VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

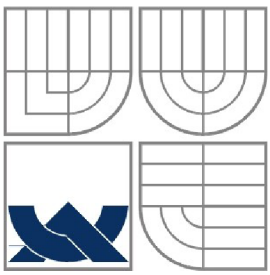
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# MONITOROVÁNÍ STAVU BEZDRÁTOVÝCH SENZOROVÝCH SÍTÍCH AGENTY

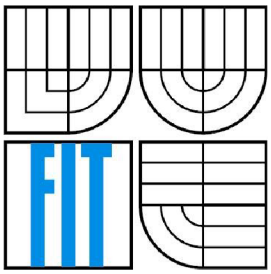
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. MAREK HOUŠŤ
AUTHOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# MONITOROVÁNÍ STAVU BEZDRÁTOVÝCH SENZOROVÝCH SÍTÍCH AGENTY

AGENT-BASED MONITORING OF WIRELESS SENZOR NETWORKS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MAREK HOUŠŤ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. FRANTIŠEK ZBOŘIL, Ph.D.

BRNO 2011

*Zadání diplomové práce/12684/2010/xhoust00*

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav inteligentních systémů Akademický rok 2010/2011

# Zadání diplomové práce

Řešitel: **Houšť Marek, Bc.**
Obor: Inteligentní systémy
Téma: **Monitorování stavu bezdrátových senzorových sítích agenty**
**Agent-Based Monitoring of Wireless Sensor Networks**
Kategorie: Umělá inteligence

Pokyny:

1. Nastudujte problematiku bezdrátových sítích a aplikacemi, pro které mohou být použity.
2. Seznamte se se systémem WSageNt a jeho podporou přes klientské webové rozhraní. Dále se seznamte s jazykem nesC a systémem TinyOS.
3. Navrhněte agentní prvky, které umožní sledovat stav celé sítě, nebo její části, případně upozorňovat klientskou aplikaci na výskyt definovaných událostí.
4. Agentní systémy implementujte a patřičně rozšiřte i jednak samotné webové rozhraní, tak i agentní platformu v jednotlivých uzlech.
5. Diskutujte výhody a nevýhody řešení projektu a uveďte možná rozšíření systému do budoucna.
6. Vytvořte poster formátu A2, který bude názorně demonstrovat vytvořené dílo, jeho architekturu, principy a použitelnost pro reálné aplikace

Literatura:

- Holger, K. and Willig A.: Protocols and Architectures for Sensor Networks
- CrossBow: XServer Manual

Při obhajobě semestrální části diplomového projektu je požadováno:

- První dva body zadání.

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

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

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

Vedoucí: **Zbořil František, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 20. září 2010
Datum odevzdání: 25. května 2011

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

L.S.

doc. Dr. Ing. Petr Hanáček
*vedoucí ústavu*

# Abstrakt

Čtenář se nejprve seznámí s historií a možnými aplikacemi pro bezdrátové senzorové sítě. Práce poté popisuje problematiku bezdrátových senzorových sítí a implementační nástroje pro bezdrátové senzorové sítě. Práce také představuje systém WSageNt a jeho klientské webové rozhraní. Praktická část práce se odvíjí od návrhu agentních prvků. Popisuje postupnou implementaci, testování a praktickou realizaci těchto agentních prvků v systému WSageNt. Na závěr práce diskutuje výhody a nevýhody řešení projektu a uvádí možná rozšíření systému do budoucna.



# Abstract

The reader is first introduced with history and possible applications for wireless sensor networks. The thesis then describes the problems of wireless sensor networks and implementation tools for wireless sensor networks. The thesis also presents the system WSageNt and its client web interface. Practical part is based on the design of agent elements. Describes gradual implementation, testing and practical realization of these agent elements in the system WSageNt. The final part discusses the advantages and disadvantages of the project and identifies possible future expansion of the system.



# Klíčová slova

Senzor, bezdrátová senzorová síť, WSN, mote, nesC, TinyOS, ALLL, Iris, Crossbow, XBow, WSageNt.

# Keywords

Sensor, wireless sensor network, WSN, mote, nesC, TinyOS, ALLL, Iris, Crossbow, XBow, WSageNt.


# Citace

Marek Houšť: Monitorování stavu bezdrátových senzorových sítích agenty, diplomová práce, Brno, FIT VUT v Brně, 2011.

# Monitorování stavu bezdrátových senzorových sítích agenty

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Františka Zbořila, Ph.D. Další informace mi poskytl pan Ing. Jan Horáček. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Marek Houšť
25.5.2011

## Poděkování

Chtěl bych poděkovat především panu Ing. Františku Zbořilovi, Ph.D. za vedení během tvorby diplomové práce a poskytnutou odbornou pomoc. Také bych chtěl poděkovat panu Ing. Janu Horáčkovi za uvedení do problematiky a další výborné konzultace, které mi při práci velmi pomohly.

# Obsah

Úvod ... 3

1 Senzorové sítě a jejich aplikace ... 5
1.1 Historie senzorových sítí ... 5
1.2 Aplikace bezdrátových senzorových sítí ... 6
1.2.1 Vojenské aplikace ... 6
1.2.2 Civilní aplikace ... 7
1.3 Problematika bezdrátových senzorových sítí ... 9
1.3.1 Topologie sítě ... 9
1.3.2 Spotřeba energie ... 11
1.3.3 Distribuované zpracování ... 11
1.3.4 Bezpečnost ... 12
1.4 Hardwarová platforma pro bezdrátové senzorové sítě ... 12

2 Implementační nástroje ... 14
2.1 Jazyk nesC ... 14
2.1.1 Komponenty ... 14
2.1.2 Rozhraní komponent ... 15
2.1.3 Generické komponenty ... 16
2.2 Systém TinyOS ... 17
2.2.1 Architektura hardwarové abstrakce ... 17
2.2.2 Překlad aplikace ... 17
2.2.3 TOSSIM ... 18
2.3 Agentní jazyk ALLL ... 19

3 Systém WSageNt ... 21
3.1 Základní popis systému ... 21
3.2 Funkce agentní platformy ... 22
3.3 Funkce webového rozhraní ... 25

4 Návrh agentních prvků ... 26
4.1 Základní popis ... 26
4.2 Externí flash paměť ... 27
4.2.1 Možnosti přístupu ... 27
4.2.2 Čtení a zápis dat ... 28
4.3 Průchod agenta sítí ... 28
4.3.1 Pomocné služby ... 28
4.3.2 Sestavení agenta ... 29
4.4 Analýza výsledků ... 31

5 Implementace agentních prvků ... 32
5.1 Výběr záznamů z platformy ... 32
5.2 Doručení záznamů do základnové stanice ... 34

5.2.1 Potřebné služby....34
5.2.2 Vytváření agenta otroka....35
5.3 Výsledná aplikace v ALLL....36
5.3.1 Popis cesty agenta otroka....39
5.3.2 Testování....41
5.4 Analýza a vykreslení tras agentů....41
5.4.1 Indikace sesbíraných záznamů v databázi....41
5.4.2 Způsob roztřídění....42
5.4.3 Způsob vykreslování....42
5.4.4 Testování roztřídění a vykreslování....44
5.4.5 Omezení vykreslování....45
5.5 Implementační problémy....45
5.5.1 Doplnění funkcionality....45
5.5.2 Problémy při přenosu agenta....46
5.5.3 Problémy ve spolehlivosti....47
6 Praktické experimenty se systémem WSageNt....49
6.1 Naměřené hodnoty....49
6.2 Zhodnocení....51
7 Výhody a nevýhody řešení a možná rozšíření do budoucna....52
Závěr....55
Literatura....56
Seznam použitých zkratek....58
Seznam příloh....59
Příloha 1. Postup zprovoznění....60
Příloha 2. Pseudokódy ALLL aplikací....62
Příloha 3. Náhledy rozšíření systému WSageNt....65
Příloha 4. Příklady zobrazení vykreslených tras agenta....70

# Úvod

Bezdrátové senzorové sítě (WSN[1]) jsou jednou z nejvíce se rozvíjejících technologií 21. století a novou třídou distribuovaných systémů, jelikož využívají potenciálu mnoha méně výpočetně výkonných jednotek, které spolu komunikují. Senzor je obecně čidlo nebo snímač a senzorové sítě se sestávají z mnoha těchto senzorů. Senzory umožňují sledovat různé fyzikální veličiny a převádět je na signál, který lze dálkově přenášet a dále využívat. Mohou měřit fyzikální veličiny elektrické, magnetické, mechanické, optické, tepelné, akustické a chemické. Celé sítě tak umožňují lépe porozumět oblasti, kde jsou nasazeny. Jejich rozvoj a nasazení ovlivňují neustále se rozvíjející technologické trendy, a tak se objevují stále nové a nové oblasti použití. Oblasti použití WSN jsou od medicíny, přes průmysl, až po domácnosti. Senzory v těchto aplikacích můžou být velké či malé a propojeny mohou být bezdrátově či drátově. Je zcela jasné, že mikrosenzory propojené bezdrátově nabízejí určitě větší potenciál než velké senzory propojené drátově.

Pro bezdrátovou komunikaci senzorových sítí by bylo teoreticky možné použít různé komunikační technologie. Bezdrátové technologie se obecně snaží být většinou co nejrychlejší, ale WSN mají oblast zájmu především v úspoře energie. Nejméně vhodnou technologií pro WSN by byla technologie WiFi, jelikož spotřebovává obrovské množství energie. Dokáže sice přenášet data velkou rychlostí, ale tento potenciál velké rychlosti není u WSN třeba. Méně vhodnou technologií je také technologie Bluetooth. Ta je nevhodná díky spotřebě energie v pohotovostním režimu. Tuto spotřebu má velkou díky tomu, že prvek s Bluetooth nelze jednoduše do tohoto režimu přepnout. Technologie Bluetooth je navíc uzpůsobena spíše pro mobilní telefony pro tzv. spojení „point-to-point“, ale WSN využívají vzájemné komunikace stovek až tisíců uzlů. Zatím nejvhodnější bezdrátovou komunikační technologií používanou v WSN je technologie ZigBee. Tato technologie se snaží vyplnit oblast zaměření, kde stávající bezdrátové technologie nejsou zcela vhodné. Cílem ZigBee je být nízkoenergetickou, levnou, nenáročnou na výpočetní zdroje a přitom spolehlivou a bezpečnou technologií. Hodí se tedy na to, pokud požadujeme bezdrátové sítě s bateriovým napájením a chceme přenášet malé objemy dat v dlouhých časových intervalech. Je totiž potřeba, aby baterie v jednotlivých uzlech vydržely řádově roky. Toto by s technologií WiFi bylo nemyslitelné a s technologií Bluetooth nemožné. S technologií Bluetooth by baterie, s takovou spotřebou jakou má, vydržely řádově dny. Na druhou stranu je třeba počítat s tím, že aplikace pro WSN využívají jednoduchý a levný hardware, a tak musí být nenáročné na výpočetní výkon, paměť a rychlost komunikace.

WSN mohou být v různých stavech podle toho, v jakém prostředí jsou a jakou aplikaci provádějí. Abychom těchto stavů využili, tak je užitečné tyto stavy vhodným způsobem monitorovat. Agent je intuitivně aktivní prvek systému. Má svou roli a tu dokáže naplňovat samostatně bez externího řízení. Jedná tedy na základě vlastního řízení, podnětů přijatých z vnějšího prostředí a komunikace s ostatními agenty. Situováním agentů do systému WSN dokážeme činnost tohoto systému monitorovat, spravovat a řídit. Zde pod pojmem agent rozumíme umělého agenta, jakož to člověkem vytvořené dílo. Agenti mohou mít obecně hardwarovou či softwarovou podobu. V této práci se vždy bude jednat o softwarovou podobu těchto agentů.

Účelem této diplomové práce je seznámit čtenáře jak s problematikou, tak se samotnými WSN. WSN práce popisuje od historie až po současné aplikace, které mohou být v těchto sítích použity. Dále se práce zabývá hardwarovými prvky WSN a jejich implementačními nástroji (programovacím

1 Wireless sensor networks (WSN).

jazykem nesC[2], systémem TinyOS a simulačním nástrojem TOSSIM). V části implementační nástroje je popsán i agentní jazyk ALLL[3], který byl vyvinut zde na VUT FIT a vytváří softwarovou podobu zmiňovaných agentů. Práce dále seznamuje se systémem WSageNt, jakož to systémem pro monitorování stavu WSN. Stručně popisuje jeho dělení a poté funkce včetně podpory přes klientské webové rozhraní. Po představení systému WSageNt následuje návrh agentních prvků, které spočívají ve sledování a monitorování pohybu samotných agentů. Kapitola implementace popisuje nejen způsob implementace daných agentních prvků, ale zabývá se i testováním a implementačními problémy. Praktická realizace agentních prvků na hardwarových uzlech IRIS je poté umístěna v další samostatné kapitole. Výhody a nevýhody dané realizace s návazností na možná rozšíření systému do budoucna jsou diskutovány v poslední kapitole a závěr práce popisuje přínosy realizace a celkové zhodnocení.

2 Network embedded systems C (nesC).

3 Agent Low Level Language (ALLL).

# 1 Senzorové sítě a jejich aplikace

Tato kapitola zprvu pojednává o historii vývoje WSN a popisuje jejich možné použití. Větší část této kapitoly se dále zabývá problematikou WSN z pohledu topologie sítě, spotřeby energie, distribuovaného zpracování a bezpečnosti. Na závěr kapitola představuje hardwarovou platformu IRIS pro WSN.

## 1.1 Historie senzorových sítí

Vznik a počátek vývoje WSN tkví ve vojenském výzkumu, jelikož tento vývoj nebyl zcela jednoduchý a levný. Spojuje celkem oblasti snímací, komunikační a výpočetní. WSN poté těží ze samostatného vývoje těchto oblastí.

Jedno z prvních použití technologie, která se dá nazvat senzorová, bylo okolo roku 1949. V tomto roce začal vývoj systému SOSUS[4] americkou armádou jakožto výzkum podmořské obrany. SOSUS byl akustický sledovací systém, který byl umístěn na strategických místech mořského dna severního Atlantského oceánu, a jeho cílem bylo sledování tichých sovětských ponorek. Avšak až v 70. a 80. letech došlo k výraznému vývoji technologií jak v pobřežních, tak podvodních systémech. Výsledkem bylo, že se ze systému SOSUS vyvinul systém IUSS[5], který se již stal integrovaným podmořským dozorovým systémem. Nyní jsou tyto systémy po odtajnění v roce 1991 používány pro pro vědecké projekty NOAA[6], jako je například sledování pohybu velryb nebo seismické aktivity oceánu [1].


Obrázek 1.: Prvky distribuované senzorové sítě okolo roku 1985 [2].

Vývoj moderních distribuovaných senzorových sítí lze datovat až na rok 1980, kdy program DSN[7] financovala agentura DARPA[8]. Přispěl k tomu předchůdce internetu Arpanet. Byla snaha o rozšíření komunikace přes Arpanet na senzorovou síť, která předpokládala mnoho distribuovaných

4 Sound Surveillance System (SOSUS).

5 Integrated Undersea Surveillance System (IUSS).

6 National Oceanic and Atmospheric Administration (NOAA).

7 Distributed Sensor Networks (DSN).

8 Defense Advanced Research Projects Agency (DARPA).

nízko nákladových snímacích uzlů, které by pracovaly samostatně a komunikovaly by navzájem. V této době bylo velmi málo technologických komponent, proto jakýkoliv vývojový program musel řešit distribuovanou výpočetní podporu, zpracování signálů a sledování (avšak řešilo se pouze akustické sledování). Největším úspěchem byl vývoj platformy pro akustické sledování nízko letícího letadla. Existovalo 9 akustických senzorů uspořádaných do tří sousledných trojúhelníků a jejich komunikace se odehrávala přes Ethernet a mikrovlnné rádio. Nutné je připomenout, že se jedná o 80. léta, takže jednotlivé uzly byly tak velké, že musely být na vozidlech, a všechny komponenty byly vyrobeny na zakázku. Na *obrázku 1.* lze vidět jednotlivé komponenty [2].

Až pokrok v počítačích a komunikacích v 21. století způsobil významný posun ve výzkumu WSN. Aktuální WSN využívají technologie a provádějí funkce, o kterých se před 20ti lety ani nesnilo. Senzory, procesory a komunikační zařízení jsou čím dál menší a levnější. Samotné senzory, obsahující snímání, výpočetní sílu a komunikaci na jednom čipu, jsou ve velikosti srovnatelné s krabičkou od sirek a hmotnosti řádově v gramech. Výzkum se také přesunul z čistě vojenského či akademického pojetí do oblasti komerční. Nejznámější společnosti vyvíjející senzorové systémy jsou Ember, Crossbow a Sensoria [2].

Budoucnost ukazuje společnost Dust Inc., Berkeley, CA. Tato společnost vyvíjí MEMS[9] senzory schopné snímání a komunikace, které by měly být velikosti krychlového milimetru. Je jasné, že WSN tvořená tímto „chytrým prachem“ bude vytvářet zcela výjimečné oblasti užití [2]. Jádro takového senzoru by mělo být poháněno speciálním nízkovýkonným a nízkoenergetickým mikroprocesorem s rozhraním do speciálního ADC[10] a odchozí a příchozí komunikace. Energetický zdroj by měl být řešen pomocí milimetrové baterie, solárního článku a kondenzátoru. Aplikační data poté budou uložena v malé integrované SRAM paměti o několika jednotkách kilobajtů. Zatím poslední reálný prototyp tohoto senzoru (rok 2003) měl celkový objem 16 $mm^3$ a místo mikroprocesoru byl použit jednoduchý FSM[11], který samozřejmě nemohl být přeprogramován [3].

## 1.2 Aplikace bezdrátových senzorových sítí

Možné aplikace WSN mohou být rozdělovány z pohledu vojenského použití a civilního použití, které lze poté dále dělit na monitoring životního prostředí, detekci katastrof, vědecké studie, zdravotnictví, stavitelství a průmysl.

### 1.2.1 Vojenské aplikace

V minulosti byl právě vojenskými aplikacemi motivován výzkum a vývoj senzorových sítí. O těchto příkladech vojenského užití v minulosti hovořila *kapitola 1.1*. Z principu jde vždy o nějakou detekci a monitoring pozemních, námořních a vzdušných sil za účelem obrany. Může ale jít i o průzkum nepřátelských vojsk za účelem výběru vhodného útoku. WSN tak můžou být nedílnou součástí vojenského velení a řízení komunikace. Příkladů nynějšího použití senzorových sítí ve vojenství může být mnoho. Zajímavou aplikací je SHM[12]. Je to protitankový minový systém, který je zajímavý v tom, že se nechová jako statická překážka. Inteligentně tvoří dynamickou překážku tak, že díky senzorové komunikaci mezi minami dokáže reagovat na pokusy nepřítele o její přerušení. Senzory

9 Micro Electro Mechanical Systems (MEMS).

10 Analog-to-digital converter (ADC).

11 Finite-state machine (FSM).

12 Self-Healing Mines (SHM).

tyto pokusy vyhodnocují a na základě tohoto vyhodnocení dojde k samovolnému přesunutí miny pomocí malých raketových trysek. Tímto dochází k samotné regeneraci minového pole. Tento projekt vede společnost SAIC[13] společně se společnostmi Sensoria, UPCO[14] a dalšími [4]. Méně zajímavé, ale užitečné použití senzorů, může být v tzv. městském válčení, kdy senzory odhalují například chemické útoky nebo pohyby nepřítele. Lze i identifikovat pozici odstřelovače pomocí akustických rázových vln, které pocházejí od zvuku výstřelu. Tato akustická senzorová síť PinPtr je vyvíjena na Vanderbilt University [5]. Příkladem použití ještě částečně vojenským může být zabezpečení kritických budov bezdrátovými senzory kvůli včasnému odhalení hrozeb teroristického útoku. Bezdrátové senzorové ad hoc sítě poskytují v tomto zabezpečení větší flexibilitu ve srovnání s fixními senzory [2].

### 1.2.2 Civilní aplikace

Až dostupnost techniky a nízkonákladových senzorů vedlo k použití distribuovaných WSN v průmyslu a komerčních či neziskových aplikacích. Příkladů takových aplikací může být několik. Časté je použití na monitoring životního prostředí, detekci či zmírnění katastrof nebo sledování provozu vozidel.

#### Monitoring životního prostředí

Při monitorování životního prostředí se mohou například studovat vegetační reakce na klimatické trendy. Senzory tak sledují a měří populaci různých druhů živočichů či rostlin. Příkladem takové aplikace může být výzkumný projekt PODS. Tento projekt prošetřuje proč ohrožené druhy rostlin rostou jen v jedné určité oblasti a ne v oblastech sousedních. Projekt je pod záštitou University of Hawaii at Manoa a jednotlivé uzly jsou rozmístěny v Hawaii Volcanoes National Park. Uzly v WSN sbírají data a posílají je do základnové stanice, která je dělá dostupnými přes síť internet. Takto lze poměrně neinvazivním způsobem sledovat stanoviště velmi citlivé na lidskou přítomnost [5].

Do oblasti použití v životním prostředí lze zahrnout také projekt Intel's Wireless Vineyard. Tato aplikace v oblasti zemědělství (vinice) sbírá a interpretuje data, která později používá k aktivnímu rozhodování na zjištění přítomnosti parazitů a umožňuje tak výběr vhodného druhu insekticidu. Sběr dat probíhá tím způsobem, že lidé nebo psi mají na sobě malá zařízení, která komunikují s uzly umístěnými přímo ve vinici [5].

#### Detekce katastrof a vědecké studie

Použití WSN při detekci katastrof či vědeckých studiích je výhodné především v těch oblastech, kde tato detekce či studium odrazuje lidskou přítomnost. Jsou to především oblasti hurikánů, lesních požárů či sopečné činnosti. Takový lesní požár může být snadno detekován pomocí integrované sítě senzorů. Tato síť kombinuje pozemní čidla pro sledování úrovně vlhkosti, směru a rychlosti větru se satelitními snímky, které dopomáhají k určení míry potenciálního vzniku požáru v cílových regionech. Taková aplikace tedy nabízí i informace o možném směru, ve kterém se daný požár šíří, a celkově tak dopomáhá ke koordinované a včasné reakci na požár díky včasnému dodání dostupných dat. Příkladem konkrétní implementace tohoto systému může být systém od Australské společnosti Telepathx Ltd, která nabízí WSN pro real time monitoring požáru. Jejich sledování umožňuje i 3D mapování požáru a požární oznámení probíhá přes klasické komunikační linky (GSM, CDMA, 3G) [6].

---

13 Science Application International Corporation (SAIC).

14 Universal Propulsion Company (UPCO) – vývoj a testování raketových motorů.

Monitorování aktivní sopečné činnosti se provádí pomocí sběru seizmických, podzvukových a nízko frekvenčních signálů. V projektu, který realizovali na Harvard University na dvou sopkách v Ekvádoru, bylo nasazeno 16 nízkoenergetických senzorů poháněných konvenčními bateriemi v otvoru o rozloze třech čtverečních kilometrů. Tento typ aplikace je jeden z náročnějších, jelikož pro monitoring aktivních sopek je typická nutnost vysoké rychlosti přenosu a vysoké věrnosti dat. Sběr dat je však také klasicky prováděn po multi-hop síti do základnové stanice, která je přenese až na observatoř [7].

Sledování provozu vozidel je další vědeckou aplikací WSN. Mnoho křižovatek má buď viditelné nebo neviditelné klasické senzory pro detekci vozidel a řízení dopravního značení. Dále obsahují často videokamery sledující pouze kritické úseky, protože tyto videokamery a klasické senzory jsou nákladné. Cílem u této aplikace je přejít na levné bezdrátové ad hoc sítě, které budou nasazeny na každé silniční křižovatce a vestavěnými funkcemi budou například detekovat vozidla, počítat vozidla a odhadovat jejich rychlost. Senzory v těchto sítích budou komunikovat, a tím vznikne celkový dopravní obraz, podle kterého může být automaticky nastavováno dopravní značení. Jiný koncept popisuje senzory nainstalované do každého vozidla, kdy pohybem těchto vozidel bude docházet k vyměňování informací, například o umístění dopravní zácpy, dopravní rychlosti či hustotě dopravy [2].

### Zdravotnictví

WSN jsou široce používány i ve zdravotnictví v oblastech lékařského výzkumu a zdravotní péče. Jsou konstruovány tak, že snímají real time fyziologická data, která jsou po nasnímání jednoduše stažena do počítače (či PDA) lékaře k vyhodnocení. Takto je možné ukládat data i s celkovou historií léčby. O tuto aplikaci se snaží Harvard University ve spolupráci s lékařskou fakultou na Boston University. Navrhli soubor zařízení založených na platformě senzorových uzlů MICA21, které shromažďují srdeční frekvence, saturace kyslíkem a EKG[15]. Pomocí WSN lze i kontrolovat pohyb léků uvnitř nemocnice pomocí identifikátorů RFID[16]. Použití WSN může být i v pečovatelské službě, kdy se redukují personální náklady tím, že se detekují pády, přechod do bezvědomí, případně i stravování a příjem léků [5].

### Stavitelství

Aplikace WSN se stále častěji objevuje i ve stavitelství (domácnostech) jako koncept tzv. chytrého domu. Variant použití integrovaných senzorů v domu je samozřejmě několik. Zajímavým použitím z hlediska WSN je sledování teploty více senzory. Na tomto základě je tak možné v domě nastavovat například zónovou klimatizaci v jednotlivých pokojích. Další variantou je použít senzory jako bezpečnostní opatření proti zlodějům, například sledováním vibrací. Pomocí těchto aplikací tak lze jednoduše snižovat výdej energie při zvýšení životních podmínek [5].

### Průmysl

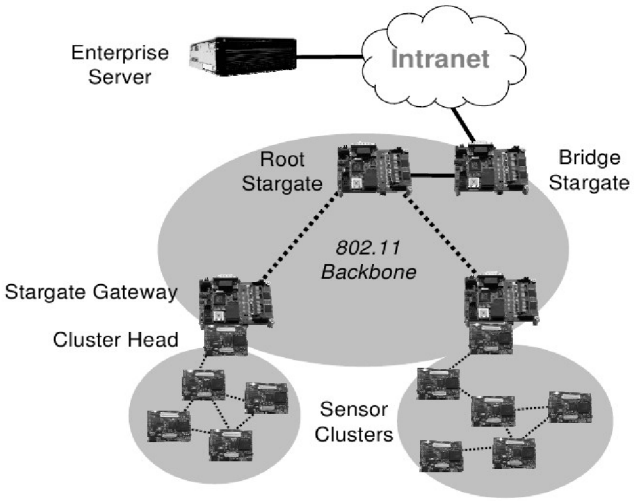
Poslední nejméně překvapivou aplikací WSN je jejich použití v samotném průmyslu. V průmyslu jde především o snížení nákladů a zvýšení výkonu strojů. Aplikace se tak zabývají efektivní a bezpečnou automatizací výrobních či montážních linek, ale i monitoringem stavu vibrací, opotřebení a mazání strojů [2].

---

15 Elektrokardiogram (EKG) - záznam časové změny elektrického potenciálu způsobeného srdeční aktivitou.

16 Radio Frequency Identification (RFID).

Monitoringem stavu jednotlivých strojů se zabývá tzv. PdM[17] technologie. Tato technologie například analyzuje frekvence vibrací v závislosti na čase, ve kterých pak hledá frekvenční vzory, které naznačují opravu či výměnu stroje. Z hlediska WSN je důležité to, že se toto monitorování provádí bezdrátově a výrazně levněji než při použití drátových senzorových sítí. Navíc tato výhoda nízkých nákladů neplatí pro klasické bezdrátové technologie, které jsou pouhou náhradou drátů. Zde je výhodná právě vlastnost multi-hop propagace výsledků měření do základnové stanice typická pro WSN. Proto se nejen v technologii PdM, ale i v celkovém průmyslu začínají objevovat multi-hop senzorová řešení. Příkladem takové architektury nasazené v průmyslu může být architektura FabApp. Jak je vidět na *obrázku 2.*, tak jednotlivé bateriově poháněné senzory sbírají data do základnové stanice (brány) a ta tyto výsledky přeposílá dále [8].




Obrázek 2.: FabApp síťová architektura [8].

## 1.3 Problematika bezdrátových senzorových sítí

Problematika WSN se dá rozdělit do následujících samostatných částí. Jedná se o část zabývající se topologií sítě, spotřebou energie, distribuovaným zpracováním a bezpečností.

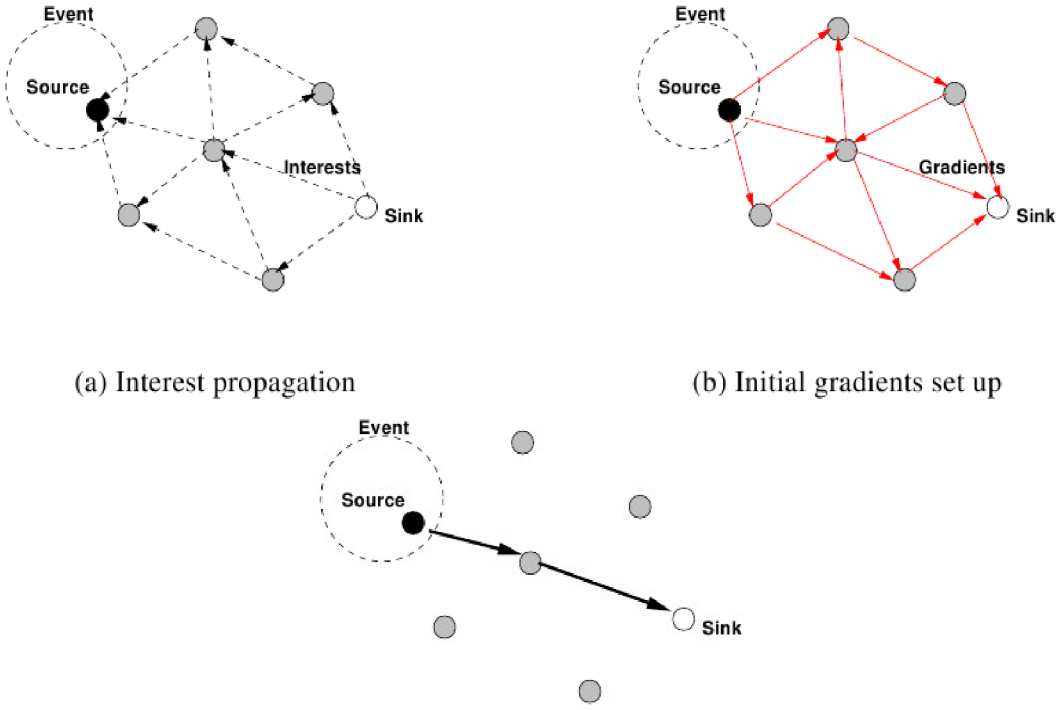
### 1.3.1 Topologie sítě

Právě ta vlastnost, která se v aplikacích popsaných v *kapitole 1.2* jeví velmi užitečná, je poměrně složitá a problematická na implementaci. Je to vlastnost, kdy topologie WSN je stavěna v reálném čase a navíc musí být pravidelně aktualizována. Díky nejistému dynamickému prostředí, energii a omezení šířky pásma totiž senzory můžou selhávat, nebo se opět objevovat.

Znalost topologie je pro funkčnost sítě a samotných senzorů velmi důležitá. Každý uzel musí znát svou identitu a identitu jeho sousedních uzlů pro vzájemnou spolupráci. Důležité je říci, že globální znalost topologie sítě obecně není nutná. Kromě takové znalosti topologie, je dobré, když každý uzel zná alespoň svou přibližnou lokaci. Opět je důležité říci, že absolutní lokace například pomocí GPS není nutná a navíc by byla nemožná nebo příliš drahá. V klasických bezdrátových sítích je topologie známa předem, proto se tato problematika vyskytla až s ad hoc sítěmi senzorovými [2].

