VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PLATFORMA PRO MOBILNÍ AGENTY V BEZDRÁTOVÝCH SENZOROVÝCH SÍTÍCH

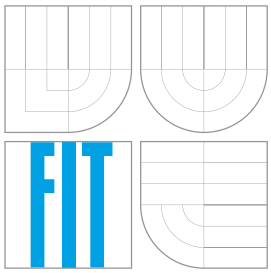
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

JAN HORÁČEK

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PLATFORMA PRO MOBILNÍ AGENTY V BEZDRÁTOVÝCH SENZOROVÝCH SÍTÍCH

PLATFORM FOR MOBILE AGENTS IN WIRELESS SENSOR NETWORKS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — JAN HORÁČEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZBOŘIL FRANTIŠEK, Ph.D.
SUPERVISOR

BRNO 2009

Vysoké učení technické v Brně - Fakulta informačních technologií
Ústav inteligentních systémů Akademický rok 2008/2009

Zadání diplomové práce

Řešitel: **Jan Horáček**
Obor: Inteligentní systémy
Téma: **Platforma pro mobilní agenty v bezdrátových senzorových sítích**
Kategorie: Umělá inteligence

Pokyny:

1. Seznamte se s tvorbou aplikací pro bezdrátové senzorové sítě, které jsou založeny na techologiích firmy CrossBow a mikrokontrolerech ATMEL128L.
2. Navrhněte architekturu platformy pro mobilní agenty implementované v jazyce ALLL, která by umožňovala existenci těchto agentů v senzorových uzlech, jejich správnou činnost a to včetně komunikace a agentní mobility mezi platformami.
3. Implementujte tuto platformu a ověřte, že splňuje výše uvedené požadavky.
4. Využijte dodaného interpretru jazyka ALLL a začeňte jej do systému. Na jednoduchých přikladech ověřte fungování platformy.
5. Zhodnoťte nároky a možnosti využití inteligentních mobilních agentů v prostředí bezdrátových sítí a uveďte případné další možnosti rozšíření těchto typů agentních platforem.
6. Zhotovte poster formátu A2, který bude názorně demonstrovat vytvořené dílo, jeho architekturu, principy a použitelnost pro reálné aplikace.

Abstrakt

Práce pojednává o návrhu agentní platformy umožnující běh agentů v bezdrátových senzorových sítích. Implementace je provozována na platformě MICAz, používající k vývoji aplikací operačního systému TinyOS. Práce obsahuje seznam vybraných částí TinyOS a názorně ukáže, jak je lze pro implementaci agentní platformy využít. Popíšeme základní charakteristiky jazyka ALLL a uvedeme příklady, jak s jeho pomocí vytvořit agenta.

Abstract

This work deals with implementation of an agent platform, which is able to run agent code in wireless sensor networks. Implementation has been done for MICAz platform, which uses TinyOS operating system for developing applications. This work contains list of chosen TinyOS parts and illustrates, how such a platform can be used for our purposes. We will describe main features of ALLL language and we will also demonstrate some examples of agents.

Klíčová slova

TinyOS, agent, BDI, senzor, bezdrátová síť, nesC, interpret, ALLL, simulace, MICAz

Keywords

TinyOS, agent, BDI, sensor, wireless network, nesC, interpret, ALLL, simulation, MICAz

Citace

Jan Horáček: Platforma pro mobilní agenty v bezdrátových senzorových sítích, diplomová práce, Brno, FIT VUT v Brně, 2009

Platforma pro mobilní agenty v bezdrátových senzorových sítích

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana Ing. Františka Zbořila. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Horáček
25. května 2009

Poděkování

Především bych chtěl poděkovat panu Františku Zbořilovi, jakožto vedoucímu práce, s jehož pomocí byla práce vypracována. Veliký podíl na celkové funkčnosti má také kolega Pavel Spáčil, jehož bakalářskou prací byla implementace části interpretu. Chtěl bych vyzdvihnout především týmovou komunikaci, spolupráci na řešených problémech a touto cestou oboum zmíněným velmi poděkovat.

Obsah

Kapitola 1

Úvod

K tomu abychom si lépe představili co to vlastně bezdrátová senzorová síť znamená a kde nachází své uplatnění by bylo dobré říci nejprve něco z historie. V dobách raného rozvoje výpočetní techniky existovala obrovská výpočetní centra velikosti budov. Z dnešního pohledu byl výpočetní výkon minimální a naprosto neporovnatelný s výkonem dnešních výpočetních center. U takovýchto typů systémů je hlavním cílem maximální výkon celé soustavy.

Pojem osobní počítač, zkráceně PC[1], vznikl v 70. letech minulého století. Jednalo se o pracovní stanice, tak jak je známe dnes. Myšlenkou bylo nabídnout počítače veřejnosti, neboli umožnit každému mít svůj osobní počítač. V průběhu několika posledních let je vidět rozvoj laptopů, u nás nazývaných taktéž notebooků. Zde je navíc kladen důraz na mobilitu, čímž se stává počítač opravdu osobním. Viditelný je taktéž trend minimalizace.

Na příkladu vybírání vlasního notebooku je zřejmá závislost jednotlivých parametrů. Podle zaměření můžeme volit jako hlavní parametr výdrž na baterie, velikost monitoru, úložnou kapacitu či typ procesoru. Pokud chceme rychlý procesor, musíme taktéž počítat s nižší výdrží na baterie. Proto pokud jeden parametr zvolíme jako klíčový, ostatní zůstanou v pozadí.

Dalším typem zařízení, které možná ani většina populace nechápe jako počítače jsou vestavěná zařízení. Většina lidí nenazve svou novou pračku, mobilní telefon, televizi či zabezpečovací systém domu počítačem. Každé takovéto zařízení má svoji specifickou funkci, kterou vykonává.

V průmyslu velmi užívanými zařízeními jsou jednoduché mikrokontroléry řídící výrobní proces. Jejich hlavní činností je získávání dat ze senzorů a počítání výstupů ovlivňující proces. Senzory mohou měřit vzdálenost, tlak, vlhkost a další fyzikální vlastnosti. Výstupem může být řízení servomotoru, signalizace obsluze, komunikace po síti a další.

Zařízení spolu mohou komunikovat prostřednictvím sítě a to jak pomocí pevného spojení či bezdrátového. Pevným spojením může být například klasický LAN kabel, sériová linka, či optický kabel. Bezdrátovým pak WiFi sítě, bluetooth, GSM síť pro mobilní telefony a další.

Síť jednotlivých uzlů, kde každý uzel má své senzory a komunikuje prostřednictvím bezdrátové sítě nazveme bezdrátová senzorová síť. Každý uzel je obvykle samostatný mikrokontrolér, na kterém běží řídící program. Funguje jako samostatná jednotka, která snímá data ze senzorů a komunikuje s ostatními uzly. Co u takovéhoto zařízení stanovíme jako základní parametr bude spotřeba zařízení. U takových zařízení je až jako druhým parame-

[1] Personal Computer

trem výpočetní výkon a další.

Využití senzorových síťí lze ukázat na jednoduchém příkladu. Do lesa umístíme několik takovýchto uzlů snímajících teplotu. Celá síť bude sloužit jako indikace požáru v lese. Uzly není možno připojit ke zdroji napájení, a proto budou napájeny pomocí dvou tužkových baterií. Komunikaci mezi jednotlivými uzly musíme tedy kvůli omezenému množství energie omezit na možné minimum. Asi si nedokážeme představit každý druhý den projít celý les a vyměňovat baterie v uzlech. Baterie musí vydržet po dobu například dvou let. Každý uzel ale musí v pravidelných intervalech snímat teplotu ze senzoru aby bylo varování vysláno včas. Ovšem vyslání zprávy do sítě je oproti získání a zpracování dat ze senzorů energeticky o mnoho řádů náročnější.

Existuje mnoho dalších oblastí, kde lze senzorové sítě využít. Je třeba si uvědomit základní parametry jednotlivých uzlů. Nemůžeme očekávat vysoký výpočetní výkon, rychlost komunikace na úrovni WiFi, ani velké kapacity úložného prostoru oproti PC. Můžeme ovšem očekávat energeticky nenáročná zařízení se schopností vzájemné komunikace a snímání dat ze senzorů.

Pojem agent vychází z latinského slova agentem[2]. Existuje více definic, ovšem zde se pro jednoduchost zaměříme pouze na jednu z nich. Agentem rozumíme celek schopný samostatného jednání v daném prostředí v závislosti na okolních podnětech. Agent má obvykle nějaký svůj cíl, kterého se snaží dosáhnout (například úkol tajného agenta CIA). Agent se tedy zadavateli snaží přidělený úkol splnit. Agent má také sociální schopnosti, tedy možnost komunikovat s ostatními agenty. Umělý agent je vytvořený program, splňující výše uvedené požadavky, který je spjat s architekturou, na které je provozován. Dále v textu budeme pro jednoduchost chápat umělého agenta pod pojmem agent[11].

V této práci bude znázorněno, jak lze tuto agentní architekturu navrhnout s využitím bezdrátových senzorových sítí. Každý uzel lze naprogramovat jako samostatného agenta, který je schopný komunikovat s ostatními agenty[3].

Posledním požadavkem bude mobilita agentů v systému, čili možnost přesunutí agenta z jednoho uzlu na jiný. Jak této vlastnosti dosáhnout bude popsáno v dalších kapitolách.

Tato práce navazuje na stejnojmenný semestrální projekt, jehož náplní bylo seznámení se s danou problematikou. Získané poznatky můžeme nalézt v kapitolách 2 až 6.

[2] Ten, kdo jedná

[3] Taktéž uzly senzorové sítě

Kapitola 2

Agenti

Jak již bylo zmíněno, agent je ten, co koná ve prospěch někoho jiného. Zakladní vlastnosti agenta jsou podle jedné z definic[6]:

- Autonomní - bez přímého vlivu z okolí jedná v daném prostředí. Má plnou kontrolu nad svým jednáním pokud se těchto výsad nevzdá.
- Reaktivní - je schopen adekvátně a pohotově reagovat na změny prostředí.
- Proaktivní - je schopen ujímat se iniciativy a sám ovlivňovat protředí za účelem dosažení svých cílů.
- Sociální schopnosti - dokáže jednat ve skupině agentů, dále spolupracovat a řešit konflikty.

Další vlastností kterou budeme požadovat bude, aby agent byl racionální. Pokud racionální agent má znalost, že některá z akcí, kterou je schopen vykonat, povede k jeho cíli, pak ji vykoná. Tato vlastnost do určité míry souvisí s proaktivním chováním.

2.1 Praktické usuzování

Praktické usuzování je narozdíl od teoretického usuzování ne o tom co je a co není pravda, ale o tom, co musíme udělat, aby se něco pravdou stalo[11]. Má dvě části:

- Zvažování - zvolení cíle, kterého má být dosaženo. Zvolený cíl se stává záměrem.
- Plánování - nalezení akce nebo sestavení plánu, který může dosáhnout záměru. Proaktivní agent se tedy bude snažit takovýto plán vykonat.

Je důležité si uvědomit, že daný plán nemusí vyústit ve splnění záměru. Klademe pouze podmínku, že při provedení daných kroků je možnost dosažení cíle. Můžeme tedy vyřadit všechny plány, po jejichž provedení již nebudeme schopni daný záměr splnit. Je v zájmu agenta aby vybral z množiny plánů ten, kterým záměr splní či se k němu co nejvíce přiblíží.

2.1.1 Plánování u racionálních agentů

Pokud je zvolen záměr, je třeba umět sestavit plán k jeho dosažení. Agent dále nemusí mít v úmyslu všechny vedlejší účinky při dosahování záměru. Určitá posloupnost akcí může vést k nemožnosti splnit nějaký vedlejší cíl.

Plánovač sestavuje plán jako (úplně) uspořádanou posloupnost akcí, která ze známých počátečních podmínek je schopna zabezpečit splnění daného cíle. Mezi v dnešní době používané plánovače lze zařadit STRIPS, GraphPlan, SetPlan a další. Plán, který byl úspěšně sestaven, nemusí být použit okamžitě, nemusí dosáhnout předpokládaného cíle, nemusí být vykonán celý a jeho vykonávání nemusí vůbec začít.

V některých případech je plánování zcela vynecháno. V takovém případě je nahrazeno knihovnou plánů. Tyto plány byly dopředu sestaveny a pouze vybereme ten, který odpovídá naší aktuální situaci.

Plánování může být jak lokální, tak skupinové. Skupinové plánování musí navíc řešit konflikty mezi agenty, případně spolupráci několika agentů pro dosažení společného cíle. Konflikty vznikají v případech, kdy jeden agent posloupností akcí znemožní jinému agentovi dosáhnout jeho cíle. Spolupráce agentů může být velmi užitečná v případech, kdy jeden agent má určitou znalost či schopnost, kterou ostatní nemají. Poté může poskytnout své služby a znalosti ostatním agentům za případnou protislužbu. Agent může danou pomoc odmítnout, pokud není v jeho zájmu ji poskytnout, či není v daném okamžiku pro agenta výhodná. Agent obvykle uvažuje o výhodnosti zisku ze spolupráce vůči náročnosti poskytnutí pomoci.

2.2 BDI agenti

Model BDI rozděluje agenta na tři části[1]:

- Belief - znalosti o okolním světě.
- Desire - cíle, kterých se agent snaží dosáhnout.
- Intention - záměr neboli cíl, který se agent rozhodl sledovat.

Agent tedy obsahuje bázi znalostí o okolním světě a zná cíle kterých se snaží dosáhnout. Záměr je cíl, který se agent rozhodl sledovat. To většinou v praxi znamená, že agent již začal vykonávat konkrétní plán, jak daného cíle dosáhnout.

Kapitola 3

Senzorové sítě

V této části se seznámíme se senzorovými sítěmi. Zaměříme se na mikrokontroléry ATMEL128L a technologie firmy CrossBow. Budou uvedeny specifikace zařízení a jejich možnosti. Dobrý zdroj na tuto tématiku lze uvést volně dostupné pdf „Introduction to Wireless Sensor Networking“ [7].

3.1 CrossBow a její technologie

Firma CrossBow nabízí řadu řešení v oblasti senzorových sítí. Jedním z nich, které využijeme, bude sada senzorových uzlů MICAz. Nabízí například také MICA2, IRIS, Imote2, TelosB, eKo a další, kterými se ale již dále nebudeme zabývat. Kompletní seznam nabízených řešení lze získat ze stránek výrobce [**?**].

Jednotlivá zařízení jsou nazvána mote. Dále v textu se budeme tohoto označení držet. Každý MICAz mote obsahuje procesor ATMEL128L schopný zpracovat 8 milionů instrukcí za sekundu[1] při pracovní frekvenci 8 MHz. Dále obsahuje IEEE 802.15.4/Zigbee kompatibilní RF vysílač s přijímačem pracující na frekvencích 2.4-2.4835 GHz. Na této bezdrátové síti je schopný dosáhnout přenosové rychlosti 250 kbps[2]. Dále obsahuje sérivé rozhraní UART či 10ti bitový ADC převodník.

Každý mote je možno rozšířit pomocí 51 pinového rozhraní. Tímto rozhraním lze připojit specifickou senzorovou desku. Mote lze také připojit k basestation, pomocí které může být programován. Basestation se připojuje pomocí USB rozhraní k PC. Program je vytvářen s využitím na zdroje nenáročného operačního systému TinyOS, o kterém bude řeč dále. Basestation může být jak v programovacím režimu, kdy nahráváme program do mote, tak i v běžném režimu. V běžném režimu může sloužit jako propojení senzorové sítě s PC, kdy využívá konvertoru sériového rozhraní (z mote) na USB sběrnici (propojující basestation s PC). Mote umožnuje nastavit několik režimů pro řízení spotřeby, čímž lze prodloužit dobu běhu na baterie. Detailnější specifikaci platformy MICAz lze vidět v tabulce 3.1.

