VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

AGENTNÍ PLATFORMA PRO BEZDRÁTOVÉ SENZOROVÉ SÍTĚ

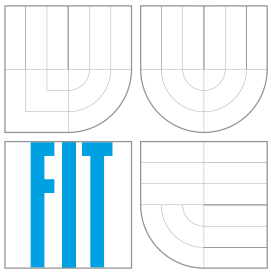
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. STANISLAV LICHÝ

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

AGENTNÍ PLATFORMA PRO BEZDRÁTOVÉ SENZOROVÉ SÍTĚ

AGENT PLATFORM FOR WIRELESS SENSOR NETWORKS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. STANISLAV LICHÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN HORÁČEK

BRNO 2013

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2012/2013

Zadání diplomové práce

Řešitel: **Lichý Stanislav, Bc.**
Obor: Počítačové sítě a komunikace
Téma: **Agentní platforma pro bezdrátové senzorové sítě**
Agent Platform for Wireless Sensor Networks
Kategorie: Vestavěné systémy

Pokyny:

1. Seznamte se s bezdrátovými senzorovými sítěmi, agentní platformou WSageNt pro senzorové uzly MICAz a IRIS a jazykem ALLL použitým pro popis agenta.
2. Navrhněte agentní platformu pro senzorové uzly Sun SPOT tak, aby byla kompatibilní s platformou WSagent pro uzly IRIS a MICAz.
3. Navrženou agentní platformu implementujte.
4. Otestujte na reálné síti běh alespoň dvou typů agentů řešících zvolenou úlohu (cyklický přesun agenta mezi uzly, průchod agenta celou sítí pro zjištění topologie sítě).
5. Zhodnoťte dosažené výsledky a kompatibilitu vaší platformy s platformou WSageNt.
6. Vytvořte poster vhodně prezentující vámi vytvořené dílo.

Literatura:

- Zbořil František, Spáčil Pavel: Automata for Agent Low Level Language Interpretation, In: Proceedings of UKSim 2009, Cambridge, GB, IEEE CS, 2009, s. 6, ISBN 978-0-7695-3593-7.
- Zbořil František, Horáček Jan, Spáčil Pavel: Intelligent Agent Platform and Control Language for Wireless Sensor Networks, In: Proceedings of 3rd EMS, Atény, GR, IEEE CS, 2009, s. 6, ISBN 978-0-7695-3886-0.
- Lin, Hong, Architectural design of multi-agent systems :technologies and techniques, Hershey : Information science reference, 2007, 421 s., ISBN 978-1-599-04108-7.

Při obhajobě semestrální části diplomového projektu je požadováno:

- Bez požadavků.

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: **Horáček Jan, Ing.**, UITS FIT VUT
Datum zadání: 17. září 2012
Datum odevzdání: 22. května 2013

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

Cílem této práce je implementace agentní platformy pro senzorové uzly SunSpot. Čtenář je nejprve seznámen s bezdrátovými senzorovými sítěmi a senzorovými uzly SunSpot. Práce dále popisuje pojmy agent, agentní plaforma a BDI agent. Následuje popis jazyka ALLL a s ním související agentní platformy WSageNt. Zbytek práce se zabývá návrhem a implementací kompatibilní agentní platformy pro senzorové uzly SunSpot. V závěru práce jsou diskutovány výsledky práce, kompatibilita s plaftormou WSageNt a možná rozšíření.

Abstract

The aim of this thesis is to implement the agent platform for SunSpot sensor nodes. Reader is firstly presented with introduction to wireless sensor networks and the SunSpot sensor nodes. The thesis then describes the terms agent, agent platform and BDI agent. Then the description of ALLL language and related agent platform called WSageNt is presented. The rest of thesis deals with the concept and implementation of compatible agent plaform for SunSpot sensor nodes. The final part discusses results of work, compatibility with WSageNt platform and possible extensions.

Klíčová slova

Bezdrátové senzorové sítě, agent, agentní platforma, BDI, SunSpot, ALLL, interpret.

Keywords

Wireless sensor networks, agent, agent platform, BDI, SunSpot, ALLL, interpreter.

Citace

Stanislav Lichý: Agentní platforma pro bezdrátové senzorové sítě, diplomová práce, Brno, FIT VUT v Brně, 2013

Agentní platforma pro bezdrátové senzorové sítě

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jana Horáčka

.....
Stanislav Lichý
2013-05-22

Poděkování

Děkuji tímto panu Ing. Janu Horáčkovi za odborné vedení, cenné rady a konzultace během tvorby práce.

Obsah

1 Úvod **5**

2 Bezdrôtové senzorové siete **7**
2.1 Inteligentný senzor . . . 7
2.1.1 Čidlo . . . 7
2.2 Vlastnosti bezdrôtových senzorových sietí . . . 8
2.3 Príklady využitia bezdrôtových senzorových sietí . . . 9
2.3.1 Armádne aplikácie . . . 9
2.3.2 Monitorovanie životného prostredia . . . 9
2.3.3 Medicínske aplikácie . . . 9
2.3.4 Iné aplikácie . . . 10
2.4 Operačné systémy pre senzorové uzly . . . 10
2.4.1 TinyOS . . . 10
2.4.2 Contiky OS . . . 10
2.4.3 Mantis . . . 11
2.4.4 LiteOS . . . 11

3 Senzorové uzly SunSpot **12**
3.1 Hardware . . . 12
3.2 Inštalácia API . . . 13
3.3 Základňová stanica . . . 13
3.4 Simulačný nástroj Solarium . . . 15
3.5 Vývoj vlastných aplikácií . . . 16
3.5.1 Vývoj aplikácií pre hostiteľský PC . . . 16
3.5.2 Prístup k zdrojom . . . 17
3.5.3 Výstup na LED . . . 18
3.5.4 Komunikácia . . . 18
3.5.5 Prístup k pamäti flash . . . 20

4 Agentný systém **22**
4.1 Praktické rozhodovanie . . . 23
4.2 BDI agent . . . 23

5 Jazyk ALLL **25**
5.1 Dátové štruktúry . . . 25
5.1.1 Zoznam, n-tica . . . 26
5.1.2 Akcie a plány . . . 26
5.1.3 Tabuľka . . . 26

5.1.4 Register . . . 27
5.2 Použitie registrov a anonymných premenných . . . 27
5.3 Agent v jazyku ALLL . . . 27
5.3.1 PlanBase . . . 27
5.3.2 InputBase . . . 27
5.3.3 BeliefBase . . . 28
5.3.4 Aktuálny zámer . . . 28
5.3.5 ID a trieda agenta . . . 28

6 Plaforma WSageNt **29**
6.1 Služby interpreta . . . 29
6.1.1 Nastavenie aktívneho registra . . . 29
6.1.2 Pridávanie do BeliefBase . . . 30
6.1.3 Odoberanie z BeliefBase . . . 30
6.1.4 Test BeliefBase na zadanú n-ticu . . . 30
6.1.5 Test InputBase na správu od zadaného uzla/senzora . . . 31
6.1.6 Priame spustenie . . . 31
6.1.7 Nepriame spustenie . . . 31
6.1.8 Odoslanie správy . . . 32
6.1.9 Zarážka . . . 32
6.2 Služby platformy . . . 32
6.2.1 Zisk prvého prvku n-tice . . . 33
6.2.2 Zisk zvyšku n-tice okrem prvého prvku . . . 33
6.2.3 Uspanie interpreta do príchodu správy . . . 33
6.2.4 Aktivovanie sledovania správ pri behu interpreta . . . 34
6.2.5 Uspanie činnosti interpreta na stanovenú dobu . . . 34
6.2.6 Ukončenie činnosti interpreta . . . 34
6.2.7 Výstup na LED diódy . . . 34
6.2.8 Operácia so vstupmi zo senzorov . . . 35
6.2.9 Detekcia susedných uzlov . . . 35
6.2.10 Pachové stopy agenta . . . 36
6.2.11 Zisk záznamov log-u . . . 36
6.2.12 Zmena identifikácie agenta . . . 37
6.2.13 Operácie so zoznamom . . . 38
6.2.14 Generovanie náhodnej hodnoty . . . 38
6.2.15 Výpočtová operácia platformy . . . 38
6.2.16 Mobilita agenta . . . 40

7 Návrh agentnej platformy **41**
7.1 Parser . . . 42
7.2 Aktuálny plán . . . 42
7.3 N-tice . . . 44
7.4 Registre . . . 45
7.5 Tabuľky . . . 45
7.6 Interpret . . . 45
7.7 Operácie s LED diódami . . . 46
7.8 Komunikácia . . . 46
7.8.1 Komunikačný protokol . . . 46

7.8.2 Typy správ . . . 47
7.8.3 Rozdelenie do tried . . . 48
7.9 Použitie flash pamäte . . . 49
7.9.1 Log agentov . . . 49
7.9.2 Log senzorov . . . 49

8 Popis implementácie **50**
8.1 Práca s n-ticami . . . 50
8.2 Práca s tabuľkou . . . 51
8.3 Registre . . . 51
8.4 Parsovanie zdrojového textu . . . 51
8.5 Zásobník plánov . . . 52
8.6 Výstup na LED . . . 53
8.7 Výpočtové operácie . . . 53
8.8 Rádiová komunikácia . . . 53
8.8.1 Adresovanie uzlov . . . 54
8.8.2 Odosielanie dátových správ . . . 54
8.8.3 Formát ostatných správ . . . 55
8.8.4 Spracovávanie prijatých dát . . . 56
8.8.5 Zisk sily signálu od okolitých uzlov . . . 57
8.8.6 Dotazovanie na pachové stopy agenta . . . 58
8.8.7 Odosielanie agenta . . . 58
8.8.8 Odosielanie správy . . . 58
8.9 Log agentov . . . 58
8.9.1 Sledovanie výskytu agenta . . . 59
8.10 Logovanie vstupov zo senzorov . . . 59
8.11 Intervalové meranie . . . 60
8.12 Uspanie interpreta na požadovanú dobu . . . 60
8.13 Interpretovanie zdrojového textu . . . 60
8.13.1 Služby implementované v interprete . . . 61
8.13.2 Služby platformy . . . 61
8.14 Aplikácia pre hostiteľský PC . . . 61
8.14.1 GUI aplikácie . . . 62

9 Testovanie **64**
9.1 Príklady testovacích agentov . . . 64
9.1.1 Blikanie LED na senzorovom uzle . . . 64
9.1.2 Blikanie LED s využitím bázy znalostí . . . 64
9.1.3 Práca so senzorom teploty . . . 65
9.1.4 Výpočtové operácie . . . 65
9.1.5 Priechod agenta sieťou . . . 66
9.1.6 Zistenie topológie siete . . . 67
9.2 Popis testovania . . . 68
9.2.1 Blikanie s LED . . . 68
9.2.2 Senzor teploty . . . 68
9.2.3 Výpočty . . . 69
9.2.4 Priechod agenta sieťou a zisťovanie topológie . . . 69

10 Záver **70**

A Výstup testu práce so senzorom teploty **74**

B Výstup testu výpočtových operácií **75**

C Výstup testu topológie siete **76**

Kapitola 1

Úvod

Za posledné desaťročia došlo k veľmi silnému rozvoju výpočtových technológií. Postupom času sa tieto technológie stále viac stávajú samozrejmou súčasťou života a prenikajú do takmer všetkých oblastí priemyslu. Znižujúca sa cena výpočtových prostriedkov umožňuje ich nasadenie takmer kdekoľvek.

Jednou z vedeckých oblastí, ktoré využívajú tieto technológie, je umelá inteligencia, resp tzv. inteligentné systémy. Tento pojem je možné chápať ako systémy, ktoré sú schopné riešiť menej, či viac náročné úlohy, pričom cieľom ich nasadenia je nahradenie ľudského uvažovania. Dá sa teda povedať, že inteligentný systém by mal danú úlohu riešiť podobne, ako keby namiesto neho rozhodoval človek.

Z rôznych uhlov pohľadu je možné na inteligentný systém nahliadať ako na systém mechatronický, sociotechnický, robotický, či agentný [7]. U mechatronického systému sa kladie dôraz na integráciu fyzického zariadenia s výpočtovou technikou a elektronikou. V prípade sociotechnického systému nastáva interakcia človeka s výpočtovými prostriedkami v rámci jedného systému. U robotického systému je dôležitá interakcia a autonómne chovanie fyzického systému vo fyzickom prostredí. Najvšeobecnejší prístup je agentný systém. Jeho vlastnosťami sú autonómia, distribuovanosť, komunikácia a situovanosť v prostredí. Prostriedkom pre prácu s takýmto systémom je využitie autonómnej entity, riešiacej danú úlohu v prostredí, zvanú agent. Týmto pojmom sa zaoberá kapitola 4.

Inteligentné systémy pre svoje fungovanie potrebujú čo najlepšie znalosti o svojom okolí. Základným spôsobom je napojenie senzorov a snímačov priamo na takýto systém. Pri väčšej komplexnosti systému a množstve senzorov je ale vhodné využiť špecializované prostriedky na tento účel. Práve v týchto prípadoch je vhodné použitie bezdrôtových senzorových sietí zložených zo senzorových uzlov. Jedná sa o malé vstavané zariadenia, ktoré v sebe integrujú senzory pre snímanie okolia, výpočtové prostriedky pre ich spracovanie na určitej úrovni, a komunikačný modul pre odosielanie týchto informácií pre ďalšie spracovanie.

Táto práca sa zaoberá spojením výhod agentných systémov a bezdrôtových senzorových sietí. Cieľom je vytvorenie agentnej platformy pre senzorové uzly, konkrétne uzly SunSpot (viď kapitola 3). Pokiaľ máme takúto platformu, je možné analyzovať prostredie priamo pomocou autonómneho agenta, ktorý sa presúva po sieti zloženej zo senzorových uzlov a rieši zvolenú úlohu. Príkladom môže byť zber informácií zo senzorov, ich analýza a vykonávanie príslušných opatrení. Prípadne potom zisťovanie stavu siete, umiestnenia jednotlivých senzorových uzlov a sledovanie zmien ich topológie.

Pre účel programovania chovania agentov bol na fakulte FIT VUT vyvinutý jazyk ALLL, ktorého podrobnejší popis je možné nájsť v kapitole 5. Tento jazyk bol navrhnutý ako jazyk nízkej úrovne abstrakcie, a špeciálne sa hodí pre prostredie senzorových uzlov. Umožňuje

napríklad priamy prístup k hodnotám senzorov na danom uzle. Jeho výhodou je jednoduchosť a možnosť tvorenia krátkeho kódu, ktorý nezaberá veľa miesta v operačnej pamäti uzla. Senzorové uzly sú totiž zariadenia, ktoré sú pomerne skromne vybavené priestorom pre ukladanie množstva dát. Pre tento jazyk je na fakulte v rámci bakalárskych a diplomových prác vyvíjaná agentná platforma WSageNt. Jedná sa v podstate o interpret jazyka ALLL, ktorý je určený pre senzorové uzly IRIS a MicaZ. Cieľom tejto práce je implementácia kompatibilnej platformy pre senzorové uzly SunSpot, ktorá umožní použitie agentov napísaných v jazyku ALLL na týchto senzorových uzloch.

Kapitola 2

Bezdrôtové senzorové siete

Bezdrôtové senzorové siete slúžia na zber informácií o prostredí, v ktorom sa nachádzajú. Tieto informácie sa potom môžu ďalej využívať pre rozhodovanie inteligentného systému o vykonávaní akcií, alebo napríklad na vytváranie štatistík a získavanie poznatkov o danom prostredí.

2.1 Inteligentný senzor

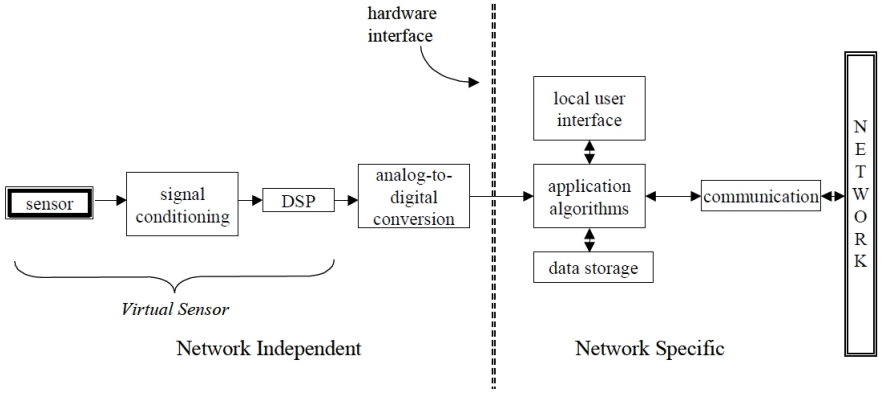
Pre problematiku senzorových sietí je jedným zo základných pojmov inteligentný senzor. Inteligentný senzor je zariadenie, ktoré poskytuje ďalšie funkcie okrem generovania korektnej reprezentácie snímanej veličiny [12]. Medzi ne patrí spracovanie signálu a vykonávanie akcií na základe nameraných veličín. Cieľom inteligentného senzora je posun inteligentnej časti bližšie k bodu, v ktorom je veličina meraná. Všeobecný model inteligentného senzora je možné vidieť na obrázku 2.1. Model sa delí na 2 časti. Prvou je modul, ktorý sa stará o snímanie a spracovávanie dát senzora, vrátane A/D prevodu[1]. Druhý modul potom slúži na spracovávanie digitálnych dát. Sem patrí rozhodovanie pre vykonané akcie na základe nameraných dát, napríklad upozorňovanie na kritické hodnoty meraných veličín. Ďalej sa táto časť stará o ukladanie nameraných hodnôt a môže obsahovať jednoduché rozhranie pre komunikáciu s užívateľom. Tento modul potom slúži tiež pre odosielanie nameraných dát do siete. Táto časť je preto závislá na konkrétnom použitom type siete. Pokiaľ je komunikačná časť bezdrôtová, je možné nazvať konkrétnu implementáciu inteligentného senzora ako bezdrôtový senzorový uzol.

2.1.1 Čidlo

Základným prvkom pre snímanie dát sú čidlá, alebo snímače. Vo všeobecnosti je možné povedať, že sa jedná o zariadenie, ktoré mení jeden typ energie na iný [12]. Typický výstup snímača je elektrický signál, ktorý sa ďalej spracúva. Na snímanie sa používajú rôzne fyzikálne vlastnosti.

Mechanické čidlá na snímanie potrebujú priamy fyzický kontakt. Sem patria čidlá, u ktorých sa využíva napríklad piezoelektrický efekt, piezorezistívny efekt, alebo napríklad kapacitné čidlá. Magnetické a elektromagnetické senzory sú potom založené na Hallovom efekte, alebo magnetorezistívnom efekte. Tepelné čidlá sú založené napríklad na termomechanickom, alebo termorezistívnom efekte. Ďalším typom čidiel sú optické čidlá, založené

[1] Prevod analógového signálu na digitálny



Obr. 2.1: Všeobecný model inteligentného senzora [12].

na fotoelektrickom efekte. Inými typmi sú potom napríklad chemické a biologické čidlá, alebo napríklad akustické senzory. Viac o jednotlivých typoch je možné nájsť napríklad v [12].

2.2 Vlastnosti bezdrôtových senzorových sietí

Bezdrôtová senzorová sieť je typicky zložená z veľkého počtu senzorových uzlov. Tieto sú nainštalované priamo v priestore, ktorý sledujú, typicky pomerne nahusto. Výhodou je, že pozícia senzorových uzlov nemusí byť predom známa [2]. Toto umožňuje ich náhodné umiestňovanie v ťažko dostupných oblastiach. Na druhú stranu z toho vyplýva, že komunikačné protokoly pre senzorové siete musia obsahovať schopnosti samoorganizácie týchto sietí. Ako bolo spomenuté, senzorové uzly obsahujú tiež výpočtové prostriedky, ktoré sa starajú čiastočne aj o spracovávanie dát a odosielanie už predspracovaných dát.

Realizácia aplikácie senzorových sietí vyžaduje techniky ad-hoc sietí[2]. Hoci existuje dostatok algoritmov pre tradičné bezdrôtové ad-hoc siete, tieto nie sú dobre prispôsobené požiadavkám bezdrôtových senzorových sietí. Špecifiká senzorových sietí sú nasledovné [2]:

- počet prvkov v sieti je násobne vyšší než u klasických sietí (napr. Wi-Fi, či Bluetooth).
- senzorové uzly sú inštalované omnoho hustejšie
- uzly sú náchylné k chybám a strate dostupnosti (napríklad dôjdu napájacie batérie uzla)
- topológia siete sa mení veľmi často
- senzorové uzly často využívajú komunikáciu typu broadcast[3]
- uzly sú silne limitované z hľadiska výpočtovej sily, kapacity pamäte a napájacích možností
- uzly nemusia byť identifikovateľné globálne

[2]Decentralizovaný typ bezdrôtovej siete, ktorý nie je závislý na predom existujúcej infraštruktúre. Každý uzol siete sa podieľa na smerovaní preposielaním dát iným uzlom. Smerovanie sa deje dynamicky na základe aktuálnej konektivity uzlov

[3]Odosielanie dát od jedného uzla ku všetkých ostatným

Dôležitým faktorom pri používaní senzorových sietí je výdrž batérií senzorových uzlov. Preto používané protokoly sa musia sústrediť na vyladenie spotreby viac, než na kvalitu komunikačnej linky. Znižovanie spotreby pritom ako následok má napríklad znižovanie prenosovej rýchlosti, či predlžovanie odozvy pri komunikácii, avšak tieto parametre u senzorových sietí nie sú kritické. Keďže vzdialenosti medzi uzlami siete bývajú relatívne malé, preferovaný typ komunikácie je multi-hop[4]. Tým je umožnené znižovanie vysielacieho výkonu jednotlivých uzlov s cieľom znižovania nárokov na napájanie. Rovnako to má kladný vplyv na celkovú kvalitu prenosu na dlhšie vzdialenosti [2].

Pre senzorové siete existujú vlastné štandardy pre komunikáciu a smerovanie v týchto sieťach. Najpoužívanejším dnes je štandard IEEE 802.15.4. Tento definuje fyzickú vrstvu a vrstvu riadenia prístupu k médiu. Tento štandard potom dopĺňa špecifikácia ZigBee, ktorá definuje sieťovú a aplikačnú vrstvu. Pre komunikáciu sa využívajú frekvenčné pásma 868 MHz pre Európu, 915 MHz pre USA a 2,4 GHz. Prenosová rýchlosť sietí je od 20 kb/s[5] do 250 kb/s.

2.3 Príklady využitia bezdrôtových senzorových sietí

Bezdrôtové senzorové siete majú široké pole možných aplikácií, od armády až po biomedicínske využitie. Využitie týchto sietí je prakticky neobmedzené a je vhodné ich nasadiť všade tam, kde je potrebný zber informácií o meraných hodnotách fyzikálnych veličín.

2.3.1 Armádne aplikácie

V armádnych aplikáciách sa senzorové siete používajú napríklad na monitorovanie vlastných aj nepriateľských zložiek, zameriavanie cieľov, alebo napríklad na detekciu útokov nukleárnymi, chemickými, či biologickými zbraňami [13]. Senzorovými uzlami sa vybavujú osoby, ale aj napríklad vojenské vozidlá, čím je možné detailné monitorovanie ich aktuálnej kondície. Uzly tiež môžu slúžiť na detekciu umiestnenia nepriateľského personálu a vozidiel, a upozorňovanie na dianie v monitorovanej oblasti. Pomáhajú tak predvídať včas možné hroziace nebezpečenstvo.

2.3.2 Monitorovanie životného prostredia

V tejto oblasti sa senzorové uzly využívajú na zber dlhodobých dát a štatistík o monitorovanom prostredí [13]. Napríklad sa môže jednať o monitorovanie správania skúmaného druhu živočíchov v prírode bez rušivej prítomnosti človeka. Iným príkladom je využitie na meteorologický, geofyzický výskum, alebo monitorovanie znečistenia prostredia [2]. Dôležité je napríklad využitie na detekciu prírodných katastrof, ako napríklad lesných požiarov, záplav, či zemetrasení.

2.3.3 Medicínske aplikácie

U medicínskych aplikácií je možné využitie senzorových uzlov na sledovanie vitálneho stavu pacientov (napríklad krvný tlak, tep), pričom títo sa môžu voľne pohybovať [2]. Pri detekcii problému tak môže byť upozornená lekárska služba o vážnosti stavu pacienta a rovnako prípadne o jeho polohe. Ako príklad možno uviesť monitorovanie hladiny cukru u diabetikov

[4] komunikácia smerovaná cez viacero uzlov

[5] kilobitov sa sekundu

[13]. Výhodou u tohto prístupu je, že jedinec je monitorovaný aj v spánku a pri probléme je okamžite na tento upozornený. Ďalej je využitie senzorových uzlov vhodné na vytváranie dlhodobých štatistík o pacientovom stave, čo napomáha k lepšej diagnostike tohto pacienta.

2.3.4 Iné aplikácie

Senzorové siete nachádzajú využitie napríklad aj na monitorovanie prostredia budov. Dáta senzorov sa pritom využívajú na ovládanie vykurovania či klimatizácie [2]. Ďalej sa môžu využívať na monitorovanie stavu ovzdušia (hladiny oxidu uhličitého) v budove a s tým spojené ovládanie vetrania miestností. Iným príkladom je využitie pri kontrole výrobných procesov v priemysle a s tým spojenou automatizáciou výroby, prevenciu havárie, sledovanie výskytu chemických prvkov a podobne.

2.4 Operačné systémy pre senzorové uzly

Bežné operačné systémy majú za úlohu správu zdrojov pre komplexné systémy. V typických systémoch sú tieto zdroje procesor, pamäť, časovače, disky, sieťové rozhrania, periférie atď. Programátorovi sú potom tieto zdroje k dispozícii cez systémové volania OS. Keďže senzorové uzly majú obmedzený hardware, operačné systémy pre tieto zariadenia sa tiež líšia od tých bežných [3].

2.4.1 TinyOS

Jedná sa voľne šíriteľný operačný systém, založený na komponentoch, určený priamo pre senzorové uzly. Podporuje súbežné programy. Samotné jadro systému v pamäti zaberá iba 400 bajtov. Jednotlivé komponenty obsahujú okrem iného sieťové protokoly a ovládače senzorov. Systém je založený na monolytickom[6] jadre.

Komponenty majú tri časti, príkazy, udalosti a úlohy. Pre interakciu medzi komponentmi slúžia príkazy (commands) a udalosti (events). Úlohy (tasks) potom bežia vo vnútri daného komponentu. Príkaz slúži ako žiadosť na vykonanie nejakej služby komponentu, udalosť potom signalizuje ukončenie danej úlohy. Jednotlivé komponenty sa potom spájajú do jedného celku pre vytvorenie konkrétnej aplikácie pre daný senzorový uzol. Aplikácie sa programujú v jazyku `nesC`. Podporované senzorové uzly sú napríklad Mica, Mica2, Micaz, Telos a Tmote. Viac informácií je možné nájsť v [10] a [3].

2.4.2 Contiky OS

Jedná sa o open-source operačný systém napísaný v jakyzu C. Je postavený na udalosťami riadenom jadre. Veľkostne zaberá na uzle okolo 2KB RAM 40KB ROM[7] [3]. Plná inštalácia tohto systému zahŕňa jadro s podporou súbežného spracovania viacerých úloh, podporu protokolu TCP/IP, IPv6. Ďalej obsahuje grafické užívateľské rozhranie, či internetový prehliadač.

[6] Druh jadra operačného systému, u ktorého všetok kód beží v jednom pamäťovom priestore, tzv. jadernom priestore

[7] Read Only Memory – pamäť iba pre čítanie. Jej vlastnosťou je tiež to, že ne nie je závislá na napájaní.

2.4.3 Mantis

Mantis je multivláknový operačný systém pre bezdrôtové senzorové siete. Je nenáročný a energeticky efektívny. Jadro systému spolu s plánovačom zaberá v pamäti 500 bajtov [3]. Jeho výhodou je tiež prenosnosť na mnoho platforiem. Aplikácie napísané pre tento systém môžu byť testované aj na klasickom PC. Podporuje tiež vzdialenú správu senzorových uzlov. Je napísaný v jazyku C a v tomto jazyku sa programujú aj aplikácie pre tento systém.

2.4.4 LiteOS

Jedná sa o systém, ktorý má za cieľ byť podobný systému Unix [3]. S tým súvisí prístup k programovaniu na tomto systéme. Umožňuje programovanie založené na vláknach. Podporuje tiež objektovo orientované programovanie pomocou jazyka LiteC++. Delí sa na 3 hlavné časti, LiteShell, LiteFS a jadro. LiteShell poskytuje interpret príkazového riadku podobný tomu v operačnom systéme Unix. LiteFS je súborový systém, ktorý okrem práce s flash pamäťou samotného uzla má aj iné funkcie. Do súborového systému totiž pripája ako súbory aj okolité uzly vzdialené na jeden skok [3]. Užívateľ potom môže pomocou tohto systému zapisovať a čítať súbory z okolitých senzorových uzlov. Jadro podporuje multithreading a umožňuje napríklad tiež registráciu obslužných funkcií pre obsluhu udalostí.

Kapitola 3

Senzorové uzly SunSpot

Senzorový uzol `SunSpot` pochádza od firmy `Sun` (aktuálne vlastnenou firmou `Oracle`). Jedná sa o pomerne kompaktné zariadenie so vstavanou batériou, nabíjateľnou priamo z USB portu. Má v sebe vstavaných niekoľko senzorov, s tým, že je rozšíriteľný o ďalšie vstupné zariadenia pomocou prídavného konektora na základnej doske.

Obr. 3.1: Senzorový uzol SunSpot.