17 Predictive Maintenance (PdM).

S topologií sítě silně souvisí směrovací algoritmy. Directed Diffusion je příklad směrovacího algoritmu, který je velmi efektivní v úspoře energie díky výběru empiricky dobré cesty. Nesměruje tedy pakety přímo na cílovou adresu uzlu, ale pakety jsou pouze předávány do sousedních uzlů. Směr tohoto předávání je založen na geografických informacích, síle signálu či množství komunikace. Například pokud uzel L potřebuje informace, které se vyskytují v místě okolí M, pak by uzly, které jsou v okolí M měly předávat tyto informace do sousedních uzlů, které jsou ve směru L. To samé by měly dělat uzly mezi okolím M a okolím L. Tyto mezilehlé uzly navíc mohou zpřesňovat informace, které posílají díky tomu, že agregují informace z více uzlů. Je vidět, že tento způsob směrování je velmi odlišný od klasického IP[18] stylu komunikace používaného v klasických sítích. Celkem se algoritmus skládá z několika prvků[19]: zájmu (interest), datových zpráv, přechodů (gradients) a výběru nejsilnější cesty (reinforced path). Nejprve je potřeba vyslovit zájem o nějaké informace. Toto je něco jako dotaz na informace, který putuje celou sítí ke zdrojovému uzlu a obsahuje i popis snímacího úkolu. Toto šíření celou sítí nastavuje přechody v rámci celé sítě, a tak tyto přechody jsou vytvořeny ve všech uzlech, které obdržely zájem. Směr přechodu je nastaven vždy směrem k sousednímu uzlu, ze kterého obdržel zájem. Síť poté vybere nejsilnější cestu případně více cest pro doručení daných informací. Vizualizaci tohoto velmi zjednodušeného postupu lze vidět na *obrázku 3.* [9].




(a) Interest propagation

(b) Initial gradients set up

(c) Data delivery along reinforced path

Obrázek 3.: Zjednodušené schéma Directed Diffusion [9].

Ve skutečnosti je popis tohoto směrovacího algoritmu samozřejmě složitější. Navíc se musí řešit to, pokud informace chce více uzlů, případně existuje-li více zdrojů, ze kterých informace chceme získat, nebo dojde k výpadku některého přechodu.

18 Internet Protocol (IP)

19 Anglické názvy v závorkách odpovídají tomu, jak byly zvoleny v článku [9].

### 1.3.2 Spotřeba energie

Problematika, která částečně souvisí s dynamikou topologie, je spotřeba energie jednotlivých uzlů. Je potřeba rozlišovat dynamickou spotřebu energie (spojenou se snímáním, zpracováváním či s vysílacím výkonem pro přenos paketů) od statické spotřeby energie v pohotovostním režimu. Ačkoli je obecně v technologiích snižována především dynamická spotřeba energie, tak v WSN je velmi důležitá také statická spotřeba vzhledem k tomu, že se přenášejí malé objemy dat v dlouhých časových intervalech. Například i návrh samotného směrování v topologii je velmi důležitý na spotřebu energie. V WSN je prakticky nepoužitelné směrování podle klasického protokolu IP. Když pomineme celkově jiný princip klasických sítí, tak by se musely v jednotlivých uzlech udržovat směrovací tabulky pro globální topologii a požadovaná aktualizace v dynamickém prostředí WSN by znamenala vysoké náklady na výpočetní čas, paměť a tedy energii [2].

Možnosti napájení jednotlivých uzlů jsou v zásadě dvě. Buď se použije akumulovaná energie, nebo se tato energie sbírá z okolního světa. U akumulované energie se nemusí jednat o klasické použití baterií. Existují i alternativní řešení, jako jsou palivové články nebo miniaturní tepelné motory. Z okolního světa se může energie sbírat pomocí solární energie, vibrací, akustického zvuku nebo piezoelektrických efektů. Jelikož se téměř všechny komerční a výzkumné platformy spoléhají na akumulovanou energii a klasické použití baterií, tak tyto baterie určují velikost jednotlivých uzlů. Alkalické baterie sice nabízejí vysokou hustotu energie a jsou levné, ale mají velkou fyzickou velikost s ohledem na senzorový uzel. Jejich výhodou je snadná výměna, ale jejich nevýhodou je neefektivnost a velký klidový odběr proudu. Lépe jsou na tom v těchto parametrech lithiové články, které jsou navíc kompaktnější. Naprosto nevhodné jsou dobíjecí baterie. Mají nízkou energetickou hustotu a jejich vlastnost dobíjení není pro WSN praktická. Potenciál mají palivové články, kde se v energii mění chemická reakce. Avšak i přes jejich velký potenciál uchování energie se nepoužívají díky faktu, že pro jejich provoz se vyžaduje vodík [3].

### 1.3.3 Distribuované zpracování

Distribuované zpracování je na rozdíl od topologie sítě problematika, která přímo souvisí se spotřebou energie WSN. Je to dáno tím, že jednotlivé uzly spolupracují na sběru a vytváření užitečných informací. Složité je především určit stupeň sdílení informací mezi uzly a postup sloučení informací z uzlů. Je jasné, že zpracování dat z více snímačů nám dá větší vypovídací hodnotu stejně jako snímání dat s větší frekvencí, ale tato hodnota je vykoupena vysokou spotřebou energie. Musí se tedy vytvořit určitý kompromis mezi vypovídací hodnotou a stupněm využití zdrojů (množstvím komunikace a spotřebou energie). Samotná kombinace informace příchozí s informací místního uzlu je závislá také na samotné aplikaci. Můžou to být kombinace od jednoduchých pravidel vybírání nejlepšího výsledku až po sofistikovaná pravidla [2].

Senzorové pole se dá také chápat jako databáze s jistými unikátními vlastnostmi. Data nejsou skladována na jednom či více centrálních místech, ale jsou rozdistribuována po jednotlivých uzlech. V závislosti na jistém příkazu jsou tak data získávána přímo z okolního prostředí. Vzhledem k tomu, že jsou tyto data distribuována přes více uzlů rozptýlených geograficky, tak je jasné, že v tomto případě jsou latence a spolehlivost na jiné úrovni než v klasických databázích. Pro maximalizaci přínosu z dané WSN by mělo existovat jednoduché rozhraní pro zadávání interaktivních dotazů a získávání odpovědí ve formě informací. Nutností je vývoj jazyku pro zadávání dotazů či úkolování databáze, která může být takto snadno dotazována nebo zaúkolována na měření určitých veličin [2].

### 1.3.4 Bezpečnost

Jelikož mohou WSN působit v nepřátelském prostředí nebo pracovat s citlivými údaji, tak by měly být schopny chránit sebe a svá data před vnějšími útoky. Bezpečnost těchto sítí by měla být vkládána již do návrhu a ne jako dodatečný nápad. Problém pro zajištění této bezpečnosti tkví v tom, že WSN mají omezení ve srovnání s klasickými počítačovými sítěmi, a také že jednotlivé uzly mají jednoduchý a levný hardware. Klasické šifrovací algoritmy vyžadují na poměry senzorů velké množství paměti a výpočetního výkonu, které senzorům nejsou dostupné. Přitom požadavky na bezpečnost WSN jsou s klasickými počítačovými sítěmi téměř shodné. Nejdůležitějším požadavkem je jistě důvěryhodnost pro uchování tajných dat. Taktéž je vyžadována integrita jako obrana proti neoprávněné manipulaci s daty v rámci paketu a důležitá je také autentizace. Např. při broadcastu musí být umožněno ověření jeho odesilatele. Každý přijímač tedy musí vědět, že daná data pochází od správného zdroje. Přitom všechny tyto požadavky nesmí ohrozit dostupnost celé sítě [10].

Velmi nebezpečným útokem na WSN může být i útok na dostupnost dat DoS[20]. Zde u WSN tento útok může být proveden např. tím, že se nepřetržitými požadavky na úkoly vyčerpají baterie uzlu. Nebezpečnost tohoto útoku spočívá v tom, pokud je útočníkem vybraná síť použita pro nějakou citlivou či kritickou aplikaci. Například při útoku na WSN monitorující vznik požáru v budově by mohl tento útok obyvatelům způsobit až smrt. Proto se věnuje mnoho času pro identifikování všech možných způsobů útoků DoS a plánování strategií proti těmto útokům. Za další ohrožení bezpečnosti může být považována také analýza toku dat k základnové stanici, replikace uzlu nebo fyzické útoky [10].

## 1.4 Hardwarová platforma pro bezdrátové senzorové sítě

Nejznámější hardwarové implementace bezdrátových senzorových uzlů[21] Berkeley stylu jsou dostupné od společnosti Crossbow Technology, Inc. Tato společnost byla založena v roce 1995 a sídlí v Kalifornii. Je to čistě soukromá společnost s investicemi od společností Cisco, Intel a Paladin Capital Group.

Obrázek 4.: IRIS mote pouze s radio-platformou [11].

20 Denial of Service (DoS).

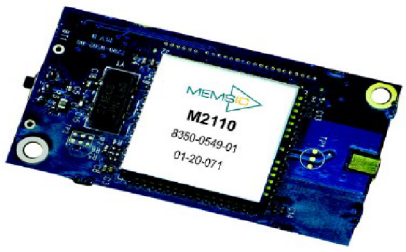
21 Tato jednotlivá zařízení se nazývají mote.

Společnost Crossbow v roce 2007 vydala novou produktovou řadu IRIS. Mote řady IRIS se vyznačují širším rádiovým rozsahem kompatibilním se standardem 2,4 GHz IEEE 802.15.4[22], velmi nízkou spotřebou energie v režimu spánku a optimalizací pro bohatší aplikace. Mají totiž dvojnásobnou paměť RAM oproti předchozím MICA produktovým řadám. Jsou napájeny pomocí dvou AA baterií a mají napětí 2,7 V - 3 V. Obsahují také uživatelské rozhraní tvořené třemi LED diodami.

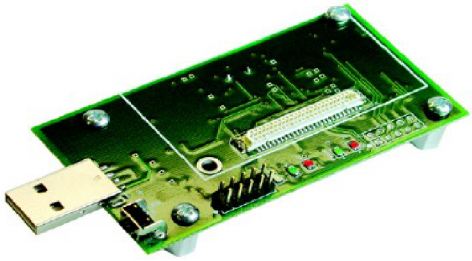
Radio-platforma IRIS mote XM2110 obsahuje:

- nízkoenergetický mikrokontrolér Atmel ATMega1281, který pracuje na frekvenci 8 MHz a zvládá 8 MIPS[23],
- radio transceiver RF230 Atmel, který má přenosovou rychlost 250 kbps,
- sériovou flash paměť velikosti 512 kB,
- flash paměť pro program velikosti 128 kB,
- RAM paměť velikosti 8 kB,
- konfigurační EEPROM paměť velikosti 4 kB,
- 10 bitový ADC převodník,
- rozšiřující 51 pinový slot s napájením na připojení senzorové desky.

Tato radio-platforma má průměrnou spotřebu 8 mA v aktivním režimu a 8 µA v režimu spánku bez započtení radio transceiveru. Samotný radio transceiver má spotřebu od 10 mA do 17 mA v závislosti na vysílacím výkonu. Dosah má kolem 300 m ve venkovních prostorách a téměř 50 m uvnitř budov.




Obrázek 5.: Radio-platforma XM2100 [11].



Obrázek 6.: Základnová deska MIB520 [11].

Senzorová deska typu MTS400 obsahuje dvouosý akcelerometr a dále umí měřit:

- teplotu (-10 C° až +60 C°),
- barometrický tlak (300 až 1100 mbar),
- okolní osvětlení se spektrální citlivostí v rozmezí 400 až 1000 nm,
- vlhkost až do velikosti 60% RH nekondenzující.

Na nahrávání programů do mote nebo na komunikaci senzorové sítě s PC pomocí USB slouží základnová deska MIB520. Programy jsou vytvářeny s využitím operačního systému TinyOS, o kterém bude řeč v *kapitole 2.2*.

---

22 Je to standard specifikující fyzickou a MAC vrstvu pro low-rate wireless personal area networks, který je základem pro technologii ZigBee.

23 Milion instrukcí za sekundu (MIPS).

# 2 Implementační nástroje

Kapitola se zabývá implementačními nástroji, které jsou používány při vývoji aplikací pro WSN. Kapitola obsahuje popis programovacího jazyka nesC, operačního systému TinyOS a agentního jazyka ALLL.

## 2.1 Jazyk nesC

Jazyk nesC je komponentně založený programovací jazyk, který se používá pro vytváření aplikací běžících v operačním systému TinyOS. Jazyk nesC je rozšířením velmi známého programovacího jazyka C. Jazyk C sice poskytuje low-level funkce potřebné pro přístup k hardwaru, ale má nevýhodu při strukturování aplikace. Proto jsou v jazyce nesC navíc komponenty, které výslednou aplikaci tvoří vzájemným spojením. Jazyk se tedy snaží zjednodušovat vývoj aplikací, snižovat velikost kódu a eliminovat mnoho zdrojů možných chyb. Aplikace napsány v nesC jsou hluboce spojeny s hardware mote, jelikož na daném mote běží vždy pouze jedna aplikace. Další důležitá vlastnost aplikací jazyka nesC je ta, že všechny zdroje aplikace jsou alokovány staticky. Jazyk nesC je tedy pouze statickým jazykem [12].

### 2.1.1 Komponenty

Jazyk nesC se vyznačuje separací konstrukce komponent od složení konstruovaných komponent. Dané komponenty tak můžou být jednoduše opakovaně použity. Celkem obsahuje dva druhy komponent [12].

Prvním druhem komponent jsou moduly (klíčové slovo `module`), které definují rozhraní a poskytují aplikační kód. V tomto aplikačním kódu, psaném v klasickém jazyce C, se vyskytuje volání příkazů a implementace reakcí na události u využívaných (`uses`) rozhraní, případně se tam vyskytuje signalizace událostí a implementace příkazů u poskytovaných (`provides`) rozhraní. Více o rozhraní bude psáno v *kapitole 2.1.2* [12].

```
module SurgeM {
  provides interface StdControl;
  uses interface ADC;
  uses interface Timer;
  uses interface Send;
}
implementation {
  uint16_t sensorReading;
  command result_t StdControl.init() {
    return call Timer.start(TIMER_REPEAT, 1000);
  }
  event result_t Timer.fired() {
    call ADC.getData();
    return SUCCESS;
  }
  event result_t ADC.dataReady(uint16_t data) {
    sensorReading = data;
    ... send message with data in it ...
    return SUCCESS;
  }
  ...
}
```

Příklad 1.: Signatura komponenty modul [13].

*Příklad 1.* ukazuje signaturu komponenty modul. V tomto příkladu lze vidět výskyt volání příkazu pomocí prefixu `call`, implementace reakcí na události pomocí prefixu `event` a implementace příkazů pomocí prefixu `command`. Je zde také díky proměnné `sensorReading` vidět, že moduly můžou mít svůj vlastní stav.

Druhým druhem komponent jsou konfigurace (`configuration`). Tyto konfigurace slouží pro již zmiňované propojení komponent prvního druhu dohromady. Pomocí konfigurace také dochází k propojení komponent aplikace se systémovými komponentami TinyOS. Konfigurace prakticky dělají to, že spojují poskytovaná rozhraní s používanými. Každá aplikace v nesC má tuto konfiguraci minimálně jednu a je v ní tedy spojení všech komponent, které potřebuje. *Příklad 2.* ukazuje signaturu komponenty konfigurace. Na tomto příkladě lze vidět, že se nejedná o top-level konfiguraci celé aplikace, ale o konfiguraci dílčí. Je to poznat díky tomu, že také poskytuje rozhraní. Konfigurace totiž můžou být na různé úrovni hierarchie a vypadat tak jako moduly, které poskytují či používají rozhraní. Rozhraní, které směřuje ven z konfigurace či z venku dovnitř, je přímo napojeno na rozhraní jedné z vnitřních komponent konfigurace (modulu) pomocí znaku `=`. Znak `->` pak udává klasické propojení rozhraní dvou modulů uvnitř konfigurace, kdy poskytovatel rozhraní se nachází na pravé straně. Existuje i opačné propojení s opačnou šipkou. Toto propojení se však používá zřídka [12].

```
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer;
  }
}
implementation {
  components TimerM, HWClock;
  StdControl = TimerM.StdControl;
  Timer = TimerM.Timer;
  TimerM.Clk -> HWClock.Clock;
}
```

Příklad 2.: Signatura komponenty konfigurace [13].

Jak moduly tak konfigurace se vyskytují v samostatných zdrojových souborech. Existuje tedy mapování 1:1 mezi komponentami a jejich zdrojovými soubory. Např. zdrojový soubor `LedsC.nc` obsahuje nesC kód komponenty konfigurace `LedsC`. Jelikož se komponenty nacházejí v globálním jmenném prostoru, tak může být pouze jedna komponenta s názvem `LedsC` [14].

### 2.1.2 Rozhraní komponent

Nyní budou podrobněji popsána již zmiňovaná rozhraní (`interface`). Tato rozhraní jsou jediným přístupovým bodem k součástem komponent a obecně modelují nějakou službu (např. zasílání zpráv). Rozhraní jsou obousměrná, jelikož např. komponenta A pomocí používaného (`uses`) rozhraní volá (`call`) příkaz (`command`), který je implementován v komponentě B poskytující toto rozhraní (`provides`), a naopak komponenta B může pomocí tohoto rozhraní signalizovat (`signal`) událost (`event`), která je implementována v komponentě A. Příkazy a události jsou v podstatě funkce. Příkazy se někdy nazývají také downcalls (vstupní body začínající operaci) a události se někdy nazývají také upcalls (výstupní body dokončující operaci). Obousměrná rozhraní, seskupující příkazy a události, tak modelují tzv. split-phase operace.

Split-phase operace umožňují to, aby se mohly softwarové komponenty chovat podobně jako hardwarové. Aplikace vyšle operační požadavek (pomocí příkazu) ke komponentě. Návratový příkaz (`return`) daného požadavku se vrátí ihned, aby program mohl pokračovat. To znamená, že zatímco je zařízení obsazeno, tak procesor může nadále vydávat instrukce a dělat jinou práci. Neblokuje se tak

zásobník kvůli čekání na dokončení I/O operace. Komponenta poté za nějaký čas signalizuje dokončení operace, typicky pomocí přerušení (vyvoláním události s již potřebnými výsledky). Technika split-phase operace umožňuje mít pouze jeden zásobník a vyhnout se vláknům. Typickým příkladem split-phase operace je odeslání paketu, kdy komponenty používají příkaz `send` k zahájení přenosu, ale informaci o dokončení tohoto přenosu se dozví přes událost `sendDone`. Pokud by komponenta implementující příkaz `send` nemohla obsluhovat více souběžných operací, tak při tomto pokusu by jednoduše pomocí události signalizovala neúspěšnost. Je vidět, že tato souběžnost vykazuje velmi nízkou režii na rozdíl od vláken, které jsou běžné v klasických OS. *Příklad 3.* ukazuje dva příklady rozhraní definující split-phase operaci [12].

```
interface Send {
  command result_t send(TOS_Msg *msg, uint16_t length);
  event result_t sendDone(TOS_Msg *msg, result_t success);
}
interface ADC {
  command result_t getData();
  event result_t dataReady(uint16_t data);
}
```

Příklad 3.: Definice rozhraní Send a ADC [13].

Rozhraní se stejně jako komponenty vyskytují v samostatných souborech a v globálním jmenném prostoru. Např. soubor `Leds.nc` obsahuje pouze rozhraní `Leds`.

### 2.1.3 Generické komponenty

Většina komponent reprezentuje služby (např. časovač) nebo části hardwaru (např. LED diody). Ve výchozím nastavení tedy existují jedenkrát (jsou tzv. singleton). Například pokud dvě komponenty jsou spojeny s komponentou stejného názvu, tak tyto dvě komponenty jsou spojeny se stejným kódem a přistupují ke stejným proměnným. Existuje ale i způsob jak vytvořit více instancí komponent. Komponenta, která chce mít více instancí, musí být deklarována jako generický prvek pomocí prefixu `generic`. Generické můžou být jak moduly, tak konfigurace jak ukazuje *příklad 4.* [14].

```
generic module SineSensorC() {
  provides interface Init;
  provides interface Read<uint16_t>;
}
generic configuration TimerMilliC() {
  provides interface Timer<TMilli>;
}
```

Příklad 4.: Generické komponenty [14].

*Příklad 5.* ukazuje použití generické konfigurace a vytváření instance pomocí klíčového slova `new`. Jedná se o top-level konfiguraci aplikace `Blink`, která ukazuje 3 bitový čítač pomocí 3 ledek a 3 časovačů.

```
configuration BlinkAppC {}
implementation {
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;
  /* Wirings below */
}
```

Příklad 5.: Použití generické komponenty [14].

## 2.2 Systém TinyOS

Systém TinyOS je událostmi řízené operační prostředí navržené s cílem běhu ve vestavěných zařízeních, které tvoří distribuovanou WSN. Důvod vlastnosti, že systém TinyOS je událostmi řízený, spočívá v tom, že mote využívající se pro sběr dat musí spíše reagovat na změny, než být řízen dávkovým zpracováním. Systém TinyOS byl napsán pomocí programovacího jazyka nesC a byl přijat velkým počtem výzkumných skupin, které se WSN zabývají. Důvodem vzniku tohoto systému je fakt, že tradiční známé operační systémy a programové modely se pro WSN absolutně nehodí. Tento systém je vyvíjen[24] společností TinyOS Alliance, ve které se nachází společnosti Intel Research, Crossbow Technology a University of California, Berkeley [14].

Skutečnost, že se systém nachází ve vestavěných zařízeních, způsobuje to, že jádro systému vyžaduje pouze 400 B svého kódu včetně paměti pro data. Díky tomu, že systém byl napsán pomocí jazyka nesC, tak se systém vyznačuje stejnými vlastnostmi. Je komponentně založený, obsahuje split-phase operace a pouze statickou alokaci paměti [14].

### 2.2.1 Architektura hardwarové abstrakce

Vývoj aplikací v systému TinyOS je platformě nezávislý. Aby tento vývoj mohl být platformě nezávislý, tak musí existovat sada hardwarově specifických knihoven, které je možno pro vývoj aplikací používat. Díky těmto knihovnám lze stejnou aplikaci použít i pro jinou platformu pouhým překladem [14].

Hardwarová abstrakce má v TinyOS zkratku HAA[25] a je tvořena pomocí třech vrstev. Úplně nejníže je vrstva HPL[26], která je silně závislá na hardwaru. K hardwaru vrstva přistupuje přes paměť nebo porty mapované do adresového prostoru. Vrstva také zpracovává přerušení. Abstrakci a jádro architektury tvoří prostřední vrstva se zkratkou HAL[27], která využívá rozhraní vrstvy HPL. Nejvyšší vrstva HIL[28] je již hardwarově nezávislá. Využívá rozhraní z předchozí vrstvy HAL a sama vytváří rozhraní, které není hardwarově závislé. Pouze u nejvyšší vrstvy se u názvů komponent nevyskytují prefixy specifikující danou platformu [14].

### 2.2.2 Překlad aplikace

Překlad aplikace je prováděn za pomoci standardního programu `make`, kterému se předá na vstup název platformy, pro kterou se daná aplikace překládá. Tento program najde v lokálním adresáři aplikace soubor `Makefile`, který lze vidět v *příkladu 6.*, a díky proměnné `MAKERULES` spojí daný lokální soubor `Makefile` se souborem `Makefile` specifickým pro zadanou platformu.

```
COMPONENT=TopLevelKonfiguraceAplikaceC
include $(MAKERULES)
```

Příklad 6.: Základní Makefile aplikace.

24 Systém je vyvíjen pod BSD licencí.

25 Hardware Abstraction Architecture (HAA).

26 Hardware Presentation Layer (HPL).

27 Hardware Abstraction Layer (HAL).

28 Hardware Independent Layer (HIL).

Po překladu je třeba ještě nahrát přeloženou aplikaci do mote. Toto se provede opět pomocí příkazu `make`, který nyní nebude obsahovat pouze název platformy, ale navíc i příkaz k instalaci, ID uzlu, název základnové desky a port, kde je základnová deska umístěna. Při tomto nahrávání/instalaci se identifikuje daný uzel podle zadaného ID uzlu[29]. Toto ID uzlu později slouží jako jeho jednoznačný identifikátor `TOS_NODE_ID`. Jelikož tzv. OTA[30] existuje pouze v operačním systému Windows, tak se musí v operačním systému Linux radio-platforma daného mote, na který chceme program nahrát, navíc vyjmout a připojit na základnovou desku. První příkaz *příkladu 7.* názorně ukazuje překlad aplikace popisovaný v předchozím odstavci. Druhý příkaz ukazuje nahrání aplikace přes základnovou stanici typu `mib520` do radio-platformy, která bude nadále umístěna na základnové desce. Tato základnová deska je připojena přes USB port. Při připojení základnové desky v linuxové distribuci Kubuntu vzniknou v adresáři `/dev` dva porty. První port `ttyUSB0` pro nahrávání programů do uzlů a druhý port `ttyUSB1` pro komunikaci základnové stanice s PC. Při nahrávání je vidět svítící červená dioda na základnové desce.

```
make iris
make iris install.0 mib520,/dev/ttyUSB0
```

Příklad 7.: Překlad a nahrání aplikace do mote.

### 2.2.3 TOSSIM

TOSSIM je diskrétním událostním simulačním nástrojem pro WSN, přesněji pro aplikace běžící v operačním prostředí TinyOS. Místo sestavení aplikace pro mote se aplikace sestaví pro framework simulátoru, který běží na PC. TOSSIM se používá pro ladění, testování a analýzu algoritmů v prostředí, které je nastavovatelné a opakovatelné. Funguje tak, že dává události do událostní fronty, jež je časově řazená, a spouští je. Prakticky se jedná o simulační knihovnu, která se konfiguruje programem. TOSSIM umožňuje konfiguraci pomocí jazyků C++ a Python. Výhodou jazyka Python je ta, že dovoluje dynamickou interakci s běžící simulací. Jazyk Python je tedy nad jazykem C++, protože používáním objektů jazyka Python se volají objekty C++ [15].

Pro ladění se používá systém `DBG`. Obsahuje kanály, přes které se posílají informace. Tyto informace jsou poté vypisovány např. na `stdout` jako text. Nejprve se však dané kanály musí nastavit, aby simulátor věděl o jejich existenci. Při tomto nastavení prakticky dojde k propojení použitých volání/maker systému `DBG` v aplikaci se souborovým deskriptorem pomocí jednoznačného identifikátoru kanálu. Prostředí WSN se nastavuje pomocí radioobjektu, kterým se nastavuje síťová komunikace mezi uzly. Nastavuje se vždy zisk[31] spojení od odesilatele k příjemci, a tak je toto spojení nutno nastavit oběma směry. Tuto topologii lze také pro jednoduchost načíst ze souboru. Simulátor používá i CPM[32] algoritmus pro simulaci radiofrekvenčního šumu od ostatních uzlů nebo okolního prostředí. Spuštění simulace se provede nastavením počtu kroků, např. pomocí cyklu `for`, který obsahuje příkaz pro spuštění dalšího kroku. Toto lze celé jednoduše umístit do jednoduchého testovacího skriptu Python či programu jazyka C++ a naráz opakovaně spouštět [15].

---

29 Programy se do radio-platformy, která zůstává na základnové desce, nahrávají většinou s ID `0`.

30 Over-the-air-programming (OTA).

31 Gain (v jednotkách dBm).

32 Closest Pattern Matching (CPM).

## 2.3 Agentní jazyk ALLL

Jak již bylo částečně zmíněno v úvodu, tak situováním agentů do systému WSN dokážeme činnost této sítě monitorovat, spravovat a řídit. Agenti nejsou pouze autonomními inteligentními jednotkami, ale vyznačují se i svou mobilitou (mohou se přesunovat přes síť). Díky tomuto požadavku, složitému přenosu nativních aplikací na zařízení mote a statické alokaci paměti v jazyce nesC nelze agenty implementovat přímo v jazyce nesC. Takový kód by totiž nemohl být jednoduše přenesen na jiný uzel a agent by už vůbec o tomto přenosu nemohl rozhodnout. Na FIT tak byl vyvinut jazyk ALLL, kterým je možné popsat chování agenta a toto chování poté interpretovat. Kvůli hardwarovým omezením WSN se však tento jazyk vyznačuje nízkou úrovní abstrakce.

Prostor, který umožňuje ukládat a interpretovat agenty napsané v agentním jazyce ALLL se nazývá agentní platforma. Je implementována na konkrétním bezdrátovém uzlu, na který se agent dokáže přesunout a jehož funkce poté dokáže využívat. Tato platforma je tedy nehybná a již může být implementována v jazyce nesC. Kód tohoto celého systému však musí být relativně malý, protože se musí vejít do paměti mikrokontroléru. Existuje 128 kB Flash programovatelné paměti, kde tato implementace platformy může být uložena. Program agenta je poté uložen v paměti RAM, která je velká pouze 4 kB (platforma MICAz) či 8 kB (platforma IRIS).

Diplomová práce [16] blíže popisuje platformu pro mobilní agenty v WSN. Tato platforma řídí činnost interpretu, který jazyk ALLL interpretuje pomocí stavového automatu. Při přechodech ze stavu do stavu se volá vždy jen jedna akce. Akce, o které se stará sám interpret, slouží především pro úpravu hodnot, jež byly získány pomocí služeb samotné platformy. Jedna z akcí interpretu totiž identifikuje to, že se jedná o službu a tyto služby jsou poté rozlišeny pomocí parametrů. Dá se tedy také říct, že i když platforma interpret řídí, tak mu poskytuje své služby. Aby mohla platforma poskytovat své služby interpretu, tak spolupracuje se systémovými knihovnami TinyOS. O aktuálních akcích a službách platformy blíže pojednává *kapitola 3.2*.

Bakalářská práce [17] se zabývá jazykem ALLL z pohledu jeho interpretace. Zjednodušeně řečeno se agent sestrojený pomocí tohoto jazyka řídí plánem s určitým cílem a je ovlivňován pouze vnějším okolím. Celkem je agent rozdělen do 7 částí. Obsahuje:

- bázi znalostí (`BeliefBase`),
- přijaté zprávy a naměřené hodnoty (`InputBase`),
- záměr (`Plan`),
- bázi plánů (`PlanBase`),
- tři univerzální pomocné registry (Register `1`, `2` a `3`).

Podstatnou vlastností agenta v jazyce ALLL je to, že své informace má rozděleny v prvních dvou zmíněných částech. Báze znalostí může být agentu naplněna na počátku, ale častěji si ji plní sám znalostmi o svém okolí pomocí služeb platformy v podobě libovolně velkých n-tic s příslušnými údaji. Nad bází znalostí poté může pracovat pomocí unifikace[33], což je prakticky test na přítomnost dané n-tice. Do druhé části `InputBase` se přijaté zprávy a naměřené hodnoty ukládají automaticky v podobě n-tic se dvěma položkami. První položka udává, zda se jedná o příjem zprávy nebo přijatá data ze senzorů, a druhá položka tato data obsahuje. Při výběru se poté zadává pouze první položka z n-tice a do aktivního registru jsou vsunuta pouze přijatá/naměřená data. Důvod oddělení těchto

33 Zde se jedná o jistou podobnost se systémem LINDA.

dvou částí spočívá v tom, aby záleželo pouze na agentovi, zda si naměřené či přijaté zprávy uloží do své báze znalostí.