3.2 Připojení senzorové desky

Jak již bylo zmíněno, senzorovou desku je k mote možno připojit pomocí 51 pinového rozhraní. Součástí tohoto rozhraní je i propojení se zdrojem napájecího napětí[3]. V našem

[1] MIPS
[2] kilobitů za sekundu
[3] 2 AA baterie napájející mote

Procesor	Atmel ATMega128L @ 8 MHz
Velikost flash pro program	128 kB
Externí (sériová) flash paměť	512 kB
RAM	4 kB
Sériové rozhraní	UART
AD převodník	10 bitů
Další rozhraní	digitální I/O, I2C, SPI
Spotřeba procesoru	8 mA v normálním módu
	1 μA v úsporném módu
Frekvence rádia	2400 MHz až 2438,5 MHz
Přenosová rychlost rádia	250 kbps
RF power (minimální síla signálu)	-24 dBm až O dBm
Citlivost přijímače	-90 dBm (minimálně), -94 dBm (typicky)
Odmítnutí při odchylce od kanálu	47 dB při +5 MHz odchylce od kanálu
	38 dB při -5 MHz odchylce od kanálu
Dosah rádia (venku)	75m až 100m
Dosah rádia (uvnitř budovy)	20m až 30m
Spotřeba rádia	19,7 mA při příjmu
	11 mA při vysílání -10 dBm
	14 mA při vysílání -5 dBm
	17,4 mA při vysílání 0 dBm
	20 μA v nečinnosti (regulátor napětí zapnut)
	1 μA v uspaném režimu (regulátor napětí vypnut)
Napájení	2 AA baterie
Uživatelské rozhraní	3x LED dioda (červená, zelená a žlutá)
Rozměry	5,7 cm x 3,2 cm x 0,6 cm
Váha	20 g
Rozšiřující rozhraní	51 pinů

Tabulka 3.1: **Specifikace platformy MICAz**[7]

případě se jednalo o senzorovou desku MTS400CA. Propojením s mote pak získáme malou meteorologickou stanici. Senzorová deska obsahuje zejména:

- Teplotní senzor
- Senzor pro měření vlhkosti a atmosférického tlaku
- Senzor pro měření intenzity okolního osvětlení
- Dvouosý akcelerometr

Kapitola 4

TinyOS

TinyOS je na zdroje nenáročný operační systém používaný při vývoji aplikací pro bezdrátové senzorové sítě. Je napsán v programovacím jazyce nesC, který bude podrobněji popsán v kapitole 5. Existují dvě větve verzí tohoto operačního systému. Od verze 1.1.7 je podporována platforma MICAz. TinyOS verze 2.0 a vyšší nabízí pozměněný styl programování, zpětně nekompatibilní, ovšem velice podobný, odlišující se v několika drobnostech. Verze 2.1 dokonce přináší podporu vláken, možnost využití mutexů a další rozšíření.

TinyOS nabízí ve většině případů platformově nezávislý vývoj aplikací. Nabízí řadu knihoven, které je možné použít pro vývoj. Tato sada knihoven utváří abstrakci nad hardwarem konkrétního zařízení. Hardware platforem MICAz a Imote2 může být různý, ale díky abstrakci, kterou nabízí TinyOS je schopný běhu naprosto stejné aplikace, pouze přeložené pro konkrétní platformu.

TinyOS není operačním systémem jaké vidíme na běžných PC. Je optimalizován s ohledem na využití zdrojů, především paměti RAM, které je v mote o několik řádů méně než v dnešních PC. TinyOS nenabízí žádnou správu dynamické paměti. Paměť nemůže být přidělena v průběhu běhu programu, ale pouze při překladu, čili staticky. Taktéž pro volání funkcí nemůžeme využít klasický ukazatel na funkci, zřejmě pro bezpečnější psaní programů. Na druhou stranu můžeme použít ukazatel na data. Pokud bychom potřebovali využít dynamické přidělování paměti, budeme muset sami implementovat dynamickou paměť ve statickém poli.

TinyOS je tedy sada knihoven, která je při překladu staticky spojena s námi napsaným programem do binárního souboru. Všechny adresy funkcí a proměnných jsou dopředu známy a nemohou být měněny. Tento soubor je pak možné nahrát do zařízení.

Pro konkrétní zařízení je možné najít i sadu specifických knihoven, které jsou hardwarově závislé. Jejich využití je doporučeno pouze v ojedinělých případech, kdy potřebujeme využít funkce, které ostatní platformy nenabízí. Tím ovšem bude takto napsaná aplikace přeložitelná pouze pro konkrétní platformu.

Na nejnižší úrovni je definováno na jakých hardwarových adresách najdeme jaké zařízení. Je zde například definováno z jaké adresy se čtou data ze senzorů, kolik časovačů je umístěno v mikrokontroléru, kolika bitové jsou atd. Pokud máme specifikaci k zařízení, které není obsaženo v TinyOS, pak jej můžeme tímto způsobem do TinyOS zařadit.

4.1 Překlad

Pro překlad programu je využito standardního programu `make`. Ten hledá v lokálním adresáři soubor "`Makefile`". Pro překlad pro platformu MICAz spustíme příkaz:

```
make micaz
```

Pro platformu Imote2 uvedeme:

```
make intelmote2
```

Program make zařídí spojení lokálního makefile, ve kterém je uveden hlavní soubor projektu s makefile z TinyOS. V TinyOS je standardní makefile zadán proměnnou `MAKERULES`. K němu je připojen makefile pro specifickou platformu[1]. Dále mohou následovat další knihovny, které chceme připojit. Pro připojení vláken u platformy MICAz přeložíme pomocí:

```
make micaz threads
```

Tímto příkazem jsou přiloženy makefile "`micaz.target`" a "`threads.extra`". Tyto knihovny a rozšíření jsou přikládány explicitně, na požádání. Knihovny využívají další zdroje, především RAM, a proto nejsou implicitně přiloženy.

Lokální makefile je velice jednoduchý. Při nestandardních požadavcích ale máme možnost si tento makefile upravit k obrazu svému. Pro většinu jednoduchých programů si vystačíme s následujícím makefile:

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

4.2 TOSSim

Pro platformu MICAz existuje simulátor, který je dokonce obsažen v balíku TinyOS. Při překladu je pouze nutno uvést

```
make micaz sim
```

Je vygenerována sdílená dynamická knihovna "`_TOSSIMmodule.so`". Dále je tato knihovna propojena s programovacím jazykem Python pomocí tříd obsažených v "`TOSSIM.py`". Je možno simulovat více uzlů najednou i například komunikaci mezi uzly a to včetně kvality spojení a šumu při přenosu dat. Z programu je možno zjistit aktuální hodnoty proměnných, jejichž název a datový typ je uveden v souboru "`app.xml`". Ten je taktéž vygenerován při překladu. Debugovací řetězce a oznámení se uvádějí ve zdrojových souborech makrem `dbg`. To obsahuje i název kanálu, do kterého bude oznámení vloženo. Výpis hlášky do kanálu `TEST` bude vypadat následovně:

```
dbg("TEST", "Promenna i se rovna: %d\n", i);
```

Tu lze následně odchytit v testovacím skriptu. Ten je psán v programovacím jazyce Python a využívá tříd z dříve vygenerovaného souboru `TOSSIM.py`. Pro správnou funkci je třeba přidat cestu k sdk do proměnné `PYTHONPATH`. Pro představu jak může testovací skript vypadat:

[1] V prvním uvedeném příkladě "`micaz.target`", ve druhém "`intelmote2.target`"

Soubor ‘‘test.py’’

```
#! /usr/bin/python
from TOSSIM import *
import sys
t = Tossim([])
t.addChannel("TEST", sys.stdout)
t.getNode(1).bootAtTime(100001);
for i in range(0, 1000):
  t.runNextEvent()
```

Na prvním řádku je uvedeno o jaký typ souboru se jedná. Na druhém řádku vložíme třídy z vygenerovaného souboru “TOSSIM.py”. Na dalším řádku vložíme systémové knihovny, abychom mohli využívat výpisu na obrazovku. Další příkaz vytvoří základní objekt, který řídí celou simulaci. Následuje navázání kanálu TEST na standartní výstup. Zbylé řádky spustí simulaci a provedou 1000 kroků. Jedná se o opravdu jednoduchý testovací skript. TinyOS verze 1 obsahovalo i grafické uživatelské rozhraní, na druhou stranu nebyli dostupné některé funkce z verze 2.

Testovací program je taktéž možno psát v C++. Objekty a jejich funkce jsou si velmi podobné, jedná se pouze o přepis syntaxe. V tomto případě bude využívána pouze sdílená dynamická knihovna “_TOSSIMmodule.so”. Stejný příklad ovšem v C++ syntaxi vypadá následovně:

Soubor ‘‘test.cpp’’

```
#include <tossim.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main() {
  Tossim* t = new Tossim(NULL);
  Mote* m = t->getNode(1);
  t->addChannel("TEST",stdout);
  m->bootAtTime(100001);
  for (int i = 0; i < 1000; i++) {
    t->runNextEvent();
  }
}
```

4.3 Nahrání programu do mote

Basestation může být k PC připojena různými způsoby (COM, USB, bezdrátově, ...). V našem případě bude využito pro platformu MICAz desky MIB520, která je připojována pomocí USB. Jedná se o konvertor sériového rozhraní na USB. V linuxu vytvoří v adresáři “/dev” dvě zařízení. Jedno slouží pro nahrávání programu a konfiguraci mote (obvykle “/dev/ttyUSB0”), druhé pak pro samotnou komunikaci s basestation (obvykle “/dev/ttyUSB1”). Samotnou konfiguraci mote provedeme příkazem:

```
make micaz install,<node_id> mib520,/dev/ttyUSB0
```

Do příkladu je pouze nutno dosadit číselnou hodnotu <node_id>. Jedná se o jedinečný identifikátor pro odlišení uzlů.

4.4 Java rozhraní pro komunikaci

TinyOS obsahuje sadu komponent, s jejichž pomocí jsme schopni komunikovat s mote v basestation. Jedná se o rozhraní a několik aplikací napsaných v jazyce Java. Pro správnou funkci je třeba přidat cestu k sdk do proměnné `CLASSPATH`. Zmíníme několik pro nás využitelných aplikací. Nejprve je ale u aplikací nutné pozměnit makefile a přidat několik řádek. Pro komunici je definován abstraktní datový typ pro pakety, komunikace bude popsána v kapitole 6.1.6. Aby třídy z sdk věděli jak jsme si sestavili datovou část paketu, musíme použít konvertoru kódu z nesC (případně C) do jazyka Java. Makefile pak bude obsahovat řádek tvaru:

```
RadioMsg.java: NasPaket.h
        mig java -target=$(PLATFORM) $(CFLAGS) -java-classname=RadioMsg \
        NasPaket.h radio_msg -o $@
```

je využito transformačního programu `mig.java`, který vezme soubor "`NasPaket.h`", zjistí hodnodnotu `radio_msg`[2] a z těchto informací vygeneruje soubor "`RadioMsg.java`". Příkazem `javac` dostáváme bytecode, který již budeme moci spojit s aplikacemi z TinyOS sdk. Na začátku makefile dále přidáme do proměnné `BUILD_EXTRA_DEPS` části, které se mají při překladu spustit také.

4.4.1 Listen

Tato hotová aplikace zobrazuje přijaté pakety. Po spuštění naslouchá a zobrazuje jednotlivé byty paketu v hexadecimálním tvaru. Každý paket je na samostatném řádku. Jedná se o jednoduchou aplikaci, která nepotřebuje pro svůj běh znát strukturu paketu[3]. Spuštění této aplikace lze provést příkazem:

```
java net.tinyos.tools.Listen -comm serial@/dev/ttyUSB1:micaz
```

Parametr `comm` udává cestu k zařízení pro komunikaci s mote a jeho typ. Zde se budeme připojovat přes zařízení "`/dev/ttyUSB1`", které simuluje sériovou linku. Dále je možno uvést přenosovou rychlost rozhraní v baudech, případně uvést skratkou `micaz`. V tomto případě bude přenosová rychlost doplněna automaticky. Parametr `comm` je možno uvést téměř ve všech aplikacích z sdk.

4.4.2 MessageListener

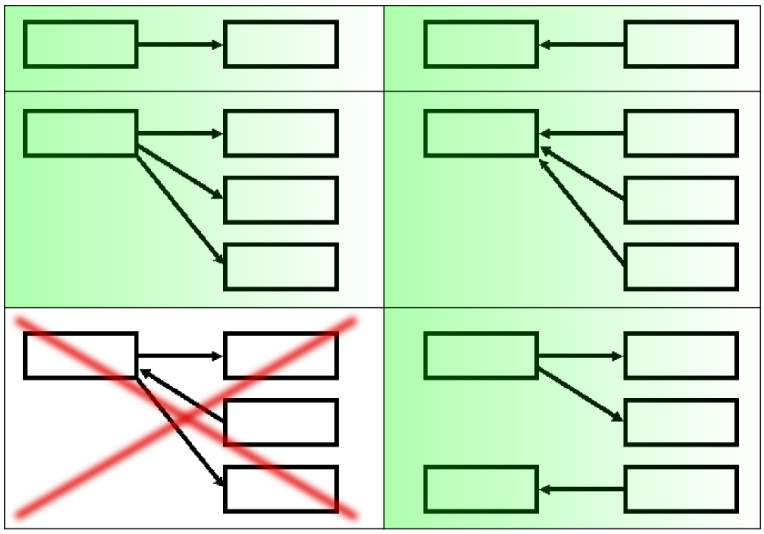
V tomto případě je možnost napsání vlastní aplikace, která bude zpracovávat příchozí pakety a taktéž je bude schopna odesílat. Odesílání a čekání na příchozí paket je odděleno pomocí dvou samostatných vláken. V příloze **A** na straně 53 je ukázána jedna vzorová aplikace. Aplikace je odvozena od třídy `MessageListener`, která také podporuje zpracování parametru `comm`. Třída musí znát, s jakým typem paketu bude pracovat, proto je nutná migrace kódu s využitím dříve zmiňovaného `mig java`.

[2] Identifikátor našeho paketu, kterým se odlišuje od ostatních paketů.
[3] Nerozkládá paket na hlavičku, datovou část, ...

Kapitola 5

Programovací jazyk nesC

Programovací jazyk nesC je do značné míry podobný a kompatibilní s jazykem C, ze kterého vychází. Důvodem bylo stanovit techniky programování takovým způsobem, aby se minimalizovala spotřeba při běhu aplikace. Hlavní odlišností a charakteristikou jazyka nesC je komponentní návrh. Každá komponenta představuje samostatnou funkční jednotku, která má své rozhraní. Pomocí rozhraní může komunikovat s ostatními komponentami. Rozhraní definuje jakési přípojné body. Každý bod obsahuje dvě strany. Na jednu stranu se připojují ty body, které vysílají událost[1]. Na druhou stranu se připojují ty body, které události přijímají. Obrázek 5.1 ukazuje, jak správně zapojit obě strany rozhraní. Na každé straně může být i více bodů, které jsou propojeny do jednoho bodu (na opačné straně). Ovšem je třeba dodržovat směr toku dat.



Obrázek 5.1: **Komponentní návrh**
Zeleně pět správných zapojení, špatné zapojení je červeně přeškrtnuto.

Tento styl programování má v senzorových sítích veliké opodstatnění. Každá komponenta může dokončit svou činnost a vyslat událost jiné komponentě, která v činnosti pokračuje, či vysílat události průběžně. Tím by se ovšem přístup nelišil od klasického volání

[1] Bodů může být více, z více komponent, ovšem se stejným identifikátorem

funkcí v čistém C. Změna přichází při čekání na nějakou událost, obvykle hardwarového charakteru. Pokud čekáme na přerušení od časovače, či na příjem paketu ze sítě, pak může komponenta, která se stará o již zmiňovaný časovač (rádio, ...) vyslat událost komponentě, která na ni čeká. Při programování v C je pro tento účel využito callback funkcí. Využití komponent, jejich rozhraní a generování událostí lépe prezentuje skutečnost a je lépe vidět tok dat[4].

Událostmi řízený přístup nutí programátora psát části komponent tak, aby po dokončení jejich činnosti mohl být procesor uspán. To vše za předpokladu že se nevyskytly další události.

5.1 Moduly v nesC

Modul je v nesC chápán jako komponenta, pro další označení budou oba výrazy ekvivalentní. Každý modul má svou specifikaci, která definuje strukturu komponenty, včetně typu rozhraní, pomocí kterého může být spojen s okolními komponentami. Specifikace může vypadat následovně:

```
module SmoothingFilterC {
  provides command uint8_t topRead(uint8_t* array, uint8_t len);
  uses command uint8_t bottomRead(uint8_t* array, uint8_t len);
}
```

Za klíčovým slovem `implementation` následuje implementační část modulu. Tato část může obsahovat mixovaný kód jazyků nesC a C. V kapitole 6 bude na příkladech názorně ukázáno jak vypadá implementační část.

