to však nemá velký smysl, neboť tato technologie je zatím podporována na malém počtu mobilních zařízení. Jako další rozšíření aplikace mohou být přidány parametry k příkazu v jazyce ALLL, který představuje službu pro přístup k senzorům, aby aplikace mohla variabilně přistupovat i k dalším senzorům a ne jen k tomu pevně zvolenému. Návrh kódů pro jednotlivé senzory je uveden v kapitole 4.3.14 v tabulce 4.4. Posledním vylepšením aplikace může být přidání podpory pro matematické operace, které již jazyk ALLL podporuje.

Senzorová síť vytvořená ze zařízení s operačním systémem Android má oproti platformě *WSageNt* výhodu v připojení *base station*. Ta komunikuje přes mobilní internet a nemusí se vůbec nacházet v blízkosti počítače s uživatelským rozhraním *Control Panel*. Velká nevýhoda u mobilního zařízení je však výdrž na baterie. To vydrží pouze pár dní a to bez zapnutých rádií pro komunikaci. Zapneme - li moduly Bluetooth a modul pro mobilní internet, můžeme počítat s výdrží baterie pouze v řádu hodin.

Literatura

[1] MEDNIEKS, Z., DORNIN, L., MEIKE, G. B., NAKAMURA, M.: *Programming Android*. O'Reilly Media, 1. vydání, Sebastopol, CA, USA, 2011, 482 s., ISBN 978-1-449-38969-7.

[2] RUSSELL, S., NORVIG, P.: *Artificial Inteligence*: *A modern Approach*. Prentice Hall, 2. vydání, New Jersey, USA, 2003. ISBN 0-1310-3805-2.

[3] WOOLRIDGE, M., JENNINGS, N. R.: Inteligent Agents: theory and practise. In *Knowledge Engineering Review*, 1995, s. 115 - 152.

[4] BUSETTA, P., BAILEY, J., RAMAMOHANARO, K.: A Reliable Computational Model For BDI Agents. In *1st International Workshop on Safe Agents, held in conjunction with AAMAS2003*, 2003. [cit. 4. ledna 2012]. Dostupný z www: <http://ww2.cs.mu.oz.au/~jbailey/papers/safeAgents03.pdf >

[5] HORÁČEK, J.: *Platforma pro mobilní agenty v bezdrátových senzorových sítích*. Diplomová práce. FIT VUT v Brně, Brno, 2009.

[6] SPÁČIL, P.: *Mobilní agenti v bezdrátových senzorových sítích*. Bakalářská práce, FIT VUT v Brně, Brno, 2009.

[7] KALMÁR, R.: *Jazyk vyšší úrovně abstrakce pro programování mobilních inteligentních agentů*. Bakalářská práce. FIT VUT v Brně, Brno, 2010.

[8] ZBOŘIL, F. jr., HORÁČEK, J., SPÁČIL, P.: Intelligent Agent Platform and Control Language for Wireless Sensor Network. In *Proceedings of 3rd EMS*, Atény, Řecko, IEEE CS, 2009, s. 6, ISBN 978-0-7695-3886-0.

[9] HORÁČEK, J., ZBOŘIL, F. jr.: Secured agent platform for Wireless Sesnsor Networks. In *Intelligent Information and Database Systems*, part 1, Berlín, Německo, Springer, 2011, s. 476-485, ISBN 978-3-642-20038-0, ISSN 0302-9743.

[10] GÁBOR, M.: *Webové rozhraní pro sledování provozu v bezdrátových sítích*. Diplomová práce. FIT VUT v Brně, Brno, 2010.

[11] DARGIE, W., POELLABAUER, Ch.: *Fundamental of Wireless Sensor Networks: Theory and Practise*. John Wiley & Sons Ltd., Chichester, West Sussex, United Kingdom, 2010, 311s. ISBN 978-0-470-99765-9.

[12] LEWIS, F. L.: Wireless Sensor Networks. In *Smart Environments: Technologies, Protocols and Aplications*. New York, USA, 2004.