Aplikace, kterou má agent provádět, je uložena v části záměr. Tento záměr může být naplněn odkazy na plány uložené v bázi plánů, kde není nutné, aby byly nějak seřazeny. Tyto uložené plány se vyvolávají pomocí tzv. nepřímého spuštění, kdy se v bázi plánů hledá plán s jistým jménem. Důležité je to, že i v těchto plánech se toto nepřímé spuštění může vyskytovat. Takto mohou být plány hierarchicky zanořovány. Pokud poté selže nějaká akce, tak selže plán na této úrovni a agentní kontrola je předána vyšší úrovni plánu.

Poslední z částí jsou univerzální pomocné registry, které slouží pro různé výpočty. Například pokud se provede nějaká akce s výsledkem, tak se v aktivním registru tato hodnota objeví.

*Tabulka 1.*, *tabulka 2.* a *příklad 8.* ukazuje jednoduché agenty pro představu, jak může takový jazyk ALLL vypadat. Konkrétní popis funkcí interpretu a samotné platformy je uveden až v *kapitole 3.2*. *Tabulka 1.* ukazuje agenta, který měří teplotu v okolí. Jelikož agent nevyužívá cyklus, tak jej lze zapsat pouze do části se záměrem (`Plan`). Agent nejprve rozsvítí červenou LED diodu pomocí `$(1,(r,1))`, což představuje akci pro volání služeb platformy s kódem operace a parametry služby. Zde se jedná o kód operace `1` a parametry služby jsou `r` (červená dioda) a `1` (rozsvícení). Agent po rozsvícení červené diody provede zjištění aktuální teploty pomocí `$(d)` a nastavení aktivity prvního registru pomocí `&(1)` na uložení hodnoty ze senzoru, kterou uloží pomocí `?(s)` až přijde. Nakonec je tato hodnota z registru poslána na základnovou stanici pomocí `!(1,&1)` a červená dioda je zhasnuta `$(1,(r,0))`. Zde má základnová stanice identifikátor uzlu číslo 1.

| **část:** | **obsah:** |
|---|---|
| `Plan` | `$(1,(r,1))$(d)&(1)?(s)!(1,&1)$(1,(r,0))` |

Tabulka 1.: Agent měřící teplotu okolí [16].

Pokud bude potřeba použít při popisu chování agenta cyklus, tak se již bez báze plánů nejde obejít. *Tabulka 2.* obsahuje agenta, který střídavě rozsvěcuje prostřední diodu (plán `stred`) a dvě diody krajní (plán `okraj`). Každá změna vydrží 500 milisekund, což je definováno službou platformy pozastavení provádění kódu na zadaný čas pomocí `$(w,(500))`. Záměrem nyní bude počáteční spuštění agenta jedním z plánů.

| **část:** | **obsah:** |
|---|---|
| `PlanBase` | `(stred,($(1,(r,0))$(1,(g,1))$(1,(g,0))$(w,(500))^(okraj)))`<br>`(okraj,($(1,(r,1))$(1,(g,0))$(1,(g,1))$(w,(500))^(stred)))` |
| `Plan` | `^(stred)` |

Tabulka 2.: Ukázka cyklu [16].

Posledním příkladem bude zápis důležité vlastnosti agentů - agentní mobility. Služba agentní mobility má označení pomocí písmene `m`. Obsahuje povinný parametr, kam se má agent přesunout. Nepovinný parametr udává to, zda se má agent po odeslání ukončit, nebo dále pokračovat v činnosti. *Příklad 8.* obsahuje dva agentní kódy. První se po odeslání kódu na uzel číslo 2 neukončí, zatímco druhý ano.

```
$(m,(2))$(1,(r,1))

$(m,(2,s))$(1,(r,1))
```

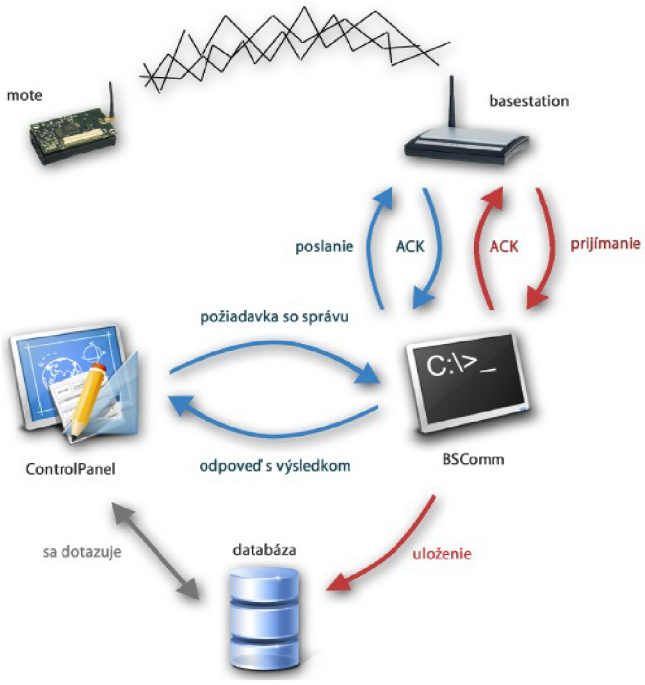
Příklad 8.: Agentní mobilita [16].

# 3 Systém WSageNt

WSageNt je agentní systém, jehož cílem je monitorování a spravování uzlů bezdrátové senzorové sítě. Je vyvíjen na FIT pod vedením Ing. Františka Zbořila Ph.D. v rámci bakalářských a diplomových prací.

## 3.1 Základní popis systému

Základem je síť bezdrátových senzorových uzlů IRIS nebo MICAz a jedna základnová stanice připojená k PC. Na senzorových uzlech je nainstalována platforma pro mobilní agenty zmíněná v *kapitole 2.3* a na základnové stanici program `Basestation`, který pouze přeposílá zprávy mezi sériovou a rádiovou linkou. Program `Basestation` pochází z balíku TinyOS, který navíc poskytuje Java rozhraní ke komunikaci s daným mote v základnové stanici. Tohoto rozhraní využívá konzolová aplikace `BSComm`. Ta tedy běží v PC a komunikuje s programem `Basestation` přes sériový port[34]. Aplikace je vyvinutá na platformě `Java SE` a základní třídou přímo komunikující s mote v základnové stanici je třída `Comm`, která je odvozena od třídy `MessageListener` původem ze zmíněného balíku TinyOS. Nad touto konzolovou aplikací běží webové rozhraní `ControlPanel` komunikující s aplikací `BSComm` přes socket[35]. Toto rozhraní bylo vyvinuto v diplomové práci [18] na platformě `Java EE` za pomoci open-source frameworku `Struts 2`. Obrázek 7. názorně ukazuje propojení zmíněných prvků a v *příloze 1.* je uveden podrobný postup zprovoznění systému včetně nástrojů potřebných k jeho provozu.




Obrázek 7.: Průběh komunikace mezi jednotlivými prvky systému [18].

34 Předpokládá se port `serial@/dev/ttyUSB1:iris` (se zkratkou `iris` pro automatické doplnění přenosové rychlosti).

35 Předpokládá se port k socketu číslo `7777`.

Systém WSageNt je stále ve vývoji. Jeho praktické použití je zatím omezené, avšak díky jeho dobrému návrhu je jeho potenciál velký. Systém je prakticky navržen tak, aby webové rozhraní `ControlPanel` posílalo jednoduché zprávy nebo požadavky na informace do konzolové aplikace `BSComm`. Tyto požadavky jsou vyjádřeny pomocí agentních zpráv, které jsou definovány pomocí jazyka ALLL. Konzolová aplikace `BSComm` se poté pokusí zprávu odeslat přes program `Basestation` na cílový uzel po paketech s potvrzováním výsledku. Daná zpráva tedy přijde na cílový uzel, kde se nachází platforma pro mobilní agenty. Pokud se jedná o jednoduchou zprávu, tak se uloží do `InputBase` agenta. Pokud se jedná o zprávu agentní, tak se na dané platformě pozastaví provádění aktuálního agentního kódu (když je nějaký přítomen) a začne se interpretovat příchozí agentní kód. Po čas interpretace daného agenta, může platforma podle typu aplikace odesílat informace jednoduchými zprávami zpět na základnovou stanici. Je možné přesunout i celého agenta na základnovou stanici, avšak tento kód se již interpretovat nebude. Pouze se celý bude brát jako příchozí jednoduchá zpráva. Program `Basestation` v základnové stanici tedy poté přepošle dané pakety do aplikace `BSComm`. `BSComm` dané pakety přijme a sestaví výslednou zprávu, kterou uloží do databáze. Webové rozhraní `ControlPanel` si pak může dané zprávy z databáze vyzvednout a podle jejich dat se překreslit, případně se jinak zachovat, aby uživatel byl o daných výsledcích seznámen.

## 3.2 Funkce agentní platformy

Je tedy vidět, že hlavní funkce a případné jejich rozšiřování spočívá především v agentní platformě a webovém rozhraní, které zjednodušeně řečeno prezentuje různé získané informace pomocí funkcí této platformy. V *kapitole 2.3* bylo naznačeno, že se agentní platforma skládá ze dvou částí. *Tabulka 3.* zobrazuje seznam typů akcí interpretu a *tabulka 4.* zobrazuje seznam služeb poskytovaných platformou pod kódem akce `$`. Pokud totiž interpret narazí na tento kód akce, tak za další závorkou hledá parametr této akce, který poslouží pro výběr toho, jaká služba platformy se zavolá.

*Tabulka 3.* a *tabulka 4.* obsahuje informace získané z diplomové práce [16], z bakalářské práce [17] a ze zdrojových souborů poskytnuté aktuální verze platformy pro mobilní agenty.

| **kód akce:** | **popis:** | **příklady:** |
|---|---|---|
| **+** | Vkládání n-tice do `BeliefBase` agenta. Při vkládání se kontroluje duplicita n-tic pomocí unifikace. Plán při duplicitě n-tice neselže. Za kódem akce může být buď celý registr, nebo se případně může objevit v libovolném prvku n-tice. | `+(NB,34,45)`<br>`+&1`<br>`+(34,&2)` |
| **-** | Odebrání n-tice z `BeliefBase` agenta pomocí unifikace. Při nenalezení dané položky plán neselže. Při zadávání může být místo libovolného prvku n-tice tzv. anonymní proměnná `_`. | `-(NB,_,_)`<br>`-(_)`<br>`-(23)` |
| ***** | Test `BeliefBase` na zadanou n-tici. N-tice může být zadaná i pomocí registru nebo anonymní proměnné. V případě anonymní proměnné je časté, že se nalezne více n-tic. Ty jsou vloženy do n-tice jedné a ta do aktivního registru. Aktivita registru musí být před tímto testem nastavena, jinak akce končí selháním plánu. Selháním plánu akce končí, i pokud není nalezena žádná n-tice. | `*(_)`<br>`*(vis,&1)`<br>`*(123,32)`<br>`*(NB,_,_)` |
| **?** | Test `InputBase` na zprávu od mote se zadanou adresou, či od senzoru se zadaným znakem senzoru. Akce má tedy pouze jeden parametr a výsledek se ukládá opět do aktivního registru, který musí být nastaven. Výsledkem se rozumí druhý prvek z nalezené dvouprvkové n-tice, která je po akci smazána. | `?&1`<br>`?(1)`<br>`?(s)`<br>`?(_)` |

| | | |
|---|---|---|
| **@** | Přímé spuštění je akce, která vše uvnitř závorek vloží na zásobník se zarážkou, ze kterého interpret bere kód. | `@(+(abc)! (2,(456)))` |
| **^** | Nepřímé spuštění provede to, že hledá plán v části `PlanBase` s daným jménem, které je uvedeno jako parametr akce. Pokud je plán nalezen, tak se vloží na zásobník se zarážkou. Pokud plán není nalezen, tak akce končí selháním aktuálního plánu. | `^(plan) ^&1` |
| **&** | Změna aktivního registru. Akce má pouze jeden parametr v závorkách, a to číslo registru, na které chceme měnit (od 1 do 3). | `&(1)` |
| **!** | Tato akce je výjimka. Jedná se o službu platformy pro odeslání klasické zprávy, která skončí v `InputBase` cílového agenta. Adresa mote/platformy je zadaná prvním parametrem a daná zpráva druhým parametrem. Oba parametry mohou být zadány pomocí registru. | `!(1,(abc)) !(&1,&2)` |
| **$** | Kód akce pro obecné volání služeb platformy, kde první parametr vždy slouží jako kód služby. Pokud volání obsahuje více parametrů (minimálně však 1), tak ostatní parametry patří volané službě. | `$(l,(r,1)) $(n)` |
| **#** | Symbol pro již zmiňovanou zarážku za plánem. Tato akce nemá sémantický význam. | |

Tabulka 3.: Seznam a popis typů akcí interpretu.

| **kód služby:** | **popis:** | **příklady:** |
|---|---|---|
| **a** | Služba pro aktivaci sledování příchozích zpráv při běžícím interpretu. | `$(a)` |
| **s** | Asynchronní služba pro uspání interpretu (provádění kódu), dokud nepřijde zpráva z rádia. Pokud byla zpráva přijata před spuštěním této služby, tak je tato služba ignorována. | `$(s)` |
| **w** | Pomocí této asynchronní služby je možné pozastavit interpret na zadaný čas v milisekundách, jež je zadán pomocí druhého parametru. Daný parametr může obsahovat i identifikátor registru. | `$(w,(100)) $(w,&3)` |
| **k** | Služba pro okamžité a definitivní ukončení činnosti interpretu, a tedy provádění kódu agenta. | `$(k)` |
| **l** | Služba pro ovládání LED na mote. Parametr služby obsahuje n-tici, která obsahuje identifikátor ledky (červená `r`, zelená `g`, žlutá `y`) a/nebo hodnotu `1` pro rozsvícení či hodnotu `0` pro zhasnutí. Pokud obsahuje pouze daný identifikátor, tak dojde ke změně dané ledky. | `$(l,(r,1)) $(l,(g)) $(l,(y,0))` |
| **d** | Pokud spustíme tuto službu bez parametrů, tak se bude jednat o asynchronní službu pro zisk aktuální teploty ze senzoru, která skončí na začátku nového cyklu interpretu v části `InputBase` ve tvaru dvouprvkové n-tice `(s, <hodnota>)`. Služba dále může mít parametr ve formě dvouprvkové n-tice. Pak služba bude pracovat s historií dat a tyto data číst z flash paměti, do které je budou v budoucnu zapisovat služby pro intervalové měření. První prvek n-tice identifikuje typ operace nad historií dat (minimum `m`, maximum `M`, průměrná hodnota `a`) a druhý prvek n-tice označuje počet posledních naměřených hodnot, ze kterých se daná operace má provádět. Pokud není dostatek hodnot pro provedení služby, tak akce skončí selháním aktuálního plánu. | `$(d)` `$(d,(m,10))` |
| **f** | Služba pro vložení prvního prvku ze seznamu do aktivního registru (nutno předem nastavit, jinak selže aktuální plán). Jedná se obdobu operátoru `cad` z jazyka LISP. Tudíž výsledkem může být buď jednoprvková n-tice nebo první ze seznamů, pokud je prvků v první n-tici více. Obsahuje parametr, který udává buď danou n-tici, nebo registr, ze kterého se má počáteční seznam brát. | `$(f,&2)` |

| | | |
|---|---|---|
| **r** | Služba pro vložení zbytku seznamu bez prvního prvku do aktivního registru. Jedná se o obdobu operátoru `car` z jazyka LISP. Parametrem je opět registr, který udává počáteční seznam. Pokud počáteční seznam obsahuje pouze jeden prvek, tak se do aktivního registru vloží prázdná n-tice. | `$(r,&2)` |
| **m** | Služba platformy pro agentní mobilitu. Jako parametr musí obsahovat buď jednoprvkovou, nebo dvouprvkovou n-tici. Pokud je pouze jednoprvková, tak obsahuje pouze jeden povinný parametr, a to adresu mote/platformy, kam se má kód agenta přesunout. Po přesunutí se pak daný kód provádí na obou platformách. Když je dvouprvková, tak obsahuje navíc nepovinný parametr `s`, který udává to, že se má ukončit provádění odesílaného kódu na zdrojové platformě po jeho úspěšném odeslání. | `$(m,(1))`<br>`$(m,(1,s))` |
| **n** | Asynchronní služba platformy pro zisk síly signálu ke svým dostupným okolním sousedům. Po spuštění této služby se hodnoty uloží ve tvaru `(NB,<adresa souseda>,<síla signálu>)` do části `BeliefBase`. Služba nemá žádný parametr. | `$(n)` |
| **o** | Tato služba je výjimka. Jedná se totiž o akci interpretu sloužící pro výpočty nad zadanými hodnotami v registrech. Za kódem této akce je jako parametr kód typu výpočtu. Výsledky těchto výpočtů se nacházejí v aktivním registru, který musí být před zahájením nastaven. Jednotlivé výpočty jsou blíže popsány v *tabulce 5.* Tato akce má vždy včetně kódu akce a zmíněného kódu typu výpočtu 7 parametrů. Dalších 5 je rozděleno na tři části (dva vstupní operandy, dvě vstupní proměnné a jedna výstupní proměnná). I když některá z částí není použita, tak musí obsahovat alespoň prázdné závorky. | |

Tabulka 4.: Seznam a popis typů služeb platformy.

Informace v *tabulce 5.* byly získány ze zdrojových souborů poskytnuté aktuální verze platformy pro mobilní agenty.

| **kód výpočtu:** | **typ operace:** | **znázornění výsledku pomocí jazyka C:** | **příklad:** |
|---|---|---|---|
| **min** | unární | `-op1` | `$(o,min,(3),(),(),(),())` |
| **not** | unární | `op1 == 0 ? 1 : 0` | `$(o,not,&1,(),(),(),())` |
| **mul** | binární | `op1 * op2` | `$(o,mul,(3),(5),(),(),())` |
| **div** | binární | `op1 / op2` | `$(o,div,(3),(3),(),(),())` |
| **mod** | binární | `op1 % op2` | `$(o,mod,(3),(2),(),(),())` |
| **add** | binární | `op1 + op2` | `$(o,add,&1,(15),(),(),())` |
| **sub** | binární | `op1 - op2` | `$(o,sub,(15),(5),(),(),())` |
| **les** | binární | `op1 < op2 ? 1 : 0` | `$(o,les,(3),(6),(),(),())` |
| **mor** | binární | `op1 > op2 ? 1 : 0` | `$(o,mor,(3),(6),(),(),())` |
| **leq** | binární | `op1 <= op2 ? 1 : 0` | `$(o,leq,(3),(6),(),(),())` |
| **meq** | binární | `op1 >= op2 ? 1 : 0` | `$(o,meq,(3),(6),(),(),())` |
| **equ** | binární | `op1 == op2 ? 1 : 0` | `$(o,equ,(3),(3),(),(),())` |
| **neq** | binární | `op1 != op2 ? 1 : 0` | `$(o,neq,(3),(6),(),(),())` |
| **and** | binární | `op1 && op2 > 0 ? 1 : 0` | `$(o,and,(1),(0),(),(),())` |
| **orr** | binární | `op1 \|\| op2 > 0 ? 1 : 0` | `$(o,orr,(0),(1),(),(),())` |
| **cpy** | unární | `op1` | `$(o,cpy,(3),(),(),(),())` |

Tabulka 5.: Seznam a popis typů výpočtů nad hodnotami v registrech.

Z *tabulky 3.* a *tabulky 4.* je zřejmé, že obecně operace (akce či služby) mohou být synchronní nebo asynchronní. Při synchronní operaci (v tabulkách není označeno) není třeba při její implementaci provádět explicitní synchronizaci. Naopak při asynchronní operaci je vyžadováno dodatečné řízení. Asynchronní operace jsou většinou služby platformy. Je u nich ihned předáváno řízení zpět interpretu a pokračuje se, jakoby se jednalo o úspěšně dokončenou operaci. Interpret tedy odebere operaci ze zásobníku a informuje platformu o ukončení kroku. Dále následuje podmínka, zda má interpret pokračovat dále v kódu. U synchronní operace by se dál pokračovalo, ale u asynchronní operace se čeká na její dokončení [16].

## 3.3 Funkce webového rozhraní

Jak již bylo zmíněno v *kapitole 3.1*, tak webové rozhraní `ControlPanel` je postaveno na technologii `Java EE`. Bližší informace o návrhu a implementaci jeho architektury je možno najít v diplomové práci [18]. Nyní z pohledu seznámení se se systémem WSageNt pro účely rozšíření jeho funkcí jsou v této kapitole popsány pouze jeho aktuální funkce. V této kapitole je abstrahováno propojení webového rozhraní s jednotlivými mote přes konzolovou aplikaci `BSComm` a program `Basestation` v základnové stanici.

Jelikož webové rozhraní zobrazuje výsledky monitorování a je nástrojem pro ovládání aktivních mote, tak poskytuje přehled těchto aktivní mote[36] připojených do sítě. Toto je v prezentační vrstvě architektury[37] vyřešeno pomocí tzv. aktivního panelu, který navíc zobrazuje i indikátor nově přijatých zpráv. Aktivní panel je vložen do všech JSP stránek, které představují jednotlivé pohledy GUI, a tvoří jej GUI komponenta, která asynchronně mění svůj obsah bez vyžádání uživatele. Toto řešení umožňuje technologie AJAX[38], kterou pro `Struts 2` využívá framework `dojo`. Webové rozhraní dané aktivní mote kontroluje pomocí uživatelem naplněného seznamu, ve kterém je nutné, aby se nacházely minimálně adresy[39] možných aktivních mote. Rozhraní poté posílá prázdné jednoduché zprávy podle tohoto seznamu a čeká, zda na ně dojde potvrzení. Rozhraní dále umožňuje posílat jednoduché nebo agentní zprávy na konkrétní adresu mote. Dané agentní zprávy (agenty) lze i ukládat do databáze a při odeslání je z dané databáze vybírat. Přijímané jednoduché i agentní zprávy od aktivních mote a mezi aktivními mote poté zobrazuje v přehledném seznamu.

Poslední funkcí je tzv. monitor topologie. Tento monitor vyšle agentní zprávu, ve které je pouze v části `Plan` obsah `$(n)$(m,(1))`, na všechny adresy mote podle výše zmíněného uživatelem vyplněného seznamu. Je tedy třeba zdůraznit, že uzly v síti musí mít přímé spojení se základnovou stanicí, což není ve WSN moc časté. Každý aktivní mote pak odešle zpět naměřené síly signálu ke svým sousedům. Naměřené hodnoty ve tvaru `(NB,<adresa mote>,<síla signálu>)` se od jednotlivých mote objeví v seznamu přijatých zpráv a také aktivní panel zobrazí nově přijaté zprávy. Monitor topologie poté pomocí analytické geometrie[40] danou topologii aktivních mote vykreslí. Obraz topologie je vykreslen do podoby `jpg` obrázku pomocí třídy `java.awt.Graphics2D`.

---

36 Pouze ty mote, na kterých je nainstalována platforma pro mobilní agenty.

37 Návrhového vzoru MVC pro webové technologie.

38 Asynchronous JavaScript and XML (asynchronní komunikace se serverem).

39 Pokud není adresa mote `TOS_AM_ADDRESS` explicitně změněna, tak je shodná s identifikátorem mote `TOS_NODE_ID`, který je definován ve fázi instalace (popisuje *kapitola 2.2.2*).

40 Silnou roli hraje experimentální určení vzdáleností uzlů od sebe na základě síly signálu.

# 4 Návrh agentních prvků

Jedním z cílů diplomové práce je návrh agentních prvků, které umožní sledovat stav celé sítě, nebo její části, případně upozorňovat klientskou aplikaci na výskyt definovaných událostí. Tato diplomová práce se zabývá sledováním výskytu agentů v systému WSageNt. Jelikož se však agenti díky agentní mobilitě mohou v síti buď přesouvat z uzlu na uzel nebo sami sebe klonovat za účelem vykonávání nějaké činnosti, tak se následně tyto výskyty budou převádět na cesty jednotlivých agentů. Je totiž dobré umět dané přesuny monitorovat a mít možnost tak například zjistit, na kterém uzlu daný agent skončil. Kapitola nejprve představuje základní popis návrhu a poté jeho jednotlivé části popisuje podrobněji.

## 4.1 Základní popis

Sledování výskytu agentů nemůže být prováděno pouze pomocí samotných agentů. Pokud bychom pouze sledovali to, které uzly agent navštívil, tak dostaneme informace o pohybu jen tohoto agenta. Jelikož potřebujeme zjistit všechny výskyty všech agentů od počátku startu systému, tak je jasné, že toto zaznamenávání adres uzlů agentem nebude stačit.

Pro dostatečné odlišení jednotlivých agentů v systému WSageNt, bude potřeba přidat k nynějšímu obecnému identifikátoru id (`ID`) agenta další identifikátor třídy (`CLASS`) agenta. Toto je praktické i z důvodu mít možnost v budoucnosti seskupit podobné agenty pod jednou třídou.

Díky tomu, že budeme zaznamenávat samotné agenty, tak je jasné, že registrace výskytu identifikátorů agentů nemůže být v samotných agentech[41]. Tyto záznamy musí ukládat samotná platforma pro mobilní agenty. Ta si při příchodu agenta musí uložit jeden záznam o jeho výskytu. Včetně identifikátorů agenta by platforma měla ukládat i odkud agent přišel a čítač jeho přesunů, aby dané záznamy mohly být využity v systému WSageNt i jako tzv. pachové stopy. Vzhledem k potencionálnímu množství těchto záznamů nevyhovuje jejich umístění v paměti RAM. Ta je silně omezená[42] a již téměř zaplněna samotným agentem. Naštěstí ještě existuje 512 kB velká externí flash paměť, kam je možné tyto záznamy ukládat, avšak je řádově pomalejší než paměť RAM. Abychom mohli výskyty agentů analyzovat a převést na cesty jednotlivých agentů, tak bude potřeba vytvořit novou službu platformy, která se bude starat o vybírání záznamů z této externí flash paměti. Bude tedy sestaven agent, co projde celou síť a tyto záznamy po výběru postupně odešle na základnovou stanici. Agent bude využívat zmíněných pachových stop při jeho průchodu. Zjistí si své okolní sousedy a na základě dalších funkcí služby pachových stop bude zjišťovat, zda určitý uzel již prošel a odeslal z něj potřebné záznamy, nebo neprošel.

Samotná analýza výskytů poté bude probíhat až ve webovém rozhraní, kam dané záznamy od všech uzlů přijdou. Uživatel rozhraní daného agenta, který se o zisk záznamů postará, vyšle pomocí jednoduchého požadavku. Poté bude uživatel upozorněn, že všechny záznamy již byly posbírány, a bude mít možnost zobrazit tabulku s přehledným soupisem všech zjištěných agentů s jejich výskyty na určitých adresách uzlů. V tomto soupisu budou uvedeny i časy posledního odeslání zjištěných agentů z rozhraní, které budou zjišťovány z databáze. Z tohoto soupisu si poté uživatel bude moci

41 Například v části `BeliefBase` nebo `InputBase`.

42 4 kB MICAz platforma nebo 8 kB Iris platforma.

vybrat vykreslení přesunů konkrétního agenta. Po výběru se jednak vypíší samotné přesuny textově a následně pod nimi budou tyto přesuny přehledně vykresleny do podoby topologie sítě.

## 4.2 Externí flash paměť

Externí flash paměť bude v agentní platformě používána pro ukládání záznamů o přijatých agentech. Tato flash paměť umožňuje trvalé uložení dat, i když je napájení odpojeno, nebo je daný mote přeprogramován. Tato kapitola popisuje možnosti přístupu k ní a způsob čtení a zápisu dat. Informace zde v kapitole vychází ze zdrojů [19] a [20].

### 4.2.1 Možnosti přístupu

Existují tři možnosti přístupu k této externí flash paměti. TinyOS poskytuje komponenty, které se vždy starají o jednotlivý přístup tím, že implementují rozhraní abstrahující služby těchto komponent.

První přístup se nazývá Config a název jeho komponenty je `ConfigStorageC`. Je určený pro malé objekty v řádu několika stovek bajtů. Tímto přístupem se tedy ukládají většinou konfigurační údaje, například identita mote, rádiová frekvence, vzorkovací frekvence atd. Tento přístup umožňuje náhodné čtení a zápis.

Druhý přístup se nazývá Logging, název jeho komponenty je `LogStorageC` a může být lineární (kdy je ukládání zastaveno, až je jednotka plná) nebo kruhový (kdy jsou po zaplnění přepisována stará data novými). Tento přístup je vhodný pro protokolování událostí a výsledků. Data totiž můžeme pouze připisovat na explicitně nastavenou pozici nebo konec a poté číst jako proud dat od začátku zápisu nebo od explicitně nastavené pozice. Problémem tohoto přístupu je právě toto neefektivní náhodné čtení a zápis.

Poslední třetí přístup se nazývá Block a název jeho komponenty je `BlockStorageC`. Jelikož tento přístup umožňuje ukládání téměř libovolně velkých objektů a zcela náhodné čtení a zápis, tak je vhodný pro ukládání záznamů o výskytu agentů. Sice nepodporuje implicitní zápis v kruhu, ale tato vlastnost lze jednoduše naprogramovat explicitně až v aplikaci. Naše záznamy se tak budou do paměti zapisovat v kruhu, aby nemohlo nikdy dojít k nemožnosti zápisu záznamu při příchodu agenta. Toto bude znamenat pouze postupné odstraňování starých záznamů z paměti. Důležitá je také vlastnost náhodného čtení a zápisu, protože tuto paměť bude používat také pomocná služba pro průchod agenta sítí (na základě pachových stop). Příklad vytvoření takové jednotky a naznačení způsobu čtení a zápisu dat je vztaženo pouze k tomuto poslednímu přístupu.

TinyOS rozděluje flash čip do jednoho či více objemů, jejíž velikost je pevně určena v době kompilace. Tuto specifikaci udává soubor XML, který musí být umístěn v adresáři aplikace. Jeho název se musí skládat z čipu, který vybraná platforma[43] používá. Obecně tedy má název `volumes-CHIPNAME.xml` a *příklad 9.* ukazuje, jak takový soubor může vypadat. Po vytvoření takového XML souboru je již potřeba pomocí konfigurace propojit vytvořenou komponentu `BlockStorageC` s naší aplikací, kde se definuje jedno z jmen úseků vytvořených v tomto XML souboru. S danou komponentou bude tedy propojena pouze jedna část. Komponenta dále vyžaduje hlavičkový soubor `StorageVolumes.h`, jak lze vidět na částečně upraveném *příkladě 10.*, takové propojující konfigurace.

43 Platforma MICAz i IRIS používá čip `at45db`. Soubor XML tedy bude mít název `volumes-at45db.xml`.

```
<volume_table>
  <volume name="CONFIGLOG" size="65536"/>
  <volume name="PACKETLOG" size="65536"/>
  <volume name="SENSORLOG" size="131072"/>
  <volume name="CAMERALOG" size="524288"/>
</volume_table>
```

Příklad 9.: Konfigurační soubor `volumes-CHIPNAME.xml` [19].

```
#include "StorageVolumes.h"

configuration RandRWAppC { }
implementation {
  components RandRWC, new BlockStorageC(VOLUME_CONFIGLOG), MainC;

  MainC.Boot <- RandRWC;
  RandRWC.BlockRead -> BlockStorageC.BlockRead;
  RandRWC.BlockWrite -> BlockStorageC.BlockWrite;
}
```

Příklad 10.: Propojující konfigurace [19].

### 4.2.2 Čtení a zápis dat

Pokud již máme vytvořenou a propojenou komponentu jednoho z přístupů, tak stačí do naší aplikace přiřadit její rozhraní jako používající[44] rozhraní. Komponenta `BlockStorageC` poskytuje jedno rozhraní pro čtení `BlockRead` a jedno rozhraní pro zápis `BlockWrite`.