3.1 Hardware

Senzorové uzly SunSpot, ktoré boli v práci použité, sú založené na procesore Atmel AT91RM9200. Jedná sa integrovaný obvod SOC[1]. Je založený na architektúre ARM [17]. Jeho maximálna frekvencia je 180Mhz. Procesor obsahuje 16KB dátovej a inštrukčnej cache pamäte.

Čo sa týka pamäte, sú uzly vybavené 512KB RAM pamäte. Na ukladanie dát väčšieho objemu a perzistentných dát je možné použiť 4MB vstavanej flash pamäte [17]. Najnovšia revízia uzlov (ktorá ale nebola k dispozícii pri tvorbe práce) je potom vybavená 1MB RAM pamäte a 8MB vstavanej flash pamäte.

Systém je napájaný vstavanou 720maH Li-ion batériou s výdržou niekoľko hodín až dní, v závislosti na pracovnej záťaži senzorového uzla. Na užívateľské výstupy je možné použiť 8 farebných LED diód umiestnených v rade vedľa seba. U každej z nich je možné nastaviť

[1] System on Chip - jednočipový systém

hodnoty jednotlivých farebných zložiek RGB[2] modelu, a teda vytvoriť akúkoľvek farbu na danom LED segmente.

Platforma obsahuje aj 2 tlačítka na užívateľské vstupy. Reakcia na stlačenie tlačidiel je programovateľná pomocou priloženého API. Ďalej je možné použiť prídavné piny na pripojenie externých zariadení. Konkrétne 4 analógové vstupné piny (spracovávajú sa pomocou A/D prevodníka), 4 výstupné piny s prúdom do 125mA (napájané externým zdrojom), a 4 GPIO piny[3] [17].

Na samotné meranie fyzikálnych veličín slúžia vstavané senzory. Jedná sa o teplotný senzor s presnosťou 0,25 stupňa Celzia. Ďalej je platforma vybavená 3-osovým akcelerometrom a svetelným senzorom. Na komunikáciu s okolím slúži rádiový modul štandardu IEEE 802.15.4.

Každý senzorový uzol má svoju jedinečnú adresu, danú výrobcom. Jedná sa o 64-bitové číslo, udávané ako štyri skupiny hexadecimálnych čísel, oddelených bodkami. Adresa tak pripomína klasickú sieťovú MAC adresu. Pomocou tejto adresy je možné s uzlami komunikovať, alebo ich vzdialene spravovať.

3.2 Inštalácia API

Predpokladom pre inštaláciu API[4] na programovanie uzlov SunSpot je nainštalované JDK[5] pre jazyk Java, poskytované zdarma firmou Sun. Ďalšou požiadavkou je inštalácia nástroja `Ant` pre kompiláciu zdrojových kódov. Pre správnu detekciu pripojených senzorových uzlov pod systémom Linux/Unix je tiež potrebný balík `hal`, dostupný v bežných distribúciách operačného systému Linux. Podporovanými operačnými systémami sú Windows, Linux, Solaris a Mac OS X [19].

Vhodným nástrojom pre vývoj, odporúčaným samotným inštalačným programom, je tiež vývojový nástroj Netbeans. Po inštalácii potrebných balíkov je možné nainštalovať aktuálne verzie API k uzlom SunSpot. Na to slúži inštalačná aplikácia `SPOTManager.jnlp`. Táto sa postará o stiahnutie všetkého potrebného z internetu, inštaláciu knižníc do zvoleného adresára, a prípadne nastavenie potrebných systémových premenných.

Tento nástroj súčasne okrem inštalácie slúži aj na správu senzorových uzlov, upgrade knižníc API v uzloch, či inštaláciu rôznych verzií API. Pomocou neho je možné tiež zobrazovať stav uzla, ktoré aplikácie sú v ňom nahrané, či spúšťanie zdieľanej základňovej stanice (bude vysvetlené ďalej).

3.3 Základňová stanica

Vývojový kit SunSpot obsahuje okrem samotných senzorových uzlov aj tzv. základňovú stanicu, angl. Basestation. Pomocou nej je možná rádiová komunikácia hostiteľského PC so senzorovými uzlami. Hardware základňovej stanice sa podobá tomu v senzorových uzloch. Výnimkou ale je, že neobsahuje batériu, ani senzory. Slúži ako komunikačný most. Základňová stanica môže operovať v dvoch módoch [19].

Prvým z nich je dedikovaný mód. V tomto prípade sa základňová stanica chová štandardne a je dostupná cez svoju hardware-ovú adresu svojmu okoliu pre komunikáciu. Tento

[2] Red-Green-Blue – červená-zelená-modrá
[3] General Purpose Input/Output – vstupné, alebo výstupné piny
[4] Application Programming Interface – rozhranie pre programovanie aplikácií
[5] Java Development Kit – vývojový kit pre jazyk Java

Obr. 3.2: Nástroj SunSpot Manager.

mód sa hodí pri priamej komunikácii základňovej stanice s fyzickými uzlami.

Druhou možnosťou je použitie zdieľaného módu. V tomto prípade má základňová stanica systémom pridelenú adresu (ktorá sa nezhoduje s fyzickou adresou) a fyzickým uzlom je dostupná cez dva skoky [19], podobne, ako keby smerovanie komunikácie prebiehalo cez nejaký ďalší uzol. V tomto móde je možné základňovú stanicu využívať s fyzickými, aj virtuálnymi uzlami súčasne. Nevýhodou ale je, že sa nedá vopred zistiť, aká bude adresa základňovej stanice, čo je nevýhodné pre komunikáciu. Ďalšia výhoda však spočíva v tom, že základňovú stanicu v PC môže súčasne používať viac procesov. Napríklad nástroj Solarium a súčasne vlastná užívateľom naprogramovaná desktopová aplikácia.

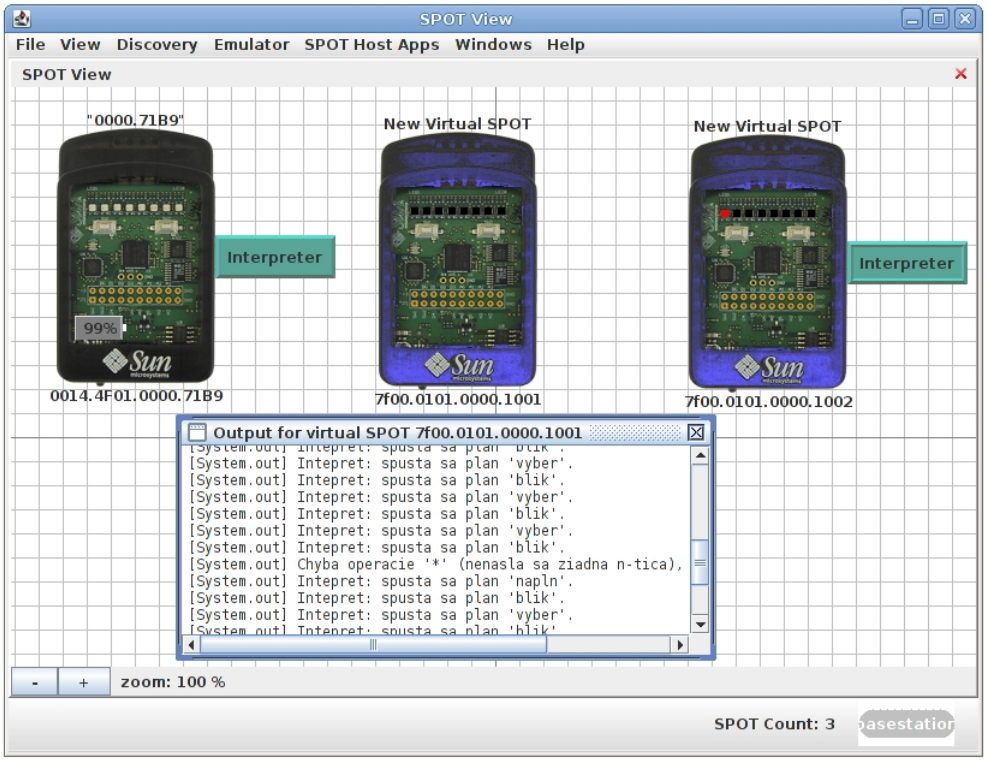
Nastavenie, či sa základňová stanica automaticky používa v zdieľanom alebo dedikovanom móde je možné pomocou súboru `.sunspot.properties`, ktorý sa nachádza v domovskom adresári užívateľa, ktorý inštaloval API na danom PC. V ňom je potrebné pridať riadok nastavujúci premennú `basestation.shared`, napríklad:

```
basestation.shared=true
```

Ak je táto premenná `true`, automaticky sa základňová stanica používa v zdieľanom móde. Toto nastavenie sa využije napríklad pri spúšťaní nástroja `Solarium` (viď časť 3.4), alebo aplikácie pre PC používajúce základňovú stanicu, viď časť 3.5.1.

3.4 Simulačný nástroj Solarium

Pri inštalácii API k platforme SunSpot sa v jeho najnovšej verzii súčasne nainštaluje aj nástroj Solarium. Tento umožňuje simulovať senzorové uzly a komunikáciu medzi nimi. Jedná sa o nástroj s grafickým užívateľským rozhraním, zobrazujúci pracovnú plochu, kde je možné vytvoriť potrebné množstvo virtuálnych uzlov (náhľad je možné vidieť na obrázku 3.3). Simulovaná je takmer kompletná funkcionalita fyzických uzlov.



Obr. 3.3: Simulačný nástroj Solarium. Modré zariadenia sú virtuálne, čierne je fyzický SunSpot.

Napríklad je možné simulovať jednotlivé LED diódy, či klikať na tlačítka simulujúce užívateľské vstupy. Ďalej je možné zobrazenie senzorového panela, kde sa dajú meniť hodnoty simulovaných senzorov teploty, intenzity svetla, či akcelerometra [18]. Pre ladenie aplikácií je tiež výhodná možnosť zobrazenia výstupov naprogramovanej aplikácie na štandardný výstup.

Každý virtuálny SunSpot dostane v simulačnom nástroji pridelenú virtuálnu adresu, cez ktorú sa dá simulovať komunikácia s okolím. Simulovaná komunikácia sa pritom navonok nijako nelíši od skutočnej. Pri použití základňovej stanice v zdieľanom móde je navyše možné kombinovať fyzické a virtuálne SunSpoty a komunikovať medzi nimi. Základňová stanica je v tomto prípade transparentná k danej komunikácii a iba preposiela pakety medzi fyzickými a virtuálnymi uzlami.

Pomocou základňovej stanice je v nástroji možná aj správa fyzických uzlov [18]. Tieto sa zobrazujú na pracovnej ploche nástroja Solarium spolu s virtuálnymi uzlami. Rovnako, ako u virtuálnych, je možné do nich nahrávať skompilované programy a spúšťať ich. Jednou z funkcií je tiež zobrazenie aktuálneho stavu fyzického uzla, napríklad stav vybitia batérie,

či množstvo voľnej RAM pamäte a pod.

3.5 Vývoj vlastných aplikácií

Programovacím jazykom pre senzorové uzly SunSpot je Java Micro Edition. Na vývoj je potrebné mať nainštalované API a nástroje uvedené v časti 3.2. Na preklad kódu pre senzorové uzly je potrebné použitie nástroja Ant. Nutnou súčasťou vlastnej aplikácie tak je súbor `build.xml`, ktorý obsahuje nastavenia, pomocou ktorých sa aplikácia prekladá. Tento typicky odkazuje na ďalšie súbory, ktoré sú priamo súčasťou SunSpot API. Samotná aplikácia na najvyššej úrovni vždy musí obsahovať objekt, ktorý dedí z triedy `MIDlet`[6]. Hlavnou metódou aplikácie je potom `startApp`, ktorá musí byť implementovaná. Táto sa ako prvá spustí pri behu naprogramovanej aplikácie.

Na vytvorenie vlastnej aplikácie je možné použiť dodávanú šablónu priamo pre programovacie prostredie Netbeans. Táto sa nachádza v adresári s nainštalovaným API, v podadresári `Demos/CodeSamples/SunSpotApplicationTemplate`. Mimo integrovaného prostredie Netbeans sa aplikácia prekladá príkazom zadaným v adresári projektu s vyvíjanou aplikáciou [19]:

```
ant compile
```

Pre nahranie aplikácie do senzorového uzla potom slúži príkaz:

```
ant deploy
```

Pomocou neho nastane detekcia senzorových uzlov pripojených k hostiteľskému PC. V prípade, že je uzlov pripojených cez rozhranie USB viac, bude užívateľ dotázaný, do ktorého zariadenia sa má aplikácia nahrať. Následne sa aplikácia nahrá do flash pamäte senzorového uzla. Automaticky bude potom spustená po reštartovaní uzla. Rovnako je možné aplikáciu spustiť priamo príkazom:

```
ant run
```

V tomto prípade sa výstup aplikácie bude zobrazovať do konzoly.

Pre vzdialené nahrávanie sa používajú tie isté príkazy, ale je potrebné identifikovať presne vzdialený uzlol pomocou jeho adresy. Na to slúži paramter `-DremoteId=nnnn.nnnn.nnnn.nnnn`, kde `nnnn.nnnn.nnnn.nnnn` je daná 64-bitová adresa cieľového uzla. Príklady nahrávania a spúšťania aplikácie na senzorových uzloch pomocou základňovej stanice:

```
ant -DremoteId=0014.4F01.0000.71A9 deploy // nahrá aplikáciu
ant -DremoteId=0014.4F01.0000.71A9 run // spustí aplikáciu
```

3.5.1 Vývoj aplikácií pre hostiteľský PC

Pre vývoj aplikácií pre hostiteľský PC, napríklad pre odosielanie a prijímanie správ od senzorových uzlov sa postupuje podobne ako pri vývoji aplikácií pre senzorové uzly. Predovšetkým je možné použiť tie isté knižnice pre komunikáciu. Principiálne preto nie je rozdiel, či sa programuje komunikácia fyzického SunSpotu s okolím, alebo komunikácia hostiteľského PC s uzlami. Naprogramovaná aplikácia beží priamo na hostiteľskom PC, ale pre svoje

[6] Aplikácia jazyka Java ME pre malé vstavané zariadenia alebo napríklad mobilné telefóny a pod.

fungovanie využíva základňovú stanicu. V tomto prípade sa hlavná trieda nededí z triedy `Midlet`.

Pre vývoj aplikácie sa odporúča použitie dodávanej prázdnej šablóny s predpripraveným súborom `build.xml`. Táto sa opäť nachádza v adresári nainštalovaného API v podadresári `Demos/CodeSamples/SunSpotHostApplicationTemplate`. Aplikácia sa prekladá a spúšťa pomocou príkazov [19]:

```
ant host-compile // preklad
ant host-run // spustenie
```

Pri spustení sa implicitne vyžaduje pripojená základňová stanica k PC, inak spustenie zlyhá. To, či sa pri spustení aplikácie základňová stanica používa v zdieľanom, alebo dedikovanom móde, závisí na nastavení premennej `basestation.shared` (viď časť 3.3). Pokiaľ je potrebné z nejakého dôvodu aplikáciu spustiť bez základňovej stanice (napríklad pri ladení), je to možné pomocou príkazu:

```
ant host-run -Dbasestation.not.required=true
```

3.5.2 Prístup k zdrojom

Prístup k senzorom, či iným perifériám v najnovšej verzii API, ktoré bolo v práci použité, sa deje pomocou triedy `Resources` [19]. Každý senzor, či napríklad vstupno-výstupný pin je programátorovi prístupný pomocou tejto statickej triedy. Prístup k zdrojom je tak možné realizovať globálne kdekoľvek v zdrojovom kóde. Na to sa využíva metóda `lookup`. Ako názov lookup naznačuje, pomocou tejto triedy sa skutočne vyhľadá daný prostriedok. Vezmime si nasledujúce volanie metódy (druhý riadok nasledujúceho kódu) [19]:

```
IAccelerometer3D am;
am = (IAccelerometer3D) Resources.lookup(IAccelerometer3D.class)
```

Toto volanie vráti prvú inštanciu triedy, ktorá implementuje rozhranie `IAccelerometer3D` (akcelerometer). Samozrejme sa predpokladá, že akcelerometer je v uzle iba jeden. Tento prístup však umožňuje existenciu viacerých implementácií daného rozhrania uložených v triede `Resources`. Výsledok, ktorý vráti metóda `lookup` je vždy potrebné pretypovať na objekt príslušnej triedy. Toto pretypovanie je ďalej v príkladoch vynechané. V prípade, že je týchto implementácií viac, je možné získať všetky pomocou príkazu:

```
IResource[] all3DAccels = Resources.lookupAll(IAccelerometer3D.class)
```

Podobne sa pristupuje pre zisk prístupu k iným senzorom, či napríklad tlačidlám, alebo LED diódam. K teplotnému senzoru, ktorý bude v práci potrebný, sa pristupuje pomocou príkazu:

```
ITemperatureInput tempSensor = Resources.lookup(ITemperatureInput.class)
```

pričom aktuálna teplota v stupňoch Celzia sa potom získava pomocou príkazu:

```
double temperature = tempSensor.getCelsius()
```

Výsledok merania teploty je číslo typu double, pričom teplota je reprezentovaná s jedným desatinným miestom. Presnosť merania teploty je na 0,25 stupňa Celzia.

3.5.3 Výstup na LED

Pre zisk prístupu k LED diódam sa pristupuje podobne ako u ostatných zdrojov, konkrétne [19]:

```
ITriColorLEDArray leds = Resources.lookup(ITriColorLEDArray.class)
```

Pomocou premennej `leds` je potom možné pristupovať k jednotlivým LED diódam. U každej z nich je možné nastaviť jednotlivé farebné zložky metódou `setRGB` na hodnoty 0 až 255 a definovať tak farby a svietivosť. Táto metóda definuje farbu, ale nemení inak stav LED (nerozsvecuje ani nezhasína). K jednotlivým LED sa pristupuje pomocou metódy `getLED`:

```
leds.getLED(0).setRGB(255,0,0); // prvá dióda v rade na červenú farbu
leds.getLED(7).setRGB(0,0,255); // posledná dióda v rade na modrú farbu
```

Na ovládanie rozsvietenia a zhasnutia LED potom slúžia metódy `setOn` a `setOff` aplikovateľné na celé pole LED, alebo jednotlivé diódy [1]:

```
leds.setOn(); // rozsvieti celé pole LED
leds.setOff(); // zhasne celé pole LED
leds.getLED(0).setOn(); // rozsvieti prvú LED
leds.getLED(8).setOff(); // zhasne poslednú LED
```

3.5.4 Komunikácia

Na komunikáciu je možné využiť dva typy komunikácie, `Radiostream` a `Radiogram`. Líšia v spôsobe použitia a spoľahlivosti. Oba umožňujú komunikáciu multi-hop a nie je teda nevyhnutné, aby príjemca dát bol v priamom dosahu rádia odosielateľa.

Radiostream

Rozhranie `RadiostreamConnection` poskytuje peer-to-peer protokol na komunikáciu. Jedná sa o spoľahlivý protokol, založený na vstupnom a výstupnom stream-e. Kód pre otvorenie spojenia tohto typu vyzerá nasledovne [19]:

```
rs_conn = Connector.open("radiostream://nnnn.nnnn.nnnn.nnnn:xxx")
```

kde premenná `rs_conn` je typu `RadioStreamConnection`, `nnnn.nnnn.nnnn.nnnn` je adresa protistrany (cieľového SunSpotu), a `xxx` je číslo portu od 0 do 255[7]. Samotný vstupný a výstupný stream sa následne vytvorí pomocou príkazov:

```
DataInputStream dis = rs_conn.openDataInputStream(); // vstupný stream
DataOutputStream dos = rs_conn.openDataOutputStream(); // výstupný stream
```

Nevýhodou tohto protokolu je to, že pokiaľ príjemca uzavrie spojenie pred prijatím paketu, bude paket potvrdený ako prijatý, ale nebude príjemcom spracovaný [19]. Pre fungovanie komunikácie je potrebné otvoriť spojenie `RadioStream` na oboch stranách, s rovnakým portom a správnymi adresami uzlov.

[7] porty 0-31 sú vyhradené pre použitie systémom

Radiogram

Radiogram je protokolom typu klient-server. Tento protokol neposkytuje žiadne záruky doručenia dát, či zachovania poradia odosielaných dát. Pakety odoslané cez viac ako jeden skok môžu byť bez upozornenia stratené, alebo doručené viackrát [19]. Klientské aj serverové spojenie môže byť otvorené na senzorovom uzle súčasne. Dôležitý je ale fakt, že v tomto prípade všetky prichádzajúce pakety sú automaticky prijímané v spojení serverovom [19].

Serverové spojenie sa vytvára pomocou príkazu:

```
rg_server_conn = Connector.open("radiogram://:<port>")
```

kde `<port> je číslo portu` a `rg_server_conn` je premenná typu `RadiogramConnection`. Toto spojenie prijíma pakety od ktoréhokoľvek uzla. Klientské spojenie sa vytvorí príkazom:

```
rg_client_conn = Connector.open("radiogram://<adresa>:<port>")
```

kde `<port>` je číslo portu (musí sa zhodovať s číslom portu serverového spojenia), `<adresa>` je adresa uzla, na ktorom je otvorené serverové spojenie.

Dáta medzi serverom a klientom sa odosielajú v štruktúre `Datagram` [19]. Odosielané dáta pomocou nej budeme v práci ďalej nazývať datagramy. Pre vytvorenie tejto štruktúry slúži príkaz:

```
Datagram dg = rg_client_conn.newDatagram(conn.getMaximumLength())
```

Predošlý príkaz vytvoril pre klientske spojenie datagram o maximálnej možnej veľkosti. Túto je možné získať metódou `getMaximumLength`. Veľkosť datagramu je možné nastaviť aj na inú hodnotu, avšak iba v rozsahu do maximálnej možnej. S aktuálnou verziou API je maximálna hodnota 1260 bajtov. Pre serverové spojenie sa datagram vytvára rovnakým spôsobom. Táto štruktúra podporuje streamové operácie, pričom sa chová súčasne ako vstupný, aj výstupný stream. Zápis, alebo čítanie je možné robiť po jednotlivých hodnotách. Slúžia na to metódy triedy `Datagram` ako `writeByte`, `readByte`, `writeInt`, `readInt` [1]. Tiež je možný zápis a čítanie celého bajtového poľa metódami `write` a `readFully`. Odoslanie štruktúry `Datagram`, resp. prijatie dát do tej istej štruktúry sa vykoná príkazmi:

```
rg_client_conn.send(dg); // odoslanie, môže a nemusí blokovať
rg_client_conn.receive(dg); // príjem, blokujúca metóda
```

Uvedené príklady platia pre klientske spojenie, ale rovnakým spôsobom sa používajú aj u serverového spojenia.

Trieda `RadiogramConnection` umožňuje tiež komunikáciu typu broadcast. Spojenie broadcast sa otvára podobne, ako klientské spojenie, ale namiesto adresy uzla je použitý identifikátor `broadcast`. Príkaz vyzerá nasledovne:

```
broadcast_conn = Connector.open("radiogram://broadcast:<port>")
```

Predvolený počet skokov datagramu typu broadcast je 2. Pri použití na zisťovanie priameho okolia uzla je preto vhodné ho nastaviť na hodnotu 1. Nastavovanie maximálneho počtu skokov sa deje pomocou metódy `setMaxBroadcastHops` triedy `Radiogramconnection`.

Č. sektorov	Poč. Adresa	Veľkosť	Použitie
0-3	0x10000000	32KB	Boot Loader
4	0x10008000	8KB	konfiguračná stránka
5	0x1000A000	8KB	FAT tabuľka
6	0x1000C000	8KB	perzistentné nastavenia
7	0x1000E000	8KB	vyhradené
8-11	0x10010000	256KB	súbory intepreta Javy
12-19	0x10050000	512KB	zavádzacie súbory
20-69	0x10140000	3200KB	priestor dynamicky alokovaný medzi súbormi API, RecordStore a FlashFile
70-71	0x103F0000	16KB	tabuľka MMU úrovne 1
72-77	0x103F4000	48KB	vyhradené

Tabuľka 3.1: Organizácia pamäte flash [19].

3.5.5 Prístup k pamäti flash

Pamäť flash senzorového uzla má, ako bolo spomenuté, celkovú veľkosť 4MB. Fyzicky je organizovaná do sektorov rozličnej veľkosti. Prvých 8 sektorov má veľkosť 8KB každý, nasleduje 62 sektorov po 64KB, za nimi opäť 8 krát 8KB sektorov. Organizácia pamäte je znázornená v tabuľke 3.1.

Na prístup k pamäti flash je možné využiť dva rozličné prístupy, a to pomocou rozhraní `FlashFile` a `RecordStore`.

RecordStore

Toto rozhranie poskytuje jednoduchý mechanizmus ukladania dát do flash pamäte. Je založený na vkladaní jednotlivých záznamov do flash, namiesto súborov. Jedná sa vlastne o pomenované úložisko, obsahujúce kolekciu záznamov. Dáta zostávajú perzistentné pri opätovnom spustení, reštartovaní uzla, či výmene batérie. Fyzické umiestnenie záznamov vo flash pamäti je aplikácii transparentné [1]. Každá aplikácia môže vytvoriť viacero úložísk. Pokiaľ je daná aplikácia (`MIDlet`) z platformy odstránená, automaticky sa odstránia aj všetky úložiská asociované s touto aplikáciou. Prístupy k dátam nie sú implicitne výlučné. Zápisy a čítanie sú atomické, a serializované, takže nehrozí nekonzistencia dát. Ak však aplikácia obsahuje viacero vlákien, ktoré pristupujú súčasne k úložisku, musí sa o synchronizáciu postarať explicitne programátor.

Jednotlivé záznamy majú vždy jednoznačné identifikátory. Prvý záznam vložený do úložiska má identifikátor 1. Každý nasledujúci vložený záznam má ID vždy o 1 väčší, než naposledy pred ním vložený záznam [19]. Toto je potrebné brať do úvahy pri implementácii kruhového prepisu záznamov, ktorá bude popísaná v kapitole 8. Systém tiež pri každom vložení, či zmene záznamu vkladá časové razítko. Samotný záznam sa vkladá ako pole bajtov, pričom každý záznam môže mať odlišný počet bajtov.

Úložisko s názvom "nazov"sa vytvorí pomocou príkazu:

```
RecordStore rs = RecordStore.openRecordStore("nazov", true)
```

Záznam sa potom vkladá nasledovne:

```
byte[] data = new byte[]{1,2,3,4,5}; // pole, ktoré sa bude vkladať
int recordid = rs.addRecord(data, 0, data.length);
```

Druhý parameter (v tomto prípade 0) metódy `addRecord` udáva index v poli, od ktorého sa majú bajty do úložiska pridať. Tretí parameter je potom počet pridávaných bajtov. V premennej `recordid` bude identifikátor novo vloženého záznamu. Hodnotu daného záznamu je potom možné získať príkazom:

```
byte[] inputData = rs.getRecord(recordid)
```

Pre prechádzanie záznamov v úložisku je možné použiť triedu `RecordEnumeration`. Viac informácií je možné získať v [1].

FlashFile

Druhým spôsobom zápisu a čítania z flash pamäte je použitie triedy `FlashFile`. Pre bežné použitie sa odporúča použiť triedu `RecordStore` [19]. Niekedy však môže byť potrebné využiť mechanizmus nižšej úrovne. Práve pre tieto prípady by mala slúžiť táto trieda. Napriek názvu, vytvorenie objektu `FlashFile` nevytvára vo flash klasický súbor. O správu samotných súborov sa stará až manažér flash pamäte (súčasť API), ku ktorému nie je možné priamo pristupovať. Tento sa stará o modifikáciu FAT tabuľky a udržiavanie deskriptorov súboru.

Objekt `FlashFile` sa vytvára pomocou konštruktora, následne je možné alokovať miesto vo flash, a to nasledovne [1]:

```
FlashFile ff = new FlashFile("nazov"); // vytvorí nový objekt FlashFile
boolean success = ff.createNewFile(500); // alokuje miesto vo flash
```

Predošlými príkazmi sa vytvoril objekt a pomocou neho následne alokoval súbor, ktorého názov bude „nazov“, o veľkosti 500 bajtov. Veľkosť tohto súboru je pevná, súbor sa alokuje na pevný počet sektorov vo flash. Túto veľkosť nie je možné meniť [1]. Pre zrušenie alokovaného miesta sa použije jeden z dvoch príkazov:

```
ff.delete();
ff.setObsolete(true);
```

Prvý príkaz súbor odstráni. Odporúča sa používať príkaz `setObsolete`, ktorý označí súbor ako neplatný. Zrušený bude po reštarte senzorového uzla.

Pre samotný zápis a čítanie alokovaného súboru je potrebné vytvoriť vstupný a výstupný stream [1].

```
FlashFileInputStream is = new FlashFileInputStream(ff);
FlashFileOutputStream os = new FlashFileOutputStream(ff);
```

Cez tieto streamy sa potom číta a zapisuje do flash nasledovne:

```
byte[] data = new byte[]{1,2,3,4,5}; // pole, ktoré sa bude vkladať
os.write(data); // zapíše celé pole
os.write('a'); // zapíše jeden znak
byte[] outputData = new byte[3]; // pole, do ktorého sa bude čítať
int count  = is.read(outputData, 0, 3); // načíta 3 bajty
int c = is.read(); // načíta jeden bajt
```

Metóda `read` pri čítaní do poľa má 3 parametre, a to buffer, do ktorého sa číta, offset v poli, kde má začať vkladanie a maximálny počet čítaných znakov. Čítanie aj zápis je tiež možný po jednotlivých znakoch, ako je ukázané na predošlých príkladoch.