5.2 Rozhraní v nesC

Rozhraní, jak již bylo naznačeno dříve, standardizuje propojení dvou či více komponent. Strany rozhraní jsou označeny. Komponenta, která poskytuje nějaké funkce, případně informuje o výskytu nějaké události, označuje stranu rozhraní pomocí klíčového slova `provide`. Pokud se podíváme do zdrojových kódů TinyOS, pak většina komponent své rozhraní poskytuje. Druhá strana rozhraní používá a je označena klíčovým slovem `uses`. V našich aplikacích to budou právě námi napsané komponenty, které budou využívat například LED, časovačů, rádia a další. Tato rozhraní budou v našich aplikací ve většině případů označena jako `uses`.

V příkladu znázorňujícím specifikaci modulu bylo ukázáno jak poskytnout ostatním komponentám funkci *topRead*. Modul bude pro svou činost využívat funkce *bottomRead*. O funkci, neboli příkaz, se jedná díky klíčovému slovu `command`. Funkce jsou poskytovány modulům, které je využívají a volají. Implementace funkce je tedy v modulu, který funkcí poskytuje. Jako příklad je možno uvést volání systémové funkce, která nám rosvítí LED diodu. Oproti tomu je možno využít signálů, označených pomocí klíčového slova `signal`. V klasickém C by se dal signál chápat callback funkcí. Signál je vyslán z komponenty, která jej poskytuje a je zachycen a zpracován v komponentě, která ho používá. Kód reagující na signál je tedy v modulu, který signál využívá. Příkladem může být informování o příjmu paketu ze sítě.

V praxi jen velmi zřídka deklarujeme jednotlivé funkce ve specifikaci modulu. Je proto vhodné využít klíčového slova `interface`. Jedná se o klasické rozhraní, jak jej chápeme

z programovacích jazyků jako je například Java. Tento abstraktní datový typ umožňuje sdružit několik funkcí a signálů a tomuto sdružení přiřadit jméno. Jednoduchý příklad rozhraní:

```
interface StdControl {
  command error_t start();
  command error_t stop();
}
```

Rozhraní definuje dva příkazy, jeden pro zastavení a jeden pro běh. Oba příkazy vracejí hodnotu datového typu `error_t`. Příklad jak využít takto pojmenované rozhraní v modulu:

```
module RoutingLayerC {
  provides interface StdControl;
  uses interface StdControl as SubControl;
}

module PacketLayerC {
  provides interface StdControl;
}
```

Na příkladu je dobré si povšimnout klíčového slova `as`. Takto jednoduše přejmenujeme název rozhraní uvnitř modulu. Příklad, kdy je nutné využít přejmenování může být využití dvou časovačů. Pokud budeme chtít využít dva časovače, každý s jiným nastavením, musíme být schopni je od sebe odlišit.

5.3 Spojování modulů v nesC

O spojování komponent se v nesC starají konfigurace, které jsou označeny klíčovým slovem `configuration`. Uvnitř každé konfigurace definujeme jaké komponenty obsahuje a jak jsou spolu propojeny. Příklad jednoduché konfigurace:

```
configuration BlinkAppC
{
}
implementation
{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  BlinkC.Boot -> MainC.Boot;
  BlinkC.Timer0 -> Timer0;
  BlinkC -> LedsC.Leds;
}
```

V tomto případě kofigurace obsahuje čtyři komponenty. Jsou označeny klíčovým slovem `components`. Komponenty MainC a LedsC jsou systémové komponenty z TinyOS. Taktéž komponenta TimerMilliC je systémovou komponentou, ovšem virtální. Vygeneruje jedinečný identifikátor, kterým se bude odlišovat od ostatních časovačů a ten předá komponentě TimerMilliP. Proto je nutné uvést klíčové slovo `new`. Opět je možné používat přejmenování. Komponenta BlinkC je námi napsanou komponentou. Obsahuje rozhraní Boot, rozhraní Timer<TMilli> přejmenované na Timer0 a poslední rozhraní Leds.

Existují tři možné operátory pro spojování komponent a to <−, −> a =. Operátory −> a <− slouží pro zapojování komponent uvnitř konfigurace. Směr šipky směřuje od uživatele k poskytovateli. Proto při uvedení poskytovatele na pravé straně, použijeme operátor −>. Důležité je, že konfigurace může, stejně jako moduly, poskytovat a používat rozhraní. Poté může být konfigurace chápána navenek jako modul a je možné vytvořit hierarchii konfigurací. Pro rozhraní, které směřují ven z konfigurace, či z venku dovnitř bude využit operátor =.

Z příkladu je vidět další vlastnost nesC. Při spojování rozhraní Boot je na obou stranách uvedeno o jaké rozhraní se jedná. Jedná-li se o stejně pojmenované rozhraní, pak je možné jeho pojmenování na jedné ze stran vynechat. Takto je definováno rozhraní Leds. U rozhraní Timer0 na straně BlinkC bohužel tuto vlastnost využít nelze. Důvodem je pojmenování u komponenty Timer0, které poskytuje rozhraní Timer<TMilli>, kdežto u komponenty BlinkC bylo rozhraní přejmenováno.

5.4 Volání funkcí a odlišnosti od jazyka C

Příkaz může být zavolán od uživatele rozhraní pomocí klíčového slova `call`. Kód je spuštěn na straně poskytovatele a návratová hodnota je vrácena zpět uživateli. Oproti tomu signál je vyslán pomocí klíčového slova `signal` ze strany poskytovatele k uživateli a návratová hodnota vrácena zpět poskytovateli.

Existuje taktéž možnost spustit novou úlohu uvozenou pomocí `task`. Spuštění nové úlohy se spouští klíčovým slovem `post`. O přepínání mezi procesy se pak stará plánovač. Bohužel takto vytvořený proces nelze pozastavit. S příchodem TinyOS 2.1 máme možnost použít vlákna, ovšem této možnosti se vzhledem k náročnosti na zdroje vyhneme.

Asi největší odlišností a jedinou zde zmíněnou je definice konstanty. Číselné konstanty se neuvozují pomocí `const` tak, jak známe z C, ale jsou definovány pomocí výčtového typu. Konstanty můžeme uložit do hlavičkového souboru a ten pomocí `#include` vložit. Příklad, jak vytvořit konstantu v nesC je následující:

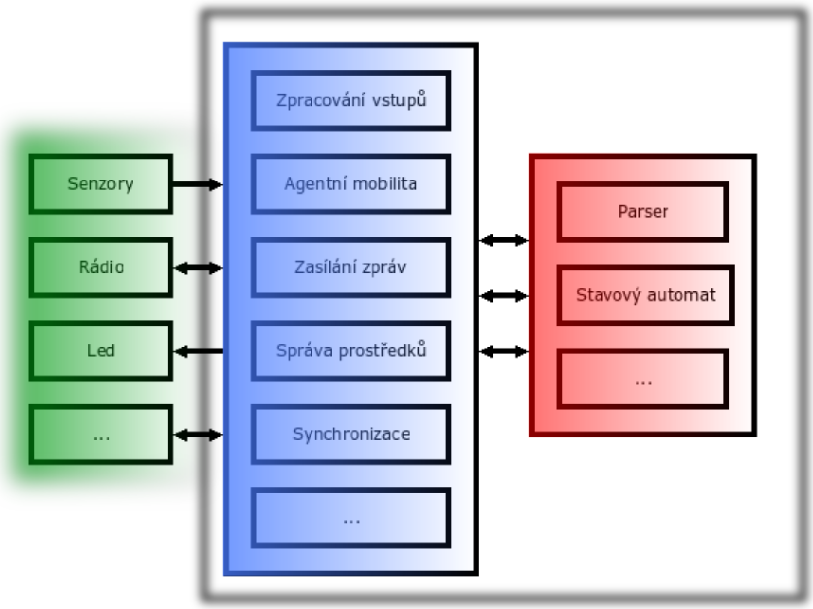
Soubor ‘‘konstanty.h’’

```
1 #ifndef KONSTANTY_H
2 #define KONSTANTY_H
3 enum {
4   NASE_KONSTANTA1 = 9,
5   NASE_KONSTANTA2 = 45,
6   NASE_KONSTANTA3 = 5,
7 };
8 #endif
```

Kapitola 6

Návrh platformy pro agenty

Problémem, který nám situaci komplikuje, je především statické přidělení paměti při překladu programu, které nelze měnit. Vzhledem k tomu, že mote MICAz neumožňují nahrání nového programu přes bezdrátovou síť, nelze vykonávat kód agenta přímo. Důvodem je agentní mobilita. V některých situacích je třeba agenta nahrát, či zkopírovat na jiný uzel. Pokud bychom napsali aplikaci v nesC a implementovali bychom v ní chování agenta přímo, nebyli bychom schopni tento kód přesunout na jiný uzel. Jediným řešením by bylo přeprogramování mote v basestation, což je velmi zdlouhavé a nepraktické. Navíc agent sám o sobě by se nemohl rozhodnout a sám sebe přesunout na jiný uzel. Proto budeme muset kód interpretovat. Návrh byl rozdělen na dvě části, které jsou znázorněny na obrázku 6.1. Zeleně



Obrázek 6.1: **Rozdělení na dva spolupracující celky**
Tlusté šedé orámování znázorňuje naši aplikaci. Modře je vyznačena část platformy (uprostřed), červeně interpretu (vpravo).

je vyznačena část systémových modulů a rozhraní poskytovaná TinyOS. V šedém orámování je celá naše aplikace, která se stará o interpretování agenta, zasílání zpráv, agentní mobilitu, čtení ze senzorů a celou řadu dalších funkcí. Aplikace byla rozdělena na dva celky. Červeně je znázorněna část samotného interpretu kódu. Modře pak blok podpůrných funkcí, které se starají o čtení dat ze senzorů a komunikaci, správnou synchronizaci a řadu dalších věcí. Tato část byla nazvána platformou a tvoří mezivrstvu mezi systémovými knihovnami a interpretem.

Interpret bude vykonávat kód agenta, napsaného v jazyce ALLL. Jedná se o velice jednoduchý jazyk, který je možné interpretovat pomocí stavového automatu. Interpret tedy vždy provede jeden přechod ze současného stavu do stavu nového. Při této akci má možnost volat pouze jednu funkci poskytovanou platformou. Tato podmínka volání maximálně jedné funkce platformy je velice důležitá. Jinak by se mohlo vyskytnout přepsání nějaké proměnné na straně platformy, či rekurentnímu volání funkcí. V prostředí senzorových sítích si nemůžeme dovolit chybně ukazující ukazatale do paměti či podobné chyby za běhu programu. Ladění aplikací pro senzorové sítě je taktéž problematičtější než na klasickém PC. Musíme proto přistoupit k jistým programovacím stylům, které výskyt chyb v programu eliminují na co možná nejměnší počet. Omezení počtu volání platformních funkcí je sice do jisté míry svazující, ovšem výpočetně ekvivalentní, ale hlavně vede ke stabilnějšímu kódu.

Platforma se stará o komunikaci se systémovými moduly. Interpretu pak nabízí abstraktnější funkce a řídí běh naší aplikace. Základní rozhraní které musí interpretu poskytnout je sada řídících signálů, především:

- Inicializace - při příjmu kódu agenta nastavit prostředí interpretu tak, aby bylo schopné začít vykonávat kód.
- Provedení jednoho kroku - předání hlavního řízení interpretu pro vykonání přechodu z jednoho stavu do druhého. Interpret nás zároveň informuje, zda-li si přeje být spuštěn znovu, či čeká na nějakou událost a přeje si být pozastaven.
- Informování o změně informací o okolí, báze znalostí a dalších událostí - probuzení z čekání na událost (např. vypršení časovače), či jiná událost, kterou informuje o změně okolí (příjem zprávy od jiného agenta).

Jak část platformy, tak i interpretu bude vložena do samostatné konfigurace. Navzájem budou propojeny pomocí rozhraní a budou vloženy v nadřazené zobalovací konfiguraci. Toto řešení umožňuje rozdělení na dvě více nezávislé části. Každá část si pak může definovat svá vnitřní rozhraní a zapojení jednotlivých komponent.

6.1 Systémové komponenty TinyOS

V dalších kapitolách bude popsána především část platformy. Část interpretu je součástí jiné práce a zde budou zmíněny pouze ty části, které implementačně souvisejí s návrhem platformy. Bude zde zmíněno jak lze pro konkrétní funkci využít systémových modulů TinyOS a případná možná řešení.

6.1.1 Inicializace

Z TinyOS bude pro tento účel využito modulu MainC, který nám nabízí rozhraní `Boot`. To vždy při spuštění mote (připojení napájení) vyšle signál *booted*. Příklad modulu, který

pouze inicializuje proměnnou může vypadat následovně:

```
module PrikladInitC {
  uses interface Boot;
}
implementation {
  int i = 0;

  event void Boot.booted() {
    i = 1;
  }
}
```

V implementační části je nejprve definována proměnná `i`. Při nahrávání programu do paměti mote je na adresu této proměnné zapsána hodnota 0. Jedná se o běžný kód v jazyce C. Stejně je možno definovat a deklarovat funkce. Následuje část zpracovávající signál *booted* z rozhraní `Boot`. Při spuštění je v modulu MainC vygenerován signál *booted* a ten je odchycen a zpracován v našem modulu. Propojení rozhraní je nutno zajistit v konfiguraci naší aplikace. Následně je změněna hodnota proměnné `i` na 1.

V inicializaci agentní platformy budeme muset připravit vše pro to, abychom následně mohli přijmout kód agenta a ten začít interpretovat.

6.1.2 Výstup na LED

V určitých případech můžeme potřebovat signalizovat nějaký stav agenta. Pro tento účel nám mote MICAz nabízí signalizaci pomocí tří LED diod. Každá z diod svítí jinou barvou, máme k dispozici červenou, zelenou a oranžovou. O tyto diody se stará systémový modul `LedsC`, který je propojen s naší aplikací rozhraním `Leds`. Jednoduchý příklad použití:

```
module PrikladLedsC {
  uses {
    interface Boot;
    interface Leds;
  }
}
implementation {
  event void Boot.booted() {
    call Leds.led1On();
  }
}
```

V příkladu je využita další možnost jak zapsat, která rozhraní modul využívá – za klíčovým slovem `uses` můžeme uvést blok obsahující použitá rozhraní. Stejnou konstrukci lze použít i pro rozhraní, která naše komponenta poskytuje. Vzorový modul po spuštění rozvítí LED diodu číslo 1. Opět je nutno zajistit správné zapojení stran rozhraní v konfiguraci. V dalších příkladech již tuto skutečnost nebudeme zmiňovat.

6.1.3 Použití časovače

Jak vytvořit instanci modulu spravující časovač bylo ukázáno v kapitole 5.3. Je propojen s naší aplikací pomocí rozhraní `Timer`, které umožnuje zadat požadovanou přesnost

časovače. Příklad modulu využívající časovač:

```
#include "Timer.h"
module BlinkC
{
  uses interface Timer<TMilli> as Timer0;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 100 );
  }

  event void Timer0.fired()
  {
    call Leds.led0Toggle();
  }
}
```

V příkladu je využit časovač, kde požadujeme přesnost na milisekundy. Možné hodnoty, které je možno dosadit jsou definovány v hlavičkovém souboru "`Timer.h`". Konkrétně se jedná o hodnoty `TMilli`, `T32khz` a `TMicro`. Soubor je nutno nejprve vložit pomocí `#define`. Vzorový modul při startu spustí časovač, který bude periodicky vysílat signál *fired* každých 100 ms. Rozhraní umožňuje časovač zastavit, spustit pouze jednou či periodicky a zjišťovat aktuální informace, jako je například doba do vyslání dalšího signálu *fired*.

Časovač lze využít pro potřeby uspání interpretu na určitou dobu či pravidelnému sběru dat ze senzorů.

6.1.4 Získaní dat ze senzorů

Hned v úvodu lze poznamenat, že platforma MICAz neobsahuje žádný vestavěný zdroj dat - senzor. K mote lze ovšem připojit specifickou senzorovou desku pomocí 51 pinového rozhraní. Pokud nemáme senzorovou desku k dispozici, lze pro simulační účely využít modulu `ConstantSensorC` či `SineSensorC`. Všechny systémové komponenty starající se o senzory poskytují rozhraní `Read`. Toto rozhraní zároveň definuje jakého typu jsou data. Například senzor poskytující data jako 16-ti bitové číslo bez znaménka bude propojen pomocí rozhraní `Read<uint16_t>`.