Důležité na čtení a zápisu je to, že obě probíhají pomocí split-phase operace a musí existovat čtecí a zápisový buffer. Pro zápis se tedy použije příkaz `write` z rozhraní `BlockWrite`. Příkaz bude mít jako parametr adresu, na kterou se má zapsat, buffer s daty a délku zapisovaných dat. I když se příkaz provede ihned, tak ještě není jasné, že jsou daná data zapsána. Toto je signalizováno pomocí události `writeDone`, která navíc obsahuje výsledek, s jakým byla daná data zapsána. Obdobným způsobem pak probíhá i čtení dat pomocí příkazu `read` a události `readDone` z rozhraní `BlockRead`.

## 4.3 Průchod agenta sítí

Jak již bylo zmíněno v *kapitole 4.1 Základní popis*, tak pro možnost vykonání sběru záznamů ze všech uzlů je důležité mít k dispozici agenta, který bude schopen projít celou sítí. Nyní nám tedy nejde ani tak o samotné vykonání sběru, ale o obecně průchod, který může být použit i pro jiné aplikace. Samotný průchod by měl být nezávislý na znalosti adres jednotlivých uzlů. Agent by měl být pouze vyslán na jeden z uzlů a dále se již sám o průchod postarat.

### 4.3.1 Pomocné služby

Pro samotný průchod agenta sítí jsou klíčové dvě služby. Za první se jedná o službu pro zisk síly signálu ke svým sousedům, která byla představena v *kapitole 3.2*. Tato služba nám dokáže naplnit `BeliefBase` agenta n-ticemi `(NB, <adresa mote>, <síla signálu>)`, kde nás budou zajímat pouze údaje o adresách okolních sousedů.

---

44 Viz *kapitola 2.1.2 Rozhraní komponent*.

Druhou potřebnou službou je služba pracující s pachovými stopami agenta, jejíž implementace bude součástí diplomové práce kolegy Bc. Petra Židka [21]. Funkce dané služby byly představeny v případové studii [22], ze které vycházejí informace o této službě. Jedná se o službu platformy pod kódem akce `$`. Kódem služby je poté písmeno `t` a její funkce silně závisí na jejich argumentech. Funkce se dají v zásadě rozdělit do třech typů.

Prvním typem funkce je tzv. dotazování[45]. Funkce slouží pro otestování toho, zda agent již navštívil konkrétní uzel. Jako argument přijímá dvouprvkovou n-tici. První prvek musí být `q` a druhý prvek udává adresu uzlu, která má být dotazována. Funkce tedy iniciuje uzel v síti, který prohledá své záznamy o příchozích agentech a odešle nám výsledek. Neprohledávají se tedy záznamy na současném uzlu, kde se agent nachází. Příkladem zápisu v jazyce ALLL je `$(t,(q,<adresa mote>))`. Funkce je vzhledem k průchodu sítí vhodná pro otestování všech okolních uzlů, aby agent nešel na uzel, který již navštívil.

Druhým typem funkce je zisk identifikace aktuálního uzlu[46]. Tato funkce pod argumentem ve tvaru jednoprvkové n-tice `(i)` pouze naplní aktivní registr agenta adresou současného uzlu, kde se agent nachází. Příklad zápisu v jazyce ALLL vypadá `$(t,(i))`. Tato funkce se hodí pro vytváření si své identifikace navštívených adres uzlů, které jsou ukládány ve formě n-tic do části `BeliefBase` agenta. Její pomocí se dá nahradit funkce dotazování, především pro testovací účely.

Třetím a posledním typem funkce je tzv. navracení[47]. Tato funkce se spouští s argumentem ve tvaru dvouprvkové n-tice, kdy první prvek musí být `b` a druhý prvek může být `l` nebo `f`. Pod prvkem `l` funkce naplní aktivní registr agenta adresou uzlu odkud tento agent naposledy přišel. Příkladem zápisu v jazyce ALLL je `$(t,(b,l))`. Pod druhým z možných prvků `f` funkce najde v záznamech první výskyt aktuálního agenta a naplní jeho aktivní registr adresou uzlu, odkud při tomto prvním výskytu přišel. Příkladem zápisu v jazyce ALLL je `$(t,(b,f))`. Tento třetí typ funkce se hodí především pro navrácení se v postupné cestě o jednu úroveň zpět po tom, co funkce dotazování zjistí, že v aktuální úrovni již všechny uzly byly navštíveny. Pro toto je vhodná především druhá varianta, avšak jeden ze dvou identifikátorů agenta (`CLASS` nebo `ID`) musí být při každém průchodu jiný.

### 4.3.2 Sestavení agenta

V této kapitole již bude popsán konkrétní agent schopný průchodu sítí. *Tabulka 6.* ukazuje tohoto sestaveného agenta v jazyce ALLL a *příklad 13.* v *příloze 2.* naznačuje jeho princip. Jeho návrh byl uzpůsoben pro pozdější použití libovolnou aplikací. Je také vidět, že jeho kód je složen ze samostatných částí/pod-plánů, které jsou samostatně otestovatelné, a to hlavně kvůli novým pomocným službám. Jeho testování bylo prováděno především v simulačním nástroji TOSSIM a poté na reálných mote pomocí jejich výstupu na uživatelské rozhraní (3 led diody). Pod-plány mají krátké názvy z důvodu ušetření co nejvíce prostoru pro aplikaci, která jej bude používat.

Agent tedy nejprve nalezne všechny své sousední uzly, dané záznamy postupně prochází pomocí plánu `cn` a pokaždé získá adresu uzlu (plány `gn` a `gi`). Tuto adresu otestuje pomocí služby dotazování a pokud agent na daném uzlu nebyl, tak se na něj přesune (plán `tm`). Když agent na daném uzlu byl, tak poté celý plán `tm` skončí neúspěchem, a jelikož na vrcholu zásobníku bude opět plán `cn`, tak se vybere další záznam. Agent přistoupí k navracení (plán `bk`) pokud s neúspěchem otestuje všechny své sousedy. V tomto případě plán `gn` není schopen začít zpracovávat žádný

45 V případové studii [22] je pod názvem Querying nodes.

46 V případové studii [22] je pod názvem Node identification.

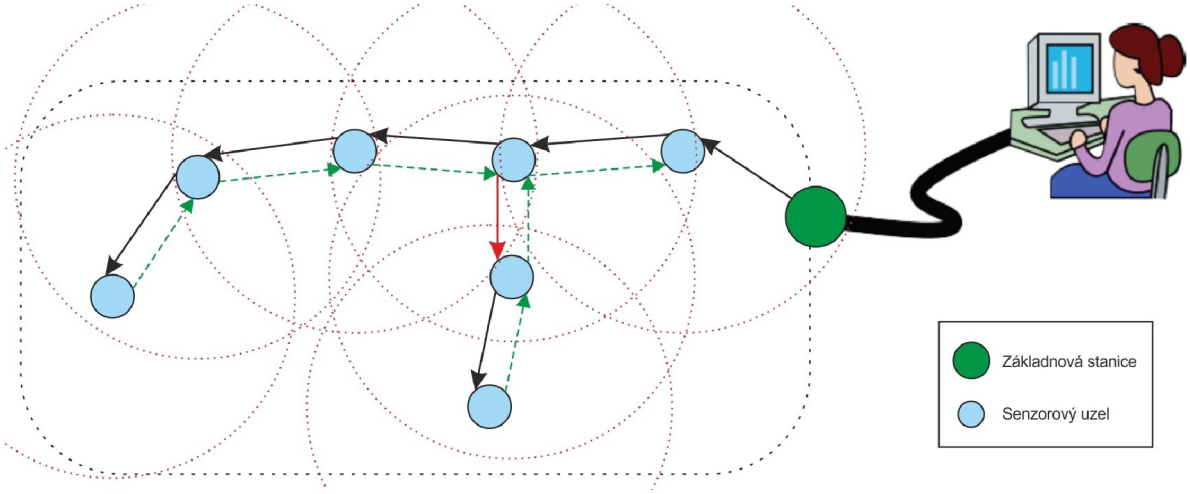
47 V případové studii [22] je pod názvem Backtracking.

záznam. Po navrácení v cestě zpět agent opět začíná od začátku novým zjištěním svých sousedů. Pouze v případě zjištění, že je těsně před základnovou stanicí s adresou 1, tak končí.

| část: | obsah: |
|---|---|
| PlanBase | (nb,($(n)^(cn)))<br>(cn,(^(gn)^(gi)^(tm)#^(cn)))<br>(gn,(+(bk)&(1)*(NB,_,_)&(2)$(f,&1)-&2-(bk)#&(1)*(bk)^(bk)))<br>(gi,(&(1)$(r,&2)&(2)$(f,&1)))<br>(tm,($(t,(q,&2))-(NB,_,_)$(m,(&2,s))^(nb)))<br>(bk,(&(1)$(t,(b,f))+(b,&1)&(2)*(b,(1))$(k)#$(m,(&1,s))^(nb))) |
| Plan | ^(nb) |

Tabulka 6.: Agent procházející sítí v ALLL.

*Obrázek 8.* znázorňuje ukázku průchodu takového agenta sítí. Černé šipky znamenají chod agenta vpřed, zelené šipky navracení a červené ohraničení dosah zjištění svých sousedů. Červená dopředná šipka znázorňuje přechod agenta do části sítě, kam se dostal jen díky jeho navracení.




Obrázek 8.: Ukázka průchodu agenta sítí. Převzato a upraveno z [22].

Agent starající se i o sběr záznamů a jejich odesílání poté bude volat plán `nb`, který se postará o správný přesun v síti. Jediná úprava tohoto kódu agenta bude spočívat v posledním příkazu nepřímého spuštění v plánu `tm`, kdy se nebude spouštět opět plán `nb`, ale základní plán tohoto agenta, jenž bude hierarchicky nad plánem `nb`. Nepřímé spuštění plánu `nb` u plánu `bk` zůstane zachováno, jelikož při přesunu zpět se aplikace, která tohoto agenta pro přesun využívá, nevykonává.

V *tabulce 6.*, konkrétně v pod-plánu `gn`, je vidět i způsob, kterým lze v jazyce ALLL vytvořit programovou konstrukci `if-then-else`. Na začátku pod-plánu se vloží n-tice `(bk)` do `BeliefBase` pomocí kódu `+(bk)`. Pokud podmínka reprezentovaná kódem `*(NB,_,_)` selže, tak dojde ke smazání plánu po zarážku `#` a za zarážkou nic nebrání vykonání nepřímého spuštění `^(bk)`, jelikož se daná n-tice `(bk)` v `BeliefBase` nachází. Pokud však podmínka neselže, tak dojde k odstranění n-tice `(bk)` a toto nepřímé spuštění pod-plánu `bk` se neprovede. Toto velmi pomáhá při tvorbě rozsáhlejších aplikací v jazyce ALLL, jelikož se pak pod-plány dají strukturovat do celků, které mají určitý účel, a tyto celky jednoduše znázorňovat. Pro toto znázorňování je nejvíce vhodný diagram aktivit z jazyka UML[48], s jehož pomocí se dá znázorňovat i vytvoření druhého

48 Unified Modeling Language (UML).

agenta, který provede práci pro agenta prvního (toto bude použito v *kapitole 5.2*). Další výhodou je to, že při přesunu agenta nenarůstá jeho velikost na zásobníku, protože se tímto odstraní části vložených plánů za zarážkou, které by se nikdy neprovedly.

## 4.4 Analýza výsledků

Jak již bylo nastíněno v *kapitole 4.1 Základní popis*, tak analýza všech přišlých záznamů ze sítě bude prováděna až ve webovém rozhraní `ControlPanel`. Zde se jednotlivé záznamy od uzlů objeví i jako příchozí zprávy v základním přehledu zpráv. Jeden záznam by měl obsahovat adresu uzlu odkud pochází, třídu a id agenta, ke kterému patří, odkud daný agent přicházel a čítač těchto přesunů. Tyto záznamy bude potřeba rozřadit podle třídy a id agenta, zjistit pro každého agenta jeho navštívené uzly a vytvořit celkové cesty všech agentů.

Po prezentaci daných výsledků a výběru vykreslení cesty konkrétního agenta uživatelem bude potřeba i zjistit vzájemné rozmístění jednotlivých uzlů v síti, abychom měli jednoznačné souřadnice pro vykreslení těchto uzlů. V této fázi bude využito obdobného principu zjištění síly signálu ke svým sousedům jako v monitoru topologie[49], avšak nyní bude třeba počítat s tím, že všechny uzly na základnovou stanici nemusí vidět.

Pak bude možnost využít již existující[50] třídu `RSSIParser` pro rozparsování zpráv s hodnotami signálů a třídu `TopologyMath`, která pomocí analytické geometrie analyzuje zjištěné naměřené hodnoty signálů a vypočítá jak 2D souřadnice jednotlivých uzlů, tak velikost potřebného 2D obrazu. Nyní již budeme mít všechny potřebné informace k vykreslení získaných výsledků.

---

49 Viz *kapitola 3.3 Funkce webového rozhraní*.

50 Třídy `RSSIParser` a `TopologyMath` jsou součástí implementace popisované v diplomové práci [18].

# 5 Implementace agentních prvků

Tato kapitola se zabývá implementačními detaily konkrétních navržených agentních prvků z *kapitoly 4*. Jelikož implementace samotného ukládání záznamů při příjmu agenta na senzorovém uzlu je součástí diplomové práce kolegy Bc. Petra Židka [21], tak se kapitola zaměřuje až na výběr těchto záznamů z platformy pro mobilní agenty. Poté kapitola seznamuje s prostředkem pro doručení vybraných záznamů a potřebných hodnot signálů ke svým sousedům[51] na základnovou stanici a dalšími nově vytvořenými službami pro účely tohoto doručení. Kapitola se také zabývá výsledným kódem agenta v jazyce ALLL, způsobem analýzy výsledků a vykreslením jednotlivých tras agenta. Na závěr kapitola popisuje implementační problémy, které vyplynuly jak po čas testování potřebných služeb platformy pro agentní aplikaci, tak ze spojení platformy pro mobilní agenty se zbytkem systému.

## 5.1 Výběr záznamů z platformy

Jak již bylo řečeno, tak výběr záznamů z platformy bude prováděn pomocí agenta, který bude procházet sítí. Stejně jako při ukládání záznamů, tak i při jejich výběru je opět problémem malá paměť RAM senzorových uzlů. Malá RAM paměť totiž způsobuje to, že celková velikost agenta byla v souboru `Agent.h` interpretu stanovena na 2081[52] znaků, což samozřejmě nestačí na výběr všech záznamů. I přes momentální skromné vymezení bloku v externí flash paměti (v souboru `volumes-at45db.xml` na 16384 B) může být uložených záznamů až 2048, jelikož se pro jeden záznam ukládají čtyři 16 bitové položky. Tyto záznamy se navíc při výběru musí převést do znakových n-tic, aby se mohly uložit do `BeliefBase` agenta. Díky tomuto problému musí docházet k výběru záznamů z platformy vícekrát po sobě.

Této skutečnosti byla uzpůsobena služba vybírající tyto záznamy. K jejímu spuštění dojde na základě toho, že interpret v modulu `AgentC` vyhodnotí službu platformy s kódem akce `$` a zavolá příkaz `svcCall` používaného rozhraní `PlatformSvcI` modulu `PlatformSvcC`. Zde se na základě kódu služby `g` zavolá funkce `ziskLogu`, která zpracuje další parametry a zavolá příkaz `readAllFootprint` používaného rozhraní `FootprintI` modulu `FootprintM`. Služba má dva parametry, a to index, od kterého se začnou záznamy vybírat, a počet záznamů kolik se má z externí flash paměti vybrat. Jelikož se pracuje s externí flash pamětí, která je výrazně pomalejší než interpretace kódu agenta na platformě, tak daná služba musí být asynchronní. Při spuštění příkazu `readAllFootprint` tedy prvně dochází k pozastavení interpretace a až poté k čtení prvního záznamu z flash paměti na základě split-phase[53] operace. V implementaci události `readDone` se vždy naplní pole prvním záznamem podle parametrů a spustí se čtení dalšího záznamu. Použití daného pole je nutné, jelikož modul `FootprintM` sám nemůže vkládat znaky do `BeliefBase`. Může pouze nastavit, že má přečtena všechna data a zpět spustit provádění interpretace. O přidání záznamů do `BeliefBase` se tak postará až modul `ControlM` (zde se nachází hlavní řídící smyčka interpretu), jelikož přes používané rozhraní `iFootprintI` zavolá příkaz `ableToAddLogsToBB`.

---

51 Pro výpočet topologie viz *kapitola 4.4*.

52 Celková velikost pole, kam se musí vejít `BeliefBase`, `InputBase`, `Plan`, `PlanBase` a obsah třech registrů.

53 O čtení a zápisu dat do externí flash paměti pojednávala *kapitola 4.2.2*.

Do `BeliefBase` agenta se záznamy ukládají ve formě n-tice např. `(LOG,3,2,1,1,0)`, kde `3` udává odkud daný záznam pochází, `2` udává odkud agent při příjmu přišel, `1` udává `ID` agenta, `1` udává `CLASS` agenta a `0` udává čítač přesunů tohoto agenta. Prvek `3` se ve flash paměti nenachází, ale je přidáván až při výběru pomocí systémové proměnné `TOS_NODE_ID`. Velikost pole pro výběr záznamů byla omezena na 128 záznamů, protože velikost n-tice minimální délky je 15 znaků, jak lze spočítat z příkladu takové n-tice. Pokud pak budeme počítat s jistou režií ve formě části `Plan`, případně `PlanBase`, tak by i v tomto ideálním případě více záznamů stejně nebylo možné do agenta uložit. Proto služba při pokusu uložit více jak 128 záznamů končí chybou a dochází ke smazání aktuálního plánu. Služba končí neúspěchem, i když je index, od kterého se mají začít číst záznamy, nastaven větší, než je počet záznamů. Tohoto je právě využito při výběru všech záznamů, jelikož se pak může daná služba umístit do samostatného plánu, který se sám volá v cyklu, a skončí až není co číst. V *tabulce 7.* je uvedena ukázka takového výběru záznamů znázorněných v *tabulce 8.*, jenž se nacházejí na uzlu `3`. *Příklad 14.* v *příloze 2.* poté naznačuje princip tohoto výběru záznamů.

| **část:** | **obsah:** |
|---|---|
| počáteční stav registru 1 | `0` |
| `PlanBase` | `(cg,($(g,(&1,3))^(ig)&(2)$(o,add,&1,(3),(),(),())&(1)$(f,&2)^(cg))` |
| `Plan` | `^(cg)` |
| stav `BeliefBase` po 1. iteraci | `(LOG,3,4,11,2,6)(LOG,3,2,23,3,4)(LOG,3,1,12,2,0)` |
| stav `BeliefBase` po 2. iteraci | `(LOG,3,5,6,8,1)(LOG,3,6,4,1,2)` |

Tabulka 7.: Ukázka výběru záznamů z platformy pro mobilní agenty.

| **pořadí:** | **záznamy:** |
|---|---|
| 0. | agent s `id = 11`, `class = 2`, přišel z uzlu 4 s čítačem 6 |
| 1. | agent s `id = 23`, `class = 3`, přišel z uzlu 2 s čítačem 4 |
| 2. | agent s `id = 12`, `class = 2`, přišel z uzlu 1 s čítačem 0 |
| 3. | agent s `id = 6`, `class = 8`, přišel z uzlu 5 s čítačem 1 |
| 4. | agent s `id = 4`, `class = 1`, přišel z uzlu 6 s čítačem 2 |

Tabulka 8.: Záznamy v paměti na uzlu s adresou 3.

V *tabulce 7.* lze vidět to, že je nutné použít službu pro ALU[54] platformy. Ta v každém cyklu plánu `cg` přičte k indexu tolik, kolik se vybralo záznamů (nyní pro jednoduchost uvažujeme pouze 3 záznamy v každém cyklu). Nutné je však před cyklem `cg` nastavit jeden z registrů na `0` jako počáteční stav indexu. V ukázce cyklus končí ve třetí iteraci a to právě na službě `g`, jelikož došlo k pokusu o výběr záznamů s indexem větším než je počet záznamů. V plánu `cg` je také vidět nepřímé spuštění plánu `ig`, který je určen pro zpracování části nasbíraných záznamů v `BeliefBase`. Plán `ig` v této ukázce není uvažován, avšak v něm vždy dochází ke smazání záznamů po jejich zpracování. Toto je v ukázce vidět na tom, že stav `BeliefBase` po druhé iteraci již záznamy z první iterace neobsahuje. Z tohoto popisu lze poznat to, že ke korektnímu ukončení služby `g` (bez smazání plánu ze zásobníků) dochází, i když je požadováno čtení více záznamů, než je k dispozici. Takto nedochází k nekonzistenci, protože tak čtení skončí vždy až v další iteraci.

54 Arithmetic logic unit (ALU).

## 5.2 Doručení záznamů do základnové stanice

V počátečních fázích implementace byly záznamy po jejich výběru ihned po jednom přímo odesílány na základovou stanici. Toto jistě nebylo zcela ideální, jelikož se tímto ztrácel význam toho, že průchod agenta sítí je na přímém spojení všech uzlů se základnovou stanicí nezávislý. V WSN celkově toto omezení není časté, a tak bylo nutné odstranit tuto podmínku přímého spojení základnové stanice s uzly v síti. Odstranění tohoto omezení si vyžádalo potřebu nových služeb, které můžou být v budoucnu použity i pro jiné typy aplikací. Pomocí těchto nově vytvořených a stávajících služeb jsou prakticky vytvářeni agenti otroci, kteří se o doručení všech záznamů z jednoho uzlu do základnové stanice postarají. Jeden agent otrok tak doručí pouze část nasbíraných záznamů z jednoho uzlu. Počet záznamů v této části odpovídá počtu záznamů, jenž se vybere v jednom cyklu plánu `cg`, který byl zmíněn v *kapitole 5.1*.

Jak již bylo zmíněno dříve, tak identifikace agenta je prováděna na základě jeho identifikátorů `ID` a `CLASS`. Protože v samotném průchodu agenta sítí na základě služby pracující s pachovými stopami je důležité to, aby každý nový agent měl tuto identifikaci unikátní, tak tuto identifikaci musí mít unikátní i daný agent otrok. Agent otrok musí mít jednoznačnou identifikaci z toho důvodu, aby po cestě k základnové stanici nezměnil pachové stopy agenta otrokáře. Pokud by je totiž změnil, tak by průchod, jak byl popsán v *kapitole 4.3*, nefungoval. Nyní je tedy pro vyslání agenta, který provede sběr záznamů, důležité to, aby měl vždy unikátní třídu[55]. Je to z toho důvodu, že agent otrok bude vytvářen vždy s identifikátorem `ID` větším o jedničku než jeho otrokář.

### 5.2.1 Potřebné služby

Tato kapitola popisuje služby, které jsou potřebné pro vytváření agenta otroka, ale i pro jeho průchod směrem k základnové stanici.

Jelikož neexistovala žádná služba pracující s hodnotami `ID` a `CLASS` agenta, tak bylo nutno takovou vytvořit. Její identifikace je pod kódem služby `c`. Modul `PlatformSvcC` na základě tohoto kódu zavolá funkci `changeIdentification`, která ovlivňuje globální proměnné `ag_id` a `ag_class`. Tyto proměnné jsou typu `uint16_t`[56] a jsou definované v hlavičkovém souboru `AMAgent.h` platformy. Služba přijímá za parametr dvouprvkovou n-tici. První prvek z n-tice určuje k jaké změně dojde. Ke změně může dojít dvěma způsoby (`i` – inkrementace, `d` - dekrementace). Druhý prvek z n-tice pak určuje, který z identifikátorů se bude měnit (`i` - id agenta, `c` - třída agenta). Pro účely testování zde byl doplněn další způsob použití `g`, co identifikátory nemění, ale pouze získává jejich hodnotu do aktivního registru, který musí být předem nastaven. *Příklad 11.* obsahuje tři agentní kódy ukazující použití této služby. První kód změní id agenta tím, že inkrementuje jeho hodnotu, druhý kód změní třídu agenta dekrementací jeho hodnoty a třetí poslední kód vloží hodnotu třídy agenta do registru `1`.

```
$(c,(i,i))
$(c,(d,c))
&(1)$(c,(g,c))
```

Příklad 11.: Agentní kódy pro ukázku služby měnící id a třídu agenta.

55 Nebude tedy platit možnost výběru jednoho z identifikátorů, jak bylo psáno v obecném průchodu agenta sítí.

56 Typ proměnné TinyOS pro 16 bitový `unsigned int` (rozsah `0-65535`).

Další službou, která musela být doprogramována byla služba pro zjištění prázdného seznamu. Důvod použití této služby bude vysvětlen až v *kapitole 5.3.1*. K jejímu spuštění dojde na základě kódu služby `e` a o její činnost se stará funkce `listOperation` modulu `PlatformSvcC`. Tato služba přijímá parametr ve formě dvouprvkové n-tice, kde první prvek musí být `e` a druhý prvek musí být registr, který je testován. Služba selže (dojde k smazání plánu ze zásobníku po zarážku), pokud registr obsahuje prázdnou n-tici. Služba může mít také tu funkci, že pouze zkopíruje daný seznam mezi registry. V tomto případě zmíněná dvouprvková n-tice musí obsahovat první prvek `c` a musí být nastaven aktivní registr. *Tabulka 9.* zobrazuje způsob použití této služby. V tabulce lze vidět, že v iteracích je jako první použita služba `$(r,&2)` pro získání zbytku seznamu. Tato služba do aktivního registru `3` vloží prázdný seznam `()`, pokud bude ve vstupním registru `2` jednoprvkový seznam. Právě na tomto seznamu se díky službě `$(e,(e,&3))` ukončí provádění plánu `l`. V registru `2` přitom zůstává prvek, který je sice pořád umístěn v seznamu (k jeho výběru dojde pomocí služby `$(f,&2)`), ale je poslední ze zpracovávaného seznamu. Tímto způsobem může docházet k výběru posledního prvku ze seznamu, který je vytvořen díky akci interpretu unifikace `*(_,v)`. Tabulka také ukazuje, kdy se hodí právě zmiňované kopírovaní mezi registry `$(e,(c,&3))`. Použije se hlavně přitom, když potřebujeme zpracovávat seznamy v cyklu a nevychází nám dané zpracování do čísel registrů.

| **část:** | **obsah:** |
|---|---|
| `BeliefBase` | `((5),v)((4),v)((3),v)((2),v)` |
| `PlanBase` | `(l,(&(3)$(r,&2)$(e,(e,&3))&(2)$(e,(c,&3))^(l)))` |
| `Plan` | `&(2)*(_,v)^(l)&(3)$(f,&2)` |
| stav registru `2` při prvním spuštění plánu `l` | `(((2),v),((3),v),((4),v),((5),v))` |
| konečný stav registru `3` | `((5),v)` |

Tabulka 9.: Výběr posledního prvku pomocí průchodu seznamem.

### 5.2.2 Vytváření agenta otroka

Vytváření obecně agenta jiným agentem, který má plnit odlišnou činnost není v jazyce ALLL jednoduché, jelikož musí být obě činnosti napsány pomocí jednoho kódu. Z důvodu toho, že platforma pro mobilní agenty může obhospodařovat pouze jednoho agenta, tak jediným způsobem jak ze senzorového uzlu vyslat dalšího agenta do sítě je použitím agentní mobility. Zatím byla agentní mobilita používána způsobem, kterým se provádění kódu na původním uzlu po přesunu agenta ukončilo. Existuje ale také způsob, jak se toto provádění kódu neukončí. Jelikož ihned po přesunu agenta bude interpret na obou uzlech zpracovávat stejný kód počínaje další operací, tak z důvodu odlišení činnosti musí následně dojít k odlišení posloupnosti zpracovávání plánů na jednom z uzlů pomocí selhání plánu. Problém částečně způsobuje také fakt, že se pachové stopy agenta ukládají automaticky při příjmu daného agenta. Aby agent otrok měl identifikaci odlišnou ihned při prvním příjmu, tak musí dojít ke změně jeho identifikace ještě před odesláním. Prakticky se tedy mění identifikace otrokáře, jelikož jak již bylo zmíněno platforma může obsahovat pouze jednoho agenta. Agent otrokář si samozřejmě po odeslání agenta otroka změní svou identifikaci zpět, aby mohl dále využívat pachových stop, které má přístupné pod svou identifikací. Potřebujeme také, aby agent otrokář počkal na dokončení práce svého agenta otroka. Mohlo by se totiž stát, že by agent otrokář

mohl být ve své práci rychlejší a mohl se buď sám potkat se svým agentem otrokem na jednom uzlu, nebo by se na jednom uzlu mohli potkat dva agenti otroci.

*Tabulka 10.* obsahuje ukázku vytvoření agenta otroka, jejíž princip naznačuje *příklad 15.* v *příloze 2.* Na ukázce je vidět, že pro odlišení kódu agenta otrokáře a agenta otroka vkládá agent otrokář svou adresu uzlu do `BeliefBase` ve formě n-tice `(<adresa uzlu>, n)`. Adresu aktuálního uzlu získá pomocí služby pachových stop `$(t,(i))`. Poté agent otrokář zjistí odkud na aktuální uzel přišel (služba `$(t,(b,l))`), změní svou identifikaci (služba `$(c,(i,i))`) a použije službu agentní mobility `$(m,(&1))` bez ukončení interpretace. Touto agentní mobilitou agent otrokář vytvoří a vyšle agenta otroka tím směrem, odkud daný agent otrokář přišel. Oba agenti následně opět zjistí adresu uzlu, kde se nacházejí a pomocí unifikace ji otestují s danou n-ticí v `BeliefBase`. Ke smazání plánu po zarážku dojde pouze u agenta otroka, a tak mu nic nebrání v nepřímém spuštění plánu `im`, který momentálně nebudeme uvažovat. U agenta otrokáře ke smazání plánu nedojde, a tak musí odstranit identifikátor[57] `(ch)` z `BeliefBase` a pokračovat v interpretaci plánu `wt`, ve kterém dojde k jeho uspání. K uspání dochází za pomoci služby `$(s)`, co jej později probudí, pokud přijde zpráva z rádia. Tuto zprávu bude posílat agent otrok a bude jí vyjadřovat ukončení své činnosti. Aby nemohl být agent otrokář vzbuzen jinou zprávou[58], tak se po probuzení kontroluje text příchozí zprávy. Když text nesouhlasí, tak je agent otrokář opět uspán. Po jeho úspěšném vzbuzení dochází k změně jeho identifikace zpět (služba `$(c,(d,i))`) a díky odstranění identifikátoru `(ch)` se daná část plánu `ch` za zarážkou zpracovávat nebude. Ke změně identifikace by samozřejmě mohlo dojít ihned po službě agentní mobility. Toto však vadí při spuštění daného kódu v simulačním nástroji TOSSIM, jelikož jsou identifikátory `ID` a `CLASS` definovány jako globální proměnné. Při jejich okamžité změně zpět v agentu otrokáři totiž dochází k jejich okamžité změně i v agentu otroku, jelikož je simulován pouze jeden programový kód pod různými identifikátory uzlů.

| **část:** | **obsah:** |
|---|---|
| `PlanBase` | `(ch,(+(ch)&(1)$(t,(i))+(&1,n)&(1)$(t,(b,l))$(c,(i,i))$(m,(&1))&(1)`<br>`$(t,(i))&(2)*(&1,n)-(&1,n)-(ch)^(wt)$(c,(d,i))#&(2)*(ch)-(ch)^(im)))`<br>`(wt,(+(wp)$(s)&(1)?(_)&(2)*(&1)-(wp)#*(wp)-(wp)^(wt)))` |
| `Plan` | `^(ch)` |

Tabulka 10.: Způsob vytváření agenta otroka.