Kapitola 4

Agentný systém

Slovo agent je v oblasti umelej inteligencie chápané ako niekto, prípadne niečo, čo pôsobí v danom prostredí v prospech svojho klienta [20]. Klient je pritom vlastník, dizajnér, či napríklad zamestnávateľ. V tejto práci sa agentom myslí tzv. umelý agent[1]. Tento agent je vytvorený človekom. Jedná sa o autonómnu entitu, ktorá je situovaná v prostredí a jedná v prospech toho, kým bol vytvorený. Tiež je možné na tento pojem nahliadať ako na vec, ktorá vníma svoje okolie pomocou senzorov, a vykonáva akcie v prostredí pomocou aktuátorov. V tomto prípade spadá tento pojem do oblasti robotiky [15]. Softwarový agent potom potom ako vstupy využíva napríklad vstupy z klávesnice, súborov, či senzorov. Ďalej týmito vstupmi môžu byť napríklad sieťové pakety. Práve tento typ agenta sa používa v navrhovanej agentnej platforme.

Vstupy agenta sa nazývajú aj vnemy. Pokiaľ je možné špecifikovať akciu agenta pre každú možnú sekvenciu týchto vnemov, je možné povedať, že správanie agenta je z matematického hľadiska popísané tzv. agentnou funkciou [15]. Pokiaľ ide o konkrétnu implementáciu tejto funkcie, nazývame ju agentným programom. Tento program beží na konkrétnej agentnej architektúre. Touto architektúrou rozumieme nejaké výpočtové zariadenie s fyzickými senzormi a aktuátormi. Agent potom pozostáva z dvojice agentný program a architektúra. Dvojicu agent a prostredie, v ktorom operuje, nazývame agentným systémom.

Typické vlastnosti agenta sú [20]:

- Autonómnosť – agent je schopný pôsobiť v prostredí bez priameho vplyvu okolia. Má plnú kontrolu nad svojím správaním, ale súčasne musí byť schopný plniť svoje stanovené ciele.
- Reaktivita – schopnosť agenta adekvátne reagovať na zmeny prostredia, v ktorom pôsobí.
- Proaktivita – agent je schopný aktívne pôsobiť na svoje okolie a meniť ho v závislosti na svojich potrebách.
- Sociálnosť – agent je schopný pôsobiť v skupine agentov, kooperovať s ostatnými agentmi či riešiť konflikty.

Ďalšou vlastnosťou, ktorá je dôležitá, je racionalita agenta. Chovanie agenta má byť také, aby sa choval racionálne vzhľadom k svojim cieľom resp. podcieľom. Racionálne jednanie agenta je potom také, ktoré s danými znalosťami o prostredí na základe rozhodovacích schopností najlepšie povedie k dosiahnutiu stanoveného cieľa [20].

[1] angl. Artificial Agent

4.1 Praktické rozhodovanie

Pred vysvetlením tohto pojmu je vhodné spomenúť pojem teoretické odvodzovanie. Príkladom je systém výrokových a predikátových logík[2]. U nich sa odvodzujú znalosti z nejakého modelu. U predikátových logík je tento model daný množinami a predikátmi, ktoré určujú vzťahy medzi prvkami týchto množín [20]. Teória je potom súbor axiómov, odvodzovacích pravidiel a formulí, popisujúcich model. Matematický zápis

$$M \models f$$

značí, že formula f je v danom modele pravdivá, t.j. je možné ju dokázať pomocou odvodzovacích pravidiel. Týmto odvodzovaním platnosti formúl daného modelu sa zaoberá teoretické odvodzovanie.

Praktické rozhodovanie je odvodzovanie znalostí z formálneho modelu. Na rozdiel od teoretického odvodzovania nie je založené na tom, čo je, alebo nie je pravda. Namiesto toho sa rozhoduje na základe toho, čo sa má vykonať, aby sa niečo stalo pravdou [20]. Praktické rozhodovanie je teda založené na odvodzovaní, čo musí byť vykonané v stave s modelu M, aby bol dosiahnutý cieľ g. Toto je vyjadrené v zápise [21]:

$$M, s, g \mid\sim Do(\pi) \textit{ iff } M, T^P(s, \pi) \models g$$

kde π je plán, $\pi \in A^*$, pričom A je množina akcií. T^P je funkcia tvaru: $S \times A^* \to S$, kde S je množina stavov.

4.2 BDI agent

BDI systémy[3] sú založené na troch mentálnych stavoch agenta [14]:

- Viera, resp. znalosti o okolitom svete. Tieto znalosti agent získava behom svojho pôsobenia v prostredí.
- Zámery, ktoré sa agent snaží dosiahnuť.
- Aktuálny zámer, ktorý sa agent aktuálne snaží dosiahnuť.

Na základe týchto stavov sa vykonáva praktické odvodzovanie a následne riadi chovanie agenta. Interpret BDI systému sa chová podľa nasledujúceho pseudokódu [14]:

```
initialize-state();
repeat
  options := option-generator(event-queue);
  selected-options := deliberate(options);
  update-intentions(selected-options);
  execute();
  get-new-external-events();
  drop-successful-attitudes();
  drop-impossible-attitudes();
end repeat
```

[2] viac informácií je možné nájsť v [20]

[3] Belief, Desire, Intention – viera, túžba, zámer

Na začiatku každého cyklu funkcia `option-generator` číta z fronty udalostí a na základe nich vráti zoznam možností pre ďalšie spracovanie. Potom funkcia `deliberate` vyberie možnosti, ktoré budú nasledované pri ďalšom spracovaní. Pomocou `update-intentions` sa na základe týchto informácií aktualizujú zámery agenta. Ak v tomto bode existuje zámer s vykonateľnou atomickou akciou, je táto akcia spustená. Následne sa do fronty externých udalostí pridajú nové nastané udalosti. Nakoniec sa modifikujú zámery a aktuálny plán, pričom funkcie `drop-successful-attitudes` a `drop-impossible-attitudes` odstránia z týchto štruktúr všetky úspešné a nevykonateľné zámery.

Kapitola 5

Jazyk ALLL

Jazyk ALLL bol vyvinutý na FIT VUT. Tento jazyk sa orientuje na rozšírenie existujúcich prístupov k agentným jazykom. Napríklad sa jedná o obsluhu výnimiek, meta-rozhodovanie, či klonovanie agentov [16]. Tým pádom je agent schopný sa vysporiadať s neočakávanými situáciami a chybami behom vykonávania plánu. Jazyk bol navrhnutý pre programovanie agentov operujúcich v bezdrôtových senzorových sieťach. Pri návrhu jazyka boli brané do úvahy nasledovné aspekty [23]:

- senzorové uzly sú založené na mikrokontroléri a majú k dispozícii relatívne malé množstvo operačnej či flash pamäte.
- je očakávané, že senzorové uzly bežia relatívne dlhú dobu bez výmeny batérie. Preto senzorové uzly väčšinu času trávia v režime spánku.
- typický agent pracujúci v bezdrôtovej senzorovej sieti je mobilný.
- agent pracuje v sieti, ktorej štruktúra sa dosť často môže dynamicky meniť.
- je dôležitá kvalita komunikačnej linky a zaručenie správneho doručenia dát.

Ako väčšina jazykov, jazyk ALLL je množina viet [23]. Každá veta reprezentuje plán agenta vo forme sekvencie akcií. Akcie sú spúšťané postupne zaradom, pričom každá z nich môže meniť stav agenta, prípadne systémových komponentov. Plán môže vytvárať hierarchickú štruktúru procesov pomocou spúšťania podplánov. Ak vykonávanie niektorej akcie zlyhá, daný (pod)plán je odstránený a riadenie sa odovzdá nadradenému plánu. Pokiaľ akcia skončí úspešne, typicky uloží do niektorého z registrov výsledok akcie. Agent končí svoju činnosť, pokiaľ zlyhá hlavný plán, alebo vykonaná posledná akcia z tohto plánu [23].

5.1 Dátové štruktúry

Základným kameňom jazyka ALLL je atóm [22]. Atómom môže byť číslo, operátor, alebo reťazec. Konkrétne, atóm je možné popísať ako regulárny výraz [22]:

```
LATOM = {[a..z ][ A..Z ][0..9]}*|[_,+,-,*,?,^,@,!,$,&,#]
```

Uvedený výraz bol upravený vzhľadom k aktuálne používanej verzii jazyka ALLL.

5.1.1 Zoznam, n-tica

Zoznam je v ponímaní jazyka ALLL chápaný ako sekvencia atómov uzatvorená v okrúhlych zátvorkách. Jednotlivé atómy sú pritom oddelené čiarkami. Tvar zoznamu je teda nasledovný: $(s_1, s_2, ..., s_n)$. Takýto zoznam nazývame lineárny zoznam [22]. Rozšíreným typom zoznamu je všeobecný zoznam. Pod týmto pojmom rozumieme zoznam v podobnom ponímaní ako lineárny zoznam. Rozdiel je v tom, že okrem atómov tento zoznam ako položky môže obsahovať iné zoznamy (lineárne, či všeobecné). Ďalej v práci budeme všeobecný zoznam nazývať n-tica.

5.1.2 Akcie a plány

Špeciálnym typom sekvencie je plán, čo je vlastne sekvencia akcií. Každá akcia jazyka ALLL je tvorená všeobecným zoznamom, pred ktorým je vždy znak označujúci danú akciu. Tento znak je z množiny `{+,-,*,?,!,^,@,$}`. Pomocou tohto znaku sa identifikuje typ akcie. Príslušný všeobecný zoznam potom reprezentuje operandy akcie. Napríklad, tvar akcie pre pridanie n-tice `(a,b,c)` do tabuľky `BeliefBase` (pojem je vysvetlený v časti 5.3) je nasledovný:

```
+(a,b,c)
```

Okrem jednotlivých akcií môže sekvencia akcií na ktoromkoľvek mieste obsahovať ešte znak `#`, tzv. zarážku, ktorej význam bude uvedený v častiach 5.3.4 a 6.1.9.

Sekvencia akcií nie je oddelená čiarkami, ale začiatok každej položky sa určuje pomocou znaku označujúceho typ akcie. Pomenovaným plánom potom rozumieme n-ticu, ktorá na prvom mieste obsahuje názov plánu a na druhom mieste sekvenciu akcií uzatvorenú v zátvorkách. Príklad konkrétneho pomenovaného plánu je nasledovný:

```
(pridaj,(+(a,b,c)*(c,_)$(l,(g,1))$(k)#$(l,(r,1))))
```

Tento príklad obsahuje plán pomenovaný `pridaj`, obsahujúci sekvenciu 5 akcií. Význam a syntax všetkých akcií jazyka ALLL bude uvedený v kapitole 6.

5.1.3 Tabuľka

Tabuľkou sa rozumie množina n-tíc jazyka ALLL. Všeobecne je možné túto množinu chápať ako neusporiadanú, kde na poradí jej prvkov nezáleží. Prvky sa z tabuľky vyberajú pomocou unifikácie [16]. Unifikácia je v podstate vyhľadanie všetkých n-tíc v tabuľke, ktoré vyhovujú zadanému predpisu. Predpisom je pritom n-tica, ktorá môže obsahovať anonymné premenné, viď časť 5.2.

Unifikáciu môžme chápať ako postupné porovnávanie všetkých prvkov vzorovej n-tice s prvkami každej n-tice v tabuľke. Pri porovnávaní prvkov dvoch n-tíc sa postupuje nasledovne. U atomickej hodnoty je porovnanie testom na zhodu dvoch reťazcov. V prípade vnorenej n-tice sa musí porovnať rekurzívne obsah vnorených n-tíc na danej pozícii. Pokiaľ je niektorý z prvkov n-tice anonymná premenná, automaticky sa považuje za zhodu na tejto pozícii. N-tica je unifikovaná, pokiaľ sa zhoduje so vzorovou na všetkých pozíciách a má rovnaký počet prvkov ako vzorová n-tica.

5.1.4 Register

Register je najvšeobecnejšia dátová štruktúra. Môže obsahovať buď atóm (jednoduchý reťazec), alebo n-ticu. Do registrov sa typicky ukladajú výsledky operácií jazyka. Jazyk ALLL predpokladá existenciu troch takýchto registrov. Výskyt registra sa označuje pomocou zástupného symbolu registra, viď časť 5.2.

5.2 Použitie registrov a anonymných premenných

Špeciálnym prípadom atómu je tzv. anonymná premenná, ktorá sa označuje znakom „_“[16]. Výskyt tejto premennej značí, že na jej mieste sa môže vyskytovať akýkoľvek prvok, či n-tica. Táto premenná sa môže vyskytovať na ktoromkoľvek mieste n-tice. Typicky sa používa pri testovaní tabuľky na množinu n-tíc určitého tvaru a počtu prvkov. Napríklad ak tabuľka obsahuje n-tice `(a,b)`, `(a,c)` a `(b,c)`, tak výsledkom unifikácie n-tice `(a,_)` v tejto tabuľke budú n-tice `(a,b)` a `(a,c)`.

Registre sa používajú na uloženie výstupných hodnôt operácií. Pred každou operáciou, ktorá vracia návratovú hodnotu, je preto potrebné nastaviť aktívny register [16]. Viac v časti 6.1.1. Uložené hodnoty v registroch je možné potom využívať ako parametre pre ďalšie operácie.

Jednotlivé registre majú zástupné symboly `&1`, `&2` a `&3`. Každá n-tica môže obsahovať na ktoromkoľvek mieste takýto zástupný symbol. Jeho význam je rovnaký, ako by sa na danom mieste nachádzala priamo hodnota, ktorú register takto zastupuje. Nech register 2 obsahuje n-ticu `(b,c)`. Potom n-tica `(a,&2)` po nahradení symbola registra bude vyzerať nasledovne: `(a,(b,c))`.

5.3 Agent v jazyku ALLL

Agent v jazyku ALLL je chápaný ako BDI agent (viď časť 4.2). Tento agent obsahuje viacero častí. Sú to tri tabuľky - BeliefBase, InputBase a PlanBase, tri registre, aktuálny plán, identifikátor a triedu. Význam jednotlivých častí je rozobraný v nasledujúcich podkapitolách.

5.3.1 PlanBase

Táto tabuľka obsahuje pomenované plány agenta. V modele BDI táto časť korešponduje s časťou `Desires`, čiže sa vlastne jedná o zámery, ktoré sa agent snaží dosiahnuť. Každý takýto plán môže obsahovať volanie podplánov. Syntax n-tíc obsahujúcich plány bola spomenutá v časti 5.1.2.

5.3.2 InputBase

Tabuľka InputBase obsahuje vstupy agenta. Týmito vstupmi môžu byť správy od iných agentov, alebo hodnoty zo senzorov. Každá n-tica v tejto tabuľke obsahuje dva prvky. Prvým je číslo uzla, od ktorého prišla správa, druhým je potom n-tica, ktorá obsahuje samotnú správu, či hodnotu senzora. Hodnoty do tabuľky ukladá vždy agentná platforma. Agent má možnosť vložiť hodnotu nepriamo napríklad tak, že vykoná príkaz merania hodnoty zo senzora.

5.3.3 BeliefBase

Tabuľka BeliefBase korešponduje s časťou `Beliefs` BDI systému. V nej sú teda uložené znalosti agenta, ktoré behom svojej činnosti získava. Agent si dáta v tejto tabuľke spravuje sám. Môže sem vkladať n-tice symbolizujúce znalosti, dotazovať sa na výskyt n-tíc pomocou unifikácie a unifikované položky z tabuľky mazať. Viac informácií o týchto operáciách bude uvedených v kapitole 6.

5.3.4 Aktuálny zámer

Aktuálny zámer, alebo `Intention` podľa modelu BDI, obsahuje aktuálne spracovávaný plán, ktorý agent vykonáva. Tento zámer je možné meniť buď spustením pomenovaného plánu, alebo priamym spustením sekvencie akcií (viac v podkapitole 6.1.6). Pri pridávaní sekvencie akcií do aktuálneho plánu sa vždy za tieto akcie vloží špeciálny znak, tzv. zarážka (označuje sa znakom `#`). Ako bolo spomenuté, zarážka sa môže vyskytovať aj kdekoľvek v pláne, pokiaľ ju tam programátor vloží explicitne.

Zlyhanie operácie v aktuálnom zámere

Zlyhaním operácie sa rozumie chyba sémantiky tejto operácie alebo prípadne nemožnosť úspešne vykonať operáciu. Príkladom môže byť snaha o odoslanie správy na neexistujúci uzol, alebo neúspešné vyhľadanie v tabuľke pomocou unifikácie. Pokiaľ nastane zlyhanie niektorej operácie, je automaticky zmazaný aktuálny zámer agenta po prvú zarážku. Tento jav budeme nazývať aj pád plánu.

5.3.5 ID a trieda agenta

Každý agent sa jednoznačne identifikuje pomocou identifikátora a triedy. Jedná sa o dve 16-bitové čísla. Identifikácia slúži pre niektoré funkcie platformy, ako napríklad pachové stopy agenta (viac v časti 6.2.10). Agent tak môže zistiť svoju predošlú prítomnosť na danom uzle. Agent si rovnako môže svoje ID a triedu meniť pomocou príslušných operácií.

Kapitola 6

Plaforma WSageNt

Táto kapitola je popisom fungovania platformy WSageNt, a to hlavne z hľadiska popisu jednotlivých funkcií, ktoré platforma poskytuje. Platforma je implementovaná v jazyku `nesC` a k svojmu behu využíva prostredie `TinyOS`. Jedná sa udalosťami riadené operačné prostredie navrhnuté s cieľom behu vo vstavaných zariadeniach, ktoré tvoria bezdrôtovú senzorovú sieť. Viac informácií o programovaní v tomto prostredí je možné nájsť v [10].

Základná implementácia prebehla v rámci bakalárskej práce [16] a diplomovej práce [4]. O ďalšie funkcie bola obohatená v rámci ďalších prác. Skladá sa z dvoch vzájomne kooperujúcich častí, interpreta a platformy. Presnejší popis týchto dvoch častí bude uvedený ďalej.

V kapitole 5 bol uvedený základný popis jazyka ALLL a princíp fungovania agentov v ňom napísaných. Táto kapitola na ňu naväzuje a zaoberá sa hlavne podrobným popisom jednotlivých operácií jazyka, ich syntaxou a sémantikou. Popis bol vytvorený podľa zdrojov, na ktoré sa tento popis odkazuje. Rovnako tiež na základe aktuálnej verzie platformy `WSageNt`, ktorej zdrojové kódy boli k dispozícii. Pre upresnenie výstupov jednotlivých operácií bola využitá možnosť simulácie behu platformy pomocou simulátora TOSSIM. Viac o jeho fungovaní je možné prípadne nájsť v [11].

6.1 Služby interpreta

V tejto časti budú popísané služby implementované priamo v interprete agentnej platformy. Jedná sa o časť platformy, ktorá bola vyvíjaná v rámci bakalárskej práce [16]. Tieto služby je možné volať bez účasti volania platformných operácií, priamo v časti, ktoré implementuje interpret.

6.1.1 Nastavenie aktívneho registra

Operácia nastaví aktívny register na zvolenú hodnotu 1-3. Aktívnym registrom sa rozumie ten, do ktorého operácie, ktoré majú výstup, ukladajú hodnoty. Táto operácia sa označuje znakom „`&`“. Príklad volania operácie:

- `&(3)` – nastaví register 3 ako aktívny.
- `&(4)` – vyhodnotí sa ako chyba syntaxe, keďže jazyk ALLL má v aktuálnej verzii 3 registre.

6.1.2 Pridávanie do BeliefBase

Pomocou tejto akcie je možné pridať zadanú n-ticu do tabuľky BeliefBase. Zadaná n-tica je najprv vyhľadaná v BeliefBase pomocou unifikácie. Pokiaľ nebola unifikovaná (nenachádza sa v tabuľke), bude do BeliefBase pridaná [16]. N-tica môže obsahovať aj anonymné premenné, či zástupné symboly registrov.

Pokiaľ n-tica obsahuje výskyty symbolov registrov, budú tieto nahradené obsahom registrov ešte pred vložením do tabuľky. Predpokladom bezchybného vloženia je správna syntax n-tice (pričom môže obsahovať n-tice ľubovoľne zanorené) a dostatok voľného miesta v tabuľke BeliefBase. Ak sa nepodarí vložiť n-ticu v dôsledku toho, že došlo miesto v tabuľke, bude to mať za následok zlyhanie operácie, viď časti 5.3.4 a 6.1.9.

Operácia je označená znakom „`+`“. Nech register 1 obsahuje n-ticu `(c,d)`. Nasledujú príklady volania operácie:

- `+(a,b,c)` – pridá do BeliefBase n-ticu `(a,b,c)`.
- `+(a,(b,&1))` – pridá do BeliefBase n-ticu obsahujúcu vnorenú n-ticu `(b,&1)`, teda po nahradení obsahu registra sa do tabuľky vloží n-tica `(a,(b,(c,d)))`.
- `+&2` – pridá do BeliefBase n-ticu `(c,d)`.

6.1.3 Odoberanie z BeliefBase

Táto akcia, ako názov hovorí, umožňuje odobrať n-ticu (resp. n-tice) z BeliefBase. Označuje sa znakom „`-`“. Odoberanie funguje na podobnom princípe, ako pridávanie. Teda najprv sa vykoná unifikácia n-tice. Keďže opäť n-tica môže obsahovať anonymné premenné, unifikovaných môže byť viacero n-tíc. Všetky takto unifikované n-tice sú nakoniec z BeliefBase odobrané [16]. Opäť je možné, aby niektorá časť, prípadne celá n-tica bola v uložená v registri. Predpokladajme, že register 1 obsahuje n-ticu `(c,d)` a v BeliefBase sú n-tice `(a,b)`, `(a,c)` a `(a,c,d)`. Nasledovné príkazy budú mať takéto chovanie:

- `-(a,b)` – odoberie BeliefBase n-ticu `(a,b)`.
- `-(a,&1)` – unifikuje n-ticu `"(a,(c,d))"`, táto sa v BeliefBase nenachádza, preto ostane bez zmeny.
- `-(a,_)` – unifikáciou sa nájdu n-tice `(a,b)` a `(a,c)` a obe budú z BeliefBase odstránené.

6.1.4 Test BeliefBase na zadanú n-ticu

Jedná sa o operáciu, ktorá unifikuje všetky n-tice, ktoré vyhovujú zadanej n-tici. Následne všetky tieto n-tice budú vložené do aktívneho registra. Výsledkom operácie je jedna n-tica, ktorá ako svoje položky obsahuje všetky unifikované n-tice. Pred touto operáciou je nutné nastaviť aktívny register. Pokiaľ sa unifikáciou nenájde žiadna n-tica, operácia zlyhá a aktuálny plán je zmazaný po zarážku [16]. Opäť je možné, aby hľadaná n-tica obsahovala anonymné premenné, vnorené n-tice, alebo aby bola jej časť, prípadne celá n-tica uložená v registri. Operácia sa označuje znakom „`*`“.

Predpokladajme, že v BeliefBase sú n-tice `(a,b)`, `(a,c)`, `(a,c,d)` a `(c)`, a register 1 obsahuje n-ticu `(c)`. Nasledujú príklady chovania tejto operácie:

- `*(a,_)` – do aktívneho registra vloží n-ticu `((a,b),(a,c))`.

- `*&1` – do aktívneho registra vloží n-ticu ((c)).
- `*(b,_)` – unifikáciou sa nenájde žiadna n-tica, preto nič do aktívneho registra nevloží, ale zmaže aktuálny plán po zarážku.

6.1.5 Test InputBase na správu od zadaného uzla/senzora

Táto operácia má ako parameter jednoprvkovú n-ticu obsahujúcu buď identifikátor uzla, alebo znak senzora, ktorého hodnota má byť získaná [16]. N-tice v InputBase majú vždy tvar `(<číslo uzla>,(správa))`, resp. `(s,(nameraná hodnota))`. Identifikátor uzla môže byť tiež zadaný zástupným symbolom registra.

Pomocou tejto operácie sa vyhľadáva v tabuľke `InputBase` s využitím unifikácie prvá n-tica, ktorá ako prvý prvok obsahuje hodnotu zadanú parametrom operácie (druhý prvok n-tice, ktorá sa v tabuľke unifikuje, je potom anonymná premenná). Unifikovaná n-tica sa následne vloží do aktívneho registra. Avšak vloží sa iba samotná správa od senzora či uzla, čiže sa z unifikovanej n-tice pred vložením do aktívneho registra odstráni adresa uzla, resp. znak senzora. V prípade, že sa unifikáciou nenájde žiadna n-tica, operácia zlyhá a plán je zmazaný po zarážku. Po výbere sa daná n-tica z InputBase automaticky odstráni. Operácia sa označuje symbolom „`?`“.

Predpokladajme, že tabuľka `InputBase` obsahuje n-tice `(2,(a,b,c))`, `(2,(sprava))` a `(s,(20))`. Chovanie operácie je ilustrované na nasledujúcich príkladoch:

- `?(2)` – do aktívneho registra vloží n-ticu `(a,b,c)`. Prípadné ďalšie volanie s tým istým parametrom do aktívneho registra vloží n-ticu `(sprava)`.
- `?(s)` – do aktívneho registra vloží n-ticu `(20)`.
- `?(3)` – unifikáciou sa nenájde žiadna n-tica, operácia zlyhá.

6.1.6 Priame spustenie

Pomocou tejto operácie je možné priame spustenie sekvencie akcií (bez spustenia konkrétneho plánu) [16]. V praxi to znamená vloženie tejto sekvencie do aktuálneho plánu so zarážkou. Operácia sa označuje znakom „`@`“a jej parametrom je sekvencia akcií uzatvorená v zátvorkách. Príklad spustenia:

- `@(+(a,b)-&2*(a,_))` – vloží na zásobník sekvenciu akcií `+(a,b)-&2*(a,_)#`, pričom znak `#` (zarážka) sa vkladá automaticky.

6.1.7 Nepriame spustenie

Akcia v podstate implementuje volanie pomenovaného plánu. Označuje sa znakom „`^`“. Parametrom akcie je jednoprvková n-tica obsahujúca názov plánu, ktorý sa má spustiť [16]. Tento plán musí existovať, inak operácia zlyhá. Keďže plány sú uložené v tabuľke PlanBase, nastane vyhľadanie plánu pomocou unifikácie v tejto tabuľke. Vyhľadaný plán má tvar uvedený v časti 5.1.2. Z vyhľadanej n-tice v tomto tvare sa následne vyberie druhá položka obsahujúca sekvenciu akcií. Táto sekvencia je vložená do aktuálneho plánu, nasledovaná zarážkou. Nasledujú príklady spustenia operácie:

- `^(mojplan)` – unifikuje v PlanBase n-ticu, ktorá má prvú položku `"mojplan"` a druhú položku tejto n-tice (obsahujúcu zoznam príkazov) pridá do aktuálneho plánu.

- `^&2` – funguje podobne ako predošlý príklad, ale n-ticu obsahujúcu názov plánu získa z registra 2.

6.1.8 Odoslanie správy

Pomocou tejto operácie je možné komunikovať s ostatnými senzorovými uzlami siete. Príkaz sa označuje znakom „`!`“a jeho parametrom je n-tica, kde na prvej pozícii sa nachádza identifikátor uzla (16-bitová číselná hodnota) a na druhej pozícii je vnorená n-tica, ktorá je samotnou správou. Na cieľový uzol sa potom odošle práve iba táto vnorená n-tica. Špeciálnym identifikátorom uzla je číslo 1 [16]. Jedná sa o symbolickú adresu základňovej stanice. V tomto prípade sa táto adresa nahradí reálnym identifikátorom základňovej stanice a následne bude na danú adresu doručená.

V prípade, že sa správu nepodarí odoslať (napríklad ak cieľový uzol nie je dostupný, alebo nemôže prijať správu), operácia zlyhá a dôjde k odstráneniu aktuálneho zámeru po zarážku. Príklady syntaxe operácie:

- `!(1,(abc))` – odošle na základňovú stanicu správu „`(abc)`“.
- `!(3,&2)` – odošle na uzol identifikátorom 3 ako správu n-ticu uloženú v registre 2. V prípade, že register neobsahuje žiadnu n-ticu, alebo sa správu nepodarí odoslať, operácia zlyhá.

6.1.9 Zarážka

Jedná sa o špeciálny symbol, ktorý sa z hľadiska sémantiky nijako neinterpretuje [16]. Pokiaľ sa pri interpretovaní zdrojového textu narazí na tento symbol, je preskočený a ignorovaný. Jeho významom je označenie konca podplánu. Po tento symbol sa zmaže obsah zásobníka v prípade, že niektorá z operácií, ktorá je na zásobníku pred ním, zlyhá. Za koniec vkladaného podplánu sa tento symbol umiestňuje automaticky (nemusí byť v zdrojovom texte).

Je však možné ho explicitne umiestniť kamkoľvek medzi príkazy. V tomto prípade sa v prípade predošlého zlyhania niektorej operácie zo zásobníka nevymaže celý podplán, ale iba jeho časť po túto zarážku. Symbolom zarážky je znak „`#`“. Vezmime si nasledovnú sekvenciu operácií jazyka ALLL, ktorú obsahuje aktuálny zámer agenta:

```
*(a,b)^(plan1)$(k)#^(plan2)
```

Pokiaľ prvá operácia (test BeliefBase) zlyhá (nenájde sa žiadna n-tica), odstránia sa z aktuálneho zámeru agenta všetky operácie po znak `#`. Je teda spustený plán `plan2`. Pokiaľ ale operácia skončí úspešne, pokračuje sa spustením plánu `plan1` a následne je interpret ukončený (operáciou `$(k)`, ktorej význam bude popísaný v časti 6.2.6). Zarážku je tak možné využívať podobne ako sekcie try-catch jazyka Java, kedy za zarážkou je možné v kóde ošetriť zlyhanie niektorej z operácií pred touto zarážkou. Rovnako je možné pomocou nej vytvárať programové konštrukcie typu if-else.