[13] PARR, Terence. UNIVERSITY OF SAN FRANCISCO. *ANTLRv3* [online]. [cit. 17.května 2012]. Dostupné z: http://www.antlr.org/

Seznam Příloh

Příloha A: Gramatika jazyka ALLL "Alll.g" pro vývojové prostředí ANTLRWorks

Příloha B: DVD se zdrojovými kódy aplikace, návod na zprovoznění a projektová dokumentace

Příloha C: Poster reprezentující vytvořené dílo

Příloha A

Zdrojový kód gramatiky jazyka ALLL definovaný ve vývojovém prostředí ANTLRWorks, pro vygenerování tříd provádějících lexikální a syntaktickou analýzu.

```
grammar Alll;

options
{
   output = AST;
}

@parser::header
{
   package cz.vutbr.fit.xskace03.dip;
}

@lexer::header
{
   package cz.vutbr.fit.xskace03.dip;
}

agent       :     seznam_planu';' seznam_akci';' seznam_ntic';' seznam_input';' reg1';'reg2';'
                  reg3';' EOF -> ^(AGENT seznam_planu seznam_akci seznam_ntic seznam_input
                  reg1 reg2 reg3);

seznam_planu:     (plan)* -> ^(PLANY plan*);
seznam_akci :     (akce)* -> ^(AKCE akce*);
seznam_ntic :     (ob_ntice)* -> ^(NTICE ob_ntice*);
seznam_input:     (in_base)* -> ^(INPUT in_base*);
reg1        :     (ob_ntice)* -> ^(R1 ob_ntice*);
reg2        :     (ob_ntice)* -> ^(R2 ob_ntice*);
reg3        :     (ob_ntice)* -> ^(R3 ob_ntice*);

in_base     :     '(' (a=NUMERIC | a=RETEZEC)','   ob_ntice ')'  -> ^($a ob_ntice);

plan        :     '(' RETEZEC ',' '(' seznam_akci ')' ')'  -> ^(RETEZEC  seznam_akci);

akce        :     '+' ob_ntice -> ^('+' ob_ntice) |
                  '-' ob_ntice -> ^('-' ob_ntice)|
                  '!' '(' NUMERIC ',' ob_ntice ')' -> ^('!' NUMERIC ob_ntice)|
                  '!' '(' registr ',' ob_ntice ')' -> ^('!' registr ob_ntice)|
                  '?' '(' NUMERIC ')'  -> ^('?' NUMERIC) |
                  '?' '(' RETEZEC ')'  -> ^('?' RETEZEC) |
                  '@' '(' seznam_akci ')' -> ^('@' seznam_akci) |
                  '*' ob_ntice  -> ^('*' ob_ntice) |
                  '$' '(' sluzba ')'  -> ^('$' sluzba) |
                  '^' '(' RETEZEC ')' -> ^('^' RETEZEC) |
                  '&' '(' NUMERIC ')'  -> ^('&' NUMERIC) |
                  '#' ;

sluzba      :     RETEZEC |
                  a= RETEZEC ',' registr -> ^($a registr)|
                  a= RETEZEC ',' ob_ntice -> ^($a ob_ntice)|
            ;

cislo       :     NUMERIC -> ^(N NUMERIC);

ob_ntice    :     '(' ob_polozky ')'  -> ^(N ob_polozky) |
                  registr ;

ob_polozky  :     ob_polozka ( ',' ob_polozka)* -> ob_polozka (ob_polozka)* ;
ob_polozka  :     RETEZEC |  ob_ntice | '_' | NUMERIC ;
```

```
registr     :     '&' NUMERIC -> ^('&' NUMERIC);

led         :     (a=RETEZEC) (','  NUMERIC)? -> ^(N  $a (NUMERIC)?);

data        :     (a=RETEZEC) ',' NUMERIC -> $a NUMERIC;

NUMERIC     :     ('0'..'9')+ ;

RETEZEC     :     ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;

AGENT       :     '-agent';

PLANY       :     '-plany';

AKCE        :     '-akce';

NTICE       :     '-ntice';

INPUT       :     '-input';

N           :     '-n';

R1          :     '-r1';

R2          :     '-r2';

R3          :     '-r3';
```