# 5.3 Výsledná aplikace v ALLL

Tato kapitola popisuje výsledný kód aplikace agenta v jazyce ALLL. Celý kód je umístěn v *tabulce 11.*, jehož princip je naznačen v *příkladě 16. přílohy 2.* Část `BeliefBase` agenta je již na počátku naplněna n-ticí ve tvaru `(mr,(<adresa uzlu>))`. Tato n-tice dává agentu znalost o uzlu, na který je vyslán jako první. Tento uzel musí mít jako jediný přímé spojení se základnovou stanicí a právě z tohoto uzlu budou na základnovou stanici posílány všechny nasbírané informace. U ostatních uzlů se již toto přímé spojení samozřejmě nevyžaduje a agent se sám i postará o jejich zjištění. Výsledná konstrukce agenta se prakticky skládá ze dvou samostatných částí, které jsou patrné v *tabulce 11.* pomocí jejich oddělení mezerou v části `PlanBase`. První část se stará o sběr

57 Způsob vytvoření programové konstrukce `IF-THEN-ELSE` byl popsán v *kapitole 4.3.2.*

58 Například prázdnou zprávou, kterou posílá webové rozhraní jako kontrolu online uzlů. Tato kontrola však byla přepracována do podoby, jak ji popisuje *kapitola 5.5.3 Problémy ve spolehlivosti*.

záznamů na aktuálním uzlu a o jejich doručení na základnovou stanici. Druhá část se poté stará o správný přesun agenta v síti, který byl popsán v *kapitole 4.3.2*. Druhá část byla pouze doplněna o závěrečné odeslání zprávy `endlog` na základnovou stanici, aby webové rozhraní `ControlPanel` vědělo, že již byly posbírány všechny záznamy.

| **část:** | **obsah:** |
|---|---|
| `BeliefBase` | `(mr,(2))` |
| `PlanBase` | `(cl,(&(1)$(t,(i))+(&1,v)&(3)$(e,(z))+(sn)$(n,m)^(cg)^(nb)))`<br>`(cg,($(g,(&3,10))$(l,(g,1))^(ig)$(l,(g,0))&(2)$(o,add,&3,(10),(),(),())&(3)`<br>`$(f,&2)^(cg)))`<br>`(ig,(+(ig)&(1)$(t,(i))&(2)*(mr,&1)^(sg)^(sn)-(sn)-(ig)#&(2)*(ig)-(ig)^(ch)))`<br>`(sg,(&(2)*(LOG,_,_,_,_,_)&(1)$(f,&2)-&1!(1,&1)^(sg)))`<br>`(sn,(&(2)*(sn)&(2)*(M,_,_,_)&(1)$(f,&2)-&1!(1,&1)^(sn)))`<br>`(ch,(+(ch)+(&1,n)&(1)$(t,(b,l))$(c,(i,i))$(m,(&1))&(1)$(t,(i))&(2)*(&1,n)`<br>`-(&1,n)-(ch)^(wt)$(c,(d,i))-(LOG,_,_,_,_,_)-(M,_,_,_)#&(2)*(ch)-(ch)&(2)`<br>`$(t,(b,l))-(&2,v)-(&1,v)^(im)))`<br>`(wt,(+(wp)$(s)&(1)?(_)&(2)*(&1)-(wp)#*(wp)-(wp)^(wt)))`<br>`(im,($(l,(y,1))+(im)&(2)*(mr,&1)^(sg)^(sn)-(sn)-(im)^(lb)#&(2)*(im)-(im)`<br>`^(lm)^(im)))`<br>`(lm,(&(2)*(_,v)^(l)&(3)$(f,&2)+(&1,b)&(1)$(f,&3)-&3$(m,(&1,s))))`<br>`(lb,($(l,(y,0))&(2)*(_,b)^(l)&(3)$(f,&2)&(1)$(f,&3)-&3$(m,(&1,s))^(lb)#`<br>`&(1)*(_,n)&(2)$(f,&1)&(1)$(f,&2)!(&1,wp)$(k)))`<br>`(l,(&(3)$(r,&2)$(e,(e,&3))&(2)$(e,(c,&3))^(l)))`<br><br>`(nb,($(l,(r,1))$(n)^(cn)))`<br>`(cn,(^(gn)^(gi)^(tm)#^(cn)))`<br>`(gn,(+(bk)&(1)*(NB,_,_)$(l,(r,0))&(2)$(f,&1)-&2-(bk)#&(1)*(bk)-(bk)^(bk)))`<br>`(gi,(&(1)$(r,&2)&(2)$(f,&1)))`<br>`(tm,($(t,(q,&2))-(NB,_,_)$(m,(&2,s))^(cl)))`<br>`(bk,(&(1)$(t,(b,f))+(b,&1)&(2)*(b,(1))!(1,endlog)$(k)#&(2)$(t,(i))-(&2,v)`<br>`$(m,(&1,s))^(nb)))` |
| `Plan` | `^(cl)` |

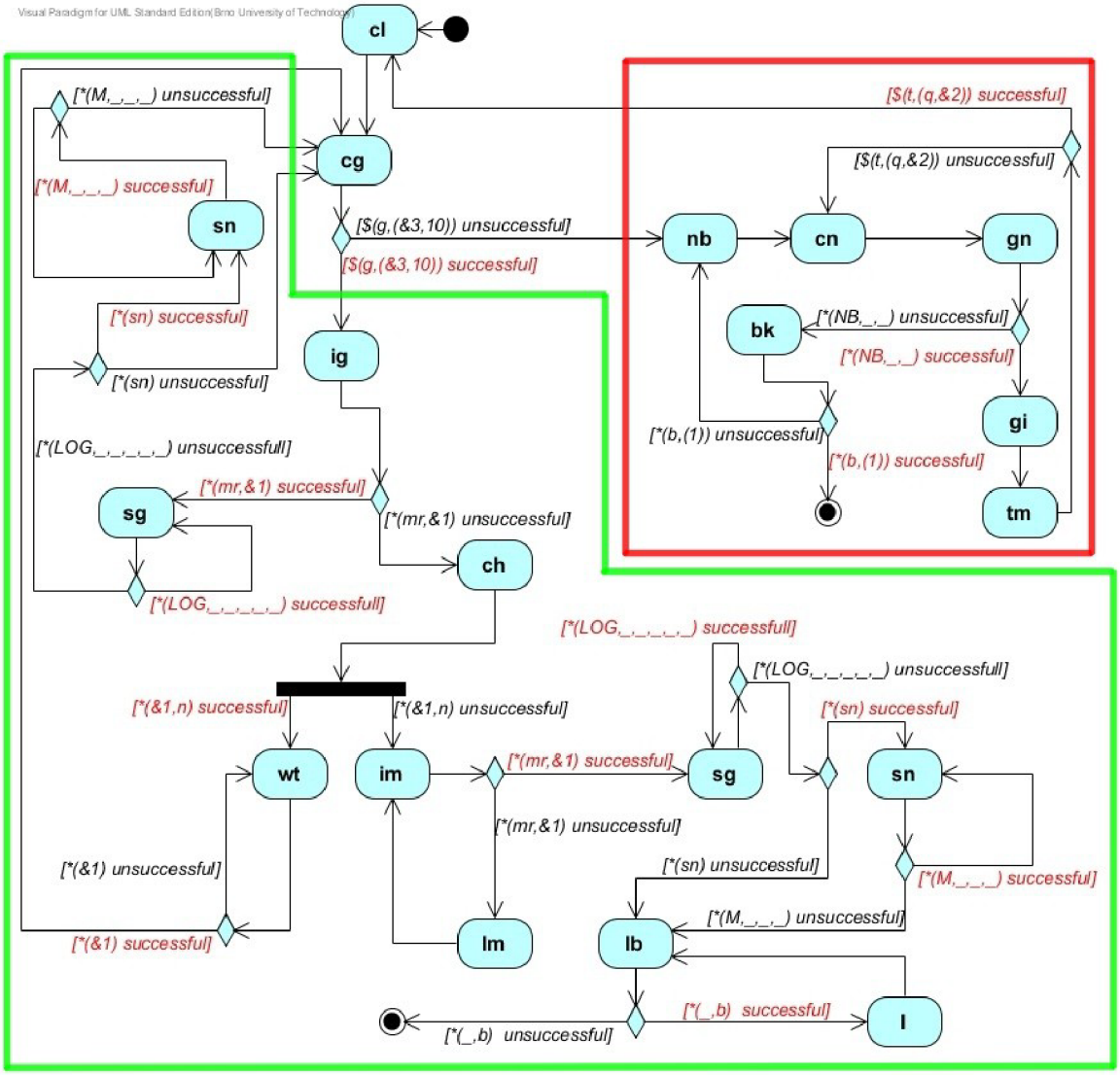
Tabulka 11.: Kód agenta výsledné aplikace v ALLL s adresou 2 počátečního uzlu v průchodu a adresou 1 pro základnovou stanici.

Agent začíná vykonáváním plánu `cl` vždy na uzlech, které doposud nenavštívil. Po příchodu na uzel vytváří n-tici ve tvaru `((<adresa uzlu>),v)`, kterou ukládá do `BeliefBase` pro pozdější použití agentem otrokem, a pomocí služby `$(e,(z))` iniciuje index na nulu, od kterého se budou číst záznamy. V této části se provede také sběr informací o sílách signálu ke svým sousedům pro potřeby vykreslení topologie. Nejprve byly tyto informace sbírány až při navracení agenta v cestě zpět, ale toto se ukázalo při použití více uzlů jako nevhodné vzhledem k množství[59] takových informací. Toto bylo příčinou toho, proč musely být i tyto informace odesílány z každého uzlu zvlášť. Informace o sílách signálu jsou sbírány pomocí služby `$(n,m)` a odesílány (pomocí plánu `sn`[60]) vždy s prvním agentem otrokem. Služba `$(n,m)` je do `BeliefBase` ukládá v jiném tvaru, než je tam standardně ukládá služba `$(n)`, z důvodu toho, aby se tyto informace vzájemně nepletly. Dalším důvodem je způsob zpracovávání topologie ve webovém rozhraní `ControlPanel` pomocí existující třídy `RSSIParser`. Do dané n-tice totiž potřebujeme nyní vložit i adresu uzlu, který toto prohledávání okolí provádí. Více o tomto zpracovávání a vykreslování topologie bude řečeno v *kapitole 5.4.3*.

59 Je třeba počítat s tím, že pokud obecně všechny informace v `BeliefBase`, které se vybírají pomocí stejné unifikace, budou velikosti například až 300 znaků, tak při jejich vyzvedávání do registru za účelem odeslání se potřeba volného místa pro ně zdvojnásobí až na 600 znaků.

60 Plán `sn` s podmínkou vykonání pouze u prvního agenta otroka se nachází vždy za plánem `sg` a již nebude dále zmiňován.

Po inicializaci agent spustí hlavní cyklus reprezentovaný plánem `cg` starající se o výběr záznamů (byl již předveden v *tabulce 7.)*. V tomto cyklu se nachází plán `ig`, který se stará o jejich doručení. Nepřímé spuštění plánu `ig` je ohraničeno rozsvícením a pohasnutím zelené ledky pro signalizaci, že daný uzel se pokouší o toto doručení. Způsob doručení je závislý na tom, jestli se agent nachází na prvním uzlu ve své cestě nebo dalších. Agent tedy porovná svou aktuální adresu uzlu s n-ticí `(mr,(2))`, která byla předem uložena do `BeliefBase`. Pokud se nachází na prvním uzlu ve své cestě, tak odesílá záznamy po jednom[61] přímo na základnovou stanici pomocí plánu `sg`, ve kterém je umístěna služba pro vytvoření jednoduché zprávy `!(1,&1)`. Když se však nenachází na prvním uzlu, ale některým z uzlů dalších, tak dochází k vytvoření agenta otroka způsobem, jak bylo popsáno v *kapitole 5.2.2*. Poslední operace (nepřímé spuštění plánu `im`) v plánu `ch` reprezentuje počátek cesty agenta otroka směrem k základnové stanici. Způsob provedení této cesty není jednoduchý, a tak samotný vyžaduje jisté kompromisy popsané dále v *kapitole 5.3.1*.




Obrázek 9.: Závislosti mezi pod-plány výsledné aplikace v ALLL.

61 Bylo zvoleno odesílání záznamů po jednom, jelikož jeden záznam může mít délku až 35 znaků.

*Obrázek 9.* ukazuje závislosti mezi pod-plány. Jak již bylo psáno v *kapitole 4.3.2*, tak na vyjádření těchto závislostí se hodí diagram aktivit z jazyka UML. Na obrázku jsou také barevnými rámy ohraničeny dvě zmiňované části agenta. Zelený rám ohraničuje pod-plány starající se o sběr a doručení záznamů a červený rám ohraničuje pod-plány starající se o přesun agenta v síti. Podmínky jsou vyjádřeny úspěšností či neúspěšností operace, na které je závislá změna pod-plánu. Jde tedy vždy o dvě možnosti cesty. V *obrázku 9.* je také naznačeno vytvoření agenta otroka pomocí symbolu `fork`. Šipky vycházející z tohoto symbolu obsahují také podmínku, která vyjadřuje právě to odlišení posloupnosti zpracovávání plánů na jednom z uzlů pomocí selhání plánu.

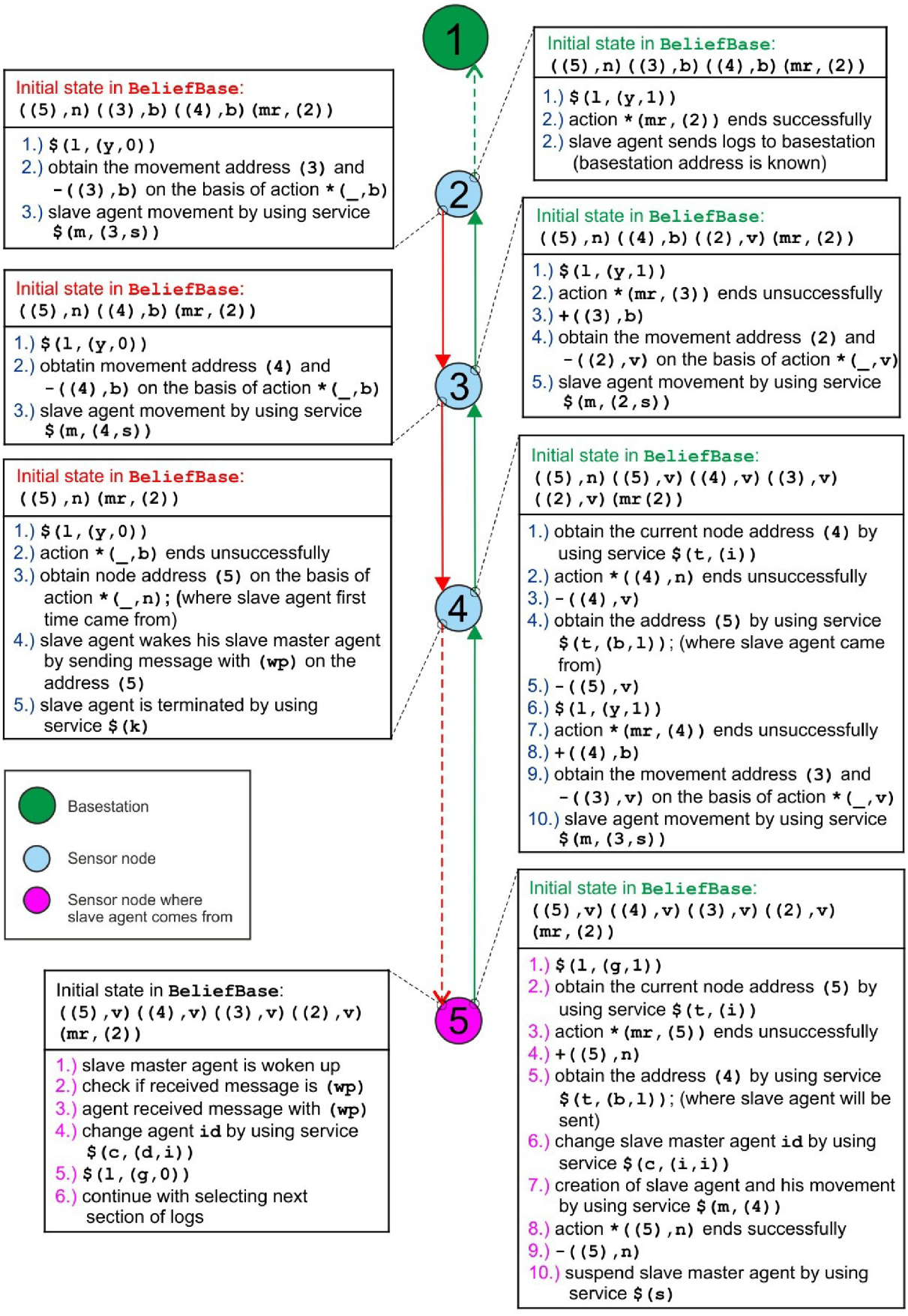
### 5.3.1 Popis cesty agenta otroka

*Obrázek 10.* zobrazuje příklad cesty[62] agenta otroka směrem k základnové stanici. Z důvodu toho, že agent otrok má novou identifikaci, tak nemá žádné pachové stopy, které by mohl při své cestě využívat. Může tedy využívat pouze n-tic navštívených uzlů tvaru `((<adresa uzlu>),v)` v `BeliefBase`, které mu pro jeho účely cesty k základnové stanici spravuje agent otrokář. Důležité je nejen přidávání daných n-tic při dopředném průchodu, ale i jejich mazání při navracení (lze vidět v pod-plánu `bk`), aby agent otrok šel přímo po cestě otrokáře k základnové stanici. Pokud by totiž agent otrokář dané n-tice v pod-plánu `bk` nemazal, tak by agenti otroci mohli chodit i na konec a zpět větvemi v síti, které sice agent otrokář navštívil a cesta přes ně vede k základnové stanici, ale tato cesta by pro ně byla zbytečným odbočením.

*Obrázek 10.* je nutno číst ve směru šipek. Před první šipkou u uzlu s adresou číslo `5` lze ještě v agentu otrokáři vidět, že se výše zmíněné n-tice nacházejí v `BeliefBase` v pořadí, v jakém má agent otrok jít směrem k základnové stanici. V této tabulce se dále nacházejí důležité akce, které agent otrokář provede pro vytvoření agenta otroka a odeslání jej na uzel s adresou číslo `4` (popisuje *kapitola 5.2.2*). Na uzlu s adresou číslo `4` lze již vidět, což je důležité zmínit, že agent otrok poté využívá pořadí, s jakým se vybírají dané n-tice z `BeliefBase` pomocí unifikace. Daná unifikace sice vybere prvky v opačném pořadí, ale agent otrok poté využívá způsobu výběru posledního prvku pomocí průchodu seznamem z ukázky *tabulky 9. kapitoly 5.2.1*. V tabulkách *obrázku 10.* je také naznačeno rozsvěcování žluté diody na uzlech v cestě směrem k základnové stanici pro jasnou signalizaci toho, kde se agent otrok momentálně nachází. Při cestě směrem k základnové stanici přitom agent otrok maže výše zmíněné n-tice a naopak je ukládá v tvaru `((<adresa uzlu>),b)` pro pozdější potřebu cesty směrem k agentu otrokáři. Toto si může dovolit, jelikož dané n-tice v agentu otrokáři stále zůstávají a při další potřebě doručení záznamů pomocí agenta otroka je opět vytvořen agent otrok nový. Pokud agent otrok přijde až k uzlu těsně před základnovou stanicí, jež pozná podle n-tice `(mr,(<adresa uzlu>))`, tak odešle záznamy do základnové stanice a obrací se v cestě zpět právě po svých ukládaných n-ticích ve tvaru `((<adresa uzlu>),b)`, kdy žlutou diodu na každém prvku pohasíná. Když pak agent otrok dojde až na uzel, na kterém začínal svou cestu, tak odešle jednoduchou zprávu ve tvaru `(wp)`, jež vzbudí agenta otrokáře, a ukončí se. Odesílání jednoduchých zpráv je v *obrázku 10.* naznačeno přerušovanou čárou. V *obrázku 10.* se také pro jednoduchost nezobrazují nasbírané záznamy v `BeliefBase`, které jsou doručovány. V reálném případě totiž nikdy nedojde k tomu, aby agent otrok byl vytvořen, pokud agent otrokář nebude mít žádné záznamy k doručení.

---

62 *Příloha 6.* (CD) obsahuje videa, která prezentují přesuny agenta otroka směrem k základnové stanici.




Obrázek 10.: Příklad cesty agenta otroka.

### 5.3.2 Testování

Výsledná aplikace v ALLL byla testována na různých topologiích pomocí simulačního nástroje TOSSIM. *Příloha 6.* (CD) obsahuje skripty těchto vytvořených topologií v jazyce Python. Jelikož v aplikaci hraje velkou roli základnová stanice, kterou simulátor TOSSIM implicitně nepodporuje, tak ji bylo nutno uměle vytvořit. Lze ji částečně odsimulovat tím, že jeden uzel se za ni bude vydávat. V skriptu je nutno vytvořit o jeden uzel více, než je počet uzlů v navržené topologii. V modulu `NeighbourDiscoveryM`, který se stará o službu zjišťování sousedů, je nutno zamezit v přidání n-tice s uzlem, který je za základnovou stanici považován. Aby první uzel topologie obsahoval i záznam o příjmu agenta, jako při reálném příjmu od základnové stanice, tak je agenta nutno umístit (v implementaci signálu `booted` modulu `AgentC`) do uzlu, který představuje základnovou stanici. Tento agent navíc bude mít jako první operaci službu agentní mobility, co jej na daný první uzel z topologie přesune.

Za popis otestování výsledné aplikace na reálných mote lze považovat *kapitolu 6*, která pojednává o praktických experimentech se systémem WSageNt. Při těchto experimentech byl z webového rozhraní posílán právě tento agentní kód výsledné aplikace.

## 5.4 Analýza a vykreslení tras agentů

Tato kapitola se již zabývá analýzou a samotným vykreslením tras agentů podle návrhu z *kapitoly 4.4*. Aby webové rozhraní `ControlPanel` mohlo tuto práci udělat, tak nejprve musí zajistit to, aby se do databáze dostaly zprávy se záznamy, které můžou být analyzovány. K tomuto účelu byl vytvořen v prezentační vrstvě architektury nový pohled `WSTracingAgents.jsp`. V tomto pohledu je možno vytvořit požadavek, kterým budou dané záznamy sesbírány. Při spuštění tohoto požadavku je vykonána metoda `execute` objektu třídy `WSRequestTracingAgents`[63]. Prakticky je zde odeslán agent z *kapitoly 5.3*, a tak je tedy nutno před tímto spuštěním navíc zadat adresu uzlu, na kterou se má agent odeslat, id agenta a třídu agenta, která musí být pokaždé unikátní[64]. Uzel s danou adresou musí být v této metodě `execute` uložen jako objekt třídy `Mote` do databáze (pokud se v ní již nenachází), jelikož je tento objekt v databázi poté provázán se zprávami, které od něj přichází. Pokud je navíc tento uzel ve spojení se základnovou stanicí, což by z hlediska funkčnosti sběru záznamů být měl, tak je zobrazen i v přehledu online mote v aktivním panelu. Po vykonání metody `execute` je ihned opět zobrazen pohled `WSTracingAgents.jsp`.

### 5.4.1 Indikace sesbíraných záznamů v databázi

Jelikož se zprávy se záznamy do databáze ukládají postupně, jak je přijímá konzolová aplikace `BSComm`, tak je zde v kapitole důležité závěrečné odeslání zprávy `endlog` agentem po jeho celkovém průchodu (bylo zmíněno v *kapitole 5.3)*. Protože ve webovém rozhraní již existoval aktivní prvek, který byl určený pro upozorňování uživatele na změny a události, tak k indikaci příjmu všech zpráv se záznamy bylo možné zvolit právě tento prvek. Do tohoto prvku (přesněji až do grafické

63 Všechny třídy s příponou `WS` jsou součástí balíčku `controlpanel`. Více o logickém rozdělení jednotlivých tříd do balíčků lze najít v diplomové práci [18].

64 Viz *kapitola 5.2.*

komponenty `MessageNotification.jsp`) tedy byla přidána třetí[65] funkce. Tato funkce plní následující:

- pokud je v databázi nějaká nová zpráva obsahující záznam, tak se zobrazí obecné oznámení[66],
- pokud je v databázi nová závěrečná zpráva `endlog`, tak se zobrazí závěrečné oznámení[67].

Pokud následně uživatel zvolí jednu z těchto ikon na aktivním panelu, tak se přepne na pohled `WSTracingAgents.jsp`. Jelikož se vždy při tomto spuštění pohledu vykonává metoda `execute` objektu třídy `WSTracingAgents`, tak ta následně zajistí roztřídění nově příchozích záznamů obsažených ve zprávách. Tento způsob indikace sesbíraných záznamů v databázi je výhodný. Pokud si totiž uživatel sám manuálně nadefinuje svého agenta, který obecně dané zprávy se záznamy do databáze dostane, tak je možné tímto způsobem uložené záznamy také roztřídit.

### 5.4.2 Způsob roztřídění

Třídou poskytující nejprve parsování a poté roztřídění záznamů je třída `LOGParser`[68]. Seznam textových zpráv se záznamy je totiž nejprve nutné rozparsovat a přeměnit na seznam objektů třídy `LOGRecord`, aby každý objekt této třídy reprezentoval jeden záznam. Poté je nutno zjistit všechny uzly (adresy mote), které agent sbírající záznamy prošel, jelikož tyto uzly budou do databáze také uloženy[69] (stejně jako uzel použitý při požadavku). Nejdůležitější použití třídy `LOGParser` spočívá ve vytvoření seznamu objektů třídy `TracingAgent`. Jeden prvek v tomto seznamu totiž představuje jednoho agenta (roztříděno podle jeho identifikátorů), ke kterému je přiřazen seznam adres uzlů, kde nyní pouze daný agent byl, a seznam objektů třídy `Trace`, které představují jednotlivé přesuny tohoto agenta.

Část těchto roztříděných informací se poté v pohledu `WSTracingAgents.jsp` zobrazí v přehledné tabulce[70] jako přehled sesbíraných agentů. V této tabulce se také u každého agenta vyskytuje čas, kdy byl vyslán do sítě, pokud se jeho identifikátory shodují s nějakým agentem uloženým v databázi. Pokud společně se zprávami se záznamy přišly i zprávy s informacemi o sílách signálu ke svým sousedům, tak je u každého agenta zobrazena ikona, která umožňuje vykreslení jeho přesunů. Po zvolení tohoto vykreslení je uživatel přesměrován na nový pohled `WSDrawTraces.jsp`, který se společně s třídou `WSDrawTraces` o toto zobrazení vykreslení[71] postará. V tomto pohledu jsou zobrazeny jednotlivé přesuny agenta také textově v tabulce[72].

### 5.4.3 Způsob vykreslování

Pro vytvoření vykreslení přesunů agenta již návrh předpokládal použití tříd `RSSIParser` a `TopologyMath`, kdy pouze první zmiňovaná musela být lehce uzpůsobena.

---

65 První funkcí je kontrola nově přijatých zpráv a druhou funkcí je monitorování dostupných (online) uzlů. Tyto funkce představuje *kapitola 3.3*.

66 *Obrázek 20.* obecného oznámení lze najít v *příloze 3.*

67 *Obrázek 20.* závěrečného oznámení lze najít v *příloze 3.*

68 Pokud nebude řečeno jinak, tak všechny třídy dále v *kapitole 5.4* byly umístěny do balíčku `util`.

69 *Obrázek 24.* příkladu tabulky s takto uloženými mote lze najít v *příloze 3.*

70 *Obrázek 21.* příkladu takové tabulky lze najít v *příloze 3.*

71 Obrázky příkladů takových vykreslení lze najít v *příloze 4.*

72 *Obrázek 22.* příkladu takové tabulky lze najít v *příloze 3.*

Do již existující třídy `RSSIParser` byla doprogramována nová veřejná metoda. Příčiny vytvoření této nové metody byly dvě. První příčina byla v tom, že je nyní potřeba parsovat jiné n-tice. Pro vytvoření vzdálenosti mezi dvěma uzly je nutná adresa zdrojového uzlu, adresa cílového uzlu a hodnota síly signálu mezi nimi. V původní metodě[73] se parsuje n-tice ve tvaru `(NB,4,130)`, kdy adresa cílového uzlu je `4`, hodnota síly signálu `130` a zdrojová adresa uzlu je taková, odkud přijde zpráva s takovou n-ticí. Protože nyní zprávy s informacemi přicházejí stále od jednoho uzlu, tak potřebujeme parsovat n-tici `(M,3,4,130)`, kde je nyní obsažena i zdrojová adresa `3`. Druhou příčinou vytvoření nové metody bylo to, že v původní metodě se hodnoty signálů parsují ze všech příchozích zpráv, jelikož monitor topologie obsahuje i historii takto sesbíraných signálů. Zde toto není třeba, a tak jsou informace parsovány pouze z nově příchozích zpráv. Tato nově doprogramovaná veřejná metoda vrací objekt třídy `RSSIRecord`, u kterého je podstatné to, že obsahuje seznam objektů třídy `RSSIObject`, které vždy obsahují zmiňovanou zdrojovou adresu, cílovou adresu a sílu signálu.

Použití třídy `TopologyMath` tkví v tom, že dokáže ze sil signálů mezi jednotlivými adresami uzlů vytvořit seznam objektů třídy `PointObject`. Tyto objekty třídy `PointObject` již představují konkrétní body (uzly topologie) s konkrétními souřadnicemi. Třída `TopologyMath` také umožňuje zjištění velikosti potřebného prostoru (šířky a délky obrázku) pro vykreslení.

Pro účely vytvoření objektů, které se již budou pro zobrazení přesunů vybraného agenta jednoduše vykreslovat, bylo nutné vytvořit novou třídu `TraceMath`. Pomocí veřejné metody této třídy lze z konkrétních bodů a roztříděných informací ze sesbíraných záznamů vypočítat konkrétní objekty tříd[74], které jsou uvedeny v *tabulce 12.*

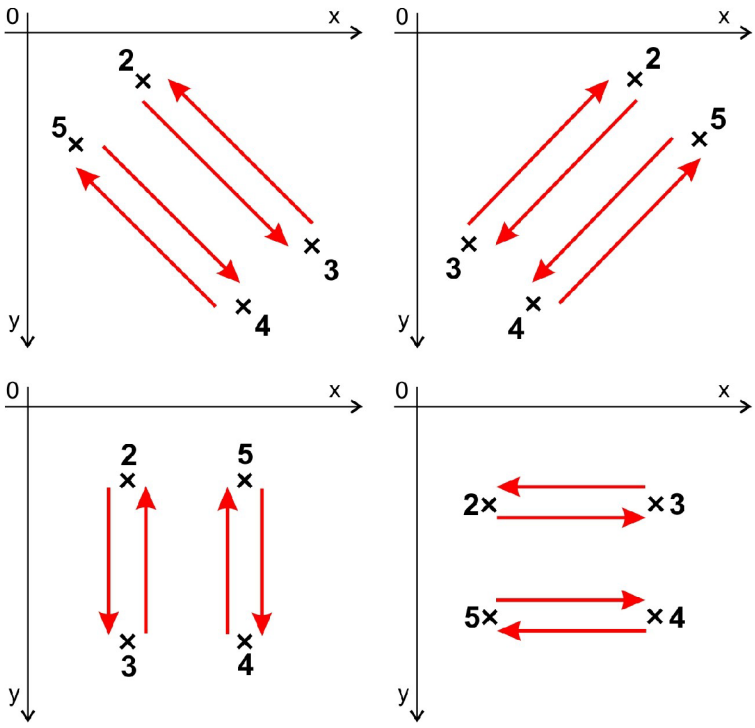
| **název třídy:** | **popis:** |
|---|---|
| `PointObject` | Instance této třídy je používána pro vykreslení uzlu, kterým agent neprošel. Jedná se o stejnou třídu, jejíž instance vytváří objekt třídy `TopologyMath`. |
| `TracePoint` | Instance této třídy je používána pro vykreslení uzlu, kterým agent prošel. Důvod vzniku této třídy je v tom, že bylo nutno odlišit agentem navštívené a nenavštívené uzly. |
| `TraceLine` | Instance této třídy je používána pro vykreslení šipky mezi navštívenými uzly. |
| `AgentPoint` | Instance této třídy je používána pro vykreslení jedné hodnoty čítače přesunu agenta. |

Tabulka 12.: Třídy, jejíž instance jsou používány k vykreslení přesunů vybraného agenta.