6.2 Služby platformy

Služba platformy je špeciálnym prípadom operácie jazyka ALLL. Vyznačuje sa tým, že pre svoje spustenie potrebuje dodatočné moduly, ktoré sa nenachádzajú priamo v interprete. Služby platformy sa delia na synchrónne a asynchrónne [4]. Synchrónne služby platformy

(ktorých je ale menšina) sú schopné výsledok operácie získať okamžite a bez čakania. Príkladom je služba pre zisk prvého prvku n-tice.

Pokiaľ je operácia asynchrónna, je potrebné na čas, kým operácia skončí, pozastaviť činnosť interpreta. Riadenie je ďalej odovzdané časti platformy, ktorá sa postará o dokončenie operácie. Následne informuje interpret, že môže pokračovať vo vykonávaní a začať interpretovať nasledujúcu operáciu (pokiaľ nie je zásobník prázdny).

Služba sa označuje znakom „`$`“. Za týmto znakom nasleduje vždy n-tica, ktorá obsahuje minimálne jeden prvok, a to označenie konkrétnej služby platformy. Vo väčšine prípadov je nasledovaný ďalšími parametrami operácie. Presnejšia syntax bude rozobraná u popisu jednotlivých služieb platformy.

6.2.1 Zisk prvého prvku n-tice

Táto služba patrí medzi synchrónne. Služba, ako vyplýva z názvu, získa prvý prvok zadanej n-tice (táto n-tica je parametrom služby) a uloží tento prvok do aktívneho registra. Výsledkom volania služby je vždy n-tica. Ak teda prvý prvok n-tice je atóm, do aktívneho registra sa vloží uzatvorený v zátvorkách. Označenie služby je „`f`“[4]. Predpokladajme, že register 1 obsahuje n-ticu `(1,2,3)` Príklady volania operácie výberu prvého prvku sú nasledovné:

- `$(f,(a,b,c))` – do aktívneho registra vloží n-ticu `(a)`.
- `$(f,((1,2),b,(c,d)))` – do aktívneho registra vloží n-ticu `(1,2)`.
- `$(f,&1)` – do aktívneho registra vloží n-ticu `(1)`.

6.2.2 Zisk zvyšku n-tice okrem prvého prvku

Služba opäť patrí medzi synchrónne služby. Jedná sa o doplnok službe zisku prvého prvku zoznamu. Výsledkom je vždy n-tica, ktorá obsahuje všetky prvky pôvodnej n-tice s výnimkou prvého prvku. Služba sa označuje znakom „`r`“[4]. N-tica môže byť zadaná samozrejme aj zástupným symbolom registra. Pokiaľ zdrojová n-tica obsahuje iba jeden prvok, operácia skončí úspešne, ale výsledkom bude prázdna n-tica. Výsledok sa opäť ukladá do aktívneho registra. Predpokladajme, že register 2 obsahuje n-ticu `(a,(b,c),d)`. Nasledujú príklady volania operácie:

- `$(r,(a,b,c))` – do aktívneho registra vloží n-ticu `(b,c)`.
- `$(r,&2)` – do aktívneho registra vloží ticu `((b,c),d)`.
- `$(r,((a,b)))` – do aktívneho registra vloží hodnotu „`()`“.

6.2.3 Uspanie interpreta do príchodu správy

Táto služba je asynchrónna. Po jej zavolaní je vykonávanie interpreta pozastavené, riadenie prevezme platforma [4]. Nasleduje čakanie po dobu, pokým nepríde správa z rádia. Po príchode správy nasleduje opätovné spustenie interpreta. Služba je označená symbolom „`s`“a nemá žiadne ďalšie parametre. Spustenie vyzerá nasledovne:

- `$(s)`

6.2.4 Aktivovanie sledovania správ pri behu interpreta

Táto služba má označenie „`a`“a nemá žiadne ďalšie parametre. Praktický význam sledovania prichádzajúcich správ je nasledovný. Ak od okamihu, odkedy sa sledovanie aktivuje príde správa, je táto udalosť zaznamenaná. Pokiaľ neskôr nasleduje volanie služby platformy pre čakanie do príchodu správy (a predtým prijatá správa nebola do toho okamihu spracovaná), je toto čakanie automaticky preskočené [4]. Služba je synchrónna. Spustenie vyzerá takto:

- `$(a)`

6.2.5 Uspanie činnosti interpreta na stanovenú dobu

Jedná sa o asynchrónnu službu, po ktorej sa interpret uspí na dobu, ktorá je daná ako parameter služby [4]. Doba je daná ako celé číslo v milisekundách. Po vypršaní požadovanej doby je činnosť interpreta obnovená. Služba má označenie „`w`“, pričom za týmto označením ešte nasleduje jednoprvková n-tica obsahujúca spomínanú dĺžku čakania. Volanie služby vyzerá nasledovne (predpokladá sa, že register 1 obsahuje n-ticu `(500)`):

- `$(w,(1500))` – uspí interpret na 1,5 sekundy.
- `$(w,&1)` – uspí interpret na 0,5 sekundy.

6.2.6 Ukončenie činnosti interpreta

Táto služba je synchrónna. Jej označenie je „`k`“[4] a nemá žiadne ďalšie parametre. Volanie tejto služby má za následok okamžité ukončenie činnosti interpreta. Činnosť môže byť znovu obnovená, pokiaľ je danému uzlu zaslaný nový agent. Spustenie služby vyzerá nasledovne:

- `$(k)`

6.2.7 Výstup na LED diódy

Služba slúži na ovládanie stavu LED diód na uzle. Platforma počíta s tromi LED diódami, červenej, zelenej, a žltej farby [4]. Pomocou nej je možné zvolenú diódu rozsvietiť, zhasnúť, alebo invertovať jej aktuálny stav (z rozsvietenej na zhasnutú a naopak). Označuje sa znakom „`l`“a ďalším povinným parametrom je n-tica, obsahujúca na prvej pozícii označenie LED diódy. Písmeno `r` onačuje červenú, `g` a `y` označujú zelenú a žltú diódu. Prípadne môže byť na druhej pozícii ešte typ operácie s LED. Tento je označený symbolom `0` resp. `1` pre zhasnutie resp. rozsvietenie danej LED diódy. Pokiaľ tento typ nie je zadaný, rozumie sa ako operácia invertovanie stavu LED. Nasledujú príklady spustenia (register 1 obsahuje n-ticu `(r)`):

- `$(l,(r,1))` – rozsvieti červenú LED diódu.
- `$(l,(y,0))` – zhasne žltú LED diódu.
- `$(l,(g))` – invertuje stav zelenej LED diódy.
- `$(l,&1)` – invertuje stav červenej LED diódy.

6.2.8 Operácia so vstupmi zo senzorov

Jedná sa o asynchrónnu službu, pomocou ktorej je možné získavať, alebo spracovávať dáta zo senzorov. Aktuálne je implementovaná iba práca so senzorom teploty [4]. Služba sa označuje symbolom „`d`“. Pokiaľ je služba spustená bez ďalších parametrov, výsledkom jej spustenia je zistenie aktuálnej teploty. Táto teplota sa uloží vo forme n-tice do časti InputBase. N-tica má pritom tvar: `(s,(teplota))`, kde `teplota` je číslo typu int reprezentujúcu hodnotu senzora.

Službu je možné použiť aj iným spôsobom. A to na zistenie priemeru, maximálnej alebo minimálnej hodnoty z daného počtu predtým nameraných hodnôt. Jednou z funkcií platformy je totiž aj intervalové meranie vstupov senzorov (resp. aktuálne senzora teploty). Tieto hodnoty sú ukladané vo forme log-u do flash pamäte senzorového uzla [4]. Pri využití operácie nad týmito uloženými hodnotami za je symbolom služby ešte dvoj-prvková n-tica. Jej prvý prvok označuje typ operácie s nameranými hodnotami. Môže to byť `m` (minimum), `M` (maximum), alebo `a` (priemer). Druhý prvok tejto n-tice je počet posledne nameraných hodnôt, s ktorými sa má počítať. Pokiaľ nie je namerané dostatočné množstvo hodnôt, operácia zlyhá. Výsledok operácie sa uloží opäť do tabuľky InputBase, vo formáte `(x,(hodnota))`, kde `x` je znak označujúci typ operácie (`m`/`M`/`a`) a `hodnota` je vypočítaný výsledok. Príklady volania operácie sú nasledovné:

- `$(d)` – získa aktuálnu teplotu a vloží ako n-ticu do InputBase vo vyššie spomenutom formáte, napríklad `(s,(25))`.
- `$(d,(M,10))` – do tabuľky InputBase vloží maximum z 10 posledných nameraných hodnôt, napr. `(M,(325))`.
- `$(d,(a,100))` – do tabuľky InputBase vloží priemer zo 100 posledných nameraných hodnôt, napr. `(a,(206))`.

6.2.9 Detekcia susedných uzlov

Táto asynchrónna služba má za účel zistenie dostupných susedných uzlov a sily signálu od týchto uzlov. Služba sa označuje znakom '`n`'. Voliteľne za znakom `n` nasleduje ešte čiarka a symbol `m` [24]. V prvom prípade je chovanie nasledovné. Uzol odošle pomocou broadcast-u správu všetkým dostupným susedom. Správa obsahuje žiadosť na zistenie sily signálu. Na túto správu uzly odpovedia zaslaním špeciálnej správy oznamujúcej silu signálu. Jednotlivé odpovede sa spracujú a výsledok sa uloží do BeliefBase vo forme n-tíc tvaru `(NB,<ID uzla>,<sila signálu>)` [6], kde `<ID uzla>` je adresa uzla v okolí a `<sila signálu>` je hodnota v intervale 0..255 (kde 0 značí minimálny signál).

V druhom prípade je chovanie volania operácie totožné s vyššie popísaným, až na spôsob výstupu. Výsledok sa opäť uloží do tabuľky BeliefBase, avšak vo formáte `(M,<ID>,<ID uzla>,<sila signálu>)`, kde `<ID>` je identifikátor aktuálneho uzla (na ktorom sa operácia volá), `<ID uzla>` je identifikátor okolitého uzla a `<sila signálu>` je opäť číslo spomínaného rozsahu. Tento typ operácie je výhodné použiť, ak sa agent presúva po uzloch, aby bolo zrejmé, na ktorom uzle sa hodnota uložila. Príklady spustenia:

- `$(n)` – do BeliefBase vloží napríklad n-tice `(NB,2,128)`, `(NB,4,65)` a `(NB,6,235)`.
- `$(n,m)` – do BeliefBase vloží napríklad n-tice `(M,3,2,128)` a `(M,3,4,200)`.

6.2.10 Pachové stopy agenta

Operácie s pachovými stopami agenta sú v zásade operácie, ktoré zisťujú predošlý výskyt agentov na danom uzle, na ktorom sú spustené. Pre svoju činnosť nutne potrebujú záznamy v log-u agentov, ktoré sa vytvárajú s príchodom každého agenta. Pomocou tohto mechanizmu je možné stopovať agentov a zisťovať, ktoré uzly pri priechode sieťou navštívili. Tým, že sa implicitne každý výskyt agenta na uzle zaznamenáva, odpadá nutnosť ukladať túto informáciu do BeliefBase agentom, a je možné priamo využívať túto službu na zistenie požadovaných informácií. Služba má označenie „`t`"a môže byť použitá tromi spôsobmi [24].

Prvým z nich je zistenie identifikátora uzla. V tomto prípade je očakávaná za názvom služby platformy `t` jednoprvková n-tica, obsahujúca znak `i`. Výsledkom volania služby je vloženie identifikátora aktuálneho senzorového uzla, na ktorom sa agent nachádza, do aktívneho registra.

Ďalšou možnosťou je zisťovanie, odkiaľ daný agent prišiel, tzv. backtracking [24]. Táto operácia pracuje s log-om uloženým na uzle, na ktorom sa agent práva nachádza. Pri volaní tejto služby nasleduje za identifikátorom služby pachové stopy dvojprvková n-tica, kde prvý prvok je znak `b`, nasledovaný druhým znakom, a to buď `f`, alebo `l`. V prvom prípade sa z log-u vytiahne prvý (najstarší) záznam o výskyte agenta na danom uzle. V druhom prípade naopak posledný (najnovší) záznam o výskyte tohto agenta. Do aktívneho registra následne operácia vloží iba samotný identifikátor uzla, z ktorého agent prišiel (uzatvorený v zátvorkách).

Poslednou možnosťou je dotazovanie na predošlý výskyt agenta na inom uzle [24]. V tomto prípade za identifikátorom služby nasleduje dvoj-prvková n-tica, kde prvým prvkom je znak `q`, a druhým prvkom identifikátor uzla, na ktorý sa chceme dotazovať. Táto služba potrebuje k svojmu fungovaniu využitie rádiovej komunikácie, pomocou ktorej sa odošle špeciálna správa na cieľový uzol. Platforma bežiaca na cieľovom uzle sa následne dotáže na svoj lokálny log agentov, či existuje záznam o agentovi s daným identifikátorom a triedou (ID a trieda agenta sú zapísané v správe, ktorú cieľový uzol prijal). Výsledok dotazu (kladný, alebo záporný) pošle cieľový uzol ako správu uzlu, ktorý sa dotazoval.

Výsledkom je zlyhanie operácie, pokiaľ aktuálny agent na cieľovom už bol. V opačnom prípade sa jednoducho pokračuje na nasledujúcu operáciu. Žiadna hodnota výsledku sa teda nikam nezapisuje. Nasledujú príklady volania operácie pachové stopy:

- `$(t,(i))` - do aktívneho registra uloží identifikátor uzla, na ktorom sa agent práve nachádza, napríklad `(2)`.
- `$(t,(b,f))` - do aktívneho registra uloží identifikátor uzla, z ktorého agent prišiel. Využije sa k tomu najstarší záznam v log-u.
- `$(t,(q,2))` - pošle dotaz na uzol s identifikátorom 2, či daný agent tento uzol navštívil. Ak navštívil, alebo nepríde odpoveď, operácia zlyhá, inak operácia skončí úspešne.

6.2.11 Zisk záznamov log-u

Jedná sa o službu pre zisk záznamov o agentoch, ktorí v minulosti navštívili uzol, na ktorom sa služba spúšťa. Služba sa označuje znakom „`g`". Za týmto znakom nasleduje povinne dvojprvková n-tica, ktorej prvý prvok označuje index, od ktorého sa majú vyberať záznamy, a druhý prvok označuje počet záznamov. Obe hodnoty sa môžu nachádzať v registroch. Avšak v tomto prípade sa predpokladá, že tieto hodnoty sú v registroch uložené ako jednoprvkové n-tice (v zátvorkách).

Výsledok operácie sa uloží do tabuľky BeliefBase, každý záznam vo forme novej n-tice. Tvar týchto záznamov je [6]:

`(LOG,<ID uzla>,<ID od>,<ID agenta>,<trieda agenta>,<počet skokov>)`

pričom význam jednotlivých položiek je nasledovný:

- `LOG` je identifikácia, že sa jedná o záznam log-u.
- `<ID uzla>` je identifikátor aktuálneho uzla, na ktorom sa agent nachádza (resp. kam prišiel).
- `<ID od>` je identifikátor uzla, z ktorého agent prišiel (bol odoslaný).
- `ID agenta` a `trieda agenta` dohromady tvoria identifikáciu samotného agenta.
- `počet skokov` je celkový počet presunov po sieti, ktorý daný agent doposiaľ absolvoval.

Pokiaľ nie je v log-u dostatočný počet záznamov, operácia nezlyhá, ale vloží maximálne možné množstvo týchto záznamov. Predpokladajme, že register 1 obsahuje n-ticu `(2)` a register 2 obsahuje n-ticu `(10)`. Príklady volania operácie sú nasledovné:

- `$(g,(5,3))` – vyberie záznamy log-u od indexu 5, v počte 3, teda záznamy z indexov 5, 6 a 7. Výsledok sa vloží vo forme vyššie popísaných n-tíc do tabuľky BeliefBase.
- `$(g,(&1,&2))` – vyberie záznamy log-u od indexu 2, v počte 10, výsledok vloží opäť do BeliefBase.

6.2.12 Zmena identifikácie agenta

Táto operácia pracuje s identifikátorom a triedou agenta. Označuje sa znakom „`c`“. Využitie operácie je dvojakým spôsobom. Buď na zistenie identifikácie aktuálneho agenta, alebo na jej zmenu [6].

V prípade zistenia identifikácie agenta nasleduje za znakom operácie dvoj-prvková n-tica, kde prvý prvok je znak `g`. Druhým prvkom môže byť znak `i`, v tomto prípade sa jedná o zistenie identifikátora agenta. Alebo je druhým prvkom tejto n-tice znak `c`, kedy sa jedná o zistenie triedy agenta. Výsledok sa uloží do aktívneho registra.

Druhou možnosťou je použitie operácie na zmenu identifikácie agenta [6]. V tomto prípade nasleduje za znakom operácia opäť dvoj-prvková n-tica. Prvý prvok určuje, aká operácia sa má vykonať. Možnosti sú dve. Pokiaľ je tento prvok `i`, bude sa jednať o zvýšenie hodnoty o 1. Ak tento prvok je znak `d`, bude sa jednať o zníženie hodnoty o 1. Druhý prvok n-tice potom určuje, s ktorou hodnotou sa má zvolená operácia vykonať. Pokiaľ je tento prvok `i`, bude sa meniť hodnota identifikátora agenta. Ak je druhým prvkom n-tice znak `c`, meniť sa bude trieda agenta. V týchto prípadoch operácia nevracia žiadnu návratovú hodnotu.

Nasledujú príklady použitia operácie:

- `$(c,(g,i))` – do aktívneho registra uloží identifikátor agenta.
- `$(c,(g,c))` – do aktívneho registra uloží triedu agenta.
- `$(c,(i,i))` – inkrementuje identifikátor agenta.
- `$(c,(d,c))` – dekrementuje triedu agenta.

6.2.13 Operácie so zoznamom

Táto služba platformy implementuje niektoré ďalšie potrebné operácie so zoznamom [6]. Označuje sa znakom „`e`“. V aktuálnej verzii agentnej platformy sú tieto operácie tri.

Prvou je vloženie čísla 0 do aktívneho registra. V tomto prípade nasleduje za znakom operácie jednoprvková n-tica `(z)`

Druhou možnosťou použitia je test na prázdny zoznam. Znak operácie je v tomto prípade nasledovaný dvojprvkovou n-ticou. Prvým prvkom je znak `e`, druhým prvkom je zástupný symbol registra, v ktorom sa má nachádzať testovaný zoznam (n-tica). Použitie zástupného symbolu registra je pritom povinné a nie je možné ho nahradiť priamo testovanou n-ticou v zdrojovom texte plánu. Výsledkom operácie je v tomto prípade jej zlyhanie v prípade, že daná n-tica je prázdna. V opačnom prípade operácia skončí úspešne, ale výsledok sa nikam neukladá.

Poslednou možnosťou je uloženie kópie n-tice do aktívneho registra. V tomto prípade za znakom `e` nasleduje dvoj-prvková n-tica, kde prvý prvok je `c`, druhý prvok je zástupný symbol registra, obsahujúceho n-ticu. Použitie registra je v tomto prípade opäť povinné. Výsledkom operácie je skopírovanie hodnoty zadaného registra do aktívneho registra.

Predpokladajme, že register 2 obsahuje n-ticu `(a,b)`, a register 3 obsahuje n-ticu `()` (prázdny zoznam), aktívny register je register 1. Príklady volania tejto služby platformy sú nasledovné:

- `$(e,(z))` – do aktívneho registra vloží n-ticu `(0)`.
- `$(e,(e,&3))` – vzhľadom k tomu, že register 3 obsahuje prázdnu n-ticu, výsledkom volania služby platformy je zmazanie plánu po zarážku.
- `$(e,(c,&2))` – do aktívneho registra (registra 1) vloží n-ticu `(a,b)`.

6.2.14 Generovanie náhodnej hodnoty

Uvedená služba platformy má označenie „`_`“, (podtržítko). Za týmto označením nasleduje 1-prvková n-tica, ktorej obsah je číslo, značiace maximálnu hodnotu generovania. Služba vygeneruje náhodné číslo v rozsahu 0 až `max`-1, kde `max` je spomínané parametrom zadané číslo. Tento parameter operácie môže byť nahradený aj zástupným symbolom registra. Vygenerované náhodné číslo je uložené do aktívneho registra uzatvorené v zátvorkách. Nech register 1 obsahuje hodnotu `'(20)'`. Nasledujú príklady volania operácie platformy:

- `$(_,(40))` – do aktívneho registra vloží hodnotu z rozsahu 0 až 39, napríklad `(5)`.
- `$(_,&1)` – do aktívneho registra vloží hodnotu z rozsahu 0 až 19, napríklad `(19)`.

6.2.15 Výpočtová operácia platformy

Táto operácia slúži na výpočet aritmetických operácií nad stanovenými operandmi. Použiť je možné binárne, aj niektoré unárne operátory [6]. Operandy tejto operácie môžu byť samostatné hodnoty, aj n-tice hodnôt. Podrobnejší popis bude vysvetlený. Táto operácia platformy sa označuje znakom „`o`“. Za týmto znakom nasleduje čiarka a reťazec, popisujúci zvolený operátor. Ďalej nasledujú n-tice, obsahujúce operandy a prípadne názvy premenných.

Kód operácie	Typ operácie	Výsledok operácie vyjadrený v jazyku C
min	unárna	`-operand`
not	unárna	`(operand == 0)? 1 : 0`
cpy	unárna	`operand`
mul	binárna	`operand1 * operand2`
div	binárna	`operand1 / operand2`
mod	binárna	`operand1 % operand2`
add	binárna	`operand1 + operand2`
sub	binárna	`operand1 - operand2`
les	binárna	`(op1 < op2)? 1 : 0`
mor	binárna	`(op1 > op2)? 1 : 0`
leq	binárna	`(op1 <= op2)? 1 : 0`
meq	binárna	`(op1 >= op2)? 1 : 0`
equ	binárna	`(op1 == op2)? 1 : 0`
neq	binárna	`(op1 != op2)? 1 : 0`
and	binárna	`(op1 && op2 > 0)? 1 : 0`
orr	binárna	`(op1 || op2 > 0)? 1 : 0`

Tabuľka 6.1: Popis výpočtových operátorov plaftormy WSageNt [6].

Unárna operácia

Pokiaľ sa jedná o unárnu operáciu, názov operátora je nasledovaný tromi n-ticami [9]. Prvou n-ticou je samotný operand. Ďalšie dve n-tice sú jednoprvkové, a obsahujú názov vstupnej a výstupnej premennej. Pokiaľ je názov vstupnej premennej prázdna n-tica, predpokladá sa, že operand je priamy, teda n-tica obsahuje priamo číslo symbolizujúce operand. Pokiaľ je názov vstupnej premennej zadaný, musí byť n-tica obsahujúca operand v tvare: `(((názov),číslo),((názov),číslo),...)`, kde `názov` je názov premennej a `číslo` je hodnota, s ktorou sa má počítať. Pokiaľ hodnota `názov` nie je zhodná s názvom vstupnej premennej, je daná n-tica preskočená a pokračuje sa na nasledujúci operand. Nad každým operandom sa vypočíta s číslom príslušná unárna operácia.

Výsledok je vložený do výslednej n-tice. Táto n-tica má opäť tvar `(((názov),číslo),((názov),číslo),...)`, kde `názov` je výstupnej premennej a `číslo` je výsledok výpočtu. Výsledky sa však vkladajú v obrátenom poradí voči vstupnej n-tici. V prípade, že namiesto výstupnej premennej je prázdna n-tica, je výstupom operácie n-tica obsahujúca iba čísla bez názvov premenných. Aktuálna verzia platformy WSagenNt obsahuje tri unárne operácie, významu uvedeného v tabuľke 6.1 (prvé tri riadky tabuľky). Operand aj názov premennej môžu byť uložené aj v registroch. Príklady volania služby výpočtu s unárnym operátorom (register 1 obsahuje n-ticu `(((a),0),((a),3))`, register 2 obsahuje n-ticu `(a)`, aktívny je register 3):

- `$(o,min,(((a),5),((a),10)),(a),(c))` – do aktívneho registra vloží n-ticu `(((c),-10),((c),-5))`.
- `$(o,cpy,&1,&2,())` – do aktívneho registra vloží n-ticu `((3),(0))`.
- `$(o,not,&1,&2,(c))` – do aktívneho registra vloží n-ticu `(((c),(0)),((c),(1)))`.

Binárna operácia

Druhou možnosťou je použitie binárneho operátora. V tomto prípade je operátor nasledovaný piatimi n-ticami. Prvé dve n-tice sú dva operandy, oba operandy majú opäť tvar: `(((názov),číslo),((názov),číslo),...)`, resp. `(číslo)`, pokiaľ sa jedná o priamy operand [9]. Nasledujú dve 1-prvkové n-tice obsahujúce názvy oboch premenných, prípadne prázdne n-tice, pokiaľ sa jedná o priame operandy. Na poslednom mieste je 1-prvková n-tica obsahujúca názov výstupnej premennej. Jednotlivé použiteľné operátory a ich význam sú popísané v tabuľke 6.1. Operandy pri výpočte sú brané v poradí od konca k začiatku n-tíc. Nad každou dvojicou operandov sa vypočíta výsledok pomocou daného operátora. Výsledná n-tica má opäť tvar `(((názov),číslo),((názov),číslo),...)`, kde `názov` je pomenovanie výstupnej premennej a `číslo` je vypočítaná hodnota. Predpokladajme, že register 1 obsahuje n-ticu `(((a),1),((a),2))`, register 2 obsahuje n-ticu `(((b),1))`. Príklady volania binárnej operácie výpočtu:

- `$(o,add,(((a),1),((a),3)),(((b),2),((b),4)),(a),(b),(c))` – do aktívneho registra vloží n-ticu `(((c),7),((c),5),((c),5),((c),3))`.
- `$(o,sub,(10),(2),(),(),())` – do aktívneho registra vloží n-ticu `((8))`.
- `$(o,equ,&1,&2,(a),(b),(c))` – ako operandy použije n-tice v registroch 1 a 2 a do aktívneho registra vloží n-ticu `(((c,0)),((c,1)))`.

6.2.16 Mobilita agenta

Jedná sa o jednu z najdôležitejších služieb platformy. Pomocou nej je totiž možné aktuálneho agenta odoslať na iný uzol. Odoslanie agenta znamená odoslanie všetkých príslušných častí ako tabuľky, registre, aktuálny plán, ID a triedu agenta [4]. Služba sa označuje znakom „`m`“, za ktorým nasleduje n-tica obsahujúca identifikáciu cieľového uzla. Pokiaľ je ID uzla v registri, musí byť uzatvorené v zátvorkách. Voliteľne za ID uzla nasleduje čiarka a znak `s`. Parameter `s` prikazuje platforme, aby po odoslaní agenta zastavila vykonávanie interpretovania na aktuálnom uzle. Pokiaľ sa teda tento parameter neuvedie, bude interpret pokračovať na oboch senzorových uzloch a agent bude existovať v sieti dvakrát.

Nech register 1 obsahuje n-ticu `(3)`. Príklady volania služby pre mobilitu agenta sú nasledovné:

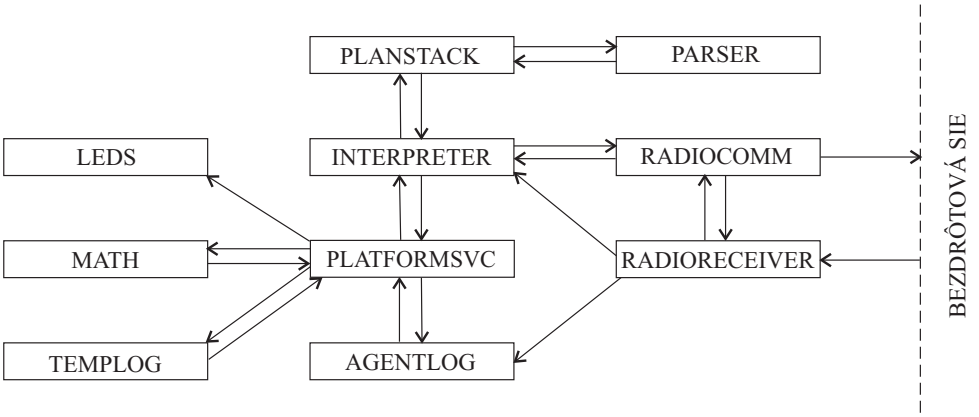
- `$(m,(2))` – odošle kód agenta na uzol s identifikátorom 2, pokračuje v interpretovaní.
- `$(m,(&1,s))` – odošle kód agenta na uzol 3, následne zastaví interpret.

Kapitola 7

Návrh agentnej platformy

Účelom tejto kapitoly je navrhnúť platformu kompatibilnú s platformou `WSageNt`. Ako bolo spomenuté, originálna platforma bola implementovaná v programovom jazyku `nesC`, s využitím prostredia `TinyOS`. V našom prípade bude využitý celkom iný prístup. Je to dané programovacím jazykom, ktorým je v tomto prípade objektovo orientovaný jazyk Java. Taktiež je rozdiel v tom, že na programovanie budú využité priamo knižnice API k platforme SunSpot. Tieto sa priamo dodávajú vo vývojovom kite k senzorovým uzlom.

Po funkčnej stránke musí navrhovaná platforma byť schopná interpretovať všetky príkazy jazyka ALLL uvedené v kapitole 6. Jednotlivé časti platformy budú rozdelené do samostatných tried, ktoré navzájom kooperujú. Toto rozdelenie je možné vidieť na obrázku 7.1. V obrázku boli vynechané triedy pre prácu s tabuľkami a n-ticami, ktoré sú zahrnuté v častiach `Interpreter` a `PlatformSvc`.