Je třeba přesně vědět jakou senzorovou desku budeme mít k dispozici a jaké senzory budeme moci využít. Příkladem budiž senzorová deska `Mts300`, obsahující kromě jiného modul `TempC`. Tento modul poskytuje rozhraní `Read<uint16_t>`, pomocí kterého lze zjistit teplotu okolí. Jak již bylo zmíněno, je tato oblast závislá na konkrétní senzorové desce. Důležité ovšem je, že všechny senzory poskytují naší aplikaci rozhraní `Read`. Příklad jak využít rozhraní `Read`:

```
module PeriodicReaderC {
  provides interface StdControl;
  uses interface Timer<TMilli>;
```

```
  uses interface Read<uint16_t>;
}
implementation {
  uint16_t lastVal = 0;
  command error_t StdControl.start() {
    return call Timer.startPeriodic(1000);
  }
  command error_t StdControl.stop() {
    return call Timer.stop();
  }
  event void Timer.fired() {
    call Read.read();
  }
  event void Read.readDone(error_t err, uint16_t val) {
    if (err == SUCCESS) {
      lastVal = val;
    }
  }
}
```

Příklad implementuje komponentu, která v pravidelném intervalu jedné sekundy snímá data ze senzoru a ty ukládá do vnitřní proměnné. Proměnná tedy obsahuje vždy poslední hodnotu čtenou ze senzoru. V příkladu je vidět, v TinyOS rozšířené, rozdělení čtení dat do dvou fází. V první požádáme o přečtení dat pomocí příkazu *read*. V druhé fázi nám systémový modul zašle data. Data získáme tedy až po nějakém čase, kdy jej má systémový modul k dispozici. Data jsou zaslána zpět pomocí signálu.

Agentní platforma pak může nabízet funkce, pomocí kterých bude možno zjišťovat hodnoty ze senzorů jak v jeden daný okamžik, tak i periodicky. Pokud budeme scanovat data periodicky, pak rozhraní musí umožňovat zvolení periody. Může být taktéž ukládána historie naměřených dat. Máme na výběr ze dvou možností kam historii uložit. První možností je paměť RAM. Druhou pak vestavěná flash paměť o velikosti 512 kB.

6.1.5 Přístup k paměti flash

Pro uložení dat máme k dispozici vestavěnou externí paměť flash přímo na platformě MICAz. Přístup k této paměti je řádově pomalejší než k paměti RAM. Nadruhou stranu nám nabízí větší úložnou kapacitu. Paměť je rozdělena na jednotky. Každá jednotka je spojitá oblast paměti specifikovaného formátu. K paměti jsou definovány dva základní přístupy:

- Logging
- Block

Logování (Logging) je využitelné, pokud potřebujeme data pouze připisovat, případně číst jako proud dat. Nová data se zapíšou vždy na konec, čímž se zvětší velikost obsazené paměti. Tento přístup neumožňuje přepisovat stávající data. Jedinou možností jak přepsat stará data, je vymazat vše a logovat od začátku.Taktéž neumožňuje efektivně číst data v náhodném pořadí. Při čtení pracuje s proudem dat. Od aktuální pozice čte data v pořadí v jakém byla logována. Umožňuje sice změnu aktuální pozice, ovšem využití pro náhodný

přístup k paměti je značně neefektivní. Na druhou stranu připsání nových dat do paměti je atomickou operací. Po zavolání funkce je garantováno zapsání dat na flash.

Blokový (Block) přístup umožnuje náhodný přístup do paměti jak pro čtení, tak i zápis. Nevýhodou je nutnost volat příkaz *sync* pro synchronizaci a jistotu zapsání dat na flash.

Vytvoření jednotky

Pro vytvoření jednotky nejprve budeme muset vytvořit soubor "`volumes-at45db.xml`". U platformy MICAz se jedná o čip `at45db`, který tvoří část názvu souboru za pomlčkou. Pokud bychom pracovali na jiné platformě než-li MICAz, poté by měl název souboru tvar "`volumes-<název_čipu>.xml`". Pro vytvoření jedné jednotky pojmenovanou jako `BLOCKTEST` s velikostí 16 kB bude soubor obsahovat:

Soubor ‘‘volumes-at45db.xml’’

```
1 <volume_table>
2   <volume name="BLOCKTEST" size="16384"/>
3 </volume_table>
```

Následující příklad obsahuje kofiguraci, která propojuje náš modul `PametC` se systémovým modulem `BlockStorageC`. Budeme tedy k paměti přistupovat v blokovém režimu. Jelikož jsme si v souboru "`volumes-at45db.xml`" definovali jednotku pojmenovanou `BLOCKTEST`, můžeme modulu `BlockStorageC` přiřadit jedinečný identifikátor `VOLUME_BLOCKTEST`. Jméno identifikátoru je tvořeno dvěmi částmi, první je pevně stanovena jako `VOLUME_` a za ní následuje název jednotky ze souboru "`volumes-at45db.xml`". Na začátku souboru také musíme vložit hlavičkový soubor "`StorageVolumes.h`". Jednoduchý příklad propojující konfigurace:

```
#include "StorageVolumes.h"
configuration PametAppC { }
implementation {
  components PametC, new BlockStorageC(VOLUME_BLOCKTEST),
    MainC;
  MainC.Boot <- PametC;
  PametC.BlockRead -> BlockStorageC.BlockRead;
  PametC.BlockWrite -> BlockStorageC.BlockWrite;
}
```

Rozhraní pro práci s flash

K práci s flash máme k dispozici několik rozhraní, podle toho k jakému účelu nám bude paměť sloužit. Rozhraní `LogRead` a `LogWrite` slouží pro práci s pamětí v logovacím režimu. Rozhraní `BlockRead` a `BlockWrite` pak slouží pro práci v blokovém režimu. Kromě těchto dvou základních existuje navíc rozhraní `ConfigStorage`, které spojuje obě rozhraní `BlockRead` a `BlockWrite` do jediného celku, přičemž obsahuje pouze nejpoužívanější funkce pro blokový přístup.

V naší aplikaci využijeme pouze blokového přístupu. Pro ukládání historie dat ze senzorů, budeme potřebovat po zaplnění celého pole začít zapisovat na začátek. Přitom ale chceme zachovat původní historii a proto je pro nás logovací režim nepoužitelný. Do pole tedy budeme zapisovat v kruhu. Blokový režim nám také nabídne větší úložnou kapacitu, kterou lze využít pro účely agenta.

6.1.6 Komunikace

V TinyOS je komunikace realizována pomocí posílání paketů. Každý paket má svou hlavičku, datovou část, záhlaví a svá metadata. Oblast komunikace je také hardwarově závislá. Dále se zaměříme pouze na platformu MICAz, používající čip `cc2420`. Základní velikost paketu je definována na 28 bytů. Tuto velikost lze změnit předefinováním `TOSH_DATA_LENGTH`. Pro naše účely velikost zůstane na původní hodnotě.

Hlavička

Každý paket je uvozen jedním bytem s hodnotou `0x00`. Hlavička paketu obsahuje především směrovací informace, jako je zdrojová adresa a cílová adresa. Adresa v síti je reprezentováná 16ti bitovým číslem. Každý mote má po spuštění přiřazenu adresu. Adresa je definována pomocí `TOS_AM_ADDRESS` a pokud není nijak změněna, pak je shodná s identifikátorem uzlu definovaným v `TOS_NODE_ID`. Je ovšem lepší do `TOS_AM_ADDRESS` nezasahovat a přímo při instalaci do mote změnit oba identifikátory tak, jak je popsáno v kapitole 4.3. Dále je definována broadcastová adresa `TOS_BCAST_ADDR`, kdy jsou zprávy odeslány na všechny uzly.

Následuje jeden byte, v kterém je uložena velikost datové části. TinyOS umožnuje sdružit skupinu uzlů dohromady. Tato skupina je označena pomocí jednoho bytu. Uzly pak komunikují pouze s těmi, které leží ve stejné skupině. Pro naše účely si vystačíme s jednou skupinou ve které budou všechny uzly dohromady. Posledním bytem hlavičky je identifikátor typu zprávy. Identifikátor typu lze vhodně využít pro odlišení typu komunikace mezi agenty. Příkladem je odlišení zaslání zprávy od zaslání kódu agenta. Pro představu jak může přijatý paket vypadat je vložena následující tabulka:

start	cíl. addr	zdroj. addr	délka dat	skupina	typ	var1 [16bit]	var2 [16bit]
00	00 02	00 01	04	22	06	8F 43	66 A0

Tabulka 6.1: **Příklad hlavičky s daty**

Datová část

TinyOS umožňuje abstrahovat přenášená data a pracovat s datovou částí jakožto se strukturou. Strukturu sami definujeme, definujeme jaké bude obsahovat proměnné a jakého typu budou. Velikost struktury nesmí být větší než je definováno v `TOSH_DATA_LENGTH`. Při překladu jsou jednotlivé složky rozmístěny, obvykle za sebou v pořadí, v jakém byli definovány. Při odesílání jsou data přetypována na pole bytů. Pokud chceme s daty opět pracovat, musíme je po přijmutí přetypovat zpět. Nic nám nebrání definovat strukturu jako pole bytů a s ním pak pracovat. Máme tedy k dispozici 28 bytů pro data na jeden paket[1].

Pro naše účely budeme potřebovat zasílání zpráv delších než 28 bytů. Větší zprávu pak rozdělíme na jednotlivé pakety. Budeme potřebovat implementovat řízení. Příjemce nesmí být zahlcen daty a taktéž je nutné reagovat na výpadky paketů. V našem případě bude příjemce každý paket potvrzovat. Čip `cc2420` sice poskytuje možnost potvrzovat pakety hardwarově ovšem v našem případě pro možnost rozšíření implementuje potvrzování vlastním způsobem. V potvrzovací zprávě pak můžeme posílat vlastní data využitelná pro

[1] Pokud nezměníme definici `TOSH_DATA_LENGTH`

řízení. Nároky na komunikaci budou ekvivalentní, hardwarové potvrzování taktéž zasílá normální zprávu zpět, ale do datové části bychom nemohli nic vložit.

Záhlaví

Záhlaví není u platformy MICAz využito. Je definováno pro ostatní platformy, které mohou tuto část využít k ukončení paketu.

Metadata

Tato část není zasílána v paketu. Obsahuje pouze informace o konkrétním paketu jako je síla přijatého signálu, vysílací výkon pro odeslání, zdali byl paket potvrzen druhou stranou, kolik bude pokusů pro odeslání paketu a další. Metadata jsou svázána s konkrétní platformou. Ta definuje, co je a co není v metadatech obsaženo.

UART

Komunikace pomocí sériového rozhraní je naprosto shodná s bezdrátovou komunikací. Jediné na co si je potřeba dát pozor je hardwarové zpracování adres u čipu `cc2420`. Proto při kopírování paketu přijatého ze sítě na sériové rozhraní musíme ručně zjistit zdrojovou a cílovou adresu paketu (z rozhraní sítě) a tyto hodnoty vložit do paketu. Nestačí tedy paket vzít a pouze zkopírovat.

Rozhraní pro komunikaci

Nejprve je nutno využít rozhraní `SplitControl` poskytované modulem `ActiveMessageC`. Toto rozhraní je potřebné pro inicializaci komunikace. Využijeme příkazu *start*. Po určité době je nám zaslán zpět signál *startDone* s parametrem, zda se komunikační modul podařilo nastartovat. V opačném případě voláme příkaz *start* znovu.

Pro práci a nastavování atributů paketu budeme využívat rozhraní `Packet` a `AMPacket`. Ta umožňují zjistit a měnit hodnoty jako například zdojová a cílová adresa, typ zprávy a další.

Rozhraní `Recieve` slouží pro příjem paketů, které nám poskytuje systémový modul `AMReceiverC`. Jako parametr při jeho vytváření uvedeme typ zprávy na kterou chceme reagovat. Můžeme tak vytvořit několik rozhraní, kdy každé bude zpracovávat jiný typ zprávy.

Pro odesílání paketů máme možnost využít rozhraní `Send` a `AMSend`. Hlavní rozdíl mezi rozhraními je ten, že `AMSend` narozdíl od `Send` umožňuje zadat cílovou adresu. Proto budeme používat především toto rozhraní. Pro odeslání paketu zavoláme příkaz *send*. O úspěšnosti odeslání jsme informováni prostřednictvím signálu *sendDone*. Komunikaci zajišťuje modul `AMSenderC`, který při vytváření taktéž přebírá typ zprávy, kterou bude posílat. Vzorový příklad nalezneme v příloze **B**, která se nachází na straně 55.

6.2 Agentní platforma

V předchozí části byly popsány vybrané komponenty. Agentní platforma do značné míry většinu těchto funkcí poskytne interpretu, pouze v abstraktnější podobě. Uvedeme seznam funkcí, které by měla agentní platforma nabízet. Jedná se především o:

- Zasílání zpráv - zde je nutno rozšířit délku zprávy na více než 28 bytů. Budeme dělit zprávu na pakety kdy první paket bude obsahovat hlavičku a následovat budou datové pakety. Každý paket bude potvrzován příjemcem. Hlavička taktéž bude obsahovat kontrolní součet. Pro správnou činnost je třeba stanovit řízení[2].
- Zaslání kódu agenta - upravená verze zasílání zpráv, zasíláme pouze typově jiná data.
- Ukládání zpráv do báze znalostí, taktéž uložení kódu agenta.
- Správa systémových prostředků, přidělovaní paměti.
- Množina algoritmů - práce se seznamy (operátory `cad` a `cdr` z jazyka LISP) a další.
- Rozhraní pro vnímání - snímání dat ze senzorů.
- Další služby - například vytvoření seznamu sousedních uzlů.

Další velmi důležitou částí bude synchronizace činnosti interpretu. Agentní platforma bude řídit běh interpretu. Interpret pracuje jako stavový automat. Interpretu bude v každou chvíli poskytnuta možnost udělat jeden krok, přičemž si sám může zažádat o provedení dalšího kroku. Agentní platforma pak v době své nečinnosti poskytne interpretu možnost provést další krok. Při každém volání je možné předat sadu atributů a dat o které bylo zažádáno v předchozím kroku, či vznikla v průběhu nečinosti interpretu. Interpret taktéž bude mít možnost uspat svoji činnost a další krok naplánovat při výskytu nějaké události[3].

[2] Například co dělat při výpadku paketu
[3] Například příjem zprávy

Kapitola 7

Jazyk ALLL pro popis agenta

V této kapitole bude popsán jazyk pro popis agenta nazvaný ALLL (Agent Low Level Language)[10]. V našem případě byly některé části mírně modifikovány, ale nejednalo se o žádné výrazné změny. Obsah této kapitoly spadá spíše do části interpretu a v této práci budou popsány pouze výrazné rysy tohoto jazyka, které jsou důležité pro implementaci platformy. Pro detailnější popis jazyka ALLL proto odkáži na bakalářskou práci Pavla Spáčila[8].

7.1 Rozdělení agenta na části

Agent byl v našem případě rozdělen na 7 částí, přičemž zachovává rysy BDI agentů. Obsahuje jak bázi znalostí o okolí, tak bázi plánů a aktuálně zvolený cíl, čili záměr. Báze znalostí je v našem případě rozdělena do dvou kategorií. První je báze znalostí, tak jak ji chápeme u BDI agentů. Druhou pak část obsahující naměřené hodnoty ze senzorů a příchozí zprávy. Zbylé tři části tvoří registry, které mohou sloužit pro různé výpočty. Části jsou pojmenovány následovně:

- `BeliefBase` - báze znalostí.
- `PlanBase` - báze plánů.
- `Plan` - záměr.
- `InputBase` - obsahuje přijaté zprávy a naměřené hodnoty ze senzorů[1].
- Registr číslo 1 - univerzální registr.
- Registr číslo 2 - univerzální registr.
- Registr číslo 3 - univerzální registr.

7.2 Struktura jazyka ALLL

Agent pracuje s jednotlivými plány, které jsou složeny z akcí. Plány mohou být hiearchicky zanořeny a v případě, že se jeden podplán nepodaří úspěšně vykonat, podplán je smazán a pokračujeme v provádění plánu na vyšší úrovni. Akcí tedy může být i spuštění podplánu.

[1] Spadá do kategorie znalostí o okolí

Jiným typem akce je například komunikace přes rádio, operace pro práci s bází znalostí, či volání služeb platformy.