Jelikož agent samozřejmě může jít mezi dvěma uzly oběma směry, tak způsob vypočítání souřadnic pro vykreslení šipek (vyjadřujících přesun) není jednoduchý. Důležité bylo zajistit to, aby se šipky mezi dvěma uzly nenakreslily přes sebe. *Obrázek 11.* naznačuje všechny možnosti, které u tohoto vykreslení můžou nastat. Prezentuje prakticky to, že pokud se agent přesunuje z uzlu s větší adresou na uzel s menší adresou, tak se šipka vykresluje na jiné straně než pokud by tomu bylo naopak. Čísla 2 a 4 představují obecně uzly s menší adresou a čísla 3 a 5 představují obecně uzly s větší adresou.

73 Původní metodu stále využívá monitor topologie (viz *kapitola 3.3 Funkce webového rozhraní*).

74 Tyto třídy jsou odvozeny od původní třídy `GeometryObject`.




Obrázek 11.: Všechny možnosti vykreslení šipek.

Podobným způsobem dochází ke korelaci souřadnic i u vykreslení hodnot čítače přesunů, jelikož je tento umístěn vždy uprostřed šipky na straně, na kterou je daná šipka posunuta. U tohoto vykreslení čítače přesunů je navíc těžší to, že se jím může vyjadřovat několika násobný přesun agenta mezi dvěma uzly. V tomto případě bude u dané šipky vykresleno takových hodnot čítačů více.

U obou tříd `TraceLine` i `AgentPoint` jsou dané souřadnice upravovány už v konstruktorech těchto tříd. Díky tomuto lze již hotové objekty předat do objektu třídy `TraceDrawer` pouze na vykreslení. Je tedy vykreslením rozlišováno, zda se jedná o uzel jímž agent procházel či neprocházel, a dále jsou vykresleny dané šipky a hodnoty čítače přesunů. Třída `TraceDrawer` k tomuto vykreslování používá mimo jiných potřebných tříd hlavně třídu `java.awt.Graphics2D`. Vygeneruje prakticky obrázek formátu `jpg`, který je poté do pohledu `WSDrawTraces.jsp` jednoduše vložen pomocí `html` tagu `img`.

### 5.4.4 Testování roztřídění a vykreslování

Testování roztřídění bylo v počátcích implementace testováno na uměle vytvořených záznamech vložených do `BeliefBase` agenta, jenž byl odeslán na libovolný uzel. Daný agent tyto záznamy poté pouze odeslal zpět. Správné vypočítání jednotlivých souřadnic u vykreslovaných objektů bylo poté testováno na základě nahrazení příjmu zpráv s reálnými signály od uzlů. Tyto zprávy tedy byly vytvářeny uměle tak, že na místech hodnot signálů (v daných n-ticích zprávy) byly vkládány náhodné hodnoty v možném rozmezí[75].

75 Možné rozmezí je v rozsahu 0-255. Hodnoty z tohoto rozsahu jsou přepočítané hodnoty, které jsou nezávislé na technologii použitých zařízení [18].

### 5.4.5 Omezení vykreslování

Na základě testování vykreslování z předchozí kapitoly, bylo zjištěno, že použitá třída `TopologyMath`, která byla součástí implementace diplomové práce [18], nedokáže s dostatečnou přesností určit souřadnice bodů (uzlů v topologii sítě). Příčinou je jistě její základ v analytické geometrii a experimentálním určení vzdáleností uzlů. Proto vzdálenosti vykreslených uzlů někdy neodpovídají realitě. Avšak cílem práce bylo pouze sledování agentů na daných uzlech a ne sledování samotných uzlů.

Někdy také třída `TopologyMath` nedokáže vůbec určit potřebné souřadnice bodů z příchozích signálů. Tato situace se však nestává často, a tak byla vyřešena tím, že se ve webovém rozhraní vytvoří chybové hlášení[76], ve kterém je uživatel požádán, aby provedl nový požadavek na sesbírání záznamů včetně informací o signálech k okolním sousedům, kvůli kterým systém nebyl schopen danou topologii vykreslit.

## 5.5 Implementační problémy

Po čas implementace se jak u platformy pro mobilní agenty, tak u celkového systému WSageNt vyskytly problémy, které musely být odstraněny pro funkčnost navrhovaného rozšíření.

### 5.5.1 Doplnění funkcionality

Snadnou spíše úpravou než problémem bylo například doplnění funkcionality služby agentní mobility o práci s registrem nebo korekce služby vracení zbytku seznamu pro zajištění její konzistence. Pro toto byly mírně upraveny pouze funkce `presun` a funkce `remain` z modulu `PlatformSvcC`.

Doplněna byla také konzolová aplikace `BSComm` a webové rozhraní `ControlPanel` o práci s identifikátory (`ID` a `CLASS`) agenta. Toto přepracování si vyžádalo zásahy jednak v samotných třídách `Agent` a `AgentMessage`, ale i v mapovacím konfiguračním souboru `agent.hbm.xml` pro ukládání[77] instancí této třídy. Dále bylo potřeba v aplikaci `BSComm` tyto identifikátory doplnit do třídy `MessageParser`, která se stará o parsování zpráv podle komunikačního protokolu, a třídy `Comm`, která slouží k odesílání paketů zpráv přes sériovou linku směrem k `Basestation`. Ve webovém rozhraní `ControlPanel` bylo třeba tyto identifikátory doplnit do grafické komponenty `WSAgentForm.jsp`, která představuje formulář pro odeslání agentní zprávy, a pohledu `WSSavedAgents.jsp`, který vytváří přehled uložených agentů. Agentní zprávy jsou přes socket odesílány do aplikace `BSComm` pomocí třídy `WSAgentSend`, která musela být také upravena. Přepracována byla také funkcionalita této třídy, jelikož bylo potřeba to, aby se nyní agent uložil, i když je pouze odesílán. Tato funkcionalita je důležitá právě ve vytváření přehledu sesbíraných agentů z *kapitoly 5.4.2*.

---

76 `Unable to determine the nodes of topology. Invalid input data! Please BACK and make new request.`

77 Ukládání probíhá pomocí programovací techniky ORM (Object Relational Mapping), kterou implementuje framework Hibernate. V *příloze 1.* je uveden popis postupu zprovoznění systému WSageNt včetně popisu potřebných nástrojů potřebných k jeho provozu.

### 5.5.2 Problémy při přenosu agenta

Jelikož má výsledný agent v jazyce ALLL z *kapitoly 5.3* v základu (bez jakýchkoliv nasbíraných informací) 1180 znaků, tak problémy nastaly také při jeho přenosu.

Aplikace `BSComm` například neuvažovala odesílání tak velkých agentů, a proto muselo být v základní třídě `Main` upraveno stanovení délky času, který je vždy vymezen na čekání na odpověď, zda byl daný agent odeslán v pořádku. Nyní je čas stanovován v závislosti na délce agenta a ne pouze staticky podle komunikačního protokolu z třídy `InternalCommProtocol`.

Těžší úpravy systému WSageNt byly v nemožnosti přeposílání takto velkých agentů mezi uzly, kdy problémy nastávaly již při agentech o délce cca 300 znaků. Je důležité zmínit, že tato skutečnost se vyskytovala pouze při přenosu takových agentů mezi reálnými uzly (ne v simulačním nástroji TOSSIM). Z tohoto důvodu byl velký problém hlavně v nalezení chyby, která toto způsobovala, jelikož takové ladění kódu je možné pouze pomocí led diod daného mote. Po takovém kompletním odladění modulů `SendM` a `ReceiveM` platformy pro mobilní agenty, bylo zjištěno, že daný problém nastával pouze pokud byl daný agent přenášen sice až mezi uzly, ale byl vyslán z webového rozhraní[78]. Program `Basestation` na základnové stanici totiž sbíral díky funkci `receive` používaného rozhraní `RadioSnoop` všechny pakety, které byly přeposílány i mezi uzly (toto bylo později také odstraněno a důvod je popsán v *kapitole 5.5.3*). Aplikace `BSComm` dané pakety poté od programu `Basestation` přebírala, potvrzovala, sestavovala a ukládala výsledné zprávy do databáze. Problém byl právě v tomto potvrzování. Platformu pro mobilní agenty totiž nezajímalo, zda potvrzení přišlo od uzlu, se kterým komunikovala, nebo z aplikace `BSComm`. Takto se až u větších agentů projevilo to, že aplikace `BSComm` byla v potvrzování rychlejší než přijímající uzel a poté daný přijímající uzel nestačil takového agenta přijímat a sestavovat. Správně přitom nepřijala ani aplikace `BSComm`, jelikož ji naopak někdy předběhl přijímající uzel. Toto bylo ošetřeno v platformě pro mobilní agenty v implementaci události `receive` v modulu `MessageAckM`, kde se nyní platforma u jakéhokoliv potvrzení zabývá od koho dané potvrzení přišlo a porovnává jej s adresou uzlu, kterému aktuálně posílá zprávu.

U přeposílání agentů, nyní tedy až o délce přes 1000 znaků, nastal další problém. Jak již bylo řečeno, tak maximální velikost agenta na platformě může být 2081 znaků. Příjem agenta je prováděn tak, že se pozastaví interpretace kódu (pokud je nějaký přítomen) a postupně se po paketech přijímá nový agent, který je vkládán přímo do pole interpretu pomocí příkazu `insertAt` používaného rozhraní `ArrayOpI`. Po příjmu všech paketů se vypočítává kontrolní součet z tohoto pole, a tak není potřeba žádné zvláštní pole pouze pro příjem. Problém nastal, pokud se daný agent vracel na uzel, na kterém již byl. Příkaz `insertAt` totiž vyžaduje, aby mu při tomto použití byla předložena celková délka agenta, i když je vkládán jen úsek určený jedním paketem. Jelikož se v příkazu `insertAt` samozřejmě kontroluje, zda vkládaná délka plus současná délka nepřesahuje maximální délku pole, tak tato podmínka byla úspěšná[79]. Toto pole interpretu totiž nebylo prázdné. Tento problém nebyl tak složitý jako předchozí, jelikož byl odladitelný v simulačním nástroji TOSSIM. Byl odstraněn pomocí příkazu `deleteAll` používaného rozhraní `ArrayOpI`, který je spouštěn v modulu `ReceiveM` v implementaci příkazu `initialReady`. Příkaz `deleteAll` smaže pole interpretu a nastaví všechny indexy k jednotlivým částem agenta na výchozí hodnoty.

---

78 Agent může být přeposlán mezi uzly i tak, že je vepsán do implementace reakce na signál `booted`, který je vyvolán při startu interpretu v modulu `AgentC`. Uzel, který má agenta odesílat, se pak musí zapnout jako druhý v pořadí.

79 Zobrazovalo se hlášení "`Vkládání přes velikost pole!`".

### 5.5.3 Problémy ve spolehlivosti

I přes odstranění výše zmíněných problémů nebylo stále možné na reálných mote navrženou aplikaci provádět spolehlivě.

Často například zamrzával program `Basestation`. Zamrzával z toho důvodu, že je nyní potřeba přenášet výrazně více dat než dříve, a tak nejspíš nestíhal odebírat a zpracovávat takové množství přenášených dat. Jednak výše zmíněný monitoring pomocí rozhraní `RadioSnoop` musel sledovat výrazně větší agenty, ale také příchozích dat bylo výrazně více. Navíc přes program `Basestation` směrem do sítě chodily i prázdné zprávy z webového rozhraní pro test online mote (toto bylo také přepracováno a je popsáno dále). Problém byl skutečně v programu `Basestation`, jelikož základnovou stanici, na které daný program běžel, stačilo restartovat a poté opět do dalšího zamrznutí vše pracovalo v pořádku. Bylo tedy rozhodnuto k odstranění daného monitoringu spojeného s rozhraním `RadioSnoop`, jelikož pro zachování monitoringu bohužel nestačilo v programu `Basestation` ani zvětšit vstupní a výstupní frontu[80]. Nyní program `Basestation` přijímá pouze ty pakety, které jsou určeny pouze pro něj. Toto přináší bohužel omezení do webového rozhraní, jelikož se z přehledu zpráv posílaných celkově v síti stal přehled zpráv určených pouze pro toto webové rozhraní (základnovou stanici s adresou `1`). Vstupní a výstupní fronta programu `Basestation` však musela být i přes tuto skutečnost nepatrně zvětšena. Toto ponížení funkcionality není až tak vážné, protože obecně v WSN stejně není časté to, že by základnová stanice viděla na všechny uzly a měla tak možnost monitorovat přenos, který se mezi nimi děje.

Dalším problémem z hlediska spolehlivosti bylo, že někdy přestávaly pracovat samotné mote. Toto bylo také neodladitelné v simulátoru TOSSIM, a tak bylo spíše určeno to, že se musí předělat způsob, jakým webové rozhraní zjišťuje své online mote v dosahu. Toto zjišťování způsobem rozesílání prázdné zprávy totiž není zcela standardní. Při každém příjmu takové zprávy se totiž v modulu `ReceiveM` zbytečně signalizoval interpret o příjmu jednoduché textové zprávy pomocí signálu `messageRecieved` (poskytovaného rozhraní `iRecieveI`) a nastavovala proměnná `have_message` na `TRUE`, která je při každém cyklu interpretu testována. V hlavní řídící smyčce interpretu (modul `ControM`) se totiž na začátku každého cyklu volá příkaz `ableToAdd` a pokud je daná proměnná hodnoty `TRUE`, tak se zpráva uloží do `InputBase`.

Zjišťování online mote bylo tedy předěláno do podoby skryté služby platformy, která je vyvolána pomocí příjmu zvláštního typu zprávy. Dříve platforma znala 5 typů zpráv, které byly definovány pomocí souboru `AMAgent.h`. Tyto zprávy jsou popsány v *tabulce 13*. Nyní k těmto pěti typům zpráv přibyly další dva typy. První typ `AM_REFRESH_MSG` slouží pro příjem testu online mote a druhý typ `AM_ACK_REFRESH_MSG` slouží jako odpověď na daný test. Poté co byly tyto typy definovány, tak bylo potřeba definovat nové komponenty `AMSenderCR` a `AMReceiverCR` v konfiguraci `PlatformaC`, která propojuje všechny jednotlivé moduly platformy mezi sebou a okolním světem. K těmto komponentám byly přiřazeny výše zmíněné typy zpráv a byly propojeny s existujícím modulem `MessageAckM` pomocí rozhraní. Pokud momentálně přijde na uzel zpráva typu `AM_REFRESH_MSG`, tak se vyvolá implementace signálu `receive` používaného rozhraní `ReceiveR` v modulu `MessageAckM`. V této implementaci se poté otestuje, zda daná zpráva přišla ze základnové stanice. Pokud zpráva přišla ze základnové stanice, tak je zpět odeslána zpráva typu `AM_ACK_REFRESH_MSG`.

---

80 Vstupní a výstupní fronta je použitá jak pro sériovou linku, tak rádiové spojení.

| typ zprávy: | popis: |
|---|---|
| AM_AGENT_MSG | Definuje agentní zprávu a při příjmu vyvolává implementaci signálu receive používaného rozhraní ReceiveA v modulu ReceiveM. |
| AM_NORMAL_MSG | Definuje normální zprávu a při příjmu vyvolává implementaci signálu receive používaného rozhraní ReceiveN v modulu ReceiveM. |
| AM_ACK_MSG | Definuje zprávu, která je používána pro potvrzení agentní nebo normální zprávy. |
| AM_NEIGBOUR_MSG | Definuje zprávu, která je používána pro odesílání dotazu a příjem výsledků u služby zjišťování sousedů. Při příjmu vyvolává implementaci signálu receive používaného rozhraní Receive v modulu NeighbourDiscoveryM. |
| AM_FOOTPRINT_MSG | Definuje zprávu, která je používána pro odesílání dotazu a příjem výsledků u služby pracující s pachovými stopami. Při příjmu vyvolává implementaci signálu receive používaného rozhraní Receive v modulu FootPrintM. |

Tabulka 13.: Typy zpráv znalé platformou a jejich popis.

Na straně webového rozhraní v objektu třídy CheckOnlineMotes byla metoda vytvářející klasickou prázdnu zprávu zaměněna za metodu vytvářející zprávu pro test online mote. Objekt třídy CheckOnlineMotes je vyvoláván cyklicky v závislosti na časovači, který je přiřazen bloku div frameworku dojo[81] grafické komponenty Right.jsp (aktivního panelu). Vždy po odeslání zprávy přes socket se daná zpráva objeví v konzolové aplikaci BSComm, kde se zjistí o jakou zprávu jde a při zjišťování online mote se zavolá nová metoda sendRefr (třídy Comm). V této metodě se vytvoří paket zprávy třídy RadioClassRefresh, jež je odvozená od třídy Message z balíku TinyOS. Při vytváření nové třídy definující typ zprávy je podstatná její hodnota AM_TYPE. Jedná se o identifikátor dané zprávy, který musí být v tomto případě shodný s identifikátorem AM_REFRESH_MSG definovaným v platformě. Takto vytvořený paket je poté v metodě sendRefr odeslán přes konektor[82] MoteIF původem z balíku TinyOS. Pro příjem odpovědi na takto odeslanou zprávu je nutno u tohoto konektoru zaregistrovat typ zprávy, který chceme přijímat. Tímto je po příjmu vyvoláno samostatné vlákno[83], které provede volání metody messageReceived. V této metodě je podle typu příchozí zprávy určeno o jakou se jedná a pomocí metody publishReportEvent je signalizováno úspěšné odeslání zprávy. Touto metodou je totiž zrušen časovač, který byl po volání funkce sendRefr nastaven. Pokud by tento časovač vypršel dříve, tak by byla daná zpráva považována za nedoručenou a v tomto případě by se takto testovaný mote považoval za nedostupný. Tato skutečnost je totiž samozřejmě odeslána i do webového rozhraní přes socket.

81 Takto je zajištěna asynchronní komunikace se serverem pomocí technologie AJAX.

82 Přesněji přes instanci třídy MoteIF.

83 Pomocí instance třídy PhoenixSource.

# 6 Praktické experimenty se systémem WSageNt

Tato kapitola popisuje praktické experimenty se systémem WSageNt z pohledu implementovaného rozšíření z *kapitoly 5*. Na různých vzdálenostních rozprostřeních jednotlivých uzlů ukazuje to, že agenti (i délky kolem 1400 znaků) byli schopni průchodu celou sítí. *Příloha 6.* (CD) poté obsahuje videa, která tyto průchody agentů sítí prezentují na osmi mote IRIS. Dané mote rozsvěcují led diody přesně podle toho, jak bylo popisováno v jednotlivých podkapitolách *kapitoly 5*. Agent při přesunu na mote nejprve v cyklu odesílá záznamy při rozsvícené zelené diodě. Pokud není na prvním uzlu, tak odesílá agenty otroky směrem odkud přišel. Tito agenti otroci rozsvěcují po cestě žluté diody a po odeslání záznamů tyto diody pohasínají. Když agent přistoupí k navracení, tak použije diodu červenou. Video s názvem `video-1.avi` přibližně odpovídá naměřeným hodnotám *1. požadavku* z *tabulky 15.* a navíc obsahuje pohledy webového rozhraní, které jsou popsány v *tabulce 14.* Video s názvem `video-2.avi` poté přibližně odpovídá naměřeným hodnotám *2. požadavku* a video s názvem `video-3.avi` hodnotám *3. požadavku*.

| **číslo pohledu:** | **popis:** |
|---|---|
| 1. | Pohled zobrazuje prázdný seznam všech mote (před prvním požadavkem na sběr záznamů). |
| 2. | Pohled zobrazuje prázdný přehled všech přijatých zpráv. |
| 3. | Pohled zobrazuje vyslání požadavku. |
| 4. | Pohled zobrazuje odeslaný požadavek, kdy všechny záznamy byly již posbírány. |
| 5. | Pohled zobrazuje roztříděné záznamy s možností volby vykreslení. |
| 6. | Pohled zobrazuje naplněný přehled všech přijatých zpráv. |
| 7. | Pohled zobrazuje uložené navštívené adresy mote v seznamu. |

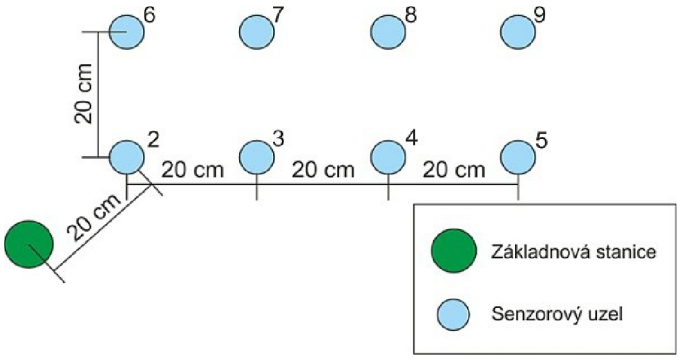
Tabulka 14.: Popis pohledů videa s názvem `video-1.avi`.

## 6.1 Naměřené hodnoty

U topologie, která je znázorněna na *obrázku 12.* byly naměřeny hodnoty z *tabulky 15.* Topologie má celkem 8 uzlů a agent byl vysílán vždy na uzel s adresou `2`. Jelikož jsou uzly blízko u sebe, tak všechny mají spojení se všemi a všechny mají přímé spojení se základnovou stanicí. Tuto topologii lze tedy považovat za ideální.

| požadavek: | celková doba průchodu [min:s]: | počet navštívených uzlů: | počet sesbíraných záznamů o přesunech agentů: | počet sesbíraných informací o okolních sousedních uzlech: |
|---|---|---|---|---|
| 1. | 1:30 | 8 | 8 | 50 |
| 2. | 2:12 | 8 | 72 | 54 |
| 3. | 3:35 | 8 | 136 | 55 |
| 4. | 5:39 | 8 | 218 | 54 |
| 5. | 9:33 | 8 | 324 | 54 |
| 6. | 14:31 | 8 | 443 | 51 |

Tabulka 15.: Naměřené hodnoty u topologie z *obrázku 12.*

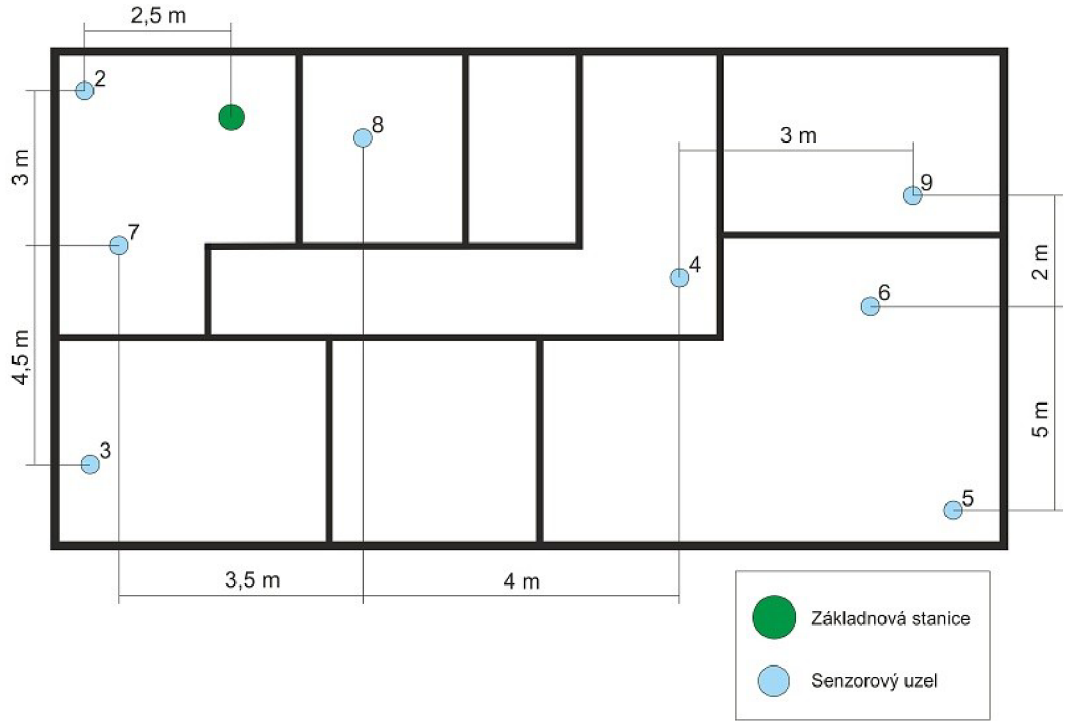



Obrázek 12.: Testovaná topologie 1.

U topologie, která je znázorněna na *obrázku 13.* byly naměřeny hodnoty z *tabulky 16.* Topologie má celkem 8 uzlů a agent byl vysílán vždy na uzel s adresou 2. Tato topologie již může představovat reálné rozmístění uzlů v budově, kde vlivem zdí již není přímé spojení všech uzlů se všemi. Toto lze také vidět v tabulce na tom, že oproti topologii z *obrázku 12.* poklesl počet sesbíraných informací o okolních sousedních uzlech.

| požadavek: | celková doba průchodu [s]: | počet navštívených uzlů: | počet sesbíraných záznamů o přesunech agentů: | počet sesbíraných informací o okolních sousedních uzlech: |
|---|---|---|---|---|
| 1. | 1:53 | 8 | 8 | 44 |
| 2. | 2:04 | 8 | 72 | 50 |
| 3. | 2:56 | 8 | 136 | 48 |
| 4. | 5:45 | 8 | 204 | 51 |
| 5. | 8:42 | 8 | 295 | 46 |
| 6. | 12:06 | 8 | 431 | 47 |

Tabulka 16.: Naměřené hodnoty u topologie z *obrázku 13.*




Obrázek 13.: Testovaná topologie 2.

## 6.2 Zhodnocení

Z obou tabulek je zřejmé, že čas potřebný k sesbírání všech záznamů o výskytu agentů velmi závisí na počtu těchto záznamů. Čím více je záznamů, tím vícekrát musí agent otrok jít od každého uzlu směrem k základnové stanici. Jelikož se vzhledem k maximální velikosti agenta pravidelně doručuje z každého uzlu pouze 10 záznamů, tak je v obou tabulkách vidět, že již u počtu kolem 400 záznamů je doba jejich doručení přes 10 minut.

Tuto dobu dokáže ovlivnit také stav baterií v daných mote, jejich vzdálenost od sebe, či různé okolní rušení. Tímto je doba ovlivňována především proto, že se pak může několikrát po sobě nepodařit[84] přesun daného agenta, což má za následek její prodloužení. U baterií s již slabší kapacitou je to poznat také při okamžitém vyslání dalšího požadavku na sběr záznamů. Největším problémem je, pokud při těchto pokusech přesunu agenta, se přijímající uzel vybije natolik, že již není schopen přijmout vůbec. Takový sběr je poté neúspěšný a musí se vyslat požadavek nový.

84 Po podmínce kontrolního součtu je odeslán požadavek na odeslání celého agenta znovu od prvního paketu.

# 7 Výhody a nevýhody řešení a možná rozšíření do budoucna

Tato kapitola se zabývá výhodami a nevýhodami řešení způsobu sledování výskytu agentů v systému WSageNt, jež je popisován v *kapitole 4* a *kapitole 5*. Následně tato kapitola uvádí možná rozšíření tohoto řešení do budoucna.

Mezi hlavní výhody bezesporu patří to, že dané řešení je nezávislé na přímém spojení všech uzlů se základnovou stanicí. Díky této výhodě tak lze danou aplikaci pro sledování výskytu agentů používat i v rozsáhlejších sítích, které budou vystavěny ad-hoc v různých prostředích. Dříve byla nutnost předem manuálně zadat všechny používané mote (minimálně jejich adresy) do seznamu ve webovém rozhraní. Toto nyní bylo odstraněno tím, že pomocí požadavku na sběr všech záznamů z jednotlivých mote v síti se všechny mote, které byly navštíveny, do tohoto seznamu automaticky uloží. Toto lze považovat také za výhodu, jelikož je dané řešení nezávislé i na tomto počátečním manuálním uložení adres mote.

Do nevýhod řešení lze zařadit to, pokud je agent, který provádí sběr záznamů, odeslán do sítě, kde mají všechny uzly spojení s ostatními. Jak lze vidět na videích v *příloze 6.* (CD), tak tento agent otrokář si vybírá uzel, na který se přesune, v závislosti na tom, jak rychle mu při službě hledání sousedů okolní uzly odpoví. Tato doba odpovědi je závislá na tom, jaká se vygeneruje náhodná hodnota při příjmu zprávy `AM_NEIGBOUR_MSG`[85]. Náhodná hodnota je v tomto případě nutná proto, aby uzel, který požadavek vysílal, stihl přijmout všechny odpovědi[86]. Díky spojení všech uzlů se všemi se tak agent otrokář přesunuje v síti zcela náhodně, jelikož síť není ve stromové topologii. Vzhledem k tomu, že všichni agenti otroci pak kopírují celou cestu agenta otrokáře, tak je jejich cesta k základnové stanici zdlouhavá. Ztrácí se zde totiž i výhoda toho, že agent otrok nemusí jít větvemi v síti, které agent otrokář navštívil. Možné rozšíření do budoucna se tak může zabývat směrováním obecně agentů v síti, aby agent otrokář procházel uzly v síti postupně a agenti otroci tak chodili skutečně přímo k základnové stanici. Pokud třída `TopologyMath` v budoucnu bude schopna vypočítat rozmístění uzlů v síti s větší přesností, tak bude možnost toto rozmístění převést do mřížky, která daným uzlům určí polohu. Tato poloha se poté pomocí agenta, který projde[87] sítí, doručí na jednotlivé uzly a poté již budou následně vyslaní agenti na jednotlivých uzlech vědět, kde se nacházejí. Pak si bude schopen každý následný agent při službě zjišťování sousedů vyžádat i jejich polohu a podle ní se rozhodnout kudy půjde.