Obr. 7.1: Interakcia jednotlivých komponentov platformy.

Hlavnou triedou je trieda `Interpreter`, ktorá bude obsahovať samotný interpret jazyka ALLL. Tento spolupracuje s triedou `PlanStack`, obsahujúcou aktuálny zámer agenta. Pri výbere operácie z aktuálneho plánu sa bude používať trieda `Parser`, starajúca sa o parsovanie aktuálnej operácie a jej parametrov. Služby platformy vzhľadom ku kapitole 6.2 budú umiestnené v triede `PlatformSvc`. O komunikáciu sa budú starať triedy `RadioComm` a `RadioReceiver`, kde v prvej z nich budú umiestnené metódy pre odosielanie správ, agenta, detekciu susedných uzlov a podobne. Druhá menovaná bude slúžiť na spracovávanie všetkých dát došlých z bezdrôtovej siete. O logovanie hodnôt senzora teploty sa bude starať trieda `TempLog`, o uchovávanie záznamov o došlých agentoch potom trieda `AgentLog`. Na

ovládanie LED diód bude slúžiť trieda `Leds`, a na výpočtové operácie trieda `Math`.

7.1 Parser

Základom interpretovania zdrojového textu je čítanie jednotlivých príkazov. Výhodou je, že na čítanie nie je potrebná zložitá lexikálna, či syntaktická analýza. Jazyk ALLL je totiž pomerne jednoduchým jazykom. Neobsahuje výrazy a pri volaní plánov nie je potrebné odovzdávanie parametrov operácie.

Vzhľadom k popisu uvedenému v kapitole 5, aktuálny zámer agenta je možné chápať ako reťazec obsahujúci akcie jazyka ALLL, z ktorého sa akcie postupne čítajú. Začiatok nasledujúcej akcie sa detekuje pomocou znaku operácie. V nasledujúcom texte budeme akcie jazyka ALLL nazývať aj príkazy.

Parser príkazov je navrhnutý ako statická metóda, ktorej sa ako parameter predloží pole obsahujúce sekvenciu príkazov a index, na ktorom má začať. Výsledkom bude priamo načítaný príkaz jazyka ALLL a index, na ktorom skončilo parsovanie. Príkaz ALLL Parser vloží do štruktúry, kde sa nachádza operačný kód a n-tica obsahujúca operandy. Nasledujúca operácia sa tak bude jednoducho čítať z nového indexu. Týmto spôsobom je možné postupne čítať jednotlivé príkazy na zásobníku.

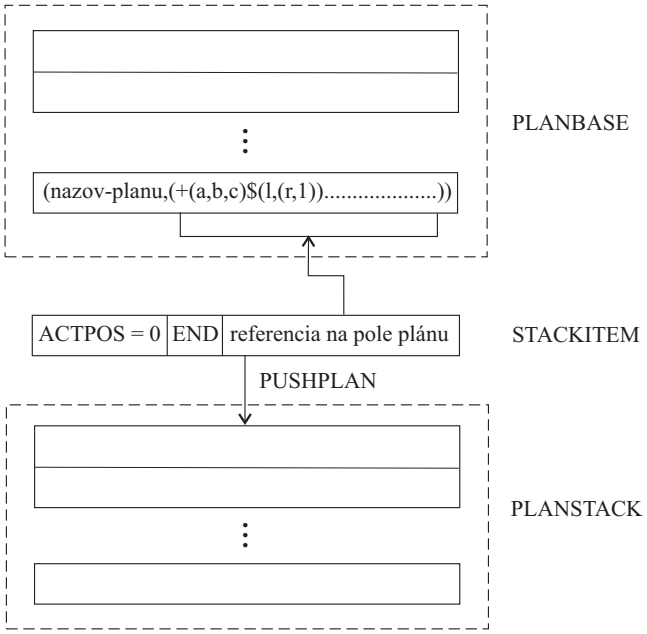
Problémom bolo to, ako sa vysporiadať s výskytom zástupných symbolov registra. Nakoniec bolo rozhodnuté, že kontrola syntaxe a sémantiky príkazu bude prebiehať v dvoch fázach. Prvá fáza je kontrola v samotnom parseri, ktorá obsah registrov neberie do úvahy. Kontroluje sa však správnosť zátvoriek n-tíc, oddeľovacích čiarok, výskyt neočakávaných znakov, správnosť zástupných symbolov registra atp.

Druhá fáza kontroly potom bude nasledovať pri volaní jednotlivých operácií interpreta, či platformy. Každá operácia dostane vždy operandy vo forme n-tice už s nahradenými zástupnými symbolmi registrov. Kontrola jednotlivých parametrov, či kontrola správnosti čísel a podobne, bude prebiehať v tejto fáze. Tým sa zjednoduší implementácia parsera a nebude potrebné použitie zložitého stavového automatu. N-tica, ktorú dostane príslušná operácia ako parameter bude totiž už v tvare dynamickej štruktúry (viac v časti 7.3), u ktorej je možný priamy prístup k jednotlivým položkám. Operandy tak bude možné pred spustením skontrolovať pomerne jednoducho.

7.2 Aktuálny plán

Návrh štruktúry obsahujúcej aktuálny plán agenta bol vytvorený s ohľadom na to, aby nebolo nutné kopírovať príkazy z pomenovaných plánov v `PlanBase` pri operácii nepriameho spustenia (viď časť 6.1.7). Zamedzuje sa tak redundancii dát, kedy každý príkaz je v operačnej pamäti senzorového uzla uložený vždy iba raz. Návrh využíva princíp zásobníka. Ďalej v texte budeme preto túto štruktúru nazývať zásobník plánov, alebo skrátene zásobník.

Pri behu interpreta väčšina akcií vkladania na zásobník pravdepodobne bude práve po volaní spustenia pomenovaného plánu. Operácia vloženia plánu na zásobník vždy vloží iba odkaz na pole, v ktorom je uložený plán. Spolu s plánom bude ďalej uložená aktuálna pozícia v pláne a koncový index poľa plánu. Na obrázku 7.2 je zobrazený princíp vkladania plánu na zásobník. Na obrázku je tiež vidieť štruktúru implementujúcu položku zásobníka (`StackItem`), ktorá obsahuje dva indexy, `actpos` a `end`, značiace aktuálnu pozíciu v pláne a spomenutý koncový index. Treťou položkou štruktúry je potom odkaz na samotný vkladaný plán, ktorý bude referenciou do tabuľky `PlanBase`.



Obr. 7.2: Vkladanie plánu na zásobník.

Toto riešenie však vždy vyžaduje, aby vkladaný plán (sekvencia operácií) bol fyzicky niekde uložený v pamäti. Pokiaľ teda bude potrebné priame vloženie sekvencie akcií na zásobník (typicky pri inicializácii agenta uloženého priamo v zdrojovom kóde interpreta, alebo pri príjme agenta z rádia), bude na to slúžiť zvláštna metóda. Pomocou nej bude možné na zásobník vložiť priamo sekvenciu príkazov jazyka ALLL reprezentovanú ako reťazec (objekt triedy String). Táto metóda vždy naalokuje potrebné pole bajtov, skopíruje príkazy do tohto poľa a vo vyššie spomenutom formáte opäť uloží akcie na zásobník. Jediný rozdiel je, že pole s akciami bude alokované priamo pri vkladaní akcií na zásobník. Toto pole bude existovať v pamäti, pokým sa daná sekvencia príkazov zo zásobníka neodstráni.

Operácia výberu nasledujúceho príkazu z aktuálneho plánu bude podobná operácii `pop` klasického zásobníka. Princíp je ale ten, že sa na plán, ktorý je na vrchole zásobníka zavolá `Parser` na aktuálnej pozícii v pláne. Parser načíta jeden príkaz ALLL a vráti index v poli plánu, na ktorom skončilo parsovanie. Pomocou tohto indexu sa aktualizuje premenná `actpos`. Pri volaní operácie `pop` teda nedôjde k fyzickému vymazaniu operácie z plánu, ale iba posunu indexu na nasledujúcu akciu plánu. Po aktualizácii sa index `actpos` porovná s maximálnym indexom `end`. V prípade, že sa indexy rovnajú (už bol spracovaný celý plán), automaticky dôjde k odstráneniu celej položky zásobníka, obsahujúcej daný plán. Toto odstránenie sa udeje ešte pred skončením operácie `pop`. Pokiaľ napríklad posledný príkaz v pláne na vrchole zásobníka bude volať ten istý plán, na zásobník sa tento plán vloží nanovo, ale ešte pred tým sa stará položka zásobníka odstráni (plán bol spracovaný). Tým sa eliminuje nežiadúci rast zásobníka pri cyklickom volaní plánu na konci toho istého plánu. Výsledkom operácie `pop` zásobníka bude príkaz jazyka ALLL, načítaný parserom. Tento príkaz bude uložený v štruktúre `ALLLCommand`. Obsah tejto štruktúry je zobrazený v tabuľke 7.1.

Operácia pre zmazanie obsahu zásobníka až po zarážku bude fungovať na podobnom princípe ako `pop`, teda dôjde k posunu indexu `actpos`, avšak nebude sa volať `Parser`, ale po znakoch sa prejde pole s plánom až kým sa nenarazí na znak zarážky. Index `actpos` sa

Premenná	Popis
`int opcode`	operačný kód operácie interpreta
`int platformAction`	kód služby platformy v prípade, že daná operácia je služba platformy
`Tuplet operands`	n-tica obsahujúca operandy príkazu
`Register inRegister`	odkaz na register použitý v prípade, že n-tica obsahujúca operandy je zastúpená registrom

Tabuľka 7.1: Štruktúra pre uloženie príkazu ALLL.

posunie na nasledujúci znak za zarážkou. V prípade, že sa dôjde na koniec plánu na vrchole zásobníka, bude odstránená celá aktuálna položka zásobníka a operácia skončí.

7.3 N-tice

Jednou z prvých vecí, ktorá bola pri návrhu riešená, je spôsob implementácie zoznamov, resp. n-tíc v pamäti senzorového uzla. Hlavným cieľom je, aby s n-ticou bolo možné pracovať čím jednoduchšie. Toto zahŕňa vytváranie n-tíc, pridávanie položiek, a v neposlednom rade prístup k jednotlivým prvkom n-tice.

Vo všeobecnosti sa dá n-tica reprezentovať vo forme reťazca, ktorý začína a končí zátvorkou, jednotlivé položky sú oddelené čiarkami, a prípadné vnorené n-tice sú reprezentované rekurzívne týmto istým spôsobom, uzatvorené v zátvorkách. Inou možnosťou, ktorá bude v práci využitá, je n-ticu reprezentovať v pamäti ako zoznam položiek, kde položka môže byť buď priamo reťazec, alebo referencia na vnorenú n-ticu.

Pri návrhu reprezentácie n-tíc sú využité možnosti jazyka Java a objektového prístupu. Cieľom je, aby bol možný jednoduchý priechod cez jednotlivé prvky n-tice. Toto umožní zjednodušenie práce s jednotlivými parametrami operácií, ktoré sú vždy n-ticami. Rovnako je možné vytvárať jednoducho vlastné n-tice, pridávať, či odoberať jednotlivé položky. Presnejší popis je možné nájsť v časti 8.1.

Na reprezentáciu n-tice v jazyku Java Micro Edition bude využitý typ Vector. Tento umožňuje vkladanie položiek všeobecného typu Object (ktorý môže byť pretypovaný na objekt akejkoľvek triedy). Preto sa pre túto reprezentáciu špeciálne hodí. N-tica bude teda reprezentovaná kolekciou týchto objektov, kde jednotlivé položky sú buď typ String, alebo odkaz na vnorenú n-ticu. Použitie typu String je výhodné z hľadiska jednoduchosti použitia (a tiež potrebné, keďže reťazcové konštanty sú automaticky tohto typu). Avšak z hľadiska pamäťovej náročnosti vzhľadom k obmedzenej pamäti senzorového uzla nie je vhodný. Bolo tak rozhodnuté na základe článku zaoberajúceho sa pamäťovou náročnosťou jednotlivých objektov jazyka Java, viac je možné nájsť v [8]. Preto je možné použiť typ String na vkladanie položiek n-tice, avšak do pamäte n-tice sa uloží jeho kópia vo forme poľa typu bajtov. Preto pri vkladaní reťazca typu String sa naalokuje pole bajtov (typ `byte`). Pri výbere daného prvku n-tice bude naopak vytvorený nový objekt typu String, do ktorého sa skopíruje obsah bajtového poľa, v ktorom je daný prvok uložený. Pokiaľ je vyberaný prvok vnorená n-tica, pri výbere tohto prvku sa vytvorí fyzická kópia celej tejto n-tice.

Jednou z dôležitých operácií je porovnanie n-tice so vzorovou n-ticou (unifikácia). Táto operácia je navrhnutá ako cyklus cez všetky prvky tejto n-tice, kde sa postupne porovnáva jej obsah so vzorovou n-ticou. Porovnávanie sa bude vykonávať podľa definície unifikácie, uvedenej v časti 5.1.3.

7.4 Registre

Register je navrhnutý ako dátová štruktúra, ktorá obsahuje buď n-ticu, alebo atomickú hodnotu (resp. reťazec). Fyzicky bude obsahovať premenné pre uloženie oboch typov hodnôt, pričom použitá bude vždy len jedna z nich. O nastavení danej premennej sa rozhodne na základe typu hodnoty, ktorá je do registra vkladaná. Pri vkladaní hodnoty do registra sa automaticky vytvorí kópia danej n-tice, alebo reťazca. Táto kópia bude potom obsahom registra.

7.5 Tabuľky

Ako už bolo spomenuté, tabuľky v jazyku ALLL slúžia na ukladanie dát troch skupín. Jednak sú to pomenované plány, ktoré je možné spustiť (tabuľka PlanBase), potom vstupy zo senzorov a správy (tabuľka InputBase) a nakoniec báza znalostí (BeliefBase). Položkami týchto tabuliek sú vždy n-tice.

Tabuľka je navrhnutá ako kolekcia n-tíc vo formáte uvedenom vyššie. V jazyku java bude pre tento účel opäť využitý typ Vector. Zvažované bolo tiež použitie napríklad hashovacej tabuľky. Avšak tu by bol problém so správnym vyhľadávaním n-tíc v tabuľke. Unifikovaná n-tica totiž, ako bolo spomenuté, môže obsahovať anonymné premenné, ktoré znemožňujú priame vyhľadanie na základe vypočítanej hash hodnoty.

V praxi teda bude tabuľka obsahovať referencie na objekty reprezentujúce jednotlivé n-tice. Trieda pre prácu s tabuľkami je navrhnutá tak, aby umožňovala všetky operácie s tabuľkou uvedené v kapitole 6.1. Jedná sa napríklad o unifikáciu n-tíc, a s tým spojené vyhľadávanie, či mazanie unifikovaných n-tíc. Operácia unifikácie n-tice v tabuľke je navrhnutá ako cyklus cez všetky n-tice v tabuľke, kde u každej z nich sa zavolá operácia porovnania s hľadanou n-ticou. Pri vkladaní n-tice do tabuľky sa najprv vykoná pokus o jej unifikáciu. K vloženiu dôjde iba v prípade, že je unifikácia neúspešná.

Spôsob vkladania pomenovaných plánov do tabuľky sa bude odlišovať od bežného pridávania n-tíc do tabuľky. V tomto prípade je plán vlastne dvoj-prvková n-tica, kde na prvej pozícii je meno plánu a na pozícii druhej potom n-tica obsahujúca reťazec príkazov plánu. Tento reťazec bude reprezentovaný poľom bajtov (typ `byte[]`) a uložený celý ako jeden prvok n-tice. Operácia pridania plánu do tabuľky dostane ako parameter namiesto n-tice reťazec typu String, obsahujúci plán vo forme n-tice reprezentovanej v čitateľnej podobne. Z tohto reťazca sa vytvorí objekt reprezentujúci n-ticu v tvare popísanom v časti 7.3. Na jej prvej pozícii bude prvok typu `byte[]` značiaci názov plánu, a druhý prvok bude vnorená n-tica s jedným prvkom. Týmto prvkom je reťazec príkazov ALLL, rovnako vo forme poľa typu `byte[]`. Pomocou tejto operácie bude možné inicializovať bázu plánov priamo v zdrojovom kóde platformy.

7.6 Interpret

Interpretovanie je možné chápať ako nekonečnú slučku kódu, kde sa postupne vyberajú príkazy, ktoré sú na zásobníku (v aktuálnom pláne) a tieto sa po jednom spúšťajú. Originálna platforma bola vyvíjaná ako dve nezávislé časti, služby interpreta a služby platformy. S ohľadom na to je navrhnuté rozdelenie implementácie služieb interpreta a platformy do dvoch nezávislých tried. Z praktického hľadiska však nie je potrebné zložité odovzdávanie riadenia medzi týmito dvomi časťami. Toto rozdelenie má hlavne logický charakter.

Pseudokód interpretovania zdrojového textu je nasledovný:

```
while not(stack.epmty()) do
   command = stack.pop(); // výber príkazu zo zásobníka
   if(command.operation == PLATFORM_ACTION)
      PlatformSvc.execute(command); // služba platformy
   else
      Interpreter.execute(command); // služba interpreta
```

Výsledkom operácie pop zásobníka bude štruktúra `ALLLCommand`, ktorá bola uvedená v kapitole 7.2. Túto štruktúru vždy ako parameter dostane daná služba, ktorá sa má spustiť. Služby interpreta budú súčasťou triedy `Interpreter`. Služby platformy sú potom implementované zvlášť v triede `PlaftormSvc`. Detaily budú popísané v kapitole 8.13.

7.7 Operácie s LED diódami

Ako bolo spomenuté v kapitole 3.1, celkovo sa na uzle SunSpot nachádza osem LED diód. Každá z nich môže mať nastavenú akúkoľvek farbu. Farba LED diód sa nastavuje pomocou intenzít jednotlivých farebných zložiek RGB modelu.

Keďže originálna platforma WSagent pre uzly Iris a MICAz počíta iba s tromi LED, je potrebné ich použitie emulovať na poli LED diód platformy `SunSpot`. Na tento účel sa použijú prvé 3 z 8 LED uzla `SunSpot`. Ich farby (červená, zelená, žltá) budú nastavené pri štarte agentnej platformy. Pre prácu s LED bude slúžiť trieda `Leds`, pričom k jednotlivé operácie s nimi budú využívať volania API, popísané v časti 3.5.3.

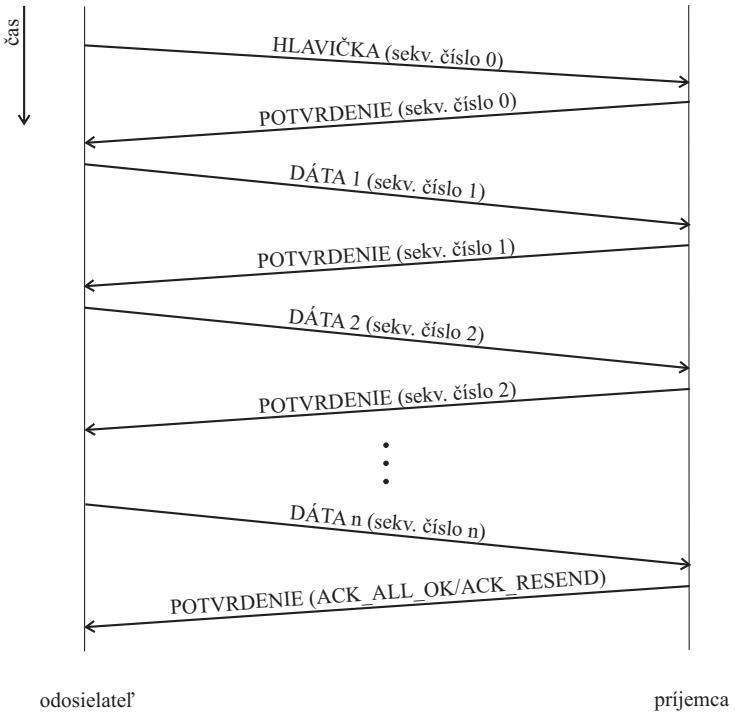
7.8 Komunikácia

Na komunikáciu bude využitá trieda `RadioGram`, ktorej popis bol uvedený v časti 3.5.4. Táto trieda používa na komunikáciou protokol typu klient-server. Zvažované bolo tiež použitie triedy `RadioStream`, ktorá na rozdiel od triedy `RadioGram` zaručuje správne doručovanie dát medzi uzlami. Avšak táto trieda sa nehodí pre implementáciu služieb ako napríklad zisťovanie sily signálu od susedov, pretože nepodporuje komunikáciu typu broadcast.

7.8.1 Komunikačný protokol

Pri návrhu služieb platformy pre komunikáciu sa zohľadňuje predpoklad, že dostupná komunikačná linka je nespoľahlivá. Pre správne doručovanie dát medzi senzorovými uzlami bol navrhnutý komunikačný protokol, ktorý toto zohľadňuje a zaisťuje správne doručenie odosielaných dát príjemcovi. Návrh predpokladá odosielanie dát reprezentovaných poľom bajtov o určitej veľkosti. Maximálna veľkosť dát, ktoré je možné odosielať pomocou metódy `send` triedy `RadioStream` je obmedzená na veľkosť cca 1100 bajtov. Takáto dĺžka by však mohla spôsobovať problémy a v prípade chýb nutnosť opätovne zasielať veľké množstvá dát. Maximálna dĺžka správy odoslanej platformou je preto stanovená na 100 bajtov. Komunikačný protokol je zobrazený na obrázku 7.3.

Majme teda dáta reprezentované poľom bajtov v pamäti. Prvým krokom pred odoslaním je rozdelenie dát na množinu menších celkov, v závislosti na maximálnej dĺžke správy. Prakticky to znamená vypočítať, koľko dátových častí sa bude odosielať. Každá dátová časť bude opatrená príslušným sekvenčným číslom, aby nedošlo k prijatiu dátových častí v nesprávnom poradí, alebo duplicitne.



Obr. 7.3: Komunikačný protokol.

Pred odosielaním dát sa odošle hlavička dát. Táto má sekvenčné číslo 0. Jej obsahom je kontrolný súčet vypočítaný z všetkých dát, ktoré sa majú odoslať. Ďalej hlavička obsahuje celkový počet bajtov, ktoré sa budú odosielať. Tieto informácie potrebuje príjemca, aby mohol správne detekovať koniec dát a skontrolovať, že dáta boli v poriadku prijaté.

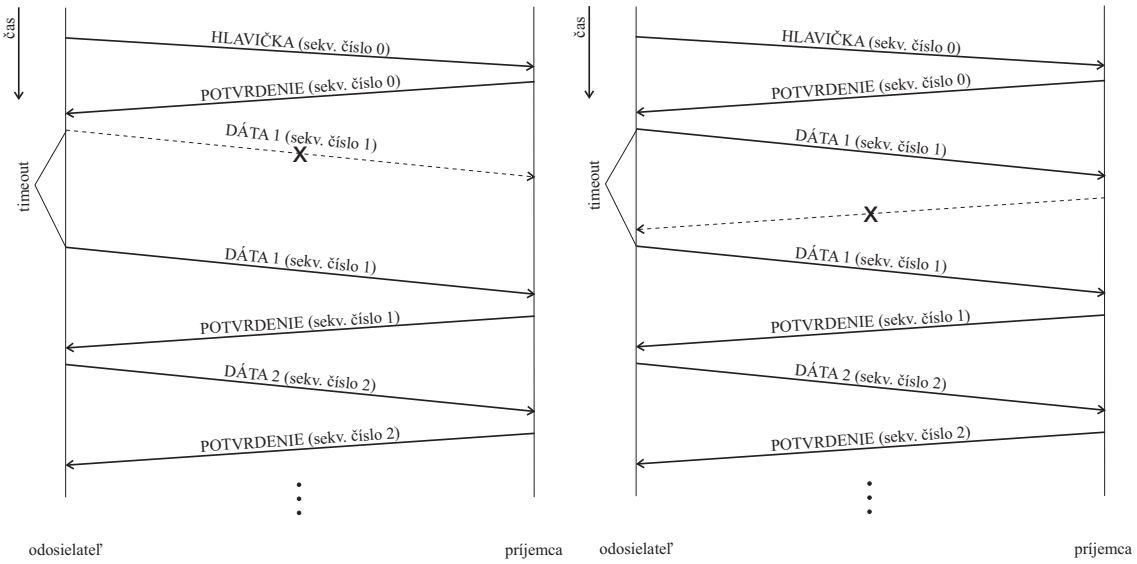
Každá časť správy, či už hlavička, alebo dátové časti, sa zvlášť potvrdzuje. Potvrdenie odosiela príjemca akonáhle prijme danú časť. Potvrdenie obsahuje sekvenčné číslo, aby bolo zrejmé, ktorá časť správy sa potvrdzuje. Po prijatí všetkých dát príjemca pošle potvrdenie `ACK_ALL_OK` v prípade, že dáta prišli v poriadku, alebo `ACK_RESEND`, ak nesedí kontrolný súčet prijatých dát a je potrebné správu poslať znovu.

Odosielateľ po odoslaní danej časti správy vždy čaká požadovanú dobu, po ktorú musí prísť potvrdenie danej časti. Pokiaľ sa tak nestane, automaticky sa daná časť odosiela znovu. Reakcia na prípad, kedy sa stratí správa, alebo jej potvrdenie, je teda rovnaká (viď obrázok 7.4). Ak príjemca dostane jednu časť správy viackrát, informuje odosielateľa, že danú časť už obdržal opätovným odoslaním potvrdenia.

7.8.2 Typy správ

V časti 7.8.1 bol predstavený komunikačný protokol, pomocou ktorého je možné zasielať správy uzlom a odosielať celých agentov. Platforma však bude potrebovať ešte iné typy správ, slúžiace napríklad pre zisťovanie topológie, či dotazy na pachové stopy. Prehľad všetkých typov správ, ktoré budú v platforme použité, je možné vidieť v tabuľke 7.2.

Správy pre prenos väčšieho objemu dát sú `DATA_AGENT` a `DATA_MESSAGE`. Ostatné typy sa budú používať na jednoduché správy, ako napríklad potvrdenia, či zisťovanie topológie siete. Pri odosielaní týchto správ sa vytvorený komunikačný protokol nebude používať.



Obr. 7.4: Reakcia na stratu dátovej správy (vľavo), alebo potvrdenia (vpravo).

Typ správy	**Popis**
`DATA_MESSAGE`	dáta klasickej správy (n-tica odosielaná príkazom `!`)
`DATA_AGENT`	dátová správa obsahujúca jednotlivé časti agenta
`ACK`	potvrdenie prijatia jednej časti správy
`ACK_ALL_OK`	potvrdenie úspešného prijatia celej správy
`ACK_RESEND`	informácia nutnosti znovuzaslania celej správy
`SIGNAL_GET`	žiadosť o zaslanie sily signálu (u príkazu `$(n)`)
`SIGNAL_GET_M`	žiadosť o zaslanie sily signálu (u príkazu `$(n,m)`)
`SIGNAL_RECEIVED`	správa oznamujúca silu signálu (u príkazu `$(n)`)
`SIGNAL_RECEIVED_M`	správa oznamujúca silu signálu (u príkazu `$(n,m)`)
`TRACE_REQUEST`	dotaz na predošlý výskyt agenta na danom uzle
`TRACE_ANSWER`	odpoveď na dotaz `TRACE_REQUEST`

Tabuľka 7.2: Typy správ agentnej platformy.

7.8.3 Rozdelenie do tried

Ako bolo spomenuté v časti 3.5.4, pri súčasnom otvorení klientského a serverového spojenia sa prichádzajúce pakety automaticky objavujú v serverovom spojení. Z toho vyplýva návrh rozdelenia implementácie komunikácie do dvoch tried. Prvou je `RadioComm`, ktorá bude mať na starosti klientskú stranu spojenia. Táto trieda bude slúžiť na odosielanie dátových správ, agentov, zisťovanie okolitých susedov a dotazy na pachové stopy agenta.

Na spracovávanie prijatých dát potom bude slúžiť trieda `RadioReceiver`. Táto bude obsluhovať serverové spojenie. V nej pobeží samostatné vlákno, ktoré bude prijímať všetky prichádzajúce pakety, vrátane potvrdení dátových správ. Spracovávanie prijatých dát bude postavené na nekonečnom cykle, ktorý volá metódu `receive` triedy `RadioGram`.

7.9 Použitie flash pamäte

Ako bolo uvedené v časti 3.5.5, s flash pamäťou uzla SunSpot je pomocou nainštalovaného API možné pracovať dvomi spôsobmi. Jedným je použitie triedy `FlashFile`, druhým je použitie triedy `RecordStore`. Prvý zmieňovaný spôsob umožňuje zápis podobne ako do klasického súboru v jazyku Java. Zápis a čítanie sa v tomto prípade vykonáva pomocou vstupného a výstupného streamu. Tento spôsob sa javil ako vhodný pre použitie. Avšak bolo zistené, že v simulačnom nástroji `Solarium` tento spôsob nie je podporovaný. Pre vývoj sa preto nehodil. Bolo by totiž nevýhodné odlaďovať prácu na fyzických uzloch. Z toho dôvodu bol zvolená práca s flash pomocou triedy `RecordStore`. Popis tejto triedy bol uvedený v časti 3.5.5.