Jazyk ALLL pracuje převážně se dvěmi základními strukturami a to tabulkou a seznamem. Tabulka je tvořena seznamy na jejichž pořadí nezáleží. Seznamem chápeme uspořádanou posloupnost akcí. Každý seznam začíná otevírací závorkou[2] a končí uzavírací závorkou [3]. Akce nejsou nijak oddělovány, protože každý typ akce je uvozen svým speciálním symbolem. Interpret má díky tomu možnost identifikovat začátek další akce.

Dalším datovým typem je n-tice hodnot. Struktura spadá do kategorie seznamu. Jediným rozdílem je, že jednotlivé položky jsou odděleny znakem „,“. N-tice jsou využívány především jako parametry platformních akcí a pro uchovávání informací v bázi znalostí.

V jazyce ALLL je možné pro účely vyhledávání svých znalostí definovat anonymní proměnné pomocí symbolu „_“ a ty následně unifikovat. Výsledkem unifikace je buď neúspěch, či substituce anonymní proměnné za konkrétní hodnotu, čímž se obě n-tice stanou identickými.

7.3 Využití pomocných registrů

V každém okamžiku je nastaven pouze jeden z registrů jako aktivní. Do tohoto registru je vložena hodnota výsledku po vykonání akce. Aktivní registr nastavíme pomocí `&(1)`, `&(2)`, respektive `&(3)`.

V každé akci můžeme namísto konkrétní hodnoty využít zástupného symbolu registru. Jedná se o `&1`, `&2` a `&3`. Při provádění akce je hodnota zvoleného registru substituována do tohoto místa.

Příklad 7.1: Registr číslo 1 obsahuje 23. Příklad substituce registru do n-tice.

$$(98, 11, \&1) \Rightarrow (98, 11, 23)$$

Při vkládání do báze znalostí se substituce neprovádí. N-tice je uložena přesně tak, jak je zapsána. Provedení substituce nastane až při výběru této n-tice z báze znalostí.

7.4 Sémantika akcí

V tabulce 7.4 máme uvedený přehled typů akcí, jejich parametry a význam. Nyní se zaměříme na specifické vlastnosti konkrétních akcí.

Jednou z podstatných vlastností je kontrola duplicity při vkládání do báze znalostí. Agent tedy udržuje ve své bázi znalostí pouze unikátní údaje. Při odebírání položek z báze znalostí jsou odstraněny všechny položky vyhovující unifikaci. Pokud báze znalostí obsahuje n-tice `(1)`, `(2)` a `(3,4)`, pak po provedení akce `-(_)` zůstane v bázi znalostí pouze `(3,4)`.

Důležitým rysem jazyka ALLL je, že je oddělena část `InputBase` a `BeliefBase`. Proto i test na jejich obsah zajišťují odlišné akce. Část `InputBase` obsahuje vždy n-tice se dvěmi položkami. Pokud je první položkou číslo, jedná se o přijatou zprávu. Nachází-li se místo tohoto čísla znak „s“, „m“, „M“ či „a“, jedná se o data přijatá ze senzorů. Při testování na obsah `InputBase` zadáváme pouze první položku n-tice. Nachází-li se v `InputBase` taková n-tice, bude do aktivního registru vložena pouze druhá položka n-tice a samotná n-tice

[2] Znakem „(“
[3] Znakem „)“

bude z `InputBase` vymazána. Příkladem budiž akce `?(1)`, která nám vybere první n-tici obsahující zprávu od uzlu číslo 1 a zprávu uloží do aktivního registru.

Oproti tomu test na bázi znalostí plně používá unifikaci a první položku, která je unifikovatelná přesouvá do aktivního registru.

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

Tabulka 7.1: **Seznam typů akcí a jejich význam**[8]

Plány mohou být jak pojmenované, tak nepojmenované. Pokud spouštíme uvnitř plánu zanořený podplán, je nutné do kódu vložit zarážku, abychom byli schopni identifikovat konec tohoto podplánu.

Příklad 7.2: Máme aktuální plán: `@(+(abc)!(2,(456)))+(123)`. Výsledkem spuštění nepojmenovaného plánu je:

```
+(abc)!(2,(456))#+(123)
```

Důležitou akcí je z našeho pohledu volání služeb platformy. V následující kapitole budou nabízené služby popsány, včetně parametrů, které konkrétní služby vyžadují.

Jak si bylo možné povšimnout, jazyk ALLL zavádí pro identifikaci akce jednopísmenné identifikátory. Celý jazyk se tak snaží být nenáročný na délku kódu, což je pro oblast senzorových sítí, které trpí nedostatkem paměti, důležité. Taktéž identifikace služby platformy je zajištěna pomocí jednoho znaku. Díky tomu je možné zapsat i poměrně složité chování agenta do vyhrazených 2 kB paměti RAM.

Kapitola 8

Hlavní řídící smyčka

V této kapitole bude popsán stavový diagram. Bude nastíněno rozdělení aplikace na část platformy a interpretu a bude ukázáno jakým způsobem platforma volá interpret. Také se budeme zabývat synchronizací asynchronních služeb.

8.1 Stavový diagram a popis jeho částí

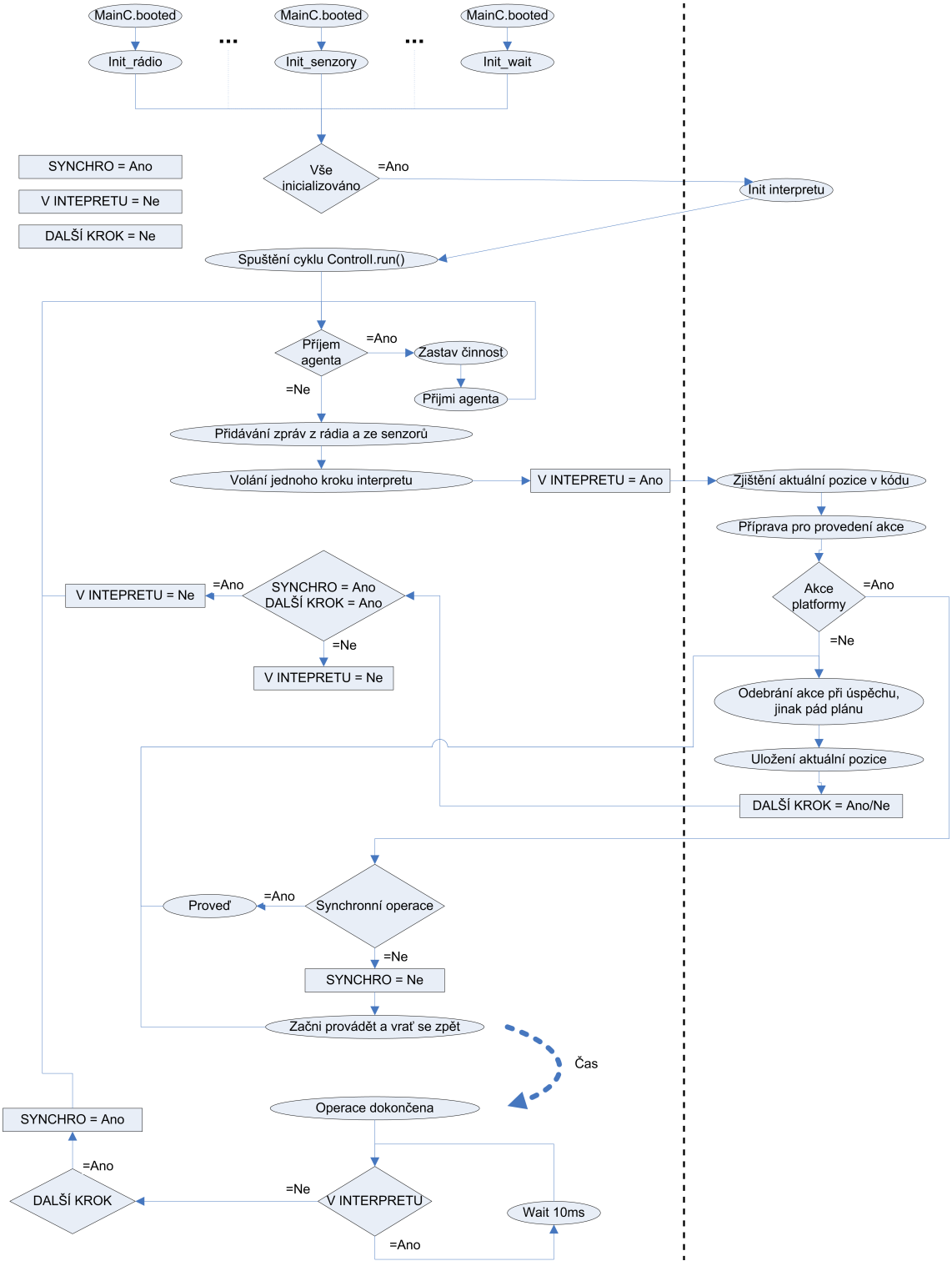
Na obrázku 8.1 můžeme vidět základní schéma řídící smyčky. Část platformy je oddělena od interpretu svislou čárkovanou čarou. Používáme tří pomocných proměnných. Proměnná `V INTERPRETU` slouží jako indikace, zdali se nacházíme v části interpretu, či platformy. Proměnná `SYNCHRO` je naplněna v případě volání služby platformy. Obsahuje hodnotu `TRUE` v případě, že se jedná o synchronní operaci a explicitní synchronizace není třeba. V opačném případě je naplněna hodnotou `FALSE` a slouží jako indikace, že byla volána asynchronní operace vyžadující dodatečné řízení. Poslední proměnná je nazvána `DALŠÍ KROK`. Je vždy naplněna na konci každého kroku interpretu. Její hodnota říká, zda si přejeme provést další krok interpretu. Interpret tedy může platformě říci, že již ukončil svoji činnost a v dalším kroku si již nepřeje být volán.

8.2 Inicializace

V horní části obrázku se nachází inicializační blok. Nejprve inicializujeme část platformy. V jednotlivých částech platformy vyžadujících inicializaci se nachází obslužná funkce zpracovávající signál *booted* z modulu `MainC`. Jakmile máme inicializovány všechny moduly platformy, můžeme zavolat inicializační část interpretu, na jejímž konci je platforma informována, že může začít v cyklu volat jeden krok interpretu. K tomuto účelu bylo definováno rozhraní `ControlI`. Obsahuje dva signály, *booted* a *nextRun*, a jeden příkaz *run*. Platforma tedy poskytuje rozhraní `ControlI` a interpret je zapojen v konfiguraci jako strana toto rozhraní používající. Vysláním signálu *booted* se začne vykonávat inicializační část interpretu. Na konci této části zavoláme příkaz *run*. Následně platforma začíná v cyklu vysílat signál *nextRun*. Ten říká interpretu, že může provést jeden krok.

8.3 Cyklické volání jednoho kroku interpretu

Platforma tedy začne vykonávat hlavní smyčku. Na začátku vždy kontroluje, zda nepožadujeme nahrání nového agentního kódu. Pokud zjistíme, že máme za úkol přijmout nového



Obrázek 8.1: **Stavový diagram**
zobrazuje způsob jakým zajišťujeme synchronizaci.

agenta, pak zastavíme činnost starého agentního kódu a přijmeme nový. V opačném případě pokračujeme ve vykonávání současného agentního kódu.

Dalším krokem, který musíme provést před předáním řízení interpretu, je přidávání zpráv ze senzorů a rádia. Důvodem je použití proměnných z části interpretu, které se v průběhu kroku interpretu mění. Jedná se především o ukazatele na jednotlivé bloky agentního kódu.

Nyní může být řízení předáno interpretu. Pomocí rozhraní `ControlI` vyšleme signál *nextRun* a nyní má interpret možnost provést jeden krok. Zároveň také nastavíme proměnnou `V INTERPRETU` na logickou hodnotu `TRUE`.

Interpret začne svou činnost tím, že si zjistí na jaké pozici v kódu se nachází a připraví vše nutné pro provedení jedné jednoduché akce. Podle typu akce je daná akce provedena buď v části interpretu, nebo je vyžadována výpomoc platformy. Jednoduchou akcí, kterou je možnost provést uvnitř interpretu, je práce se seznamy a různé matematické operace. Operací vyžadující kooperaci s platformou je například uspání interpretu po nějaký čas, či čekání na příchozí zprávu.

Nejprve popíšeme chování po provedení jednoduché akce nevyžadujících spolupráce platformy. Jedná se vždy o synchronní operace. Zvolená akce může být dokončena jak úspěšně, tak i neúspěšně. Pokud se nepodařilo akci vykonat, pak aktuálně zvolený plán končí taktéž neúspěchem. Interpret se pak musí postarat o obsluhu neúspěšně provedeného plánu a následnou reakci na nastalou situaci. V případě úspěšného dokončení akce je tato akce odebrána a připravíme vše potřebné na to, abychom mohli v dalším kroku zavolat interpret znovu. Součástí je i uložení aktuální pozice.

Poslední operací kterou interpret vykoná je nastavení proměnné `DALŠÍ KROK` a navrácení řízení zpět platformě. Logická hodnota `FALSE` značí, že interpret si přeje ukončit svou činnost a proto nastavíme proměnnou `V INTERPRETU` na `FALSE`. V opačném případě budeme muset volat interpret znovu. Opět nastavíme proměnnou signalizující ukončení části interpretu (`V INTERPRETU=FALSE`) a začneme provádět smyčku od začátku, tedy dotazem zda přijímáme agentní kód. V implementaci systému se jako proměnná `DALŠÍ KROK` označuje návratová hodnota z funkce *nextRun*. Nejedná se tedy o samostatnou proměnnou, ale pro jednoduchost byla do diagramu vyznačena tímto způsobem.

8.4 Volání platformní akce

Nyní se zaměříme na akce vyžadující spolupráce platformy. Tyto akce lze dělit do dvou kategorií a to synchronní a asynchronní. Hned z úvodu je potřeba říci, že většina služeb platformy jsou asynchronní služby. Jednoduchým příkladem může být již zmíněné čekání na příchozí zprávu. Interpret v první fázi informuje platformu, že volá službu platformy. Ta zjistí o jakou službu se jedná a v případě, že je asynchronní, nastaví proměnnou `SYNCHRO` na logickou hodnotu `FALSE`. Důležité je, že v tomto okamžiku vrátí řízení zpět interpretu a pokračuje jako by se jednalo o úspěšně dokončenou akci. Ten provede odebrání akce ze zásobníku a uloží si svou pozici v kódu. Následuje informování platformy o ukončení kroku spolu s hodnotou, zda požadujeme krok další. V tuto chvíli následuje porovnání, zda máme pokračovat ve smyčce. Zde je dobré si povšimnout složené podmínky, jejíž součástí je test proměnné `SYNCHRO`.

Jelikož se nejednalo o synchronní operaci, cyklus v tuto chvíli končí a je nastavena proměnná `V INTERPRETU` na hodnotu `FALSE`. Nyní je hlavní smyčka pozastavena a čekáme na dokončení dané operace. Po určitém čase, kdy dokončíme operaci, otestujeme proměnnou `V INTERPRETU`, která nám jinými slovy říká, zda-li již byla hlavní smyčka ukončena. Pokud

se do nynější doby smyčka neukončila, čekáme 10ms a znovu v cyklu provádíme stejný test. Je potřeba říci, že v praxi se toto čekání nevyskytuje. Důvodem je velmi rychlé ukončení hlavní smyčky (cca jednotky instrukcí). Oproti tomu dokončení akce trvá řádově mnohem delší dobu. Tato čekací smyčka je tedy navržena spíše z formálních a simulačních účelů.

Nyní se nacházíme ve stavu, kdy již je hlavní smyčka ukončena a zároveň je zvolená operace dokončena. Nezbývá nám tedy nic jiného, než znovu pokračovat v hlavní smyčce za předpokladu žádosti o další krok. Jelikož můžeme s jistotou říci, že byla hlavní smyčka ukončena, pak také víme, že byla nastavena proměnná `DALŠÍ KROK`. Nyní již v závislosti na této proměnné provádění ukončíme natrvalo, či pokračujeme dalším krokem.

Posledním typem akce, kterou lze volat je synchronní operace platformy. Její zpracování je stejné jako zpracování akce náležící interpretu. Jediným rozdílem je umístění dané operace. Příkladem je služba, vracející průměrnou hodnotu z dat ze senzorů. V případě, že pole těchto dat máme nacachováno v paměti RAM, můžeme operaci provést synchronně. Data v tomto případě náleží části platformy a proto se také jedná o její operaci.