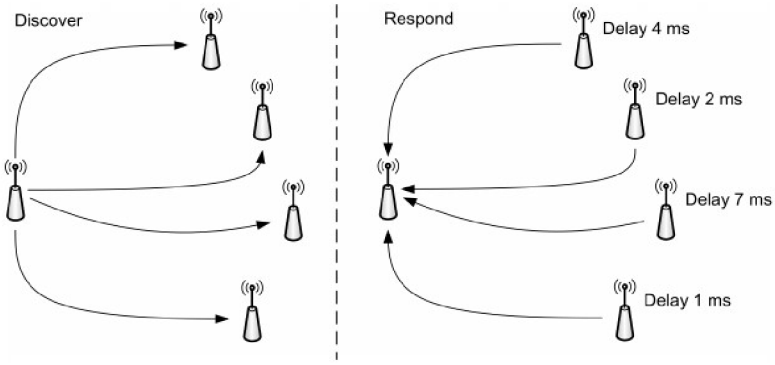
Za dlouhou dobou doručení všech záznamů k základnové stanici stojí také fakt, že agenti otrokáři musí čekat na agenta otroka, až se vrátí zpět k uzlu odkud vycházel. Pokud by nemusel agent otrokář na dané agenty otroky čekat, až dokončí svou práci, tak by se dané doručení výrazně zrychlilo. Toto čekání je nutné, protože nyní obecně agentu při přesunu na jakýkoliv uzel nic nezabrání v tom, aby na daném uzlu přepsal jiného agenta, který může ještě vykonávat nějakou činnost. Bylo by možné do systému přidat služby pro dorozumívání agentů a poté umožňovat toto přepsání pouze na základě priorit. Když by tedy agent měl prioritu větší, tak by mu toto přepsání bylo umožněno, ale pokud menší nebo rovnu, tak by musel čekat, až na daném uzlu agent svoji práci

85 Viz *tabulka 13.*

86 Viz *obrázek 14.*

87 Tento agent bude procházet ještě náhodně.

dokončí. Dané doručení záznamů by mohlo tedy probíhat tak, že by agent otrokář směrem k základnové stanici vytvářel agenty otroky s větší prioritou, než má on samotný. Takto by agent otrokář čekal jen ve dvou případech. V prvním případě by čekal, pokud by vytváření agenti otroci byli pomalejší a on nemohl daným směrem vyslat dalšího agenta otroka, jelikož by na následném uzlu agent otrok již byl. Druhý případ by se týkal situace, kdyby se sám chtěl daným směrem vrátit a na daném uzlu by byl ještě jeho agent otrok.




Obrázek 14.: Požadavek a odpovědi služby pro zjištění sousedů [22].

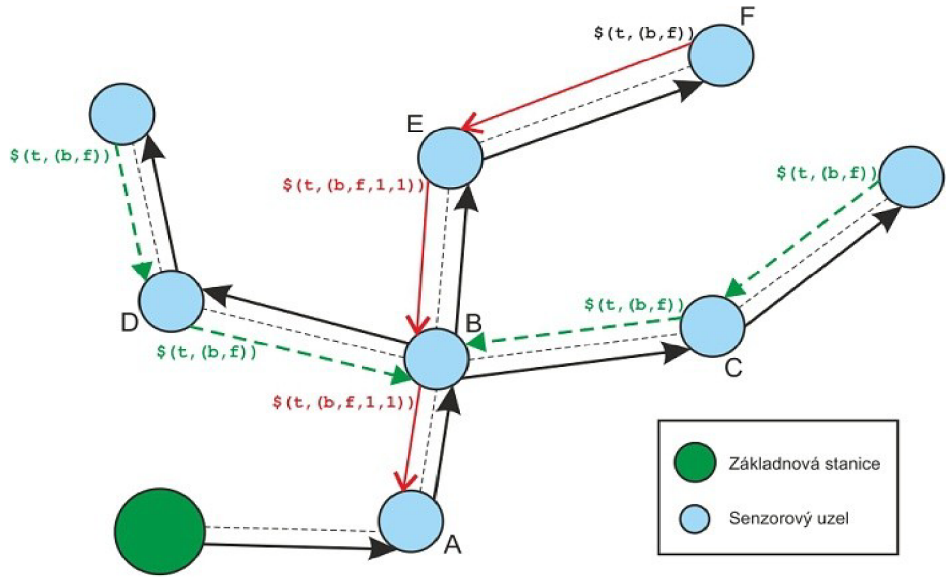
Za částečnou nevýhodu lze identifikovat to, že doručování záznamů agentem otrokem se spoléhá na pořadí, jakým jsou n-tice s hodnotami navštívených uzlů do `BeliefBase` agenta vkládány a vybírány pomocí unifikace. Tento způsob použití n-tic při cestě agenta otroka k základnové stanici sice nesnižuje funkcionalitu, ale dal by se také vyřešit jinak. Při této cestě agenta otroka by šlo využít toho, že agent otrokář vychází k jednotlivým listům ve stromové topologii vždy z kořene (základnové stanice). Takto na jednotlivých uzlech i při jeho odbočení, navrácení ale i cestě do jiné větve stále jako první záznam zůstává jeho příchod ze směru základnové stanice. Pokud by služba pracující s pachovými stopami, konkrétně její funkce navracení[88], umožňovala se ptát na jinou identifikaci, tak by se agent otrok mohl ptát na identifikaci svého agenta otrokáře a byl by schopen se tak navrátit až k základnové stanici. Tímto by odpadla režie s prací s n-ticemi v `BeliefBase`, ale přibyla by režie s prací s identifikátory. Místo v registrech by na ně totiž jistě nebylo, a tak i tyto identifikátory by agent otrok musel mít v `BeliefBase`. *Obrázek 15.* ukazuje způsob této cesty agenta otroka na příkladu rozvětvené topologie, kde čárkované spojnice mezi uzly vyjadřují to, které uzly jsou se sebou schopny komunikovat, černé šipky cestu agenta otrokáře vpřed, zelené čárkované šipky cestu agenta otrokáře zpět a červené šipky cestu agenta otroka směrem k základnové stanici. V obrázku je uvažován až stav, kdy ze všech uzlů byly záznamy doručeny. Obrázek tak ukazuje způsob cest posledních agentů otroků z uzlu `F`. Důležitý je přechodný uzel `B`, kdy u cest agentů otroků z uzlu `F` (ale i uzlu `E` a uzlů z větví `C` a `D`) by byl využíván při cestě z uzlu `B` do `A` první záznam, který agent otrokář (s identifikací id `1` a třídy `1`) vytvořil při přechodu z uzlu `A` do `B` při cestě do jeho první větve `C`.

Pachových stop by bylo možno využít i při cestě agenta otroka zpět k uzlu odkud vycházel. Nyní by se agent otrok ptal funkcí navracení již na své identifikátory. Tyto identifikátory by však musely být vytvářeny tak, aby byly stejné pouze v rámci agentů vysílaných z jednoho uzlu[89]. V rámci uzlů by musely být unikátní, protože by se agent otrok musel funkcí navracení ptát na svoji první

88 Činnosti funkce navracení popisovala *kapitola 4.3.1*. Implementaci této funkce popisuje diplomová práce kolegy Bc. Petra Židka [21].

89 Nyní jsou identifikátory agentů otroků stejné v rámci všech uzlů.

pachovou stopu. K vytváření unikátních identifikací agentů otroků by byla možnost využít adres jednotlivých uzlů tak, že by se id agentů otroků vytvářelo součtem id agenta otrokáře a hodnoty adresy uzlu, ze kterého by byli vysíláni.




Obrázek 15.: Způsob cesty agenta otroka směrem k základnové stanici pomocí pachových stop.

# Závěr

Tato práce díky návrhu a implementaci sběru záznamů o pohybech agentů z jednotlivých uzlů poskytla vyvíjenému systému WSageNt nejen přehled o všech výskytech a cestách agentů v systému, ale i několik přínosů navíc.

Mezi tyto hlavní přínosy práce bezesporu patří to, že na reálné aplikaci měla možnost ukázat důvod, proč je třeba mít možnost vytvořit agenta, který vykoná práci pro agenta jiného. Práce měla možnost i následně ukázat způsob, jakým lze tuto skutečnost opakovaně provádět. Tento fakt je zajímavý i z pohledu toho, že zatím v žádné aplikaci systému WSageNt nebyl předveden, a tak tato práce může sloužit jako podklad pro aplikace budoucí. Práce také představila vlastní implementaci agenta schopného průchodu sítí, který je prakticky bez úprav aplikovatelný i pro jiné budoucí aplikace. Tento agent byl začleněn do hlavní aplikace v ALLL starající se o sběr záznamů jako jeden z pod-plánů, a tak byla jeho aplikace i předvedena. Práce na konstrukci hlavní aplikace ukázala i způsob, jakým lze provádět návrh složitějších aplikací v ALLL pomocí diagramu aktivit. Tyto aplikace by měly být již implementovány bez vážnějších problémů, jelikož byl systém díky danému řešení odladěn pro použití výrazně větších agentů, než tomu bylo dříve. Uvedené použití bylo reálně předvedeno, a tak lze tento fakt považovat za další přínos.

Práce také nastínila možnosti dalšího rozšíření systému v závislosti na jeho nevýhodách. Rozšíření byla určena z pohledu toho, aby mohla být základem pro další především diplomové či bakalářské práce. Bude ještě trvat pár let, než bude systém WSageNt použitelný pro nějakou praktickou aplikační oblast, avšak práce opět potvrdila fakt, že agenti jsou vhodným nástrojem pro monitoring, správu a řízení systémů WSN.

# Literatura


[1] WHITMAN, Edward. SOSUS : The „Secret Weapon“ of Undersea Surveillance. *UNDERSEAWARFARE : The Official Magazine of the U.S. Submarine Force* [online]. Winter 2005, vol. 7, no. 2, [cit. 2010-11-18]. Dostupný z WWW: <http://www.navy.mil/navydata/cno/n87/usw/issue_25/sosus.htm>.

[2] CHONG, Chee-Yee; KUMAR, Peter. Sensor Networks : Evolution, Opportunities, and Challenges. *Proceedings of the IEEE* [online]. Aug. 2003, vol. 91, no. 2, [cit. 2010-11-18]. Dostupný z WWW: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1219475>. ISSN 0018-9219.

[3] SCOTT, M.D.; BOSER, B.E.; PISTER, K.S.J. An ultralow-energy ADC for Smart Dust. *IEEE Journal of Solid-State Circuits* [online]. July 2003, vol. 38, no. 7, [cit. 2010-11-25]. Dostupný z WWW: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1208460>. ISSN 0018-9200.

[4] ROLADER, Glenn. *SAIC Team Program* [online]. 2002 [cit. 2010-11-21]. Self Healing (Antitank) Minefield (SHM). Dostupné z WWW: <http://www.dtic.mil/ndia/2002mines/rolader.pdf>.

[5] PUCCINELLI, D; HAENGGI, M. Wireless sensor networks : applications and challenges of ubiquitous sensing. *IEEE : Circuits and Systems Magazine* [online]. 2005 , vol. 5, no. 3, [cit. 2010-11-21]. Dostupný z WWW: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1507522>. ISSN 1531-636X.

[6] *An Australian company, Telepathx Ltd* [online]. 2010 [cit. 2010-11-21]. Real Time Fire Mapping. Dostupné z WWW: <http://www.telepathx.com/solutions_fire.htm>.

[7] WERNER-ALLEN, Geoffrey; LORINCZ, Konrad; WELSH, Matt. Deploying a Wireless Sensor Network on an Active Volcano : Sensor Network Applications. *IEEE Internet Computing* [online]. April 2006, vol. 10, no. 2, [cit. 2010-11-21]. Dostupný z WWW: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1607983>. ISSN 1089-7801.

[8] KRISHNAMURTHY, Lakshman, et al. Design and deployment of industrial sensor networks : experiences from a semiconductor plant and the north sea. *In SenSys ’05 : Proceedings of the 3rd international* [online]. 2005, vol. 5, [cit. 2010-11-22]. Dostupný z WWW: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.138.3160>.

[9] INTANAGONWIWAT, Chalermek, et al. Directed Diffusion for Wireless Sensor Networking. *IEEE/ACM Transactions on Networking* [online]. 2003, vol. 11, [cit. 2010-12-07]. Dostupný z WWW: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.3918>.

[10] WALTERS, John Paul, et al. *Wireless sensor network security : A survey,” in book chapter of Security* [online]. United States : CRC Press, 2007 [cit. 2010-12-06]. Dostupné z WWW: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.77.3003>.

[11] *Memsic Powerful Sensing Solutions for a Better Life* [online]. 2010 [cit. 2011-03-18]. Wireless Modules. Dostupné z WWW: <http://www.memsic.com/products/wireless-sensor-networks/wireless-modules.html>.

[12] GAY, David, et al. NesC 1.3 Language Reference Manual [online]. July 2009, [cit. 2010-12-09]. Dostupný z WWW: <http://sourceforge.net/projects/nescc/>.

[13] GAY, David; LEVIS, Philip; BEHREN, Robert. The nesC Language : A Holistic Approach to Networked Embedded Systems [online]. San Diego, California, USA : PLDI, June 2003 [cit. 2010-12-08]. Dostupné z WWW: <http://www.cs.berkeley.edu/~pal/pubs/nesc-pldi03.pdf>.

[14] LEVIS, Philip; GAY, David. *TinyOS Programming* [online]. Stanford University, July 16, 2009 [cit. 2010-12-09]. Dostupné z WWW: <http://csl.stanford.edu/%7Epal/pubs/tos-programming-web.pdf>.

[15] LEVIS, Philip; LEE, Nelson. TOSSIM : A Simulator for TinyOS Networks [online]. September 17, 2003, [cit. 2010-12-11]. Dostupný z WWW: <www.cs.berkeley.edu/~pal/pubs/nido.pdf>.

[16] HORÁČEK, Jan. *Platforma pro mobilní agenty v bezdrátových senzorových sítích*. Brno, 2009. 55 s. Diplomová práce. VUT FIT.

[17] SPÁČIL, Pavel. *Mobilní agenti v bezdrátových senzorových sítích*. Brno, 2009. 34 s. Bakalářská práce. VUT FIT.

[18] GÁBOR, Martin. *Webové rozhraní pro sledování provozu v bezdrátových sítích*. Brno, 2010. 60 s. Diplomová práce. VUT FIT.

[19] *Storage - TinyOS Documentation Wiki* [online]. 2009 [cit. 2011-03-26]. TinyOS. Dostupné z WWW: <http://docs.tinyos.net/index.php/Storage>.

[20] GAY, David; HUI, Jonathan. *TinyOS* [online]. 2010 [cit. 2011-03-26]. Permanent Data Storage (Flash). Dostupné z WWW: <http://www.tinyos.net/tinyos-2.x/doc/html/tep103.html>.

[21] ŽIDEK, Petr. *Použití inteligentních agentů v bezdrátových senzorových sítích.* Brno, 2011. Diplomová práce. VUT FIT.

[22] HORÁČEK, Jan; ZBOŘIL, František. WSageNt: A case study. *Proceedings of CSE 2010 International Scientific Conference on Computer Science and Engineering*. 2010, vol. 1, s. 258-264. ISBN 978-80-8086-164-3.

[23] *Getting Started with TinyOS* [online]. 2010 [cit. 2011-05-01]. TinyOS. Dostupné z WWW: <http://docs.tinyos.net/index.php/Getting_Started_with_TinyOS>.

# Seznam použitých zkratek

| | |
|---|---|
| WSN | Wireless Sensor Networks |
| nesC | network embedded systems C |
| ALLL | Agent Low Level Language |
| SOSSUS | Sound Surveillance System |
| IUSS | Integrated Undersea Surveillance System |
| NOAA | National Oceanic and Atmospheric Administration |
| DNS | Distributed Sensor Networks |
| DARPA | Defense Advanced Research Projects Agency |
| MEMS | Micro Electro Mechanical Systems |
| ADC | Analog-to-digital converter |
| SRAM | Static Random Access Memory |
| FSM | Finite-state machine |
| SHM | Self-Healing Mines |
| SAIC | Science Application International Corporation |
| UPCO | Universal Propulsion Company |
| EKG | Elektrokardiogram |
| RFID | Radio Frequency Identification |
| PdM | Predictive Maintenance |
| IP | Internet Protocol |
| DoS | Denial of Service |
| MIPS | Milion instrukcí za sekundu |
| HAA | Hardware Abstraction Architecture |
| HPL | Hardware Presentation Layer |
| HAL | Hardware Abstraction Layer |
| HIL | Hardware Independent Layer |
| OTA | Over-the-air-programming |
| CPM | Closest Pattern Matching |
| AJAX | Asynchronous JavaScript and XML |
| UML | Unified Modeling Language |
| JDK | Java Development Kit |
| JRE | Java Runtime Edition |
| ORM | Object Relational Mapping |

# Seznam příloh

Příloha 1. Postup zprovoznění
Příloha 2. Pseudokódy ALLL aplikací
Příloha 3. Náhledy rozšíření systému WSageNt
Příloha 4. Příklady zobrazení vykreslených tras agenta
Příloha 5. Plakát prezentující rozšíření systému WSageNt
Příloha 6. CD se všemi soubory systému WSageNt, testovacími skripty třech vytvořených topologií pro simulátor TOSSIM, videi prezentujícími přesuny agentů otroků na osmi mote IRIS a zdrojovými soubory této diplomové práce.

# Příloha 1. Postup zprovoznění

Tato příloha popisuje postup zprovoznění celého systému WSageNt na linuxové distribuci Kubuntu. Stručně popisuje i software potřebný k jeho zprovoznění.

Základem je instalace systému TinyOS, který byl představen v *kapitole 2.2*. Jelikož Kubuntu podporuje balíčky Debian, tak je výhodné použít TinyOS repositáře. Poté v názorném tutoriálu na stránkách wiki TinyOS [23] existuje postup, jak nastavit prostředí a zkompilovat jeden z ukázkových programů.

Následně je nutné do operačního systému nainstalovat platformu `Java EE` ve verzi 6 (s technickým označením 1.6). Jedná se platformu pro provoz a vývoj podnikových/webových aplikací a je rozšířením základní verze platformy `Java SE`. Musí se nainstalovat balíček `JDK`[90], který obsahuje nástroje pro vývoj programů včetně balíčku `JRE`[91], který by obsahoval pouze programy a knihovny potřebné pro běh programů. Systém pro běh aplikace potřebuje aplikační server kompatibilní s `Java EE`. Je tedy nutné jej do systému nainstalovat. Vyžadován je open-source aplikační server Tomcat ve verzi 6. Tomcat samotný je platformě nezávislá implementace servletů a JSP stránek. Vyžaduje se, aby Tomcat pracoval ve spolupráci s webovým serverem Apache. Musí se tedy nainstalovat i balíček Apache ve verzi 2. Systém používá databázi `MySQL`, proto nainstalujeme `MySQL` server a `MySQL` klienta. Pomocí `MySQL` klienta se později bude vytvářet databáze `wsagent`. Posledním balíčkem (co je potřeba doinstalovat) je v *kapitole 3* zmiňovaný open-source framework `Strutc 2`.

Ve složce, kde je Tomcat nainstalován je nutno přidat do souboru `tomcat-users.xml` role `manager` a `admin` se jménem a heslem. Toto heslo se bude později zadávat pro spuštění `Tomcat Web Application Manager`. Dále se musí povolit oprávnění pomocí kopie souboru `20wsn.policy` do adresáře `/var/lib/tomcat6/conf/policy.d/`. Nyní pomocí `MySQL` klienta se vytvoří databáze `wsagent` pomocí příkazů z *příkladu 12.*

```
~$ mysql -u root -p
mysql> create database wsagent;
mysql> grant usage on *.* to wsagent@localhost identified by 'hesloveslo';
mysql> grant all privileges on wsagent.* to wsagent@localhost;
```

Příklad 12.: Vytvoření databáze wsagent [18].

Jelikož systém ukládá do databáze `MySQL` přímo vytvořené instance objektů a používá na to framework Hibernate, který tak implementuje techniku ORM[92], tak je nutné tento framework[93] nastavit. Existuje konfigurační soubor `hibernate.cfg.xml`, do kterého se nastaví přístup k databázi `MySQL` pomocí uživatele `wsagent` a hesla `hesloveslo`.

Dále je nutné nové překompilování konzolové aplikace `BSComm` pomocí vývojového prostředí `NetBeans`. Před tímto překompilováním se musí po načtení aplikace ještě nastavit linky na knihovny `tinyos.jar` a `mysql-connector-java-5.0.8-bin.jar`. Případně pokud daná kompilace hlásí chyby kvůli nové verzi TinyOS 2.1.1, tak přejmenovat interface RSSI pro získání síly signálu.

---

90 Java Development Kit (JDK)

91 Java Runtime Edition (JRE)

92 Object Relational Mapping (ORM)

93 Framework Hibernate se nainstaloval spolu s aplikačním serverem Tomcat.

Nyní máme v operačním systému všechno nastaveno ke spuštění. Před tímto spuštěním však musíme do svých mote a základnové stanice nahrát příslušné programy. Pro radio-platformu, která bude připojena na základnové desce, je nutné přeložit a nahrát program `Basestation` (nachází se ve složce `/WSageNt/BaseStation`) s identifikací ID uzlu `1`. Poté je potřeba překompilovat platformu pro mobilní agenty (nachází se ve složce `/WSageNt/wsnagent/src`), radio-platformy ostatních mote postupně vydělávat a připojovat na základnovou desku k nahrání této platformy. Důležitá je taková instalace, která definuje odlišné ID uzlů. Každý mote totiž musí mít odlišné ID uzlu, jelikož toto ID slouží jako jeho adresa. Více o kompilaci a nahrávání programů do mote hovořila *kapitola 2.2.2*.

Pro konečné spuštění je nutné nejprve spustit konzolovou aplikaci (nachází se ve složce `WSageNt/BSComm/dist`) pomocí příkazu `java -jar BSComm.jar serial@/dev /ttyUSB1:iris 7777` a až poté můžeme spustit webové rozhraní `ControlPanel` tak, že do prohlížeče napíšeme `localhost:8080`, spustíme manažera (zde se zadává jméno a heslo nastavované v souboru `tomcat-users.xml`) a v tomto manažeru deploadneme soubor `ControlPanel.war`.

# Příloha 2. Pseudokódy ALLL aplikací

Tato příloha obsahuje pseudokódy jazyka C, které názorňují princip aplikací napsaných pomocí jazyka ALLL. Na příklady z této přílohy se odkazuje text diplomové práce.

```
void movement_in_network() {
    neighborhood detection;
    tuples_number = number of detected neighbours in BeliefBase;
    while (1) {
        if (tuples_number >= 1) {
            tuple selection from BeliefBase;
            tuples_number--;
            neighbour address detection;
            using querying nodes service;
            if (agent was not there) {
                agent movement by agent mobility;
                movement_in_network(); /* If function movement_in_network()
                    uses an application, the main function of this
                    application is called here. */
                return;
            }
        } else {
            using backtracking service;
            if (agent is not before basestation) {
                agent movement by agent mobility;
                movement_in_network();
                return;
            } else {
                agent termination;
                return;
            }
        }
    }
}
```

Příklad 13.: Pseudokód naznačující princip agenta procházejícího sítí (umístěn v *tabulce 6.*).

```
void log_selection() {
    log_index = 0;
    NUMBER_OF_SELECTED_LOGS = 3;
    do {
        using service for selection logs with
            parameters log_index and NUMBER_OF_SELECTED_LOGS;
        if (log_index < TOTAL_NUMBER_OF_LOGS) {
            processing of selected logs;
            log_index = log_index + NUMBER_OF_SELECTED_LOGS;
        } else {
            log_index is greater or equal then TOTAL_NUMBER_OF_LOGS;
            end of selection logs;
            break;
        }
    } while (1);
}
```

Příklad 14.: Pseudokód naznačující princip výběru záznamů (umístěn v *tabulce 7.*).

```
void slave_creation() {
    using node identification service;
    storing identification to BeliefBase;
    using backtracking service for obtain the address
       where slave agent will be sent;
    using change identification service for increment slave master agent id;
    (fork) - agent mobility without terminating interpretation of
       code on transmitting node;
    using node identification service;
    checking the node identification against the address storing in BeliefBase;
    if (slave master node identification - same node) {
       suspend_slave_master();
       using change identification service for decrement slave master agent id;
       other activities of slave master agent;
    } else {
       other activities of slave agent;
    }
}

void suspend_slave_master() {
    while (1) {
       suspend until radio message come;
       message pickup from InputBase;
       if (it is message with "wp")
           break;
    }
}
```

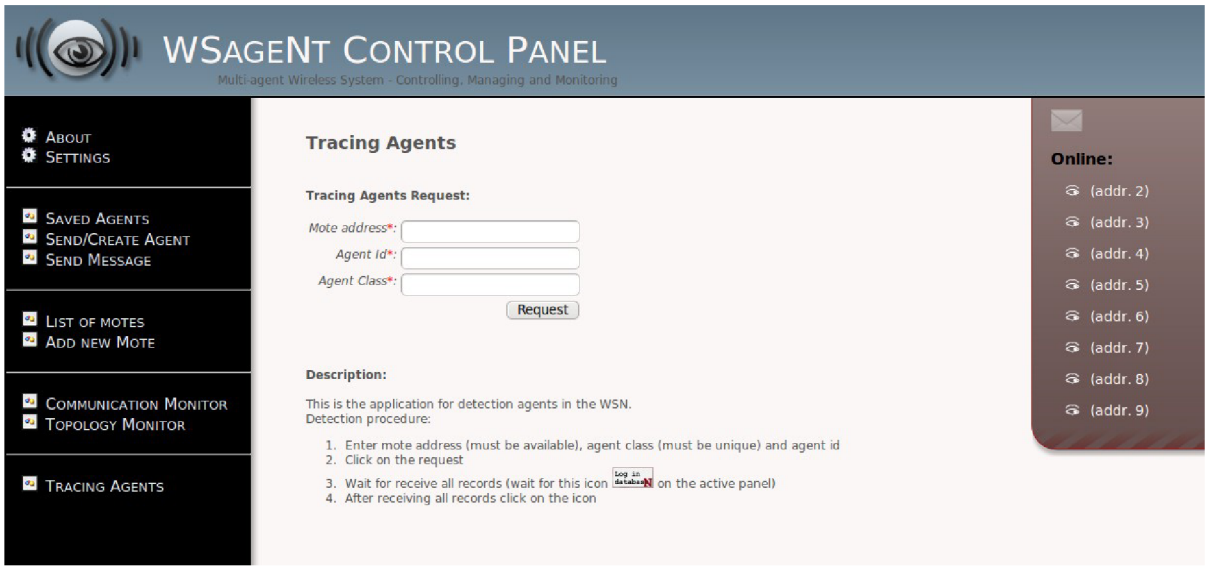
Příklad 15.: Pseudokód naznačující způsob vytváření agenta otroka (umístěn v *tabulce 10.*).

```
void collection_of_logs() {
    while (1) {
       log_selection();
       movement in network;  // This is now considered as means to move.
       if (agent is before basestation) {
           agent termination;
           return;
       }
    }
}
void log_selection() {
    log_index = 0;
    NUMBER_OF_SELECTED_LOGS = 3;
    do {
       using service for selection logs;
       if (log_index < TOTAL_NUMBER_OF_LOGS) {
           if (agent is not before basestation) {
              slave_creation();
           } else {
              sending selected logs to basestation;
           }
           log_index = log_index + NUMBER_OF_SELECTED_LOGS;
       } else {
           end of selection logs;
           break;
       }
    } while (1);
}
void slave_creation() {
    storing identification to BeliefBase;
    using change identification service;
    (fork) - agent mobility;
    checking the identification against the address storing in BeliefBase;
    if (slave master identification) {
       suspend slave master agent;
       using change identification service;
    } else {
       while (1) {
           if (slave agent is not before basestation) {
              move slave agent near to basestation;
           } else {
              sending selected logs to basestation;
              break;
           }
       }
       while (1) {
           if (slave agent is not before node where he came out) {
              slave agent movement near to node where he came out;
           } else {
              send message with "wp" to node where he came out;
              slave agent termination;
              return;
           }
       }
    }
}
```

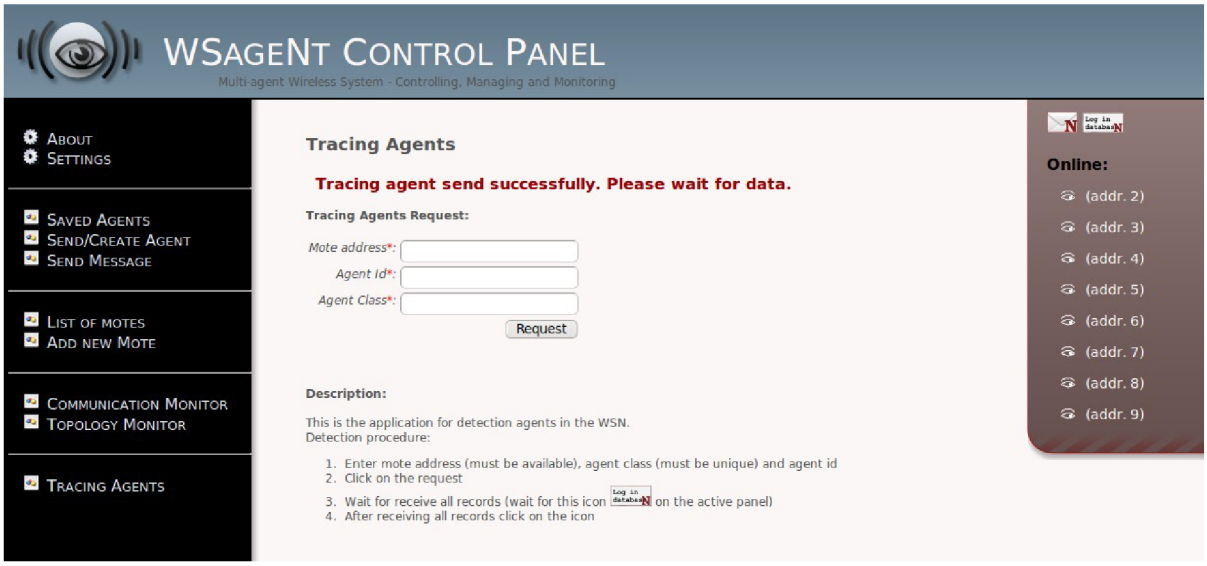
Příklad 16.: Pseudokód naznačující výslednou aplikaci (umístěna v *tabulce 11.*).

# Příloha 3. Náhledy rozšíření systému WSageNt

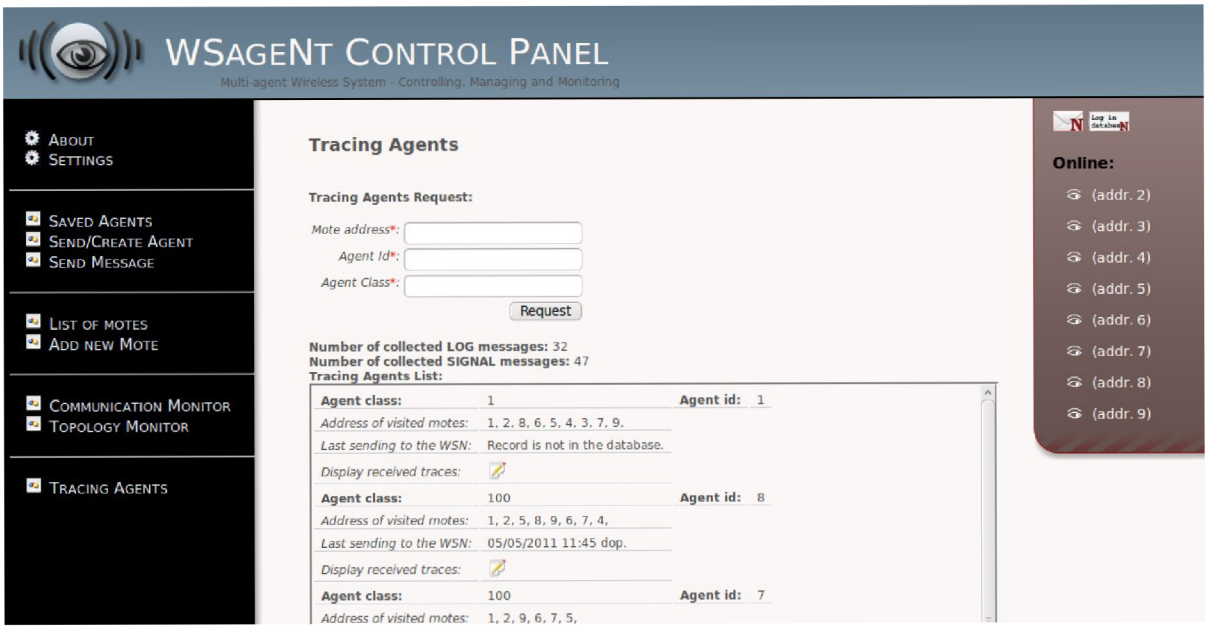
V této příloze lze najít náhledy tykající se rozšíření systému WSageNt. Na řadu těchto náhledů se odkazuje text diplomové práce.