Do pamäte flash sa budú ukladať záznamy logov. Aktuálne sú v agentnej platforme potrebné logy dva, a to log vstupov zo senzorov, a log prichádzajúcich agentov. Dôvody na ukladanie práve do flash pamäte sú dva. Jednak je to šetrenie miesta v operačnej pamäti. Ďalej je to perzistencia týchto dát. Logy sú totiž navrhnuté tak, aby bolo možné s dátami pracovať aj po vypnutí, či reštartovaní senzorového uzla. Oba logy boli navrhnuté ako kruhové, a teda pri dosiahnutí maximálnej veľkosti logu sa kruhovo budú prepisovať staré záznamy novými.

7.9.1 Log agentov

Log agentov bude umiestnený v triede `AgentLog`. Aktuálna veľkosť logu je zvolená na 16KB. Fyzicky bude log uložený v pamäti flash pod názvom `agent_log`. Zo špecifikácie použitej triedy `RecordStore` vyplýva, že tento log bude existovať do zmazania triedy `Midlet` implementujúcej agentnú platformu.

Každý záznam o agentovi bude pozostávať zo štyroch 16-bitových čísel. Konkrétne sa jedná o identifikátor uzla, z ktorého agent prišiel, ID, triedu agenta a počet presunov agenta. Ako bolo spomenuté, tento log bude kruhový. Znamená to, že pri jeho zaplnení sa najstaršie záznamy budú prepisovať novými. Preto bude nutné ukladať informáciu o tom, kde aktuálne začína log, t.j. index najstaršieho záznamu. Pred vypnutím uzla sa táto hodnota uloží vždy do flash pamäte.

7.9.2 Log senzorov

Trieda `TempLog` bude slúžiť na logovanie hodnôt nameranej teploty. Veľkosť log-u je stanovená na 32KB. Dáta sa budú fyzicky ukladať do súboru s názvom `temp_log`. Na rozdiel od log-u agentov sa u každého záznamu bude ukladať iba jedna hodnota, a to 32-bitová hodnota senzora (typu `int`). Aktuálne sa pri návrhu počíta iba s logovaním hodnôt teploty, ale použitie tejto triedy bude možné pre hodnoty ktoréhokoľvek senzora.

Hodnoty sa do flash pamäte nebudú zapisovať priamo. Na zápis bude slúžiť buffer, do ktorého sa bude ukladať zvolené množstvo záznamov. Tento počet je zvolený na hodnotu 32. K zápisu do flash vždy dôjde až po naplnení tohto buffera. Tým sa eliminuje príliš veľký počet prístupov do flash pamäte.

Prístup do flash sa bude bufferovať aj pri výbere hodnôt log-u. Tým, že každý záznam obsahuje 32 hodnôt, pri prístupe na daný záznam sa tento celý uloží do operačnej pamäte. Pokiaľ bude nasledujúci prístup na hodnotu, ktorá spadá na ten istý záznam vo flash, použije sa hodnota zo záznamu v bufferi. Mechanizmus kruhového zápisu je podobný, ako u triedy `AgentLog`. Detaily implementácie budú uvedené v časti 8.10.

Kapitola 8

Popis implementácie

Rozdelenie do jednotlivých tried prebehlo na základe návrhu popísaného v kapitole 7. Táto kapitola popisuje implementáciu jednotlivých tried a ich metód.

8.1 Práca s n-ticami

Trieda implementujúca n-ticu bola pomenovaná `Tuplet`. Každá n-tica, s ktorou interpret pracuje, je vždy objektom tejto triedy. Táto trieda obsahuje metódy pre všetky potrebné operácie s n-ticami, prehľad tých najdôležitejších je v tabuľke 8.1.

Názov metódy	**Popis**
`void add(Object item)`	pridá do n-tice položku `item`, t.j. buď n-ticu, alebo reťazec
`Object get(int i)`	vráti i-tu položku n-tice (reťazec, alebo n-tica)
`Object get_reference(int i)`	vráti referenciu na i-tu položku n-tice (typ `Tuplet`, alebo `byte[]`)
`void remove(int i)`	odstráni i-tu položku n-tice
`boolean isEqual(Tuplet t)`	porovná aktuálnu n-ticu s n-ticou `t`, pričom n-tica t môže obsahovať anonymné premenné
`boolean replaceRV()`	nahradí výskyty zástupných symbolov registra v n-tici hodnotami týchto registrov
`Object first()`	vráti prvý prvok n-tice
`Tuplet rest()`	vráti zvyšok n-tice okrem prvého prvku

Tabuľka 8.1: Dôležité metódy triedy Tuplet.

Pre výber daného prvku n-tice slúži metóda `get`. Výsledkom výberu je buď objekt typu `String`, alebo `Tuplet`, ak sa jedná o vnorenú n-ticu. Vždy sa jedná o fyzickú kópiu prvku. Návratovou hodnotou metódy je typ `Object`. Typ návratovej hodnoty, pokiaľ je to potrebné, sa rozlišuje pomocou metódy `instanceof` jazyka Java. Pre prístup priamo k položkám n-tice bez ich kopírovania potom slúži metóda `get_reference`, ktorá vráti iba odkaz na danú položku. Návratová hodnota v prípade, že sa jedná o atomickú hodnotu je potom typ `byte[]`, inak je to typ `Tuplet`. Je to dané spôsobom uloženia dát n-tice, ktorý bol popísaný v časti 7.3.

Zvlášť stojí za zmienku metóda `isEqual`. Táto slúži pre operáciu unifikácie. Ako parameter metóda dostane vzorovú n-ticu, ktorá môže obsahovať anonymné premenné. Metóda

vráti `true`, ak je možné danú n-ticu unifikovať so vzorovou n-ticou, inak vráti `false`.

8.2 Práca s tabuľkou

Dáta tabuľky sa ukladajú ako kolekcia objektov triedy `Tuplet`. V jazyku java je pre tento účel opäť využitý typ Vector. Trieda implementujúca tabuľku bola pomenovaná `Table`. Táto trieda poskytuje metódy pre všetky operácie s tabuľkou, ktoré používa jazyk ALLL.

Prehľad dôležitých metód triedy je v tabuľke 8.2. Unifikácia n-tíc sa využíva pomocou metód `findAll` a `removeAll` pre vyhľadanie resp. vymazanie všetkých n-tíc vyhovujúcich zadanej n-tici. Operácie pracujú tak, že v cykle porovnávajú všetky uložené n-tice pomocou metódy `isEqual` triedy `Tuplet`. Pri vyhľadávaní je vytvorená nová n-tica, ktorá obsahuje všetky tieto n-tice, pri mazaní sú tieto n-tice postupne odstránené.

Na pridávanie pomenovaného plánu slúži metóda `addPlan`. Táto dostane ako parameter namiesto n-tice reťazec typu String, ktorý obsahuje plán vo forme n-tice reprezentovanej v čitateľnej podobne.

Názov metódy	**Popis**
`boolean add(Tuplet t)`	pridá n-ticu `t` do tabuľky
`void remove(int i)`	odstráni n-ticu s indexom `i` z tabuľky
`void removeAll(Tuplet t)`	odstráni všetky n-tice unifikovateľné s n-ticou `t`
`Tuplet findAll(Tuplet t)`	vráti všetky n-tice unifikovateľné s n-ticou `t`
`void addPlan(String s)`	pridá do tabuľky plán reprezentovaný n-ticou v čitateľnej podobe v reťazci

Tabuľka 8.2: Dôležité metódy triedy Table.

8.3 Registre

Na prácu s registrami bola implementovaná trieda `Register`, ktorá obsahuje metódy pre nastavovanie hodnoty, získanie aktuálnej hodnoty a výpis registra do reťazca (viď tabuľka 8.3). Výpis je využívaný pri odosielaní agenta, alebo napríklad pre ladiace účely.

Názov metódy	**Popis**
`SetValue(String s)`	nastaví hodnotu registra - v registri bude priama hodnota
`SetValue(Tuplet t)`	nastaví hodnotu registra - v registri bude n-tica
`Object getValue()`	vráti aktuálnu hodnotu registra (n-tica, alebo reťazec)
`String printToString()`	vypíše hodnotu registra do reťazca

Tabuľka 8.3: Dôležité Metódy triedy `Register`.

8.4 Parsovanie zdrojového textu

Parsovanie kódu jazyka ALLL je implementované v triede `Parser`. Metóda, ktorá sa pri parsovaní volá, má nazov `parse`. Parametrami metódy sú pole bajtov, nad ktorým má pracovať, index v tomto poli, kde sa má začať, a štruktúra pre uloženie výsledného príkazu. Príkaz jazyka ALLL je pritom definovaný štruktúrou `ALLLCommand`, ktorej popis bol uvedený

v časti 7.2. Metóda `parse` okrem naplnenia štruktúry `ALLLcommand` tiež vráti index v poli, na ktorom skončilo parsovanie. Tento index sa využije pri potrebe parsovať nasledujúci príkaz plánu. Jedno volanie tejto metódy vždy vráti jeden príkaz jazyka ALLL.

Táto metóda používa k svojej činnosti metódu `readTuplet`. Táto má za úlohu na aktuálnej pozícii v kóde načítať n-ticu do štruktúry `Tuplet`. Je to dané tým, že všetky akcie jazyka ALLL majú operandy uzatvorené v zátvorkách. A práve parsovanie n-tice (a kontrola zátvoriek, čiarok medzi zátvorkami atď.) vykonáva táto pomocná metóda. Výsledkom metódy `readTuplet` je vytvorený objekt typu `Tuplet`.

V prípade, že nastane pri parsovaní chyba, typicky pri chybnej syntaxi zdrojového kódu, metóda `parse` vyhodí výnimku. Táto výnimka obsahuje operáciu, na ktorej parsovanie zlyhalo, index operácie v poli, a prípadne operáciu platformy. Tieto údaje je následne možné použiť pre výpis chyby, aby sa ľahšie lokalizovala.

8.5 Zásobník plánov

Implementácia zásobníka plánov je v triede `PlanStack`. Teoretický návrh bol spomenutý v časti 7.2. Táto trieda implementuje aktuálny zámer agenta. Prehľad metód tejto triedy je v tabuľke 8.4.

Názov metódy	**Popis**
`void pushPlan(Tuplet plan)`	vloží na zásobník plán (n-tica `plan` pochádza z tabuľky `PlanBase`)
`void push_str(String s)`	vloží na zásobník postupnosť príkazov ALLL obsiahnutých v reťazci `s`
`ALLLCommand pop()`	vráti príkaz ALLL zo začiatku zásobníka
`void popToBP()`	zmaže zo zásobníka všetky príkazy až po zarážku
`boolean empty()`	test na prázdny zásobník
`String printToString()`	vypíše obsah zásobníka do reťazca a jednotlivé plány oddelí zarážkami

Tabuľka 8.4: Dôležité Metódy triedy `PlanStack`.

Na vkladanie plánu na zásobník slúži metóda `pushPlan`. Parametrom metódy je pritom referencia na n-ticu v tabuľke `PlanBase`, ktorá obsahuje daný plán. Pri vkladaní na zásobník sa vytvorí štruktúra `StackItem`, popísaná v časti 7.2. Index `actpos` sa inicializuje na hodnotu 0. Index `end` sa inicializuje na veľkosť poľa obsahujúceho daný plán v `PlanBase` zníženú o 1. Položka `plan`, tj. odkaz na pole plánu sa inicializuje vložením referencie na pole plánu. Takto inicializovaná štruktúra sa potom vloží na vrchol zásobníka plánov.

Na priame vloženie sekvencie príkazov na zásobník slúži metóda `push_str`, ktorá funguje na podobnom princípe ako `pushPlan`, ale pole plánu je v tomto prípade obsiahnuté priamo v reťazci, ktorý je parametrom metódy. Presnejší popis bol uvedený v časti 7.2.

Metóda zásobníka `pop` pracuje v súčinnosti s triedou `Parser`. Návratovou hodnotou je totiž štruktúra `ALLLCommand`, ktorej obsah bol uvedený v tabuľke 7.1. Túto štruktúru vracia operácia `parse` triedy `Parser`. Preto reálne pri operácii `pop` sa na danom poli plánu a aktuálnom indexe zavolá Parser, ktorý vráti aktuálnu operáciu. Táto je rovnako návratovou hodnotu operácie `pop` zásobníka. Trieda `PlanStack` si tiež v statickej premennej `plan` udržuje názov aktuálne volaného plánu. Tento sa použije v prípade chyby, aby bolo možné vypísať pozíciu aktuálnej chyby a názov plánu.

Na výpis obsahu zásobníka do reťazca príkazov slúži metóda `printToString`. Táto vypíše všetky doposiaľ nespracované časti plánov vložených na zásobník, pričom ich oddelí zarážkami. Metóda sa používa pri odosielaní agenta, tiež je možné ju využiť napríklad pri ladiacich výpisoch.

8.6 Výstup na LED

Výstup na LED je implementovaný v triede `Leds`. Táto trieda obsahuje metódy pre rozsvietenie, zhasnutie, či zmenu stavu zvolenej LED diódy. Parametrom je vždy znak, ktorý určuje danú LED. Tento znak je `r`, `g`, alebo `y` pre označenie červenej, zelenej, alebo žltej LED. Návrh triedy bol popísaný v časti 7.7. Prehľad implementovaných metód pre prácu s LED je v tabuľke 8.5.

Trieda `Leds` obsahuje tiež premennú `brightness`. Tá udáva intenzitu, akou majú LEDky svietiť. Hodnota je od 0 do 255, pričom viditeľne LED diódy svietia od intenzity okolo 20. Účel zníženia intenzity je najmä šetrenie batérie senzorového uzla. Predvolená hodnota je 255.

Názov metódy	**Popis**
`setOn(char led)`	rozsvieti zvolenú LED diódu
`setOff(char led)`	zhasne zvolenú LED diódu
`invert(char led)`	zmení stav LED na opačný

Tabuľka 8.5: Metódy triedy `Leds`.

8.7 Výpočtové operácie

Výpočtové operácie platformy boli implementované vzhľadom k popisu v časti 6.2.15. Pre tento účel slúži trieda `Math`. jedinou verejnou metódou tejto triedy je `compute`. Táto metóda očakáva ako parameter n-ticu (objekt triedy `Tuplet`) v tvare uvedenom v časti 6.2.15. V závislosti na tom, či sa jedná o operáciu s binárnym, alebo unárnym operátorom, zavolá sa jedna z privátnych metód `compute_unary`, alebo `compute_binary`. Na čítanie operandov používa každá z nich metódu `getOperand`. Keďže operandy sú vždy n-tica, táto metóda slúži vždy na načítanie samotného čísla, reprezentujúceho daný operand.

Pre každý operand u unárnej operácie sa následne zavolá metóda `eval_unary`, ktorá vráti číselnú hodnotu výsledku. Podobne sa deje u binárnej operácie pre každý pár operandov. Na tento účel slúži metóda `eval_binary`. Po vypočítaní hodnoty sa každá výsledná hodnota pridá do výslednej n-tice, ktorá má tvar uvedený v časti 6.2.15. Výsledok metódy je n-tica (objekt triedy `Tuplet`) obsahujúca výsledok v potrebnom tvare.

V prípade, že nastane chyba výpočtu, metóda vyhodí výnimku. Výnimka môže byť vyvolaná buď nesprávnym prevodom reťazca obsahujúceho číslo na číselnú hodnotu, prípadne chybným formátom niektorej z n-tíc obsahujúcich operandy, alebo názvy premenných. Tieto výnimky je potrebné ošetriť a spracovať pri interpretovaní.

8.8 Rádiová komunikácia

Jednou z najdôležitejších častí agentnej platformy je implementácia rádiovej komunikácie. Pomocou nej sú implementované služby pre zisťovanie susedných uzlov, odosielanie správ a

presun agentov. O komunikáciu sa starajú dve triedy `RadioComm` a `RadioReceiver`. Komunikácia prebieha na stanovenom porte (nastaviteľnom v triede `RadioComm`), aktuálne 111 a každý paket odoslaný z platformy je ošetrený identifikátorom o veľkosti 4 bajty, ktorého hodnota je v premennej `SPOTCOMM_IDENTIFIER`. Tým je zaručené, že nedôjde k prijímaniu iných, než platforme určených paketov.

8.8.1 Adresovanie uzlov

V rámci kompatibility s platformou `WSageNt` sa senzorové uzly adresujú pomocou 16, resp. 32-bitových dekadických čísel. Formát adresy uzla SunSpot je však 64-bitový (ako bolo spomenuté v časti 3.1). Preto je potrebný prevod adries pri komunikácii. Využíva sa fakt, že prefix adresy (prvých 32 bitov) je pre každý fyzický uzol rovnaký. Nasledujúcich 16 bitov sú väčšinou samé nuly, ale dá sa predpokladať, že u novších výrobných sérií bude využitá aj táto časť adresy. Posledných 16 bitov je potom vždy unikátna adresa uzla. Momentálne je možné používať len 16-bitové adresovanie uzlov, pričom zvyšok adresy sa automaticky doplní nulami. Virtuálne uzly používané v nástroji Solarium majú prefix odlišný od fyzických uzlov, ale v rámci virtuálnych uzlov je tiež nemenný. Tieto prefixy sú nastavené priamo v zdrojových kódoch platformy. Virtuálne uzly majú hodnou na posledných 16 bitoch adresy nižšiu než fyzické uzly. Tým je možné odlíšiť virtuálne od fyzických uzlov. Pokiaľ je teda zadaná adresa uzla 16, resp. 32-bitovým číslom, výsledná adresa sa vytvorí spojením prefixu pre daný typ uzla a tejto adresy.

Špeciálnym prípadom je adresa 1, čo je symbolická adresa základňovej stanice. Trieda `RadioComm` obsahuje premennú `basestation_address`, do ktorej je potrebné nastaviť fyzickú 64-bitovú adresu základňovej stanice, ktorá sa bude s platformou používať. V tom prípade sa adresa 1 vždy automaticky preloží na túto adresu. Rovnako pri prijatí správy, alebo agenta od základňovej stanice táto fyzická adresa prevedie na hodnotu 1.

8.8.2 Odosielanie dátových správ

Implementácia odosielania správ vznikla v súlade s navrhnutým komunikačným protokolom, uvedeným v časti 7.8. Pri odosielaní správy je táto najprv vložená do poľa bajtov, odosielacieho buffera. Jeho veľkosť je stanovená v triede `RadioComm`. Rovnako je v tejto triede definovaná maximálna veľkosť odosielaných dát. Vzhľadom k návrhu je táto veľkosť 100 bajtov, ale je možné ju zmeniť v zdrojových kódoch platformy. Správa, ktorá sa má odoslať, sa rozdelí na potrebný počet častí, vzhľadom k maximálnej veľkosti správy.

Časť	**Dĺžka**
`SPOTCOMM_IDENTIFIER`	4B
sekvenčné číslo (hodnota 0)	4B
typ dát	1B
kontrolný súčet	4B
dĺžka samotných dát	4B

Tabuľka 8.6: Štruktúra hlavičky dátovej správy.

Nasleduje postupné odosielanie správy. Najprv je odoslaná hlavička (formát viď tabuľka 8.6). Sekvenčné číslo hlavičky je 0. Nasleduje postupné odosielanie jednotlivých častí dát (formát je uvedený v tabuľke 8.7) so sekvenčným číslom `1` až `n`. Ako bolo popísané v časti 7.8.1, je každá časť potvrdzovaná protistranou. Formát potvrdenia je v tabuľke 8.8, pričom

typ správy je v tomto prípade `ACK`. Posledné 4 bajty potvrdenia obsahujú sekvenčné číslo správy, ktorá je potvrdzovaná. Na toto potvrdenie sa čaká po stanovenú dobu (aktuálne zvolená doba je 500ms). Pokiaľ v tejto dobe nepríde potvrdenie, aktuálna časť správy s tým istým sekvenčným číslom sa pošle znovu.

Časť	**Dĺžka**
`SPOTCOMM_IDENTIFIER`	4B
sekvenčné číslo	4B
dáta	92B

Tabuľka 8.7: Štruktúra dátovej správy.

Po odoslaní poslednej časti správy, prijímacia strana skontroluje prijaté dáta. Vypočíta sa kontrolný súčet a tento sa porovná s tým, ktorý bol prijatý v hlavičke. Pokiaľ je kontrolný súčet v poriadku, prijímacia strana odošle správu typu `ACK_ALL_OK`. V opačnom prípade prijímateľ odošle správu typu `ACK_RESEND`. Obe správy majú tiež formát uvedený v tabuľke 8.8. V tom prípade je potrebné znovu odoslať celú správu a opakuje sa celý vyššie uvedený postup. O odosielanie dát vyššie uvedeným spôsobom sa stará metóda `SendData` triedy `RadioComm`. Maximálny počet opätovných odoslaní jednej časti správy, ako aj celej správy, je možné nastaviť v triede `RadioComm`. Aktuálne zvolené hodnoty sú 3 znovuodoslania časti správy a 2 opätovné odoslania dátovej správy. Pokiaľ sa po tomto počte opakovaní správu nepodarí odoslať, metóda skončí neúspešne.

Časť	**Dĺžka**
`SPOTCOMM_IDENTIFIER`	4B
sekvenčné číslo (hodnota 0)	4B
typ	1B
sekvenčné číslo potvrdzovanej správy	4B

Tabuľka 8.8: Štruktúra potvrdzovacej správy.

8.8.3 Formát ostatných správ

V časti 8.8.2 bol predstavený formát hlavičky, potvrdenia správy a dátovej správy. Platforma potrebuje však ešte iné typy správ, slúžiace napríklad pre zisťovanie topológie, či dotazy na pachové stopy. Formát všetkých týchto správ je podobný, pričom všetky obsahujú prefix uvedený v tabuľke 8.9. Typ správy je vždy daný položkou `typ`. Všetky možné typy správ boli uvedené v časti 7.8.2.

Časť	**Dĺžka**
`SPOTCOMM_IDENTIFIER`	4B
sekvenčné číslo (hodnota 0)	4B
typ	1B

Tabuľka 8.9: Prefix spoločný pre ostatné správy.

Správy pre zisťovanie topológie, t.j. `SIGNAL_GET`, `SIGNAL_GET_M`, `SIGNAL_RECEIVED`, `SIGNAL_RECEIVED_M` sú zhodné presne s formátom uvedeným v tabuľke 8.9 a žiadne ďalšie časti neobsahujú. Odlišné sú iba správy `TRACE_REQUEST` a `TRACE_ANSWER` pre službu pachové

stopy. U správy typu `TRACE_REQUEST` nasledujú za prefixom (tabuľka 8.9) obsahuje ešte 4 bajty, ktoré obsahujú ID a triedu agenta (2 krát 2 bajty). U správy `TRACE_ANSWER` je prefix nasledovaný jedným bajtom, ktorý určuje odpoveď platformy na dotaz `TRACE_REQUEST`.

8.8.4 Spracovávanie prijatých dát

Prijaté dáta sú spracovávané v samostatnom vlákne, ktoré je implementované v triede `RadioReceiver`. Ako bolo uvedené v časti 7.8, toto vlákno automaticky prijíma aj potvrdenia odoslaných dát funkciou `SendData`.

Spracovávanie dát je postavené na nekonečnom cykle, ktorý volá metódu `receive` komunikačného rozhrania platformy SunSpot. Po prijatí dát, o maximálnej veľkosti stanovenej v triede `RadioComm`, nastane spracovanie podľa toho, o aký typ dát sa jedná.

Prijímanie správy, alebo agenta

Prijímanie správy, alebo agenta, začína prijatím hlavičky. Samotná hlavička sa detekuje pomocou sekvenčného čísla, ktoré je v tomto prípade 0. Z hlavičky sa načíta kontrolná suma, veľkosť dát, ktoré sa majú prijať a typ dát. Následne sa overí, či veľkosť buffera bude stačiť na prijatie správy (porovná sa veľkosť buffera a veľkosť dát z hlavičky). Pokiaľ je všetko v poriadku, prejde prijímacie vlákno do stavu `receiving` (stav, kedy prijíma dáta od protistrany). V tomto stave sú automaticky dátové správy od iných uzlov ignorované.

Nasleduje postupné prijímanie jednotlivých častí samotných dát. Prijímacie vlákno si vždy po prijatí danej správy zvýši počítadlo značiace aktuálne sekvenčné číslo. Toto sa vždy kontroluje s tým, ktoré je v danej prijatej správe. Ak sekv. čísla súhlasia, odošle sa potvrdenie. Pokiaľ sa stane, že sekvenčné číslo prijatej správy nesedí, správa sa nespracuje. Pokiaľ je toto menšie, než aktuálne očakávané, znamená to, že daná časť dát už bola prijatá. Preto sa odošle prípadne znovu potvrdenie prijímacej strane.

Akonáhle je prijatá posledná časť dát, vypočíta sa kontrolný súčet, porovná sa s tým v hlavičke, a odosielateľovi je odoslané potvrdenie celej správy - pozitívne, alebo negatívne. Predpokladajme teda, že všetko prebehlo v poriadku a všetky dáta boli úspešne prijaté. Pokiaľ sa jedná o klasickú správu, nastaví sa premenná `receiver_messageready` na hodnotu `true`. V tomto stave sa čaká do vyzdvihnutia správy interpretom.

Ak bol doručený nový agent (hlavička obsahuje typ `DATA_AGENT`), zastaví sa vykonávanie interpreta. Prijímací buffer v tomto prípade obsahuje agenta, uloženého vo formáte uvedenom v časti 8.8.7. Je preto potrebné z tohto poľa agenta načítať. Na to slúži metóda `parseAgent` triedy `Parser`. Metóda postupne načíta všetky časti agenta (tabuľky, registre, zásobník, identifikáciu a počet skokov). Všetky n-tice, ktoré agent obsahuje, sa načítavajú pomocou metódy `ReadTuplet`, volanej metódou `parseAgent`. Po načítaní agenta je vykonávanie interpreta opäť spustené. Záznam o prijatom agentovi a vloží do log-u (viac informácií v časti 8.9). Nakoniec sa nastaví premenná `receiving` na hodnotu false. Tým je opäť umožnené prijímať nové dáta od iných odosielateľov.

Prijímacie vlákno sa musí tiež vyrovnať so situáciou, kedy odosielateľ z nejakého dôvodu skončí odosielanie skôr, než sú prijaté všetky dáta. V tomto prípade by sa totiž zablokovalo natrvalo prijímanie od iných uzlov. Preto je implementovaný timeout prijímania dát. Hodnota, po ktorú sa maximálne čaká na nasledujúcu časť správy, je stanovená v triede `RadioComm`. Po uplynutí tejto doby je automaticky zrušené prijímanie dát. Po každej prijatej časti správy sa časovač resetuje.

Spracovávanie ostatných správ

Všetky ostatné správy sa detekujú, podobne ako hlavička, na základe sekvenčného čísla hodnoty 0. Pokiaľ sa jedná o potvrdenie správy (typ `ACK`, `ACK_ALL_OK`, alebo `ACK_RESEND`), do stavových premenných sa uloží typ potvrdenia, adresa, z ktorej potvrdenie prišlo, a tiež sekvenčné číslo potvrdenia. Následne odosielacie vlákno môže hodnotu stavových premenných získať pomocou metódy `getAckReceived`.

Pokiaľ príde správa `SIGNAL_GET`, t.j. žiadosť o zistenie sily signálu od aktuálneho uzla k protistrane, je zavolaná metóda `SendSignalDatagram`, ktorá odošle žiadateľovi správu `SIGNAL_RECEIVED`. Na správu `SIGNAL_RECEIVED` potom spracujúce vlákno reaguje zistením sily signálu od odosielateľa tejto správy pomocou metódy `GetSignal`.

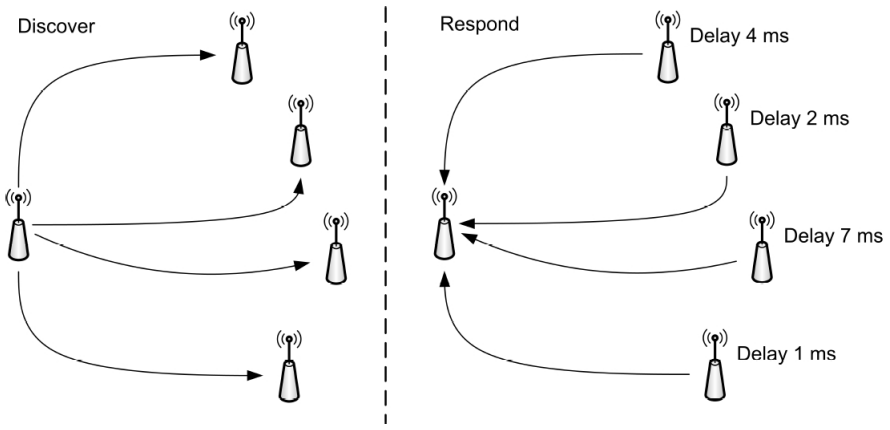
Poslednými možnosťami sú príjem dotazu alebo odpovede na dotaz ohľadne predošlého výskytu agenta na danom uzle. V prípade dotazu (typ `TRACE_REQUEST`) je zavolaná metóda `WasHere` triedy `AgentLog` (viď kapitola 8.9). Tejto metóde sa ako parameter predloží identifikácia agenta, ktorá je súčasťou prijatej správy. Metóda vyhľadá v log-u predošlý výskyt daného agenta a vráti výsledok. Tento výsledok je následne odoslaný ako správa dotazujúcemu uzlu, pomocou metódy `sendTraceAnswer`.

V prípade, že naopak príde odpoveď na dotaz o výskyte agenta (typ `TRACE_ANSWER`) na vzdialenom uzle, je nastavená stavová premenná obsahujúca výsledok dotazu. Táto je odosielaciemu vláknu dostupná pomocou metódy `getTraceAnswer`.