8.5 Atomicita přiřazení do proměnné

Hojně využívanou vlastností v celém programu je atomicita operace přiřazující hodnotu do proměnné typu pravda/nepravda[1]. Jelikož se jedná o jednoduchou operaci, která v asembleru pro Micaz zabírá jednu instrukci, je tato operace vždy dokončena atomicky. Pokud bychom pracovali na procesoru, kde dané přiřazení není atomickou operací, museli bychom atomicitu operace vyjádřit explicitně pomocí bloku `atomic`.

Tuto skutečnost zde zmiňujeme z důvodu provázanosti jednotlivých modulů platformy. V jednotlivých modulech můžeme nalézt spoustu podobných přiřazení, které fungují pouze díky tomu, že dané přiřazení je atomické.

[1] boolean

Kapitola 9

Služby poskytované platformou

V této kapitole budou popsány jednotlivé služby, které platforma nabízí. Bude nastíněno, jakým způsobem služby pracují a pod jakou zkratkou je lze v jazyce ALLL spustit.

9.1 Výstup na LED

Každý mote obsahuje 3 LED diody (červená, zelená a žlutá). Agentní platforma tedy nabízí služby pro jejich ovládání. Rozhraní `LedkyI` poskytuje interpretu základní operace jako například rozsvícení červene LED diody, její zhasnutí a dále také operaci pro překlopení stavu. Pokud tedy byla dioda v rozsvíceném stavu, bude po zavolání příkazu *redToggle* zhasnuta.

Rozhraní také poskytuje pro komlexní manipulaci příkaz *controlLed*. Ten požaduje zadání dvou parametrů. Prvním je identifikace LED diody a druhým operace, kterou požadujeme se zvolenou diodou provést. Oba argumenty jsou výčtového typu a hodnoty kterých mohou nabývat najdeme v hlavičkovém souboru "`Ledky.h`". Jako příklad uvedeme rozsvícení červené LED diody:

```
call LedkyI.contolLed(LED_RED, LED_ON);
```

Příkaz *controlLed* je pouze jakousi zaobalovací funkcí a celé rozhraní `LedkyI` víceméně kopíruje systémové rozhraní `Leds`. Toto rozhraní je jako jedno z mála synchronní a proto i služby nabízené platformou nevyžadují explicitní synchronizaci.

Služby mají v jazyce ALLL jeden společný identifikátor „l". Následuje n-tice parametrů, kdy první určuje LED s kterou chceme pracovat a druhým volitelným parametrem je hodnota „1" pro rosvícení a „0" pro zhasnutí. Pokud druhý parametr neuvedeme, platforma pouze přepne stav dané LED.

Příklad 9.1: Rozsvícení červené LED

```
$(l,(r,1))
```

Příklad 9.2: Zhasnutí zelené LED

```
$(l,(g,0))
```

Příklad 9.3: Přepnutí stavu žluté LED

```
$(l,(y))
```

9.2 Služby pro řízení uspání

Platforma nabízí také možnost uspat interpret po určitý časový okamžik. Služby tohoto charakteru lze volat z interpretu pomocí rozhraní `WaitI`.

9.2.1 Uspání na požadovanou dobu

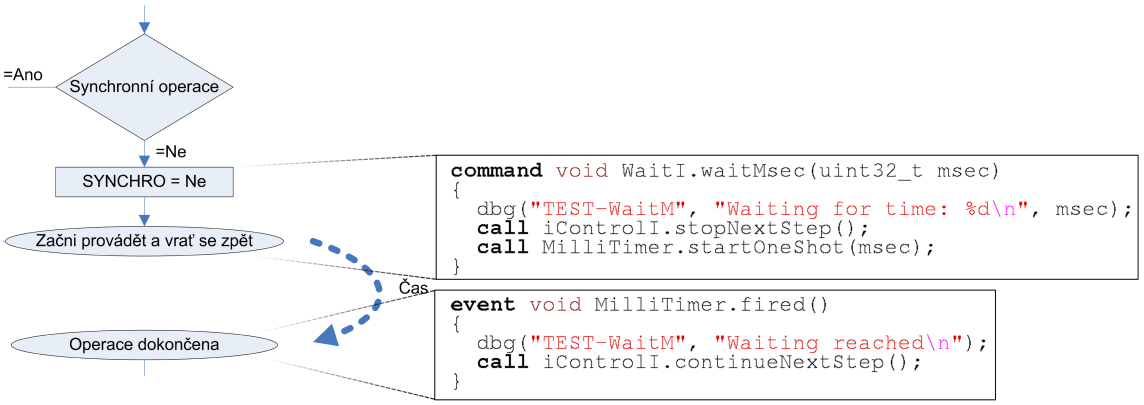
První službou, kterou si nyní popíšeme bude jednoduché uspání interpretu po zvolený časový úsek v milisekundách. Interpret je možno uspat zavoláním příkazu *waitMsec*. Služba požaduje zadání čísla, kterým volíme dobu v milisekundách. Tento argument je 32-bitové čislo bez znaménka. Můžeme vypočítat maximální možnou dobu, po kterou lze interpret uspat jako:

$$t_{max} = 2^{32}ms = 4294967296ms \approx 49.71dní$$

Pokud požadujeme zadání delšího časového úseku, budeme muset volat tuto službu vícekrát. Pro většinu běžných aplikací je ale tato hodnota více než dostatečná. Tato služba patří do kategorie asynchronních operací a proto před začátkem provádění musíme o této skutečnosti informovat hlavní smyčku. Pro dodání těchto informací hlavní řídicí smyčce z obrázku 8.1 slouží vnitřní rozhraní platformy `iControlI`. Využijeme příkazu *stopNextStep*, který nastaví hodnotu proměnné `SYNCHRO` na `FALSE`.

Následně nastavíme časovač tak, aby nám za požadovaný časový interval zaslal signál *fired* značící vypršení časovače. V tuto chvíli jsme úspěšně zahájili vykonávání platformní akce. Můžeme tedy informovat interpret, že zvolená akce byla zahájena. Interpret jak je vidět z obrázku 8.1 odebere akci a uloží svou pozici v kódu. Vrátí návratovou hodnotu, zda si přeje byt znovu zavolán a nyní, poněvadž byla nastavena proměnná `SYNCHRO` na `FALSE`, je ukončena hlavní smyčka.

V tuto chvíli je interpret uspán a platforma čeká na zaslání signálu *fired* od nastaveného časovače. Jakmile časovač vyprší, můžeme znovu začít volat interpret z hlavní smyčky. Modul starající se o hlavní smyčku o této situaci informujeme zavoláním příkazu *continueNextStep* opět z rozhraní `iConrolI`.



Obrázek 9.1: **Služba *waitMsec*.**
Reálná implementace vyznačená v diagramu

Na obrázku 9.1 je názorně vidět provádění asynchronní služby. Obrázek obsahuje kód asynchronní služby *waitMsec* a jemu odpovídající výřez stavového diagramu.

Služba má v jazyce ALLL identifikátor „w“ a vyžaduje zadání jediného parametru, určujícího po jakou dobu má být interpret uspán.

Příklad 9.4: Rozsvícení zelené LED, uspání interpretu na 1 sekundu a následné zhasnutí diody

```
$(l,(g,1))$(w,(1000))$(l,(g,0))
```

9.2.2 Čekání na příchozí zprávu

V některých aplikacích budeme požadovat, aby měl agent možnost uspat interpret do té doby než je mu zaslána nějaká zpráva. Interpretu je tato akce zpřístupněna jakožto příkaz *waitOnMessage*. Ten informuje hlavní smyčku podobně jako při uspání na zvolenou dobu o tom že se jedná o asynchronní operaci. Zavolá tedy příkaz *stopTilMessage* z rozhraní `iControlI`. Příkaz *stopTilMessage* se od příkazu *stopNextStep* liší v tom, že hlavní smyčka bude kooperovat s dalšími moduly, konkrétně s modulem obsluhujícím příchozí pakety. Tento modul a jeho rozhraní bude popsáno v samostatné kapitole. Řízení se opet navrací do části interpretu a hlavní smyčka testem proměnné `SYNCHRO=FALSE` končí svou činnost.

Konkrétní posloupnost akcí vykonaných při příjmu zprávy bude uvedena v kapitole věnující se zasíláním zpráv. Pro nás je v tuto chvíli důležité, že při příjmu zprávy je z příslušného modulu informována hlavní smyčka, že může opět začít cyklicky volat interpret. Čekání na příchozí zprávu lze v jazyce ALLL zapsat pomocí symbolu „s“.

Příklad 9.5: Rozsvícení zelené LED, čekání na zprávu a následné zhasnutí diody

```
$(l,(g,1))$(s)$(l,(g,0))
```

V souvislosti se službou umožňující čekat na příchozí zprávu byla implementována také služba *startOfMsgWaiting*. Po zavolání této služby začne platforma kontrolovat, zda nebyla přijata nějaká zpráva. V případě, že jsme obdrželi po zavoláni *startOfMsgWaiting* zprávu a naším dalším krokem bude akce *stopTilMessage*, bude samotné čekání na zprávu ignorováno.

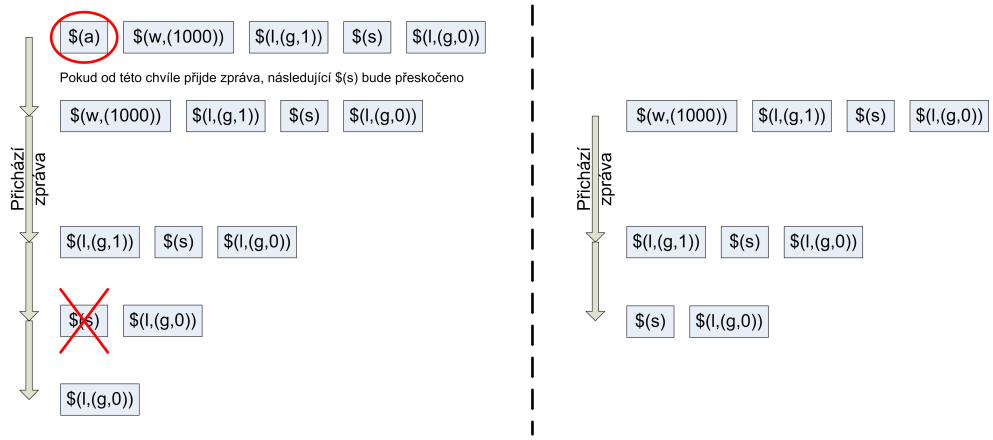
V jazyce ALLL lze službu identifikovat symbolem „a“. Obrázek 9.2 demonstruje využití této služby. Zatímco na levé straně jsme využili služby „a“, nalevo nikoli. Výsledkem je, že na levé straně je přeskočena služba „s“, čímž dojde k následnému provedení akce a zhasnutí zelené LED. Na pravé straně se interpret zastaví a čeká na příchozí zprávu. Jeho zelená LED dioda zůstává rozsvícená.

9.2.3 Ukončení činnosti agenta

Poslední službou nabízenou rozhraním `WaitI` je služba *kill*. Interpret volá tuto službu v ojedinělých připadech, kdy si přeje zcela ukončit svoji činnost. Stejného efektu je možno dosáhnout nastavením proměnné `DALŠÍ KROK` na hodnotu `FALSE`. Zavolání této služby je pouze jiným způsobem, jak ukončit provádění agentního kódu. Služba je má v jazyce ALLL identifikátor „k“.

Příklad 9.6: Rozsvícení zelené LED a ukončení činnosti agenta před rozsvícením červené diody

```
$(l,(g,1))$(k)$(l,(r,1))
```



Obrázek 9.2: **Využití služby startOfMsgWaiting.**
Na levé straně je ukázán agentní kód využívající služby startOfMsgWaiting.

9.3 Služby pro zasílání zpráv

Dalšími operacemi, které platforma nabízí je služba pro zaslání zprávy a služba pro zaslání agenta. V podkapitolách se dozvíme, jakým způsobem jsou zprávy zasílány včetně příjmu na druhé straně.

9.3.1 Zaslání zprávy

Modul starající se o zasílání zpráv se nazývá `SendM` a poskytuje rozhraní `SendI`. Nyní popíšeme službu *sendMessage*. Nejdříve se seznámíme s parametry, které tento příkaz přijímá. Jedním z parametrů je cílová adresa. Skrytým parametrem, neuvedeným v definici služby, je samotná zpráva. Ta je umístěna ve sdílené proměnné `output_message`. Jedná se o pole bytů s pevně stanovenou velikostí. Velikost tohoto pole je uvedena v hlavičkovém souboru Send.h definicí `MAX_MSG_LEN`.

Důvod proč je zpráva uložena ve sdílené proměnné je ten, že interpret vyžadoval těsně před odesláním určité části poskládat do statického pole. Jednalo se především o expanzi obsahu registrů (viz. &1). Z důvodu úspory paměti a snadné práce s proměnnou, především ze strany interpretu, byl zvolen tento způsob[1]. Posledním parametrem je předání délky zprávy, která byla umístěna do proměnné `output_message`.

Služba si po zavolání uloží parametry do pomocných proměnných. Jelikož se jedná o asynchronní službu, informujeme hlavní smyčku příkazem *stopNextStep*, aby pozastavila svůj běh. Následně pošleme první paket, obsahující hlavičku. Hlavička obsahuje délku zasílané zprávy a taktéž kontrolní součet, který je nutno před posláním hlavičky vypočítat. Obě hodnoty jsou reprezentovány jako 16-ti bitové číslo bez znaménka a pro kontrolní součet použijeme vztah:

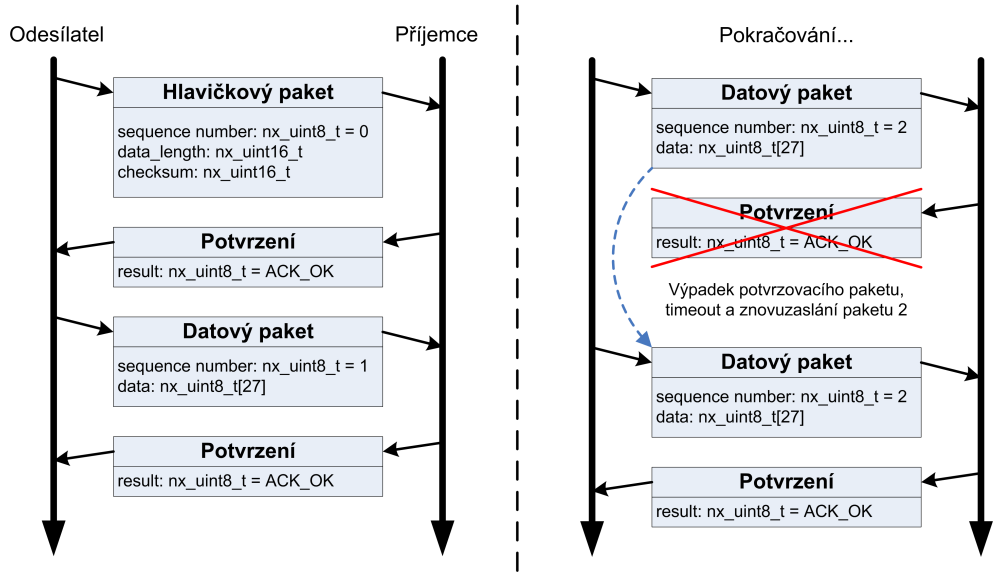
$$kontrolní\ součet = (\sum_{i=0}^{n-1} znak_i)\ mod\ 2^{16}$$

Každý paket obsahuje také své pořadové číslo. To je vždy umístěno na začátku paketu a zabírá jeden byte. Hlavičká má tedy vždy pořadové číslo 0 a každý následující datový paket

[1] Dalším možným řešením je předání ukazatele do pole.

toto číslo inkrementuje o 1.

Nastavíme časovač na jednu sekundu[2] a následně čekáme dokud nám přijímající strana nezašle potvrzení. Po přijetí potvrzení příjmu začneme postupně zasílat datové pakety obsahující rozdělenou zprávu. Pokud nám příjemce nezašle potvrzení do vypršení časovače, pošleme ten samý paket znovu. Musíme tedy vždy před odesláním jak hlavičky, tak i každého datového paketu, znovu nastavit časovač.



Obrázek 9.3: **Reakce na výpadky paketů**
Ukázka jak platforma zareaguje při nepřijetí potvrzovacího paketu.