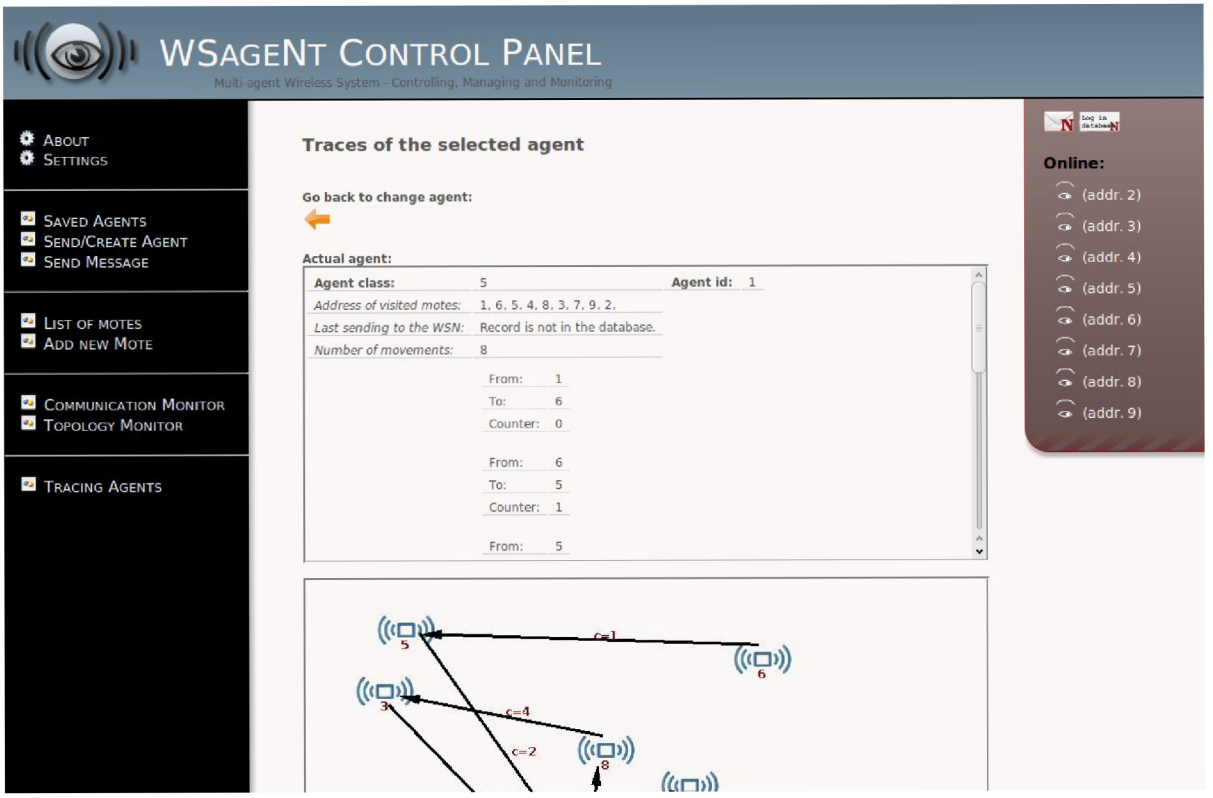

Obrázek 16.: Pohled (`WSTracingAgents.jsp`) s popisem, ve kterém je možno vytvořit požadavek na sesbírání všech záznamů z uzlů v síti.



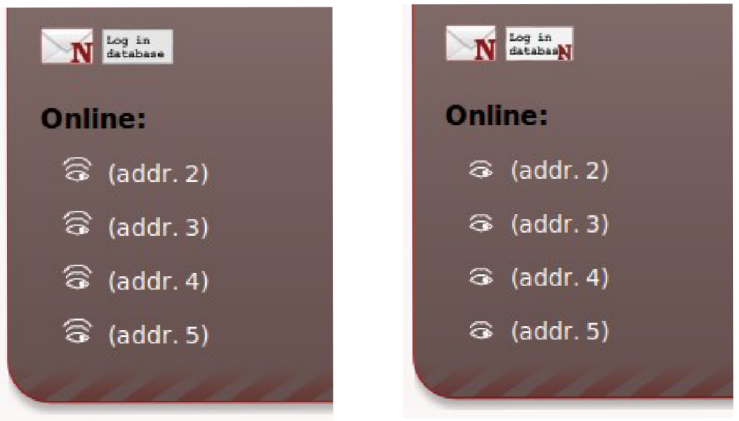

Obrázek 17.: Pohled (`WSTracingAgents.jsp`) s odeslaným požadavkem na sběr všech záznamů, kdy všechny záznamy již přišly.



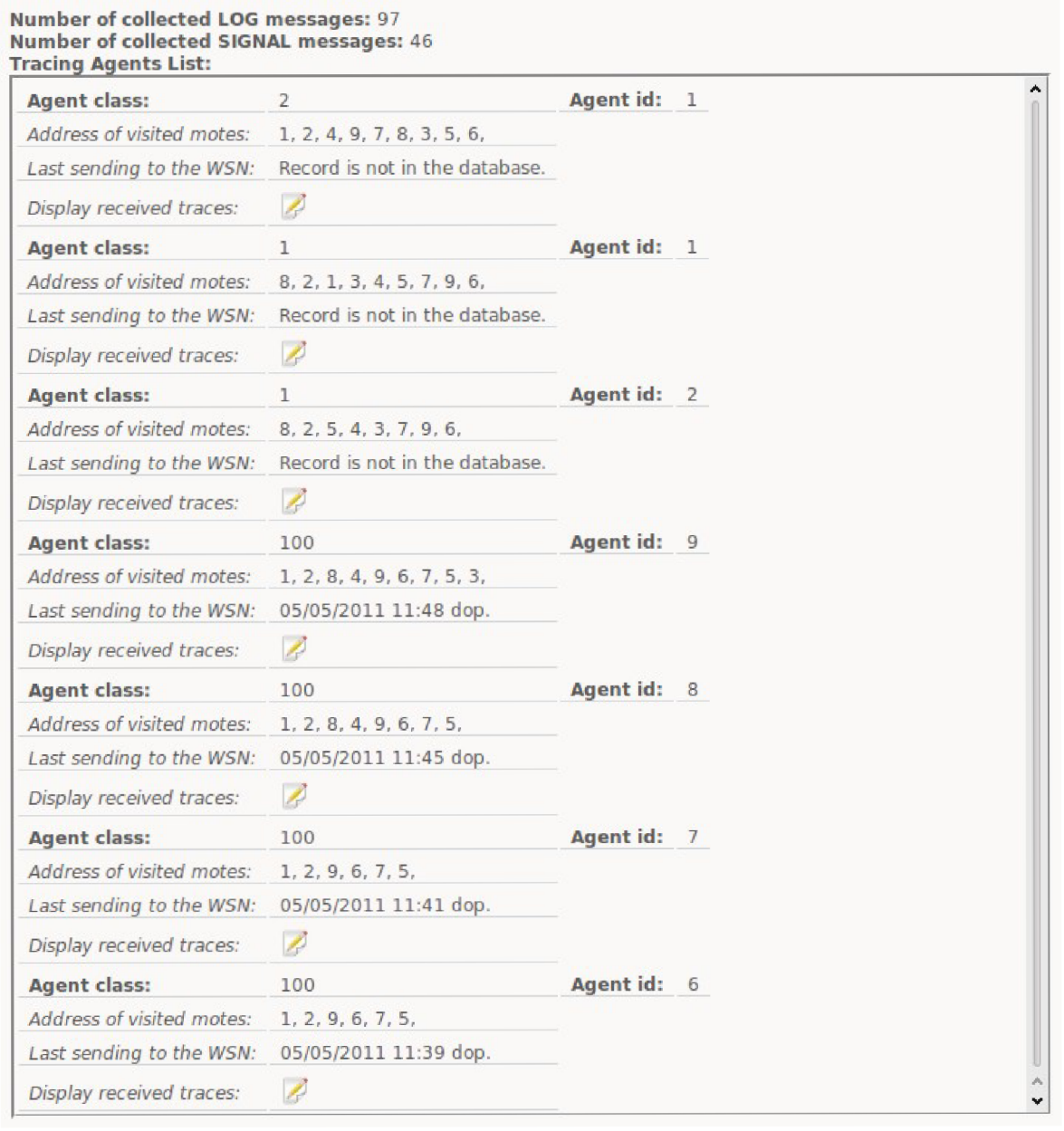

Obrázek 18.: Pohled (`WSTracingAgents.jsp`) s roztříděnými záznamy ze všech uzlů.



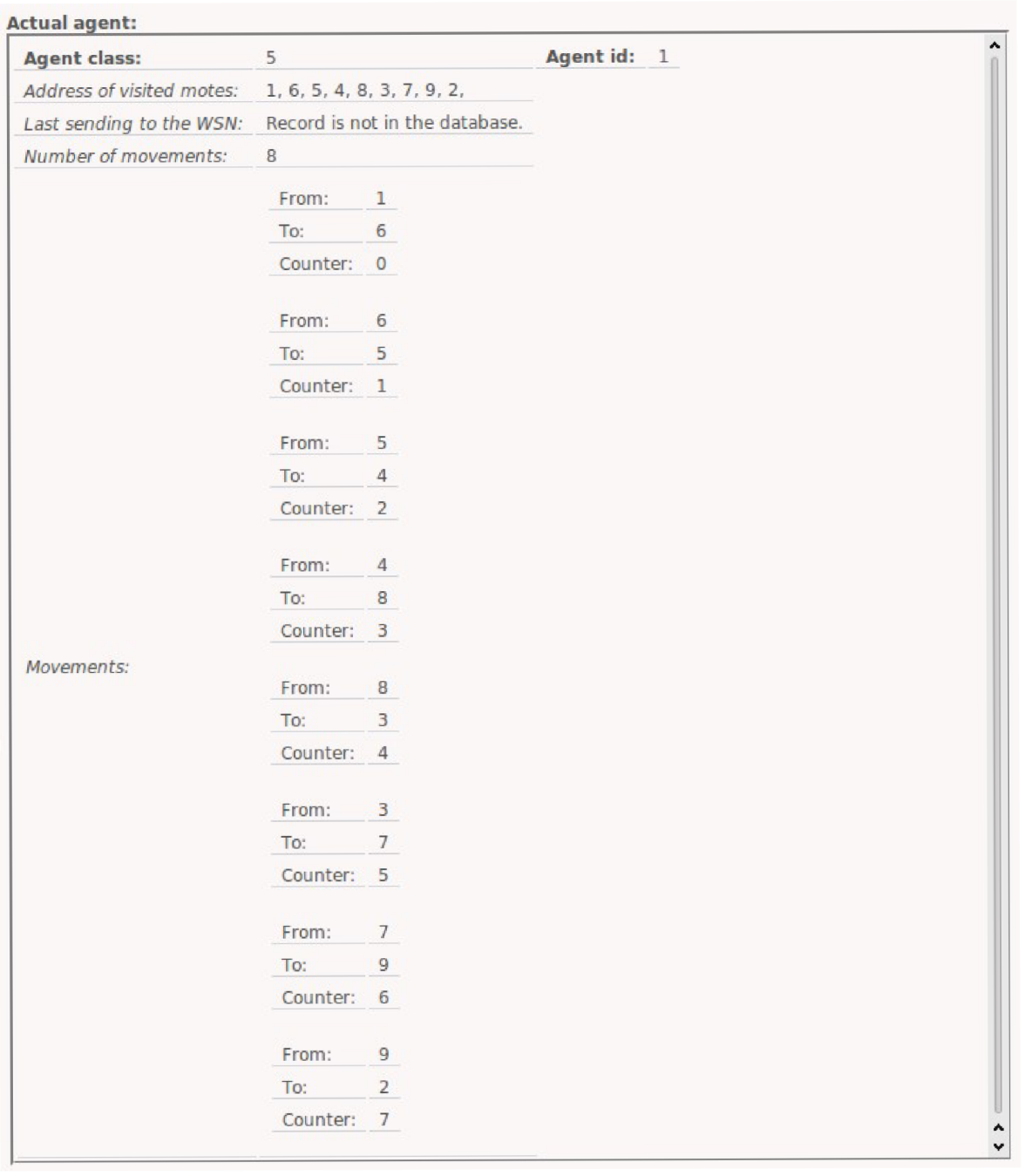

Obrázek 19.: Pohled (`WSDrawTraces.jsp`), který je zobrazen, když uživatel zvolí vykreslení přesunů konkrétního agenta.



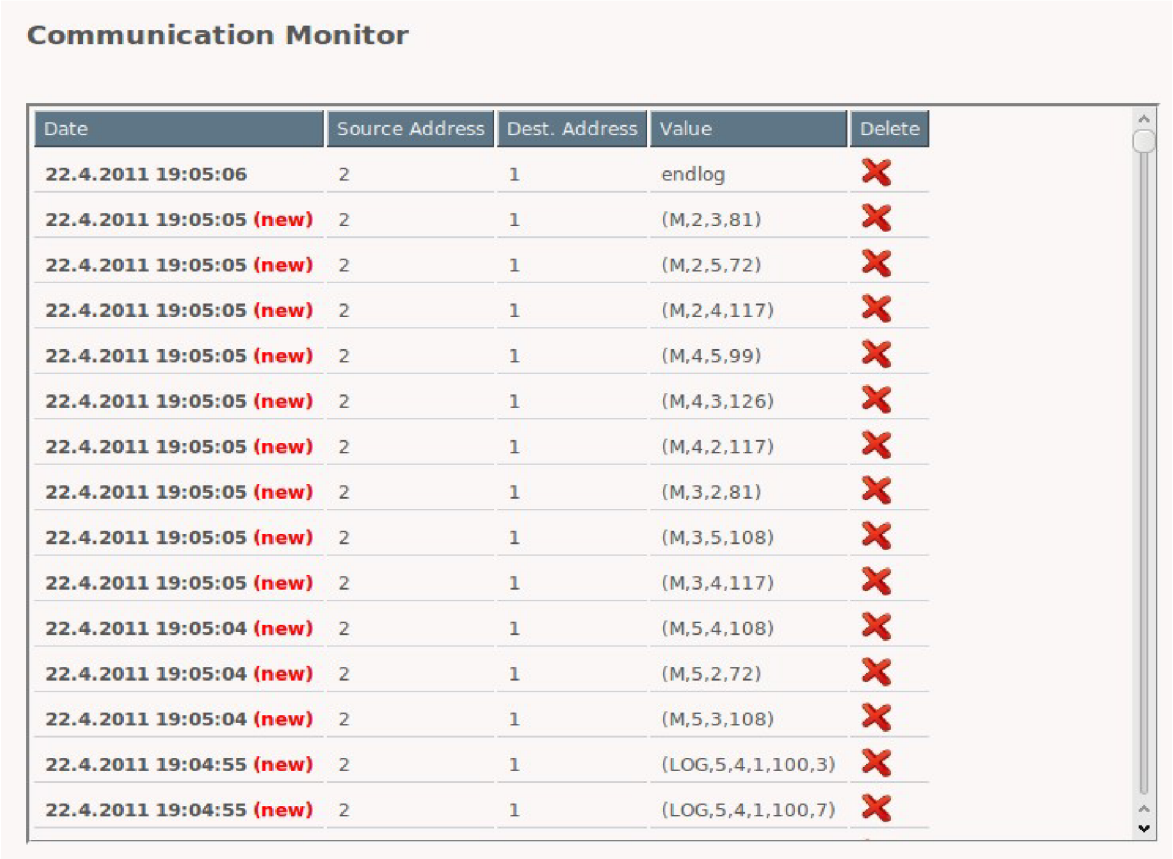

Obrázek 20.: Vlevo aktivní panel (`Right.jsp`) s obecným oznámením, vpravo aktivní panel se závěrečným oznámením.



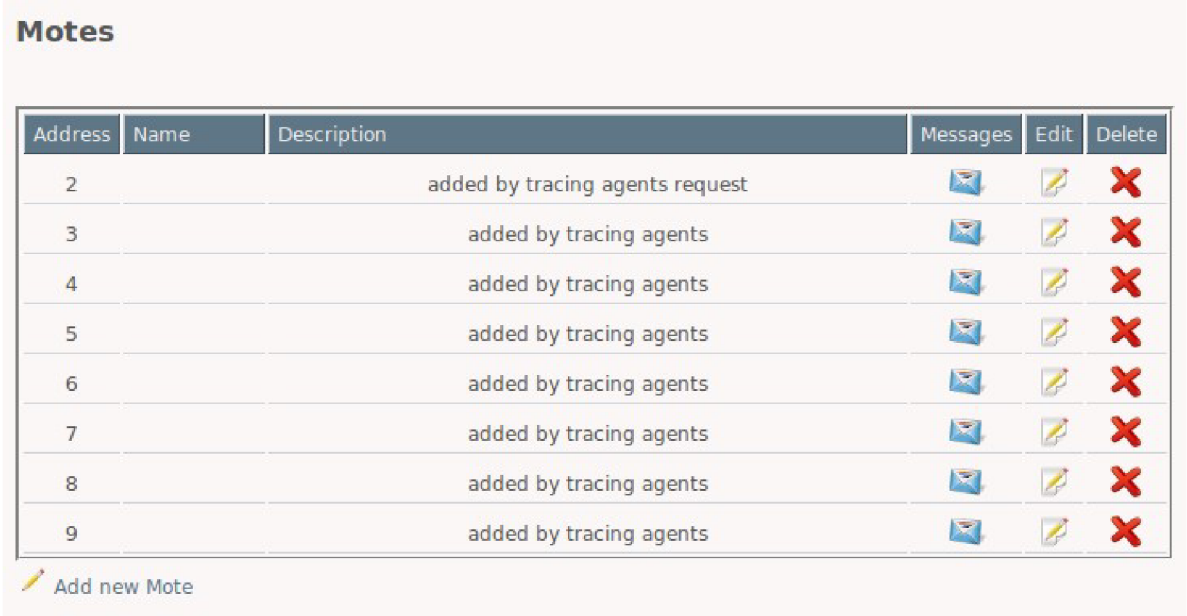

Obrázek 21.: Detail příkladu tabulky sesbíraných a roztříděných záznamů.




Obrázek 22.: Detail příkladu tabulky jednotlivých přesunů agenta vypsaných textově.

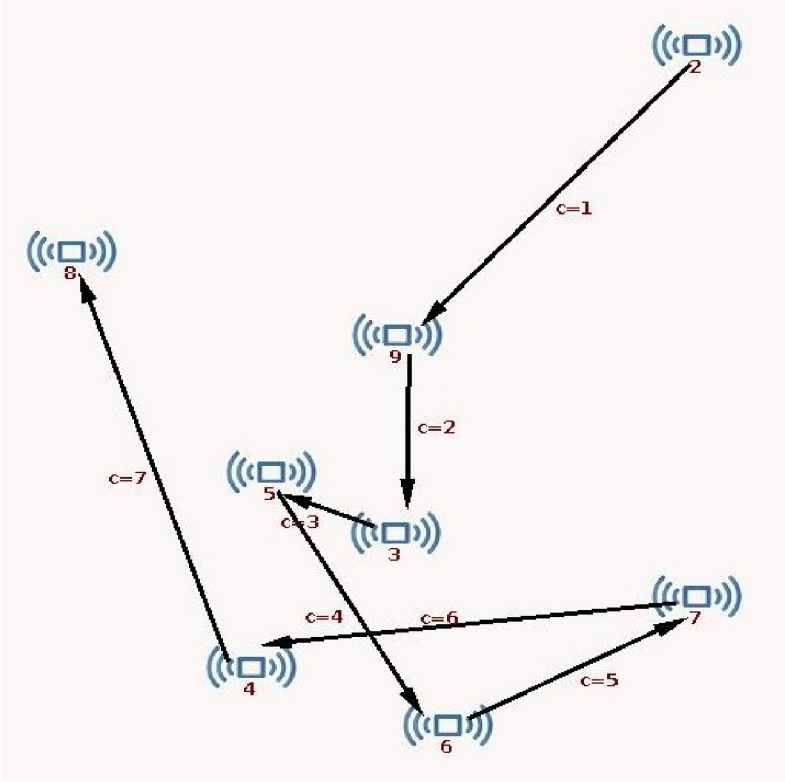



Obrázek 23.: Ukázka, že se zprávy s informacemi po vytvoření požadavku zobrazí i v seznamu příchozích zpráv.

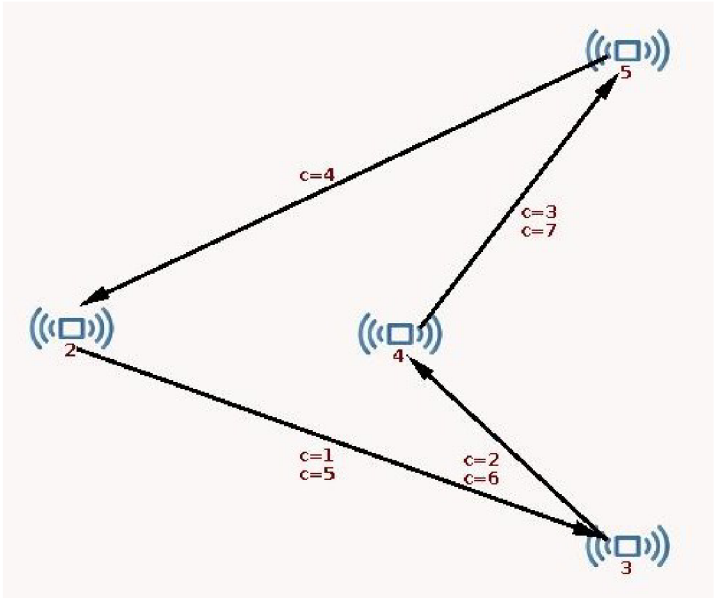



Obrázek 24.: Ukázka, že jsou nově zjištěné uzly automaticky přidány do celkového seznamu všech mote.

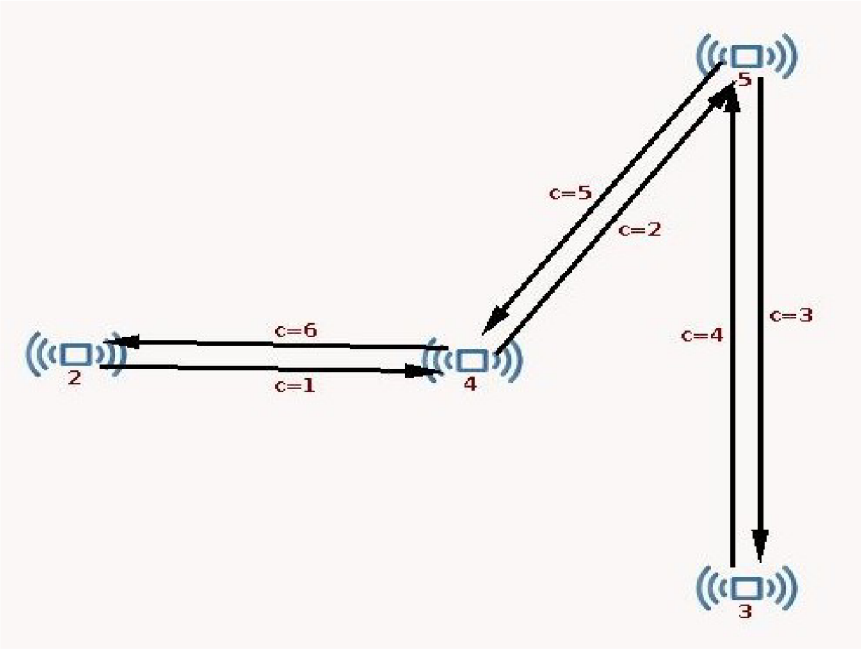
# Příloha 4. Příklady zobrazení vykreslených tras agenta



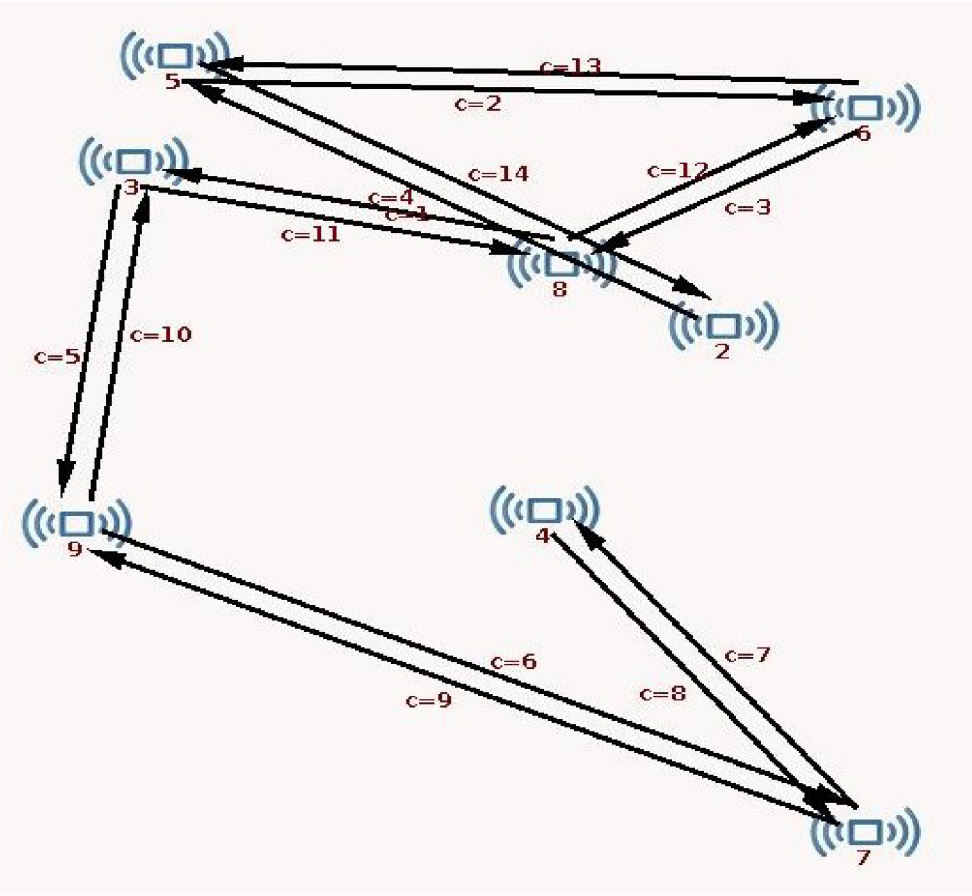

Obrázek 25.: Trasa agenta s pohyby v pořadí uzlů s adresami - 2, 9, 3, 5, 6, 7, 4, 8. Navštíveny byly všechny uzly.



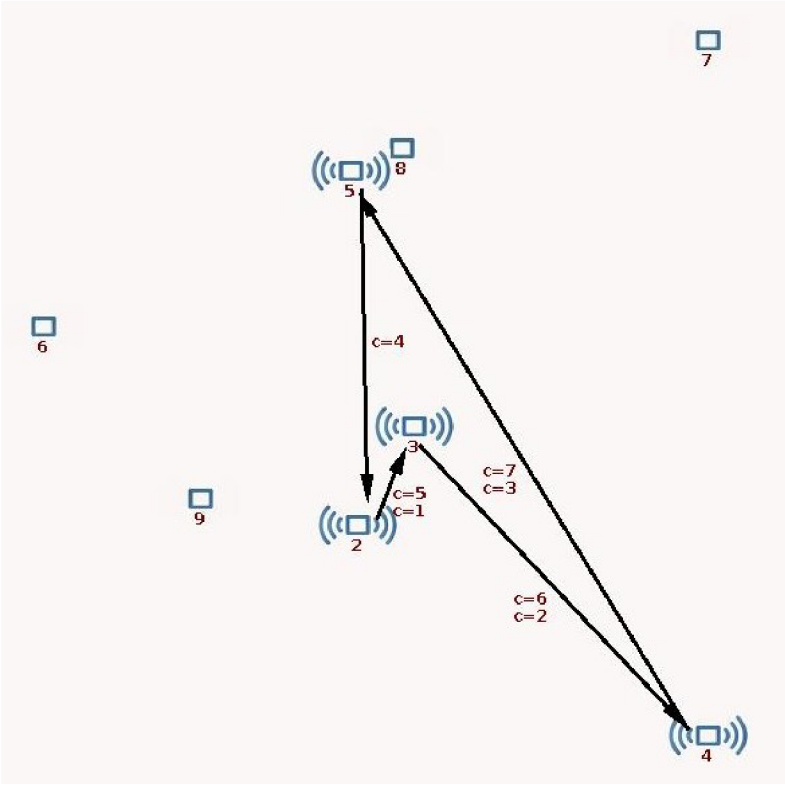

Obrázek 26.: Trasa agenta s pohyby v pořadí uzlů s adresami - 2, 3, 4, 5, 2, 3, 4, 5. Navštíveny byly všechny uzly.



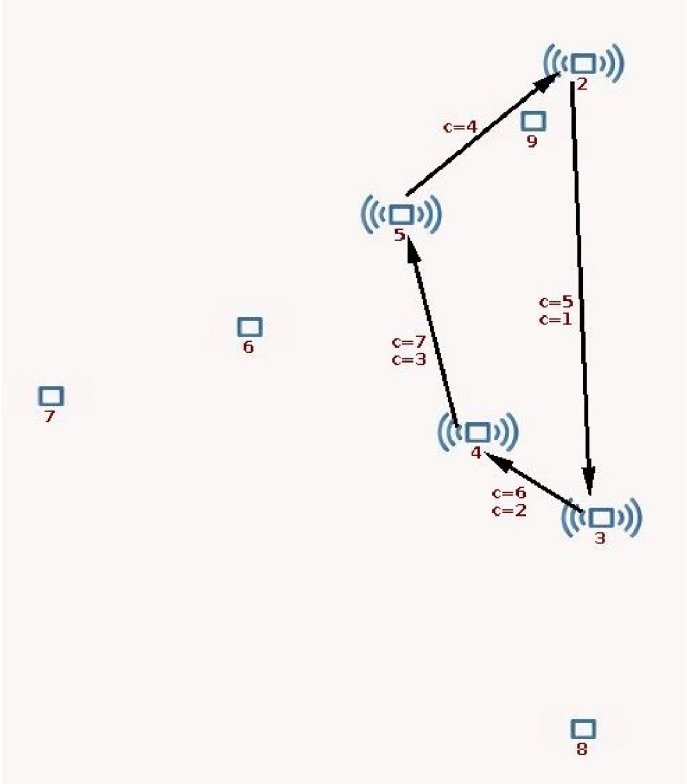

Obrázek 27.: Trasa agenta s pohyby v pořadí uzlů s adresami - 2, 4, 5, 3, 5, 4, 2. Navštíveny byly všechny uzly.




Obrázek 28.: Trasa agenta s pohyby v pořadí uzlů s adresami - 2, 5, 6, 8, 3, 9, 7, 4, 7, 9, 3, 8, 6, 5, 2. Navštíveny byly všechny uzly.




Obrázek 29.: Trasa agenta s pohyby v pořadí uzlů s adresami - 2, 3, 4, 5, 2, 3, 4, 5. Navštíveny byly pouze uzly s adresami 2, 3, 4, 5.




Obrázek 30.: Trasa agenta s pohyby v pořadí uzlů s adresami - 2, 3, 4, 5, 2, 3, 4, 5. Navštíveny byly uzly s adresami 2, 3, 4, 5.