8.8.5 Zisk sily signálu od okolitých uzlov

Zisk sily signálu je implementovaný v metóde `getSignalFromNeighbours` triedy `RadioComm`. Táto metóda otvorí špeciálne spojenie, pomocou ktorého je možné posielať broadcastové správy. Odosielaná správa má typ `SIGNAL_GET`.

Uzol teda odošle túto správu pomocou broadcast-u. Každý dostupný uzol v okolí správu dostane a vyšle odpoveď. Pred odoslaním každý uzol počká po dobu danú náhodnou hodnotou od 0 do 20 milisekúnd, aby nedošlo k zahlteniu príjemcu. Výsledkom volania je, že pre každú odpoveď sa zistí sila signálu a s príslušnými ďalšími údajmi sa táto hodnota vloží priamo do BeliefBase agenta. Viac o formáte vkladanej n-tice bolo uvedené v časti 6.2.9.



Obr. 8.1: Zisk sily signálu od okolitých susedov [5].

Po odoslaní broadcast-u je nutné počkať stanovenú dobu, kým prídu odpovede od jednotlivých uzlov a dáta sa uložia do BeliefBase. Dĺžku čakania je možné nastaviť v triede

RadioComm. Aktuálne je zvolená hodnota 700 milisekúnd, ktorá sa pri testovaní ukázala ako postačujúca.

8.8.6 Dotazovanie na pachové stopy agenta

Tento dotaz je v podstate zistenie predošlej prítomnosti aktuálne interpretovaného agenta na cieľovom uzle. Implementácia je v metóde traces_request_wasthere triedy RadioComm. Metóda odošle správu na cieľový uzol a počká stanovenú dobu na odpoveď. Ak odpoveď nepríde, pokúsi sa dotaz odoslať znovu. Návratová hodnota, ktorú odpoveď protistrany nesie, sa vyzdvihne pomocou metódy getTraceAnswer triedy RadioReceiver. Táto hodnota je následne metódou vrátená ako návratová hodnota dotazu.

8.8.7 Odosielanie agenta

Pri odosielaní agenta je potrebné naskladať všetky jeho časti za sebou do odosielacieho buffera. Celý agent pozostáva z tabuliek PlanBase, BeliefBase, InputBase, registrov, aktuálneho plánu, identifikácie, a počtu skokov. Všetky n-tice v tabuľkách, či registroch sa pred odoslaním prevedú do čitateľnej podoby (vo formáte uvedenom v časti 5.1.1) pomocou metódy printToString.

Odosielací buffer je alokovaný v objekte triedy RadioComm a má obmedzenú kapacitu. Aktuálne nastavená hodnota je 30 kilobajtov, ale je možné ju podľa potreby zmeniť. Prvých 20 bajtov buffera sa použije na uloženie pozícií jednotlivých častí agenta. Pozície sú 32-bitové, preto každá z nich zaberá 4 bajty buffera. Na 8-bitové hodnoty sa indexy častí prevedú pomocou bitových posunov.

Samotný agent sa ukladá od indexu 20. Ako prvé sa vkladajú tabuľky PlanBase, BeliefBase a InputBase. Všetky zhodne vo forme postupnosti n-tíc, ktoré nie sú nijako oddeľované. Po vložení každej tabuľky sa do buffera uloží koncový index tejto tabuľky, spôsobom spomenutým v predošlom odstavci. Nasleduje vloženie obsahu registrov. Hodnoty registrov sú oddelené znakom '|'. Následne sa vloží jeden bajt, značiaci číslo aktívneho registra. Potom nasleduje zásobník (aktuálny plán). Tento sa vloží vo forme reťazca príkazov jazyka ALLL, ktorý sa získa pomocou metódy printToString triedy PlanStack. Nasleduje vloženie koncového indexu zásobníka. Nakoniec sa vloží ID, trieda agenta a počet skokov, opäť rozdelením týchto čísel do 1-bajtových častí. Tým je agent pripravený na odoslanie.

Agent je potom odoslaný pomocou metódy SendData. Ako parameter sa metóde predloží typ dát, ktorý je v tomto prípade hodnota DATA_AGENT. Celé odoslanie tak prebehne pomocou vytvoreného komunikačného protokolu.

8.8.8 Odosielanie správy

Odosielanie správy je v zásade implementované podobne ako odosielanie agenta. Slúži na to metóda SendMessage. Ako parameter sa metóde predloží n-tica, teda objekt typu Tuplet. Pomocou metódy printToString triedy Tuplet sa n-tica prevedie na reťazec. Tento sa následne vloží do odosielacieho buffera. Nasleduje volanie metódy SendData. Jediné čo sa pri odoslaní líši, je typ dát. V tomto prípade to bude DATA_MESSAGE.

8.9 Log agentov

Na prácu s logom agenta slúži trieda AgentLog. Táto v sebe zapúzdruje prácu s flash pamäťou a obsahuje metódy pre pridávanie záznamov a vyhľadávanie v záznamoch. Veľkosť log-u

bola stanovená na 16KB, ale je možné ju zmeniť v zdrojovom kóde triedy. Ako bolo spomenuté v časti 3.5.5, pri každom vložení nového záznamu sa vygeneruje v triede `RecordStore` identifikátor záznamu. Tento je jedinečný pre každý záznam. Pomocou neho je možné daný záznam neskôr prepísať novou hodnotou.

Na pridávanie záznamu bola implementovaná metóda `addRecord`. Táto ako parameter dostane štruktúru obsahujúcu jednotlivé položky záznamu. Z nich sa vytvorí 8-bajtové pole. Následne sa zistí, či je ešte miesto v log-u (nebol prekročený maximálny počet záznamov). Pokiaľ nie, je záznam klasicky pridaný. Pokiaľ už je log plný, bude sa prepisovať najstarší záznam. ID najstaršieho záznamu je v premennej `start_recordid`. Tento záznam je následne prepísaný a premenná `start_recordid` sa inkrementuje. Tým sa posunie začiatok kruhového log-u.

Na vyberanie záznamov slúži metóda `getRecord`. Jej parametrom je index záznamu. Tento index je hodnota reprezentujúca klasický index v poli. Preto pre vybranie správneho záznamu, pokiaľ sa log už predtým prepisoval v kruhu, je potrebné vypočítať identifikátor správneho záznamu vo flash. Na tento účel slúži metóda `get_recordid`. Následne je potrebný záznam vybraný z flash. Výsledkom je štruktúra obsahujúca jednotlivé položky daného záznamu. Je nutné podotknúť, že metódy pre vkladanie a výber musia zaručovať výlučný prístup k dátam pomocou monitora, aby nedošlo k nekonzistentnému stavu flash pamäte.

8.9.1 Sledovanie výskytu agenta

Jednou z funkcií agentnej platformy je tzv. Backtracking, teda spätné sledovanie pohybu agenta sieťou. Popis bol uvedený v časti 6.2.10. V podstate sa jedná o zistenie, z ktorého uzla agent prišiel. Na tento účel sú priamo v triede `AgentLog` implementované metódy `backTrackFirst` a `backTrackLast`. Každá z nich vyhľadá záznam o aktuálnom agentovi v log-u. Prvá spomínaná nájde prvý (najstarší), druhá posledný záznam. Zo záznamu sa získa identifikátor uzla, odkiaľ agent prišiel. Tento je aj návratovou hodnotou metódy `backTrackFirst` resp. `backTrackLast`.

Ďalej je agenta možné sledovať pomocou dotazovania, či sa záznam o danom agentovi už nachádza v log-u, teda či daný agent bol na tomto uzle. Na tento účel slúži metóda `wasHere`. Metóda vyhľadá výskyt daného agenta a vráti príslušnú návratovú hodnotu. Na rozdiel od Backtracking-u, dotazuje sa vždy vzdialený uzol, preto volaniu tejto metódy vždy predchádza správa z rádia, požadujúca túto odpoveď. Výsledok je opäť zaslaný cez rádio.

8.10 Logovanie vstupov zo senzorov

Aktuálna verzia platformy `WSageNt` pracuje iba so senzorom teploty. Preto bolo implementované v tejto práci iba intervalové meranie a logovanie teploty. Ako ďalšie rozšírenie by mohla byť implementácia merania a logovania napríklad nameranej intenzity svetla. Pre ukladanie záznamov o teplote bola implementovaná trieda `TempLog`. Trieda vychádza pri návrhu z implementácie triedy `AgentLog`.

Na vkladanie hodnoty teploty slúži metóda `AddValue`. Teplota sa ukladá vo forme čísla typu int. Pomocou metódy `AddValue` v podstate nastane pridanie tejto hodnoty do buffera, ktorý je alokovaný priamo v triede `TempLog`. Zápis do flash pamäte nastane v momente, kedy je vstupný buffer naplnený hodnotami. Potom vždy nasleduje zápis pridaním záznamu do kruhového logu vo flash. Mechanizmus je podobný ako u log-u agentov. Vloží sa buď nový záznam, alebo je prepísaný najstarší záznam novými hodnotami.

Výber hodnoty z log-u je implementovaný v metóde `GetValue`. Ako parameter je index hodnoty podobne, ako by hodnota bola uložená v poli o veľkosti rovnej maximálnemu počtu položiek log-u. Výber danej hodnoty sa mierne komplikuje použitím buffera hodnôt. Daná hodnota sa totiž okrem flash pamäte môže vyskytovať v bufferi. Je preto potrebný prepočet, či najnovšia hodnota daného záznamu je vo flash, alebo v bufferi. Toto platí hlavne pri kruhovom prepise, kedy sa zápisom do buffera postupne zneplatňujú jednotlivé položky jedného záznamu vo flash pamäti.

Pri zatváraní log-u je potrebné uložiť informáciu, ktorý záznam vo flash je aktuálne najstarším záznamom. Rovnako je potrebné uložiť obsah nekompletného buffera do flash, pokiaľ obsahuje nejaké namerané dáta. Pri opätovnom otvorení log-u sa hodnoty v bufferi obnovia.

8.11 Intervalové meranie

Služba implementuje meranie hodnôt teploty opakovane v stanovenom časovom intervale. Na tento účel sa používa trieda `Timer` jazyka Java. Samotné meranie teploty je implementované v triede `SensorTask`, čo je trieda odvodená od triedy `TimerTask` v jazyku Java.

Interval je momentálne pevne nastavený na hodnotu 1 sekunda a je ho možné zmeniť v triede `PlatformSvc`. Pri konštrukcii triedy `PlatformSvc` sa naplánuje opakujúca sa časová udalosť `SensorTask` do premennej `timer`. Pri spustení tejto udalosti dôjde k zisteniu aktuálnej teploty z teplotného senzora. Nameraná hodnota z teplotného senzora typu `double`, preto sa prevedie na typ `int` (bude sa vkladať v tomto formáte do logu). Deje sa tak vynásobením hodnoty číslom 10 (nameraná teplota má jedno desatinné miesto). Následne je táto teplota pridaná do log-u teploty.

8.12 Uspanie interpreta na požadovanú dobu

Na uspanie interpreta je použitý mechanizmus čakania na monitore pomocou volania funkcie `wait` jazyka Java. Metóda starajúca sa o uspanie interpreta má názov `sleep` a je umiestnená v triede `Interpreter`. Dôležitá vlastnosť je možnosť predčasného ukončenia uspania interpreta. Toto je potrebné v prípade, že je interpret uspaný na nejakú dlhšiu dobu a medzitým príde cez rádio nový agent. Následne je potrebné čakanie zrušiť a obnoviť vykonávanie interpreta. Na tento účel slúži metóda `cancelSleeping` triedy `Interpreter`. Metóda je postavená na volaní funkcie `notify` jazyka Java.

8.13 Interpretovanie zdrojového textu

Hlavnou triedou agentnej platformy je trieda `Interpreter`. Trieda implementuje samotný interpret jazyka ALLL. Interpret funguje ako nekonečný cyklus, v ktorom sa odohráva nasledovný postup. V každom priechode sa najprv skontroluje, či je na zásobníku nejaký kód pre interpretovanie. Pokiaľ nie, interpret sa zastaví. Zastavenie znamená nastavenie stavovej premennej `running` na hodnotu false metódou `stop`, a uspanie vlákna interpreta na monitore. Toto sa opäť deje pomocou volania funkcie `wait` jazyka Java. Opätovné spustenie interpreta je možné prijatím nového agenta. Vtedy prijímacie vlákno (objekt triedy `RadioReceiver`) zavolá metódu `start`, ktorá sa postará o obnovenie činnosti interpreta.

Pokiaľ interpret beží, nasledujúcim krokom je získanie prípadnej správy z rádia. Metódou `isMessageAvailable` triedy `RadioComm` sa preto skontroluje prijímacie vlákno, či bola

prijatá nejaká správa. Ak áno, je táto správa vložená do tabuľky `InputBase`.

Nasledujúcim krokom je získanie príkazu jazyka ALLL, ktorý sa má spustiť. Zavolá sa teda metóda `pop` zásobníka plánov. Fungovanie zásobníka bolo popísané v časti 8.5. Výsledkom je štruktúra obsahujúca príkaz jazyka ALLL, konkrétne operačný kód a operandy vo forme n-tice. Keďže operandy môžu obsahovať na ktoromkoľvek mieste zástupné symboly registra, nasleduje zavolanie metódy `replaceRV` na n-ticu symbolizujúcu operandy. Metóda, ako bolo uvedené v časti 8.1, nahradí výskyty registrov v n-tici hodnotami týchto registrov.

Tým je príkaz pripravený na spustenie. Podľa toho, či sa jedná priamo o operáciu platformy, alebo interpreta, je zavolaná príslušná metóda, buď priamo triedy `Interpreter`, alebo triedy `PlatformSvc`. Pokiaľ pri interpretovaní nastane chyba, zavolá sa metóda `print_error`. Tejto sa ako parameter predloží popis chyby. Metóda vypíše chybu spolu s názvom aktuálneho plánu, operáciou a pozíciou v pláne, na ktorej chyba nastala. Toto je výhodné z hľadiska lepšej orientácie pri ladení vlastného kódu jazyka ALLL.

8.13.1 Služby implementované v interprete

Príkazy ALLL, ktorých implementácia je priamo v interprete, korešpondujú s návrhom platformy `WSageNt`. Tabuľka 8.10 obsahuje prehľad týchto metód. Jednotlivé metódy implementujúce dané príkazy volajú metódy už implementovaných tried pre manipuláciu s tabuľkami, n-ticami a komunikáciu. Každá metóda dostane ako parameter n-ticu, v ktorej sú operandy pre daný príkaz jazyka ALLL.

Názov metódy	**Popis**
`addBB(Tuplet t)`	pridá zadanú n-ticu do BeliefBase
`removeBB(Tuplet t)`	odoberie n-tice vyhovujúce zadanej z BeliefBase
`testBB(Tuplet t)`	nájde všetky n-tice vyhovujúce zadanej n-tici v BeliefBase
`testIB(Tuplet t)`	otestuje InputBase na správu od senzora/uzla
`setRegister(Tuplet t)`	nastaví aktívny register
`runPlan(Tuplet t)`	spustí pomenovaný plán
`runDirect(Tuplet t)`	priamo vloží množinu príkazov na zásobník
`sendMessage(Tuplet t)`	odošle správu na iný uzol

Tabuľka 8.10: Príkazy ALLL implementované v interprete.

8.13.2 Služby platformy

Služby platformy tak, ako boli definované v platforme `WSageNt`, sú implementované v triede `PlatformSvc`. Podobne, ako u služieb interpreta, všetky metódy implementujúce služby platformy prevažne volajú metódy ďalších tried, v ktorých je daná funkcionalita implementovaná. Popis týchto tried bol uvedený vyššie v texte. Prehľad jednotlivých metód triedy platformSvc a ich významu je v tabuľke 8.11.

8.14 Aplikácia pre hostiteľský PC

Na komunikáciu so senzorovými uzlami cez hostiteľský PC bola vytvorená aplikácia s grafickým užívateľským rozhraním v jazyku Java. Ako bolo spomenuté v časti 3.5.1, na komunikáciu sa využíva ten istý prístup ako pri programovaní aplikácie pre senzorové uzly.

Názov metódy	Popis
`first(Tuplet t)`	vloží do aktívneho registra prvý prvok n-tice
`rest(Tuplet t)`	vloží do aktívneho registra zvyšok n-tice bez prvého prvku
`led(Tuplet t)`	vykoná výstup na zvolenú LED diódu
`wait(Tuplet t)`	uspí interpret na danú dobu
`sensors(Tuplet t)`	odmeria teplotu, alebo vykoná operáciu nad nameranými hodnotami
`compute(Tuplet t)`	výpočtová operácia platformy
`waitForMessage()`	počká na prijatie správy z rádia
`activateMessageMonitoring()`	aktivuje sledovanie prišlých správ
`getSignalFromNeighbours(Tuplet t)`	zistí silu signálu od okolitých uzlov
`moveAgent(Tuplet t)`	presunie aktuálneho agenta na iný uzol
`kill()`	zastaví vykonávanie interpreta
`listOp(Tuplet t)`	operácia so zoznamom (napr. test na prázdny zoznam)
`random(Tuplet t)`	generovanie náhodného čísla
`traces(Tuplet t)`	operácia s pachovými stopami agenta
`getLog(Tuplet t)`	zisk záznamov log-u agenta

Tabuľka 8.11: Popis metód implementujúcich služby platformy.

Pre komunikáciu boli využité triedy `RadioComm` a `RadioReceiver` vyvinuté pre senzorové uzly. Tieto boli upravené pre dané použitie, a hlavne boli z nich odstránené nepotrebné funkcie a metódy, ktoré vyžadovali použitie ďalších tried. Ostal však celý princíp odosielania dát s potvrdzovaním jednotlivých datagramov.

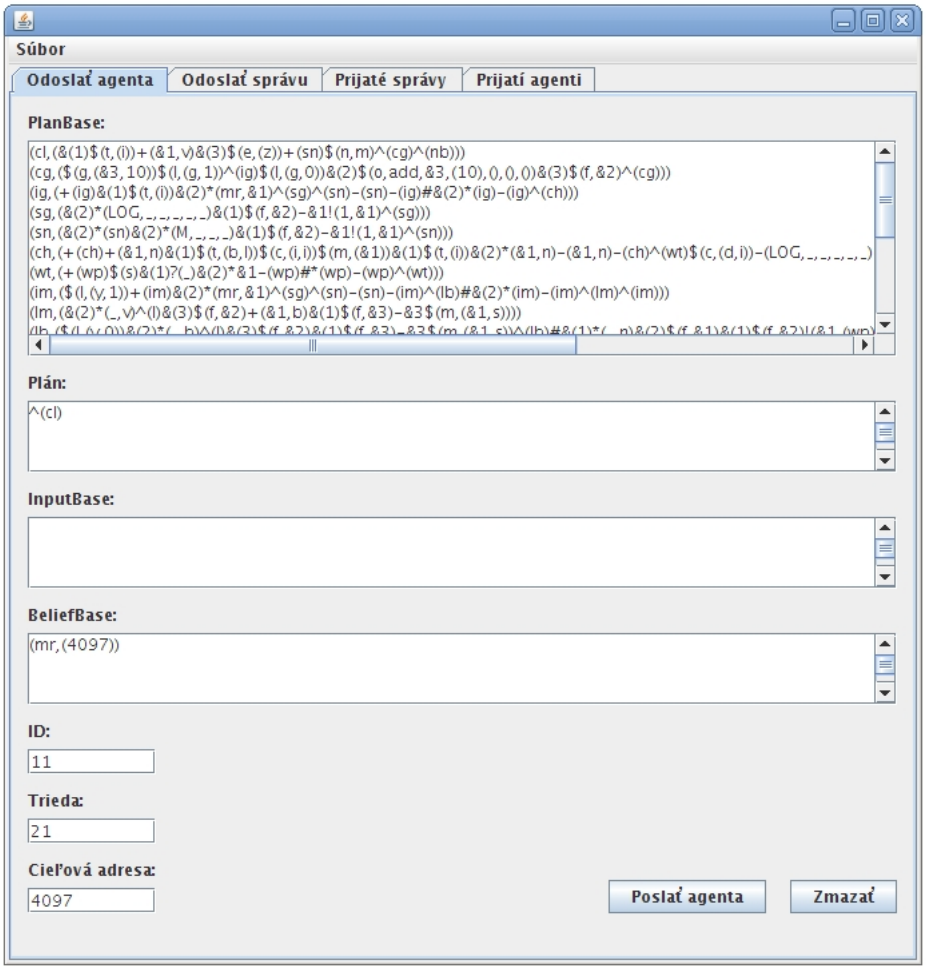
8.14.1 GUI aplikácie

Okno grafického užívateľského rozhrania aplikácie obsahuje 4 karty. Prvá z nich umožňuje odosielať agenta. Časti agenta sa vkladajú do jednotlivých textových polí. Náhľad je možné vidieť na obrázku 8.2. Po odoslaní agenta je užívateľ je informovaný o výsledku odoslania, úspešnom či negatívnom.

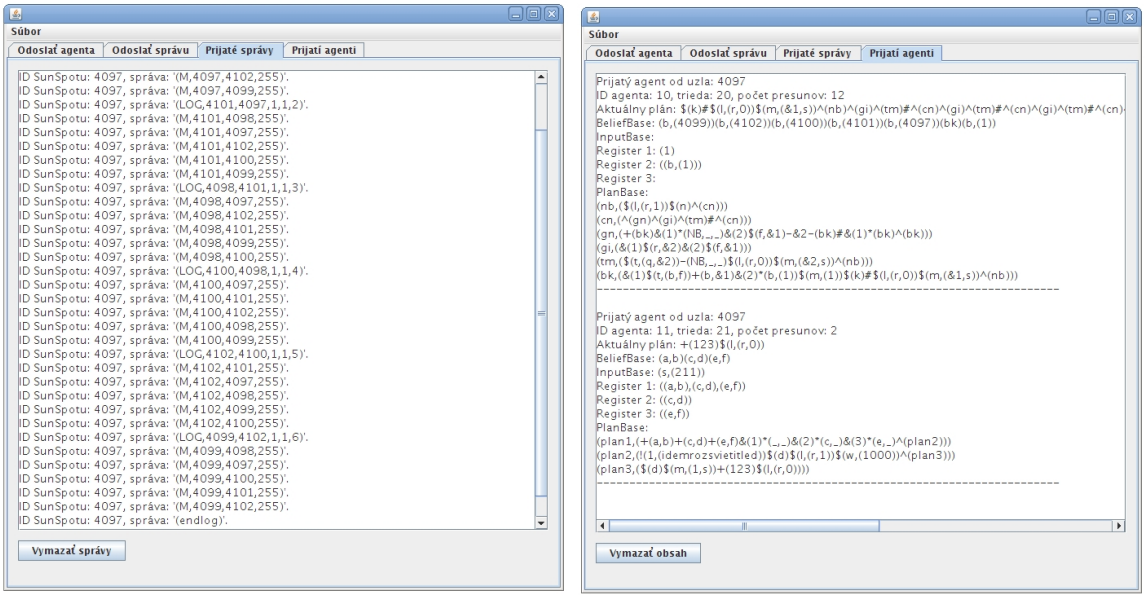
Druhá karta potom umožňuje odosielanie samostatných správ senzorovým uzlom. V tomto prípade sa udáva iba reťazec obsahujúci danú správu a identifikátor uzla, na ktorý sa správa má poslať. O správnom odoslaní správy je užívateľ opäť informovaný.

Tretia karta obsahuje textové pole, do ktorého aplikácia vkladá prijaté správy od senzorových uzlov. Formát výpisu obsahuje adresu uzla, od ktorého správa prišla, nasledovanú n-ticou obsahujúcou samotnú správu, viď obrázok 8.3. Je tiež možnosť obsah okna kedykoľvek vyčistiť príslušným tlačítkom.

Posledná karta obsahuje výpisy agentov došlých na základňovú stanicu. Opäť sa jedná o textové pole, pričom záznam o každom agentovi obsahuje všetky jeho časti, teda tabuľky, registre, identifikáciu a počet presunov.



Obr. 8.2: GUI aplikácie: Odosielanie agenta.



Obr. 8.3: GUI aplikácie: Prijaté správy (vľavo) a agenti (vpravo).

Kapitola 9

Testovanie

Záverečnou časťou práce je popis testovania implementovanej platformy. Platforma bola testovaná v simulátore `Solarium`, ako aj na fyzických uzloch. Testovanie jednotlivých častí prebiehalo samozrejme aj behom implementácie. V nasledujúcej podkapitole sú uvedené niektoré príklady testovacích agentov a princíp ich činnosti. Príklady boli zvolené tak, aby dohromady pokryli všetky funkcie implementovanej agentnej platformy. Presnejší popis testovania bude uvedený v podkapitole 9.2.

9.1 Príklady testovacích agentov

9.1.1 Blikanie LED na senzorovom uzle

Tento kód (viď tabuľka 9.1) testuje fungovanie zásobníka, volania pomenovaných plánov a základné operácie platformy (vzhľadom k deleniu na služby interpreta a platformy, viď kapitola 6), konkrétne výstupy na LED a čakanie interpreta po stanovenú dobu.

Časť	Obsah danej časti

`PlanBase`	`(cervena,($(l,(r,1))$(w,(500))$(l,(r,0))$(w,(500))))`
	`(zelena,($(l,(g,1))$(w,(500))$(l,(g,0))$(w,(500))))`
	`(blikaj,(^(cervena)^(zelena)^(blikaj)))`

`Plan`	`^(blikaj)`

Tabuľka 9.1: Agent, ktorý v nekonečnom cykle bliká s LED.

V počiatočnom stave aktuálny zámer agenta obsahuje volanie plánu `blikaj`. Po jeho spustení sa v cykle volajú plány `cervena` a `zelena`. Každý z nich obsahuje rozsvietenie zvolenej LED diódy (služba `l`), čakanie po stanovenú dobu (služba `w`), a opätovné zhasnutie a čakanie pomocou tých istých služieb. Po vykonaní oboch plánov `cervena` a `zelena` za zavolá opäť plán `blikaj` a tak sa cyklus nekonečne opakuje.

9.1.2 Blikanie LED s využitím bázy znalostí

Tento príklad (umiestnený v tabuľke 9.2) obsahuje iný prístup k programovaniu cyklu. Informácia o tom, ako sa má s LED diódami blikať, je umiestnená v tabuľke `BeliefBase`. Jednotlivé n-tice symbolizujúce predpisy pre blikanie sa vložia pomocou plánu `napln`. Nasleduje volanie plánu `blik`. Tento plán otestuje `BeliefBase` na n-tice s predpisom pre blikanie. Tu sa využíva fakt, že operácia výberu `*` spôsobí pád plánu, ak sa nenájde žiadna

n-tica. V tomto prípade dôjde k zmazaniu obsahu zásobníka po zarážku (#) a pokračuje sa opätovným volaním plánov napln a blik. Tým je zaistený cyklus.

Časť	Obsah danej časti
PlanBase	(blik,(&(1)*(led,_,_)^(vyber)$(l,&1)$(w,&2)$(l,&1)^(blik) #^(napln)^(blik))) (vyber,(&(2)$(f,&1)-&2&(3)$(r,&2)&(1)$(f,&3)&(2)$(r,&3))) (napln,(+(led,r,200)+(led,g,400)+(led,y,600)))
Plan	^(napln)^(blik)

Tabuľka 9.2: Agent blikajúci s LED na základne predpisu v BeliefBase.

Po úspešnom výbere z BeliefBase sa zavolá plán vyber. V registri 1 je pritom uložená n-tica obsahujúca všetky predpisy pre blikanie. Pomocou služby f sa do druhého registra vloží prvá z týchto n-tíc. Následne sa táto z BeliefBase odstráni operáciou -. Pre výpočet sú potrebné iba posledné dve položky n-tice, a to znak LED a dĺžka čakania. Preto sa pomocou služby r získa zvyšok n-tice obsahujúci tieto dva prvky. Z tejto (aktuálne dvojprvkovej) n-tice sa prvý a druhý prvok získa pomocou služieb f a r, pričom tieto dve hodnoty budú v registroch 1 a 2, ktoré sa pred danými operáciami nastavili ako aktívne. Skončením plánu vyber pokračuje ďalej plán blik. Tento zavolá služby l a w, pričom ako parametre služieb sa použijú registre, v ktorých sú uložené hodnoty získané v pláne vyber.

9.1.3 Práca so senzorom teploty

Kód v tabuľke 9.3 využíva služby platformy pre meranie aktuálnej teploty a tiež pracuje s hodnotami nameranými pomocou intervalového merania (všetko pomocou služby d). Plán teplota zmeria aktuálnu teplotu, túto hodnotu vyberie z InputBase a odošle ju na základňovú stanicu. Ostatné plány potom zistia minimum, maximum a priemer z posledných 100 hodnôt teploty, nameraných intervalovým meraním. Každá z týchto hodnôt je rovnako odoslaná na základňovú stanicu.

Časť	Obsah danej časti
PlanBase	(teplota,($(d)&(1)?(s)!(1,(teplota,&1)))) (max,($(d,(M,100))&(1)?(M)!(1,(max,&1)))) (min,($(d,(m,100))&(1)?(m)!(1,(min,&1)))) (priemer,($(d,(a,100))&(1)?(a)!(1,(priemer,&1))))
Plan	^(teplota)^(max)^(min)^(priemer)

Tabuľka 9.3: Meranie teploty a práca s históriou nameraných hodnôt.

9.1.4 Výpočtové operácie

Tento kód (viď tabuľka 9.4) využíva všetky dostupné výpočtové operácie, ktoré sú v platforme implementované. Počiatočný plán (časť agenta Plan) sa stará o vloženie testovacích n-tíc do registrov 1 a 2. Jednotlivé operandy sa vložia do BeliefBase, následne sa pomocou operácie * vložia do registrov vo forme n-tíc obsahujúcich dané operandy. Register 1 tak obsahuje n-ticu (((a),0),((a),100)), register 2 potom (((b),100),((b),6)). Plány t1 až t16 potom obsahujú výpočty pre rôzne operátory. Výsledok sa vždy vloží do registra 3 a jeho obsah je odoslaný na základňovú stanicu.