Reakce na výpadky paketů je znázorněna na obrázku 9.3. V levé části obrázku odešleme hlavičku a první datový paket zprávy. Oba pakety jsou příjemcem potvrzeny a tento potvrzovací paket je v pořádku doručen odesílací straně, před vypršením timeoutu. Zasílání zprávy pokračuje na pravé straně obrázku. Datový paket číslo 2 je v pořádku zaslán příjemci a ten pošle potvrzení. Bohužel tento potvrzovací paket není doručen odesílací straně. Tato situace může vzniknout vlivem okolního prostředí v reálném světě. Odesílací strana tedy po vypršení timeoutu znovu zasílá datový paket číslo 2. Z obrázku je taktéž zřejmé, že nezáleží zda dojde k výpadku potvrzovacího paketu, či samotného datového paketu putujícího k příjemci. Odesílací strana reaguje na oba typy výpadků stejně.

Jak si je možné povšimnout, potvrzovací paket obsahuje proměnnou `result`. Hodnota result může nabývat jedné ze 3 hodnot:

- `ACK_OK` - Úspěšný příjem paketu
- `ACK_RESEND` - Po příjmu celé zprávy a špatném kontrolním součtu
- `ACK_HAVE_IT` - Po úspěšném příjmu celé zprávy a znovupřijetí posledního datového paketu

Přijetí potvrzovacího paketu `ACK_OK` a `ACK_HAVE_IT` značí úspěšné přijetí paketu a není třeba žádné další řízení. Po příjmu `ACK_RESEND` je třeba zaslat celou zprávu od začátku, včetně hlavičky.

[2] V další verzi aplikace by bylo dobré tuto hodnotu měnit dynamicky.

Po úspěšném odeslání celé zprávy, je hlavní řídící smyčka znovu spuštěna zavoláním příkazu *continueNextStep*. Tím je odesílající strana hotova.

Službu v jazyce ALLL voláme poněkud odlišněji než ostatní služby. Pro zasílání zpráv se namísto symbolu „$“ používá „!“. Následuje n-tice parametrů, obsahující cílový uzel a další n-tici obsahující samostatnou zprávu.

Příklad 9.7: Odeslání zprávy „ahoj“ na uzel číslo 2

```
!(2,(ahoj))
```

9.3.2 Příjem zprávy

Příjem příchozí zprávy je prováděn nezávisle na činnosti interpretu uvnitř modulu `RecieveM`. Platforma obsahuje vstupní buffer `input_message`, do kterého jsou postupně vkládány přicházející části zprávy. Jeho velikost je stejná jako `output_message`. Přijemce tedy nejprve obdrží paket obsahující hlavičku. Jednotlivé parametry[3] uloží do pomocných proměnných a pošle potvrzení příjmu odesílající straně.

Od této chvíle uzel komunikuje pouze s tímto odesílatelem a pokud přijde paket od někoho jiného, je automaticky zahozen. Nastala-li situace, že s námi chtěl komunikovat i jiný uzel, jeho hlavičkový paket bude zahozen a on, po vypršení timeoutu, znovu zasílá hlavičku. Situace se opakuje do té doby, kdy příjemce nepřijímá zprávu od jiného uzlu.

Nyní tedy máme zajištěno, že každý další datový paket bude od jednoho stejného odesílatele. Z příchozího paketu zjistíme jeho pořadové číslo a pokud je rovno 0, jedná se o hlavičku, v opačném případě o datový paket. Při opakovaném příjmu hlavičky vždy aktualizujeme obsah pomocných proměnných [3]. Z datového paketu určí jeho pořadové číslo a obsah zprávy zkopíruje na příslušnou pozici do `input_message`. Zároveň má informaci o délce zprávy a v okamžiku, kdy je přijata kompletní zpráva, porovná kontrolní součet a pokud se i ten shoduje s kontrolním součtem z hlavičky, můžeme zprávu předat interpretu.

Každý paket je potvrzován hodnotou `ACK_OK` za předpokladu úspěšného přijetí. V poslední fázi, kdy porovnáváme kontrolní součet mohou nastat dvě situace. První je již zmíněná shoda indikující, že celá zpráva byla doručena v pořádku. Zasíláme tedy taktéž hodnotu `ACK_OK`. Druhou možností je rozdílná hodnota kontrolního součtu z hlavičky a hodnoty vypočtené z přijaté zprávy. V tomto případě budeme žádat odesílatele, aby zprávu poslal znovu a to pomocí hodnoty `ACK_RESEND`.

Poslední hodnotou, kterou může potvrzovací paket nést je `ACK_HAVE_IT`. Může nastat situace, kdy je poslední datový paket úspěšně doručen příjemci, ale potvrzení odesílateli nedojde. Odesílatel tedy posílá poslední paket znovu. Hodnota má pro odesílatele význam podobný `ACK_OK`, neboli příznak úspěšného zaslání zprávy. Odlišná hodnota je spíše pro testovací účely, kdy sledujeme vzájemnou komunikaci uzlů.

Nyní předpokládáme, že platforma úspěšně přijala kompletní zprávu. Jako další krok budeme o této skutečnosti informovat interpret. První informaci zasíláme hlavní smyčce a jedná se o probuzení při čekání na příchozí zprávu. Modul `RecieveM` vyšle signál *iRecieveI.messageRecieved*. Ten je zpracován v hlavní smyčce a pokud opravdu čekáme na příchozí zprávu je spuštěn další cyklus. Druhým krokem je samotné předání zprávy. Budeme se tedy snažit obsah bufferu `input_message` přidat do části `InputBase`. Jak je vidět ze stavového diagramu ze strany 30, můžeme přidávat hodnoty do `InputBase` pouze před voláním jednoho kroku interpretu. Těsně před voláním nás hlavní smyčka informuje, že

[3] Délka zprávy, kontrolní součet, ...

můžeme tuto zprávu přidat. To se děje spuštěním příkazu *iRecieveI.ableToAdd*. Při reakci zjistíme, že máme zprávu, kterou je třeba přidat do `InputBase` a zavoláme příkaz *TableI.addInputBase*. Jako parametry uvedeme samotnou zprávu, její délku a adresu uzlu, ze kterého byla zpráva zaslána. Tím je zpráva úspěšně vložena do `InputBase` ve tvaru n-tice `(<odesílatel>, <zpráva>)`.

V případě předešlého využití služby *startOfMsgWaiting* z kapitoly 9.2.2, nastavíme příznak, že jsme obdrželi zprávu. Při příštím volání služby *stopTilMessage* je tento příznak využit a samotné čekání na zprávu přeskočeno. Od tohoto okamžiku můžeme začít komunikaci s dalším účastníkem.

9.3.3 Agentní mobilita

Služba se opět nachází v modulu `SendM` pod názvem *sendAgent*. Agentní mobilita je až na drobné úpravy totožná se službou pro zaslání zprávy. Hlavním rozdílem je typ přenášených dat. Abychom rozlišili, zda se jedná o běžnou zprávu, či zprávu obsahující agentní kód, využijeme postupu popsaného v kapitole 6.1.6. Další odlišností je, že hlavička obsahuje kromě délky a kontrolního součtu indexy na jednotlivé části agenta. Starý agentní kód je možné přepsat až po dokončení kroku interpretu. Proto si tyto indexy uložíme do pomocných proměnných. Počkáme na spuštění *iRecieveI.ableToAdd*, čímž víme že řízení je určitě v části platformy a proto můžeme starý kód přepsat. Další možností je, že při příjmu hlavičky byl agent již ve stavu, kdy interpret ukončil svoji činnost. V tom případě můžeme začít přepisovat kód okamžitě. Zaměníme tedy indexy na jednotlivé části agenta a následně zasíláme potvrzení s hodnotou `ACK_OK`.

Od této chvíle nám odesílatel postupně zasílá datové pakety obsahující kousky agentního kódu. Namísto plnění bufferu `input_message` přepisujeme starý agentní kód. Rozhraní `ArrayOpI` nám poskytuje příkaz *insertAt*, kterému předáme pozici, na kterou chceme vložit nový kousek kódu, samotný kód a jeho délku. Dále nám poskytuje přikazy pro změnu indexů, které jsme využili po přijetí hlavičky. Jakmile přijmeme celý agentní kód, spočítáme jeho kontrolní součet a pokud je vše v pořádku, spustíme znovu hlavní smyčku. Případně opakujeme zaslání celé zprávy.

Agentní mobilita je v jazyce ALLL označena pomocí písmene „m“ a přebírá jeden povinný a jeden nepovinný parametr. Povinným parametrem je adresa cílového uzlu. Nepovinným parametrem udáváme, zda si přejeme po dokončení služby pokračovat v činnosti, či zadáním „s“ ukončit svoji činnost.

Příklad 9.8: Poslání našeho agentního kódu uzlu číslo 2. Oba uzly následně rozsvítí červenou LED diodu.

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

Příklad 9.9: Poslání našeho agentního kódu uzlu číslo 2. Odesílající uzel ukončí činnost a pouze uzel č. 2 rozsvítí červenou LED diodu.

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

9.4 Služby pro získávání dat ze senzorů

Odlišností našich agentů pracujících v senzorových sítích, oproti agentům jak je známe z jazyků jako je Jason, je přístup ke snímání dat z reálného prostředí pomocí senzorů. Většina prostředí s touto možností při návrhu jazyka nepočítalo. Pro tato prostředí mohly

být informace o okolí uloženy většinou pouze v bázi znalostí. Mohlo tak docházet ke kolizi s ostatními informacemi zde uloženými. Naproti tomu agent v jazyce ALLL má pro tato data vyhrazenou samostatnou část `InputBase`, kterou sdílí spolu s příchozími zprávami. Data, která přišla z okolí jsou pohromadě v oddělené části a záleží jen na samotném agentovi, zda si tyto informace vyzvedne a uloží do své báze znalostí[4].

Je nutno říci, že ostatní jazyky pro popis agenta k této problematice přistupují různě a agent může taktéž reagovat na změny okolí. Oproti tomu použití samostatné části `InputBase` umožňuje kompaktnější zakomponování do jazyka ALLL.

V následujících kapitolách budou popsány služby pro snímání dat ze senzorů. Platforma taktéž nabízí intervalové měření těchto dat, nezávisle na chodu interpretu. Díky velikosti jsou jednotlivé hodnoty uchovávány ve flash paměti. Postup bude také uveden v následujících podkapitolách.

9.4.1 Senzorová deska MTS400CA

Spolu s mikrokontrolérem je možno spojit různé druhy senzorových desek. Jak již bylo popsáno v kapitole 3.2 jednalo se o senzorovou desku MTS400CA. Bohužel ovladač pro tuto senzorovou desku v TinyOS chybí. Je sice možné na internetu najít ovladač pro tento typ desky, ovšem z licenčních důvodů nebyl do TinyOS přiložen. Ze stejných důvodů nebyl ovladač použit. Navíc, vzhledem k tomu že platforma bude dále vyvíjena na nových mote, nejspíše i s jinou senzorovou deskou, nebyl kladen až tak velký důraz na tuto oblast. V budoucnu se jedná o jednu z oblastí, kterou lze značně rozšířit schopnosti agenta.

Naštěstí s univerzálním ovladačem `basicsb` fungovalo alespoň snímání dat z teplotního senzoru. Díky tomu mohla být ověřena funkčnost snímání dat v reálných aplikacích. Služby tedy pracují pouze se senzorem snímajícím teplotu a rozšíření na ostatní senzory bude možné až při použití jiného ovladače, případně jiné podporované desky.

9.4.2 Získání aktuální teploty ze senzoru

Všechny služby pro získávání dat ze senzorů jsou uvedeny v modulu `SensorsM`. Při žádosti o aktuální hodnotu ze senzoru volá interpret službu *getTemp*. Služba je asynchronní a proto je pozastavena hlavní smyčka voláním příkazu *stopNextStep*. Pomocí systémového rozhraní `Read` zavoláme příkaz *read* a v reakci na signál *readDone* uložíme naměřenou hodnotu do pomocné proměnné. Nakonec znovu spustíme hlavní smyčku zavoláním *continueNextStep*.

Ze stavového diagramu 8.1 vidíme, že stejně jako příchozí zprávy, tak i data ze senzorů mohou být přidávána do `InputBase` pouze na začátku každého cyklu. Hlavní smyčka nás o vhodné době pro přidání dat informuje spuštěním příkazu *iSensorsI.ableToAdd*. Pokud tedy máme nějakou hodnotu pro přidání, můžeme ji uvnitř příkazu *ableToAdd* vložit do `InputBase` a to ve tvaru n-tice `(s, (<hodnota>))`.

Služba je v jazyce ALLL označena pomocí „d“. A v tomto případě nepřebírá žádný parametr.

Příklad 9.10: Zjištění aktuální hodnoty a přesunutí z `InputBase` do registru číslo 1

```
$(d)&(1)?(s)
```

[4] `BeliefBase`

9.4.3 Intervalové měření

Pro intervalové měření využijeme časovače, který je pevně nastaven na 1 sekundu, tak jak bylo uvedeno v kapitole 6.1.4. V budoucích verzích budou implementovány i služby, které intervalové měření spouští a vypínají, případně mění intarval časovače. Každou novou naměřenou hodnotu postupně ukládáme do historie. Ta je z objemových důvodů uložena ve flash paměti. Abychom do paměti nezapisovali vždy jenom jednu hodnotu, což by bylo neefektivní, zapisujeme vždy skupinu 16-ti naměřených hodnot. Implementujeme tedy zapisování do paměti se spožděným zápisem.

Tato část platformy pracuje nezávisle na činnosti interpretu a jejím účelem je pouze plnit a uchovávat historii naměřených dat.

9.4.4 Virtuální modul a rozhraní pro simulaci flash

Při testování intervalového měření se naskytl problém absence simulátoru flash paměti. Museli jsme tedy napsat vlastní virtuální modul, který nabízel stejné rozhraní, se stejnými schopnostmi, jako modul starající se o reálnou flash paměť. Námi implementovaný modul `VirtualBlockM` se nachází v podadresáři "`sim`". Pokud zkompilujeme aplikaci pro simulaci, pak se soubory z adresáře "`sim`" nahradí všechny stejně pojmenované soubory pro reálná zařízení. Toho využijeme pro předefinování hlavičkových souborů "`Storage_chip.h`" a "`StorageVolumes.h`". Jejich obsah je zcela prázdný. Obsah původních hlavičkových souborů vkládal další moduly potřebné pro obsluhu reálné flash, ovšem v simulačním režimu tyto moduly způsobovaly problémy. TinyOS při použití rozhraní pro práci s flash automaticky vkládá oba hlavičkové soubory a při původním obsahu obou souborů nešlo aplikaci zkompilovat. Předefinováním obou souborů zamezíme hiearchickému vkládání modulů blokujících překlad a aplikaci je již možno sestavit.

Samotný modul simulující flash je generický. Využívá statického pole, jehož velikost je možné změnit při vytváření komponenty. Komponenta nabízí standartní rozhraní jak pro čtení i zápis v blokovém režimu. Konkrétně se tedy jedná o rozhraní `BlockRead` a `BlockWrite`. Obě rozhraní poskytují asynchronní operace. Nejdříve je tedy požádáno o danou operaci a po dokončení nás modul informuje zasláním příslušného signálu s případnými požadovanými hodnotami. Abychom zajistili stejné chování využijeme časovače. Operace je sice úspěšně dokončena už ve fázi požadavku o provedení, ale samotný signál je zaslán až po vypršení časovače. Časovač vždy nastavujeme na dobu jedné milisekundy, ale při událostmi řízené simulaci na této hodnotě nezáleží.

Příklad 9.11: Příklad vytvoření virtuální flash o velikosti 16kB (uvnitř konfigurace).

```
components new VirtualBlockM(16384) as StorageTeplota;
components new TimerMilliC() as casovac_VirtualBlockM;
StorageTeplota.MilliTimer -> casovac_VirtualBlockM;
```

9.4.5 Služby pracující s historií dat

V některých aplikacích budeme potřebovat zjistit kromě aktuální teploty i například průměrnou teplotu za poslední hodinu. Následně se může dle průměrné hodnoty rozhodnout, zda aktuální teplota byla běžnou hodnotou, případně detekovat extrémní nárůst teploty a vyvolat poplach. Takové chování může posloužit například k detekci požárů. Platforma proto nabízí tři funkce - zjištění minimální, maximální a průměrné hodnoty. Každá pracuje s hodnotami uloženými do historie dat a přebírá parametr, jak dlouhé období bude bráno v potaz.