Časť	Obsah danej časti
PlanBase	(t1,(&(3)$(o,min,&1,(a),(min))!(1,&3))) (t2,(&(3)$(o,not,&1,(a),(not))!(1,&3))) (t3,(&(3)$(o,cpy,&1,(a),(cpy))!(1,&3))) (t4,(&(3)$(o,mul,&1,&2,(a),(b),(mul))!(1,&3))) (t5,(&(3)$(o,div,&1,&2,(a),(b),(div))!(1,&3))) (t6,(&(3)$(o,mod,&1,&2,(a),(b),(mod))!(1,&3))) (t7,(&(3)$(o,add,&1,&2,(a),(b),(add))!(1,&3))) (t8,(&(3)$(o,sub,&1,&2,(a),(b),(sub))!(1,&3))) (t9,(&(3)$(o,les,&1,&2,(a),(b),(les))!(1,&3))) (t10,(&(3)$(o,mor,&1,&2,(a),(b),(mor))!(1,&3))) (t11,(&(3)$(o,leq,&1,&2,(a),(b),(leq))!(1,&3))) (t12,(&(3)$(o,meq,&1,&2,(a),(b),(meq))!(1,&3))) (t13,(&(3)$(o,equ,&1,&2,(a),(b),(equ))!(1,&3))) (t14,(&(3)$(o,neq,&1,&2,(a),(b),(neq))!(1,&3))) (t15,(&(3)$(o,and,&1,&2,(a),(b),(and))!(1,&3))) (t16,(&(3)$(o,orr,&1,&2,(a),(b),(orr))!(1,&3))) (testy,(^(t1)^(t2)^(t3)^(t4)^(t5)^(t6)^(t7)^(t8)^(t9)^(t10) ^(t11)^(t12)^(t13)^(t14)^(t15)^(t16)))
Plan	+((a),0)+((a),100)+((b),100)+((b),6)&(1)*((a),_)&(2)*((b),_) ^(testy)

Tabuľka 9.4: Výpočtové operácie.

9.1.5 Priechod agenta sieťou

Kód nasledujúceho príkladu (viď obrázok 9.5) bol prevzatý z práce [6], ktorá sa zaoberá zisťovaním topológie siete. Uvedený kód funguje nasledovne [6]. Agent najprv nájde všetky susedné uzly. Pomocou plánu `cn` postupne prechádza jednotlivé záznamy. Pre každý záznam pomocou plánov `gn` a `gi` získa adresu uzla. Túto adresu otestuje pomocou služby pachové stopy. Pokiaľ agent na danom uzle nebol, tak sa tam pomocou plánu `tm` presunie. Pokiaľ agent na danom uzle bol, nastane pád aktuálneho plánu (`tm`) a pokračuje sa na výber ďalšieho záznamu plánom `cn`. Pokiaľ agent neúspešne otestuje všetky okolité uzly (všetky už navštívil), presunie sa nazad na uzol, z ktorého prišiel pomocou plánu `bk`, a celý cyklus sa opakuje. Agent končí svoju činnosť, keď sa vráti na uzol, z ktorého začínal presuny, teda na uzol, na ktorý bol poslaný zo základňovej stanice.

Časť	Obsah danej časti
PlanBase	(nb,($(l,(r,1))$(n)^(cn))) (cn,(^(gn)^(gi)^(tm)#^(cn))) (gn,(+(bk)&(1)*(NB,_,_)&(2)$(f,&1)-&2-(bk)#&(1)*(bk)^(bk))) (gi,(&(1)$(r,&2)&(2)$(f,&1))) (tm,($(t,(q,&2))-(NB,_,_)$(l,(r,0))$(m,(&2,s))^(nb))) (bk,(&(1)$(t,(b,f))+(b,&1)&(2)*(b,(1))$(k)#$(l,(r,0)) $(m,(&1,s))^(nb)))
Plan	^(nb)

Tabuľka 9.5: Priechod agenta celou sieťou.

9.1.6 Zistenie topológie siete

Kód ALLL v tabuľke 9.6 je finálnou aplikáciou z práce [6], slúžiacou na zisťovanie topológie siete. Skladá sa z dvoch hlavných častí. Jednak je to časť starajúca sa o priechod agenta všetkými uzlami siete, kde sa využíva kód uvedený v tabuľke 9.5. Táto časť je oddelená medzerou a je umiestnená v dolnej časti tabuľky. Zvyšok kódu slúži na zber informácií o topológii.

Na začiatku tabuľka `BeliefBase` obsahuje n-ticu určujúcu adresu uzla, ktorý ma priame spojenie so základňovou stanicou. Pri testovaní to bol fyzický uzol 29097, takže v tabuľke `BeliefBase` bude n-tica `(mr,(29097))`. Pomocou tohto uzla sa odosielajú všetky informácie na základňovú stanicu, a na tento uzol musí byť agent odoslaný zo základňovej stanice [6]. Agent začína vykonávaním plánu `cl`. Po príchode na daný uzol agent uloží do `BeliefBase` n-ticu `((adresa_uzla),v)`. Táto n-tica sa slúži pre orientáciu agenta otroka (bude vysvetlené ďalej).

Časť	Obsah danej časti
PlanBase	(cl,(&(1)$(t,(i))+(&1,v)&(3)$(e,(z))+(sn)$(n,m)^(cg)^(nb))) (cg,($(g,(&3,10))$(l,(g,1))^(ig)$(l,(g,0))&(2)$(o,add,&3,(10),(),(),())&(3) $(f,&2)^(cg))) (ig,(+(ig)&(1)$(t,(i))&(2)*(mr,&1)^(sg)^(sn)-(sn)-(ig)#&(2)*(ig)-(ig)^(ch))) (sg,(&(2)*(LOG,_,_,_,_,_)&(1)$(f,&2)-&1!(1,&1)^(sg))) (sn,(&(2)*(sn)&(2)*(M,_,_,_)&(1)$(f,&2)-&1!(1,&1)^(sn))) (ch,(+(ch)+(&1,n)&(1)$(t,(b,l))$(c,(i,i))$(m,(&1))&(1)$(t,(i))&(2)*(&1,n) -(&1,n)-(ch)^(wt)$(c,(d,i))-(LOG,_,_,_,_,_)-(M,_,_,_)#&(2)*(ch)-(ch)&(2) $(t,(b,l))-(&2,v)-(&1,v)^(im))) (wt,(+(wp)$(s)&(1)?(_)&(2)*&1-(wp)#*(wp)-(wp)^(wt))) (im,($(l,(y,1))+(im)&(2)*(mr,&1)^(sg)^(sn)-(sn)-(im)^(lb)#&(2)*(im)-(im) ^(lm)^(im))) (lm,(&(2)*(_,v)^(l)&(3)$(f,&2)+(&1,b)&(1)$(f,&3)-&3$(m,(&1,s)))) (lb,($(l,(y,0))&(2)*(_,b)^(l)&(3)$(f,&2)&(1)$(f,&3)-&3$(m,(&1,s))^(lb)#&(1) *(_,n)&(2)$(f,&1)&(1)$(f,&2)!(&1,(wp))$(k))) (l,(&(3)$(r,&2)$(e,(e,&3))&(2)$(e,(c,&3))^(l))) (nb,($(l,(r,1))$(n)^(cn))) (cn,(^(gn)^(gi)^(tm)#^(cn))) (gn,(+(bk)&(1)*(NB,_,_)&(2)$(f,&1)-&2-(bk)#&(1)*(bk)^(bk))) (gi,(&(1)$(r,&2)&(2)$(f,&1))) (tm,($(t,(q,&2))-(NB,_,_)$(l,(r,0))$(m,(&2,s))^(nb))) (bk,(&(1)$(t,(b,f))+(b,&1)&(2)*(b,(1))$(k)#$(l,(r,0)) $(m,(&1,s))^(nb)))
Plan	^(nb)
BeliefBase	(mr,(29097))

Tabuľka 9.6: Zber informácií o topológii siete.

Na každom uzle agent zistí svojich susedov príkazom `$(n,m)`, ktorý do `BeliefBase` uloží v tvare `(M,<adresa uzla>,<adresa uzla z okolia>,<sila signálu>)`, ako bolo uvedené v časti 6.2.9. Následne je spustený plán `cg`, ktorý sa stará o výber záznamov z logu prišlých agentov. Nerozlišuje sa, ktoré záznamy sa majú vybrať, preto nastane vloženie všetkých dostupných záznamov do časti `BeliefBase`. Záznamy sa ukladajú v tvare, ktorý bol uvedený v podkapitole 6.2.11.

O doručenie záznamov sa stará plán `ig`. Agent najprv otestuje, či sa nachádza na uzle s priamym spojením so základňovou stanicou (príkazom `$(t,(i))` zistí adresu aktuálneho uzla a túto porovná s hodnotou vloženou v `BeliefBase` ako informáciou o uzle so spo-

jením so zákl. stanicou). Pokiaľ je na tomto uzle, tak jednotlivé záznamy z `BeliefBase` postupne odošle na základňovú stanicu pomocou plánu `sg`. Odosielajú sa všetky záznamy log-u (podplán `sg`), a tiež záznamy o zistených susedoch uzla (podplán `sn`). Pomocou týchto informácií je možné zistenie topológie celej siete.

Pokiaľ sa agent nachádza na inom než spomenutom uzle, je potrebné doručiť záznamy nepriamo. Na tento účel slúži agent otrok [6]. K jeho vytvoreniu dôjde v pláne `ch` volaním služby platformy `$(m,(adresa uzla))`, ktoré spôsobí odoslanie agenta (v tomto prípade otroka) a súčasne pokračovanie jeho činnosti agenta. Preto je ešte pred odoslaním nutné zmeniť identifikáciu agenta, aby sa odlíšil agent otrok. Agent sa potom uspí pomocou plánu `wt`, a počká, kým mu otrok odošle správu, že doručil záznamy na základňovú stanicu.

Otrok sa na uzol, ktorý je najbližšie k základňovej stanici dostane s použitím záznamov v `BeliefBase`, ktoré mu agent zanechal v tvare n-tíc `((<adresa uzla>),v)`. Pri svojej ceste rozsvecuje žlté diódy na uzloch, ktorými prechádza. Počiatok cesty otroka je po zavolaní plánu `im`. Na čas, kým otrok doručí záznamy, je na danom uzle (z ktorého otrok bol odoslaný) rozsvietená zelená dióda. Po tom, čo sa otrok dostane na uzol najbližšie k základňovej stanici, postupne odošle jednotlivé záznamy opäť pomocou plánov `sg` a `sn`. Potom otrok odošle správu agentovi, ktorý ho vytvoril, ktorou potvrdí doručenie dát a skončí svoju činnosť. Následne sa tento agent presunie na ďalší uzol a celý postup sa opakuje, až kým agent neprejde celú sieť a odošle informácie z každého uzla. Nakoniec agent zašle správu `(endlog)`, čím dá najavo, že zber informácií o topológii je kompletný. Presnejšie informácie je možné nájsť v práci [6].

9.2 Popis testovania

Pri vývoji aplikácie sa postupovalo nasledovne. Najprv bolo odlaďované fungovanie samotného interpreta, práca s tabuľkami, zásobníkom a niektoré služby platformy, konkrétne práca so zoznamom a výstup na LED. Neskôr boli implementované ostatné služby platformy a modul pre komunikáciu. V rôznych fázach vývoja slúžili na testovanie krátke úseky kódu ALLL, ktoré sa používali na testovanie jednotlivých operácií, s operandmi priamo a v registroch, prípadne rôzne varianty, pokiaľ daná operácia má viacero možností spustenia. Tieto krátke úseky kódu v práci nie sú uvedené. Dôležité bolo tiež porovnanie výsledkov operácií s výstupmi platformy WSageNt, ktoré boli dostupné pomocou simulátora TOSSIM. Niektoré testovacie príklady boli uvedené v tejto kapitole a nasleduje popis, ako boli pre testovanie použité.

9.2.1 Blikanie s LED

Po prvej fáze vývoja bolo možné otestovať blikanie s LED diódami, pomocou volania plánov a v druhom prípade ešte operácií s `BeliefBase` agenta n-ticami. Popis týchto príkladov je v tabuľkách 9.1 a 9.2. Výstupom je vizuálne overenie blikania podľa zadaných predpisov, ktoré bolo úspešne vykonané.

9.2.2 Senzor teploty

Prácu so senzorom teploty a logom nameraných hodnôt testuje kód tabuľky 9.3. Výsledky testov sú v prílohe A vo forme správ prijatých na základňovej stanici pomocou desktopovej aplikácie popísanej v časti 8.14. Ako bolo uvedené v kapitole 8.11, hodnoty bez desatinnej čiarky sa získavajú vynásobením teploty desiatimi. Teda napr. hodnota 196 značí 19,6

stupňa Celzia. Senzorový uzol s ID 29097 bol pre testovanie vystavený slnečnému svitu, čím došlo k jeho ohriatiu. Výstup testu, aktuálne nameraná teplota, minimum, maximum a priemer z posledných 100 nameraných hodnôt (z posledných 100 sekúnd) overuje, že teplota behom testovania naozaj stúpala.

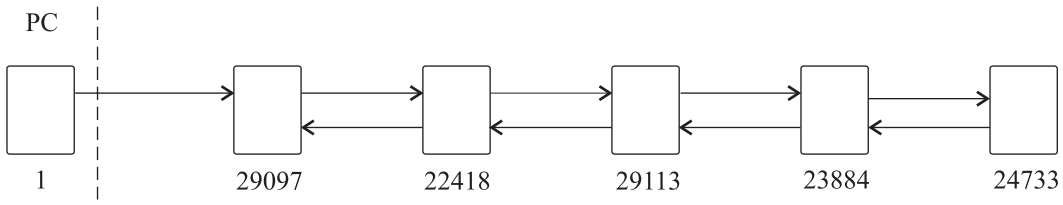
Fyzický uzol s ID 29113 potom bol pri testovaní ochladený v chladničke. Výsledky meraní ukazujú, že v danom časovom intervale 100 sekúnd teplota klesla z 24,3 na 19,6 stupňa Celzia.

9.2.3 Výpočty

Test výpočtových operácií obsahuje kód v tabuľke 9.3. Výstup tohto testu vo forme správ prijatých na základňovej stanici obsahuje príloha B. Je vhodné podotknúť, že operácie berú svoje operandy v smere od konca vstupných n-tíc k ich začiatkom, preto sú výsledky tiež v tomto poradí. Jednotlivé výstupy sú overením výpočtu nad tými istými operandmi (ktoré sú tiež obsiahnuté v tabuľke 9.3) pomocou rôznych operátorov.

9.2.4 Priechod agenta sieťou a zisťovanie topológie

Kód agenta, ktorý navštívi všetky uzly siete, je v tabuľke 9.5. Príslušná podkapitola tiež obsahuje stručný popis fungovania tohto kódu. Priechod agenta bol overený na sieti obsahujúcej 5 fyzických uzlov. Agent navštívil jednotlivé uzly v poradí, ktoré je uvedené na obrázku 9.1. Agent bol, rovnako ako u predošlých testov, odoslaný do siete zo základňovej stanice, ktorá má symbolickú adresu 1. Tento kód testuje jednak funkčnosť presunov celého agenta. Ďalej potom služby pre pachové stopy a logovanie príchodu agentov, či zisťovanie sily signálu okolitých uzlov.



Obr. 9.1: Cesta agenta po sieti.

Následne bol vykonaný najzložitejší test, ktorý používa kód v tabuľke 9.6. Bol vykonaný na tej istej sieti senzorových uzlov. Tento test zozbieral dáta pre zistenie topológie fyzických uzlov v sieti. Výstup testu je v prílohe C. Ako bolo uvedené v popise tohto kódu, agent odosiela vždy na základňovú stanicu všetky záznamy z logu na každom uzle. Preto je možné vo výstupe vidieť aj záznamy predošlého agenta a sledovať, v akom poradí uzly navštívil. Pomocou tohto výstupu bol aj zostavený obrázok 9.1. Prvý agent mal ID 2 a triedu 3. Druhý agent (zisťujúci topológiu siete) mal potom ID 10 a triedu 20. Pre zostavovanie topológie sú potrebné záznamy log-u tohto agenta pre každý uzol, a informácie o okolí, rovnako pre každý uzol. Všetky tieto záznamy agent odoslal na základňovú stanicu vo forme správ. Samotné zostavenie topológie už nebolo predmetom testovania. Tento test komplexne otestoval funkčnosť platformy a využíva takmer všetky implementované príkazy ALLL okrem práce so senzormi.

Kapitola 10

Záver

V rámci tejto práce bola úspešne implementovaná agentná platforma pre senzorové uzly SunSpot. Funkčnosť riešenia bola overená pomocou testov a porovnávaním výstupov s originálnou platformou. V rámci riešenia bola snaha o čo najväčšiu kompatibilitu s platformou WSageNt. V aktuálnom stave sú obe platformy schopné interpretovať ten istý kód jazyka ALLL s rovnakým výsledkom. Boli implementované všetky príkazy ALLL, ktoré boli v čase vývoja funkčné v originálnej platforme. Vytvorená platforma tiež emuluje použitie červenej, zelenej a žltej diódy na troch z ôsmich farebných LED. Výhodou je, že použitý hardware výrazne prevyšuje možnosti uzlov IRIS a MicaZ čo sa týka RAM a flash pamäte. Je tak umožnené vytváranie dlhšieho kódu, či ukladanie väčšieho množstva dát do bázy znalostí agenta.

V niekoľkých detailoch sa platformy odlišujú. Interpret v implementovanej platforme odlišuje 2 typy chyby operácie. Prvým z nich je sémantická chyba, alebo zlyhanie operácie (napríklad nemožnosť odoslať správu a podobne), na ktorú reaguje bežným spôsobom, teda pádom aktuálneho plánu. Na syntaktickú chybu potom reaguje priamo zastavením celého interpretovania. Toto je možné považovať za menšie rozšírenie, uľahčuje sa tým napríklad ladenie napísaného kódu ALLL. Interpret v tomto prípade vždy zahlási chybu a určí pozíciu a názov plánu, kde chyba nastala. Pri interpretovaní syntakticky správneho kódu by sa tento rozdiel nemal prejaviť.

Ďalším rozdielom je, že logy senzorov a prichádzajúcich agentov boli navrhnuté tak, aby informácie v nich obsiahnuté boli k dispozícii aj po vypnutí, či reštartovaní uzla. Nie je však problém toto zmeniť jednoduchým zmazaním príslušných súborov vo flash pri zapínaní uzla. Posledným detailom je, že na rozdiel od originálnej platformy, odosielaná správa musí mať vždy tvar n-tice, v opačnom prípade nastane pád plánu. Z rozdielu hardware tiež nutne vyplýva, že hodnota senzora teploty môže mať rozdielny formát (je udávaná v stupňoch Celzia). Inak je možné považovať platformu za kompatibilnú s platformou WSageNt.

Možné rozšírenie je navrhnutie a implementácia zvláštnych príkazov ALLL pre prácu so svetelným čidlom a akcelerometrom. Rovnako by bolo možné v jazyku umožniť využitie všetkých 8 LED diód uzlov SunSpot, prípadne tiež možnosť nastavenia rôznych farieb pre každú z nich. Ďalším rozšírením by mohla byť možnosť ukladania a načítavania agentov zo súborov v PC aplikácii, ktorá slúži na komunikáciu s uzlami.

Literatúra

[1] SunSPOT API v6.0 [online]. 2010 [cit. 2013-01-07].
URL http://www.sunspotworld.com/docs/Yellow/javadoc/index.html

[2] Akyildiz, I. F.; Su, W.; Sankarasubramaniam, Y.; aj.: Wireless sensor networks: a survey [online]. *Computer Networks*, 2002 [cit. 2013-01-26]: s. 393–422.
URL http://www.larces.uece.br/~celestino/RSSF%20I/Wireless%20Sensor%20Networks%20a%20Survey.pdf

[3] Farooq, M. O.; Kunz, T.: Operating Systems for Wireless Sensor Networks: A Survey [online]. *Sensors*, ročník 11, č. 6, 2011 [cit. 2013-03-15]: s. 5900–5930, ISSN 1424-8220, doi:10.3390/s110605900.
URL http://www.mdpi.com/1424-8220/11/6/5900

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

[5] Horáček, J.; Zbořil, F.: WSageNt: A case study. In *Proceedings of CSE 2010 International Scientific Conferenceon Computer Science and Engineering*, Volume 1, The University of Technology Košice, 2010, ISBN 978-80-8086-164-3, s. 258–264.

[6] Houšť, M.: *Monitorování stavu bezdrátových senzorových sítích agenty.* Diplomová práce, FIT VUT v Brně, Brno, 2011.

[7] Janoušek, V.: Inteligentní systémy. 2012, podklady k přednáškám předmětu SIN na FIT VUT.

[8] Kabutz, H. M.: Determining Memory Usage in Java [online]. 2001 [cit. 2012-11-25].
URL http://www.javaspecialists.eu/archive/Issue029.html

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

[10] Levis, P.; Gay, D.: *TinyOS programming.* Cambridge University Press, 2009 [cit. 2013-02-19].
URL http://csl.stanford.edu/~pal/pubs/tos-programming-web.pdf

[11] Levis, P.; Lee, N.; Welsh, M.; aj.: TOSSIM: accurate and scalable simulation of entire TinyOS applications [online]. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, ACM, 2003 [cit. 2013-03-14], ISBN 1-58113-707-9, s. 126–137.
URL http://doi.acm.org/10.1145/958491.958506

[12] Lewis, F. L.: Wireless Sensor Networks [online]. 2004 [cit. 2013-01-17].
URL http://arri.uta.edu/acs/networks/WirelessSensorNetChap04.pdf

[13] Martincic, F.; Schwiebert, L.: Introduction to Wireless Sensor Networking [online]. 2005 [cit. 2013-01-22].
URL
http://media.wiley.com/product_data/excerpt/24/04716847/0471684724.pdf

[14] Rao, A. S.; Georgeff, M. P.: BDI Agents: From Theory to Practice [online]. In *1st International Conference on Multi Agent Systems (ICMAS 1995)*, San Francisco, CA, USA: The MIT Press, jún 1995 [cit. 2013-02-03], ISBN 0-262-62102-9, s. 312–319.
URL http:
//www.upv.es/sma/teoria/teoria_ag/bdi%20agents%20from%20theory-Rao.pdf

[15] Russell, S. J.; Norvig, P.; Candy, J. F.; aj.: *Artificial intelligence: a modern approach*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996, ISBN 0-13-103805-2.

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

[17] Sun Labs: *Sun SPOT Theory of Operation*. Jún 2009 [cit. 2012-11-20].
URL http://www.sunspotworld.com/docs/Red/SunSPOT-TheoryOfOperation.pdf

[18] Sun Labs: *Solarium User's Guide*. Júl 2009 [cit. 2012-12-08].
URL http://www.sunspotworld.com/docs/Red/SolariumUsersGuide.pdf

[19] Sun Labs: *Sun SPOT Programmer's Manual*. November 2010 [cit. 2012-11-05].
URL
http://www.sunspotworld.com/docs/Yellow/SunSPOT-Programmers-Manual.pdf

[20] Zbořil, F.: Agentní a multiagentní systémy. 2006, studijní opora k předmětu AGS na FIT VUT.

[21] Zbořil, F.: Agent systems, basic terms, architectures, models and realizations. 2011, podklady k přednáškám předmětu AGS na FIT VUT.

[22] Zbořil, F.; Spáčil, P.: Automata for Agent Low Level Language Interpretation. In *Proceedings of UKSim 2009*, IEEE Computer Society, 2009, ISBN 978-0-7695-3593-7, str. 6.

[23] Zbořil, F.; Zbořil, V. F.: Simulation of Wireless Sensor Networks with Intelligent Nodes. In *10th International Conference on Computer Modelling and Simulation*, IEEE Computer Society, 2008, ISBN 0-7695-3114-8, str. 6.

[24] Žídek, P.: *Použití inteligentních agentů v bezdrátových senzorových sítích*. Diplomová práce, FIT VUT v Brně, Brno, 2011.

Zoznam príloh

Príloha A: Výstup testu práce so senzorom teploty
Príloha B: Výstup testu výpočtových operácií
Príloha C: Výstup testu topológie siete
Príloha D: CD obsahujúce zdrojové texty agentnej platformy a textu tejto diplomovej práce
Príloha E: Plagát prezentujúci vytvorenú agentnú platformu

Dodatok A

Výstup testu práce so senzorom teploty

```
ID SunSpotu: 29097, správa: '(teplota,(281))'.
ID SunSpotu: 29097, správa: '(max,(281))'.
ID SunSpotu: 29097, správa: '(min,(257))'.
ID SunSpotu: 29097, správa: '(priemer,(270))'.

ID SunSpotu: 29113, správa: '(teplota,(196))'.
ID SunSpotu: 29113, správa: '(max,(243))'.
ID SunSpotu: 29113, správa: '(min,(196))'.
ID SunSpotu: 29113, správa: '(priemer,(216))'.
```

Dodatok B

Výstup testu výpočtových operácií

```
ID SunSpotu: 29097, správa: '(((min),0),((min),-100))'.
ID SunSpotu: 29097, správa: '(((not),1),((not),0))'.
ID SunSpotu: 29097, správa: '(((cpy),0),((cpy),100))'.
ID SunSpotu: 29097, správa: '(((mul),600),((mul),10000),((mul),0),((mul),0))'.
ID SunSpotu: 29097, správa: '(((div),16),((div),1),((div),0),((div),0))'.
ID SunSpotu: 29097, správa: '(((mod),4),((mod),0),((mod),0),((mod),0))'.
ID SunSpotu: 29097, správa: '(((add),106),((add),200),((add),6),((add),100))'.
ID SunSpotu: 29097, správa: '(((sub),94),((sub),0),((sub),-6),((sub),-100))'.
ID SunSpotu: 29097, správa: '(((les),0),((les),0),((les),1),((les),1))'.
ID SunSpotu: 29097, správa: '(((mor),1),((mor),0),((mor),0),((mor),0))'.
ID SunSpotu: 29097, správa: '(((leq),0),((leq),1),((leq),1),((leq),1))'.
ID SunSpotu: 29097, správa: '(((meq),1),((meq),1),((meq),0),((meq),0))'.
ID SunSpotu: 29097, správa: '(((equ),0),((equ),1),((equ),0),((equ),0))'.
ID SunSpotu: 29097, správa: '(((neq),1),((neq),0),((neq),1),((neq),1))'.
ID SunSpotu: 29097, správa: '(((and),1),((and),1),((and),0),((and),0))'.
ID SunSpotu: 29097, správa: '(((orr),1),((orr),1),((orr),1),((orr),1))'.
```

Dodatok C

Výstup testu topológie siete

```
ID SunSpotu: 29097, správa: '(LOG,29097,1,10,20,1)'.
ID SunSpotu: 29097, správa: '(LOG,29097,22418,2,3,9)'.
ID SunSpotu: 29097, správa: '(LOG,29097,1,2,3,1)'.
ID SunSpotu: 29097, správa: '(M,29097,24733,136)'.
ID SunSpotu: 29097, správa: '(M,29097,29113,165)'.
ID SunSpotu: 29097, správa: '(M,29097,22418,144)'.
ID SunSpotu: 29097, správa: '(M,29097,23884,157)'.
ID SunSpotu: 29097, správa: '(LOG,22418,29097,10,20,2)'.
ID SunSpotu: 29097, správa: '(LOG,22418,29113,2,3,8)'.
ID SunSpotu: 29097, správa: '(LOG,22418,29097,2,3,2)'.
ID SunSpotu: 29097, správa: '(M,22418,24733,161)'.
ID SunSpotu: 29097, správa: '(M,22418,23884,157)'.
ID SunSpotu: 29097, správa: '(M,22418,29113,144)'.
ID SunSpotu: 29097, správa: '(M,22418,29097,144)'.
ID SunSpotu: 29097, správa: '(LOG,23884,22418,10,20,3)'.
ID SunSpotu: 29097, správa: '(LOG,23884,24733,2,3,6)'.
ID SunSpotu: 29097, správa: '(LOG,23884,29113,2,3,4)'.
ID SunSpotu: 29097, správa: '(M,23884,29097,157)'.
ID SunSpotu: 29097, správa: '(M,23884,29113,144)'.
ID SunSpotu: 29097, správa: '(M,23884,22418,157)'.
ID SunSpotu: 29097, správa: '(M,23884,24733,172)'.
ID SunSpotu: 29097, správa: '(LOG,29113,23884,10,20,4)'.
ID SunSpotu: 29097, správa: '(LOG,29113,23884,2,3,7)'.
ID SunSpotu: 29097, správa: '(LOG,29113,22418,2,3,3)'.
ID SunSpotu: 29097, správa: '(M,29113,23884,138)'.
ID SunSpotu: 29097, správa: '(M,29113,29097,163)'.
ID SunSpotu: 29097, správa: '(M,29113,22418,148)'.
ID SunSpotu: 29097, správa: '(M,29113,24733,157)'.
ID SunSpotu: 29097, správa: '(LOG,24733,29113,10,20,5)'.
ID SunSpotu: 29097, správa: '(LOG,24733,23884,2,3,5)'.
ID SunSpotu: 29097, správa: '(M,24733,23884,159)'.
ID SunSpotu: 29097, správa: '(M,24733,22418,153)'.
ID SunSpotu: 29097, správa: '(M,24733,29113,144)'.
ID SunSpotu: 29097, správa: '(M,24733,29097,131)'.
ID SunSpotu: 29097, správa: '(endlog)'.
```