Služby jsou v jazyce ALLL přidruženy ke službě pro získání aktuální hodnoty senzoru. Ta bez uvedených parametrů funguje tak, jak bylo posáno v kapitole 9.4.2. Pro naše účely je možné zapsat další parametr a od této chvíle služba bude pracovat s historií dat. Parametrem je n-tice obsahující dvě hodnoty. První je znak identifikující typ operace nad historií dat a druhým počet naměřených hodnot. Pro zjištění minimální hodnoty je vyhrazen symbol „m“, pro maximální hodnotu „M“ a pro průměrnou hodnotu „a“. V případě, že nemáme k dispozici dostatek hodnot pro provedení služby, nebude služba dokončena a interpret na tuto situaci reaguje pádem aktuálního plánu. Hodnota je opět uložena do `InputBase` uvnitř příkazu *ableToAdd*. Vložená n-tice má tvar `(<typ_operace>, (hodnota))`.

Příklad 9.12: Zjištění minimální teploty za posledních 10 sekund. Vložená hodnota do InputBase může vypadat (m,(412))

```
$(d,(m,10))
```

9.5 Služby pro práci se seznamy

Následně byly implementovány dvě služby pro práci se seznamy. Jedná se o obdobu operátorů `cad` a `cdr` z jazyka LISP. Obě služby bylo možné implementovat přímo uvnitř interpretu.

Služba vracející první prvek seznamu je v jazyce ALLL označena symbolem „f“, služba vracející zbytek seznamu symbolem „r“. Jako parametr je uveden samotný seznam.

Příklad 9.13: Seznam je uložen v registru číslo 2. První prvek tohoto seznamu je vložen do registru číslo 1.

```
&(1)$(f,&2)
```

Kapitola 10

Vytvoření nového agenta

V této kapitole budou popsány způsoby, jakými lze vytvořit nového agenta pracujícího v síti.

10.1 Naprogramování statického agenta

První možností a také historicky nejstarší je statické naprogramování agenta. Tato možnost byla využívána především v ranných verzích aplikace a využívá části inicializující interpret. V době, kdy nebyla implementována agentní mobilita se jednalo o jediný možný způsob otestování aplikace v reálném prostředí.

Součástí inicializace interpretu je naplnění příslušných registrů a tabulek agentním kódem. Po tomto naplnění standartně zavoláme platformu pomocí *ControlI.run* a začneme vykonávat daný kód. Inicializace interpretu se nachází v adresáři "`Interpret`" v souboru "`AgentC.nc`".

Příklad 10.1: Využití inicializace interpretu k vytvoření statického agenta

```
event void ControlI.booted() {
  char c[200];
  dbg_clear("Boot", "Boot: Mote booted at %s\n",sim_time_string());
  call PoleCntrl.start();
  // vlozeni pocatecniho planu a dalsich casti do tabulek

  strcpy(c,"(blik,(&(1)*(led,_,_)&(2)$(f,&1)-&2&(1)$(r,&2)"
  "&(3)$(f,&1)$(l,&3)&(2)$(r,&1)&(1)$(f,&2)$(w,&1)$(l,&3)^(blik)))");
  call Tabulka.add_str(TBL_PLANBASE, c, (uint16_t)strlen(c));

  strcpy(c,"(napln,(+(led,y,800)^(blik)#^(napln)))");
  call Tabulka.add_str(TBL_PLANBASE, c, (uint16_t)strlen(c));

  strcpy(c,"$(l,(r,0))$(l,(g,0))$(l,(y,0))^(napln)");
  call Zasobnik.push_str(c, (uint16_t)strlen(c));
  call ControlI.run();
}
```

Tento způsob naprogramování agenta se dá využít pro otestování funkčnosti v simulátoru. Můžeme si napsat agentní kód a ten následně odsimulovat. Pro tento účel byl

napsán jednoduchý testovací skript v jazyce Python s názvem "`test.py`". Ten umožňuje simulovat i vzájemnou komunikaci více uzlů včetně výpadků paketů.
Aplikaci přeložíme pomocí:

```
make micaz sim
```

Následně spustíme testovací skript a výstup uložíme do souboru:

```
./test.py > out.txt
```

10.2 Agent posílající agenta

V době, kdy již byla k dispozici agentní mobilita jsme mohli agenta zaslat do sítě za pomoci jiného agenta. Platforma již byla schopna přijmout agentní kód pomocí rádiového rozhraní a taktéž agenta zkopírovat z daného uzlu na uzel jiný. Díky této skutečnosti jsme mohli naprogramovat v basestation „prázdného" agenta, který nebude obsahovat žádný kód. Takto naprogramovaný mikrokontrolér vyjmeme z basestation a umístíme jej do mote. V tuto chvíli „prázdný" agent čeká, dokud mu není zaslán agentní kód.

Následující postup bude podobný předcházející sekci 10.1. Vytvoříme statického agenta, jehož první instrukce bude zkopírování sama sebe na jiný uzel, čili poslání „prázdnému" agentu. Následně tento agent ukončí svoji činost, což zajistíme příznakem „s" při volání služby pro posílání agentního kódu. Cílový uzel obdrží agentní kód, který začíná následující instrukcí.

Příklad 10.2: Poslaní agenta na uzel 2, který rozsvítí červenou LED diodu

```
event void ControlI.booted() {
  char c[200];
  call PoleCntrl.start();
  strcpy(c,"$(m,(2,s))$(1,(r,1))");
  call Zasobnik.push_str(c, (uint16_t)strlen(c));
  call ControlI.run();
}
```

Tento způsob posílání agenta vyžaduje přeložení aplikace vždy, když chceme poslat nového agenta do sítě. Mikrokontrolér vložený v basestation má vždy identifikátor uzlu roven 1 a sám se bude chovat jako normální agent. Může tedy přijímat zprávy od okolních agentů a vzájemně s nimi komunikovat. Ovšem nemá možnost tyto informace předat do PC přes USB rozhraní. V následující sekci bude uvedena varianta umožňujicí i komunikaci s hostitelským PC.

10.3 Basestation jako komunikační most

Uvažujme agenta jehož cílem je naměřit nějaká data a ty následně poslat přes basestation až do hostitelského PC. TinyOS obsahuje hotový program pro basestation, který zajistí přeposílání paketů z rádia na UART rozhraní[1] i opačně. Tento program najdeme v adresáři "`<Cesta k TinyOS>/apps/BaseStationCC2420`".

[1] Pomocí převodníku USB/UART propojeno s PC

Narozdíl od aplikace v adresáři "<Cesta k TinyOS>/apps/BaseStation" je počítáno s hardwarovým zpracováním adres u rádiového čipu CC2420. Příchozí pakety si můžeme zobrazit v hexadecimální podobě s využitím Java aplikace `Listen` z kapitoly 4.4.1.

Jelikož ale v komunikačním protokolu používáme potvrzování zpráv, bude ze zpráv zaslaných agenty obdžena pouze hlavička. Aplikace `Listen` neví jak a ani proč potvrzovat příchozí pakety. Důsledkem toho je, že daná hlavička nebude potvrzena, čímž dojde po vypršení timeoutu k opětovnému zaslání paketu. Tento program může posloužit pro testovací účely, ale pro plnohodnotné zasílání zpráv a agentního kódu budeme muset napsat vlastní Java aplikaci.

Využijeme třídy `MessageListenner` popsané v kapitole 4.4.2. V aplikaci můžeme definovat kompletní komunikační rozhraní, tak jako by se jednalo o část agentní platformy běžící na reálném mote. V podstatě se jedná o přepis komunikačního protokolu z jazyka NesC do jazyka Java. Následně popíšeme dva typy Java aplikací, které byly implementovány za tímto účelem.

10.3.1 Konzolová aplikace

Ve skutečnosti mluvíme o dvou konzolových aplikacích. Jedna slouží pro vyslání agenta do sítě, druhá pro příjem normálních zpráv. Nejprve se zaměříme na část posílající agenta do sítě.

Agent se skládá z celkem sedmi základních částí, tak jak je popsáno v kapitole 7. Námi implementovaná aplikace umožňuje nahrání agenta ze souboru a následné vyslání na zvolený uzel. Soubor je procházen po řádcích, kdy počáteční písmeno řádku určuje o jakou část agenta se jedná. Význam počátečních písmen jednotlivých řádků:

- B - Belief, konkrétně část `BeliefBase`.
- D - Desire, konkrétně část `PlanBase`.
- I - Intention, konkrétně část `Plan`.
- i - Hodnoty `InputBase`[2].
- 1 - Registr číslo 1
- 2 - Registr číslo 2
- 3 - Registr číslo 3

Části „B“, „D“ a „i“ mají navíc vlastnost, že pokud aplikace narazí na další část, která byla již v minulosti definována, připojí novou definici za původní obsah. Naopak u částí „I“, „1“, „2“ a „3“ znamená nová definice přepsání původního obsahu.

Příklad 10.3: Soubor obsahující agenta, který nejdříve zhasne všechny LED, počká 1 sekundu a následně rozsvítí červenou diodu

```
D(rozsvitCervenou,($(l,(r,1))))
D(smazLed,($(l,(r,0))$(l,(g,0))$(l,(y,0))))
I^(smazLed)$(w,(1000))^(rozsvitCervenou)
```

[2]Spadá do kategorie znalostí o okolí

Na příkladu 3 je možné demonstrovat využití této vlastnosti. Každý plán můžeme definovat na samostatném řádku a aplikace se sama postará o složení obou částí dohromady.

Aplikace se nachází v podadresáři "`Basestation`" a je možno ji spustit pomocí příkazu:

```
java SendAgent <soubor> <moteID> -comm serial@/dev/ttyUSB1:micaz
```

Druhá aplikace, která byla implementována, slouží pro příjem zpráv. Aplikace naslouchá na zvoleném rozhraní[3] a umožňuje od okolních agentů přijmout zprávu. Tu následně zobrazí. Aplikace je umístěna taktéž v podadresáři "`Basestation`" a spouští se příkazem:

```
java TestSerial -comm serial@/dev/ttyUSB1:micaz
```

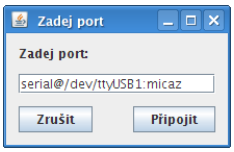
Oba programy využívají definice paketů pro normální zprávy, agentní kód a potvrzovací zprávy. Ty můžeme najít v souborech "`RadioClassMessage.java`", "`RadioClassAgent.java`" a "`RadioClassAck.java`". Tyto soubory vznikly postupem popsaným v kapitole 4.4 a následnou ruční úpravou vygenerovaných souborů.

10.3.2 Gui nadstavba

Pro zpříjemnění práce při posílání agentů do sítě byla taktéž implementována aplikace s grafickým uživatelským rozhraním. Uživatelské rozhraní bylo implementováno kolegou pracujícím na interpretu a část starající se o samotnou komunikaci byla převzata z konzolové aplikace. Aplikace umožňuje jak posílání zpráv do sítě a zpět, tak i odesílání agentního kódu na zvolený uzel. Aplikaci najdeme v podadresáři "`Basestation`" a spustíme příkazem:

```
java CommGui
```

Program nás přivítá dialogem pro zadání cesty k zařízení. Jedná se o parametr `comm`, tak jak jej známe z předchozích příkladů. Jak dialog vypadá, můžeme vidět na obrázku 10.1.

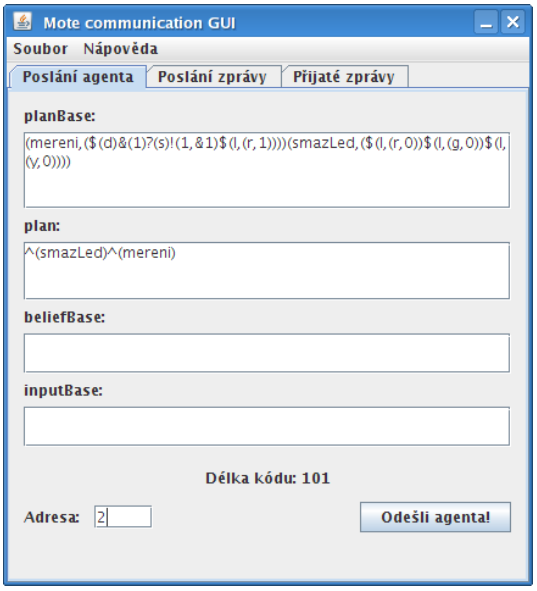


Obrázek 10.1: **Zadání cesty k zařízení.**
Změna parametru `comm`.

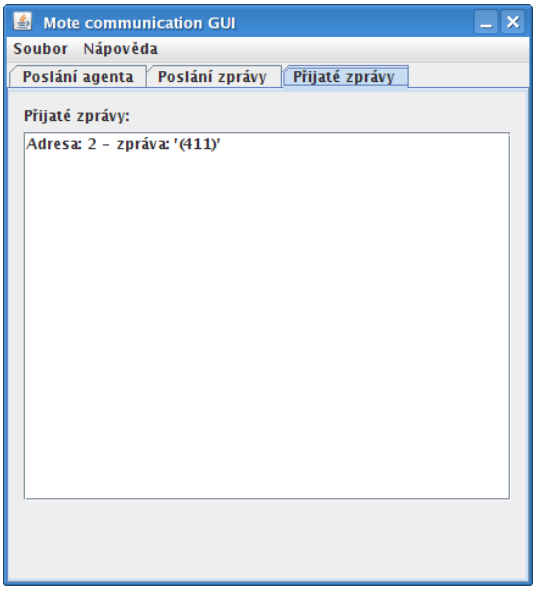
Následně se zobrazí hlavní okno celé aplikace. Na jednotlivých záložkách můžeme nalézt rozhraní pro zapsání agenta, odesílání zpráv a také zobrazit příchozí zprávy. Obrázek 10.2 obsahuje záložku pro posílání agentního kódu. Lze zapsat jednotlivé složky, adresu cílového uzlu a takto vytvořeného agenta poslat do sítě. Na obrázku 10.3 najdeme zobrazeny příchozí zprávy. Na obou obrázcích se nachází vzorový příklad agenta, který změří okolní teplotu a naměřenou hodnotu pošle zpět do PC.

Aplikace začleňuje reakci na všechny typy příchozích paketů pod jednu jedinou třídu `MessageListener`. To je důležité z hlediska přístupu na zařízení uvedené parametrem `comm`. Rozhraní neumožňuje přístup více aplikací s oddělenou definicí `MessageListener`, a proto byla situace tak jak ji známe z kapitoly 10.3.1 poněkud komplikovaná.

[3] pomocí parametru -comm



Obrázek 10.2: **Okno aplikace.**
Posílání agenta měřícího teplotu okolí. Agent nám tuto hodnotu zašle zpět.



Obrázek 10.3: **Okno aplikace.**
Přijaté zprávy. Agent nám zaslal data získaná ze senzoru.

Nejprve jsme mohli agenta zaslat do sítě pomocí aplikace `SendAgent`, ale bez možnosti zároveň přijímat zprávy. Aplikaci na příjem zpráv[4] bylo možno spustit až po úspěšném odeslání kódu. Díky implementaci reakce na výpadky paketů na straně mote bylo možné zasílanou zprávu doručit dodatečně. Agent se jednoduše snažil zprávu doručit a když na druhé straně (PC) nikdo neodpovídal, zaslal zprávu znovu.

Problém mohl nastat při komunikaci s více uzly a nutnosti odeslat nového agenta do sítě. Aplikace pro přijímání zpráv musela být ukončena, což ovšem mohlo být v době, kdy byla zahájena komunikace s některým z uzlů. Celý komunikační protokol se tak mohl dostat do nekonzistentního stavu. Grafická aplikace tímto problémem netrpí a v současné době se jedná o nejlepší způsob z výše nabízených. Nedostatkem aktuální verze je absence ukládání a načítání agentů ze souborů, což může být dobrým vylepšením budoucí verze.

[4] `TestSerial`

Kapitola 11

Příklady jednoduchých agentů

V této kapitole nalezneme několik vzorových aplikací, které ověřují funkčnost námi navržené aplikace.

11.1 Měření teploty v síti

Vzorový agent bude obsahovat pouze část `Plan`, která je zapsána v tabulce 11.1. Agent nejprve rozsvítí červenou LED a zjistí aktuální teplotu ze senzoru. Nastaví první registr jako aktivní a uloží do něj naměřenou hodnotu. Tu zašle zpět do PC přes basestation. Po odeslání zprávy zhasne červenou LED.

Část	**Obsah**
`Plan`	`$(l,(r,1))$(d)&(1)?(s)!(1,&1)$(l,(r,0))`

Tabulka 11.1: **Agent měřící teplotu okolí**

11.2 Využití plánů

Agent zapsaný v tabulce 11.2 obssahuje dva plány. Plán `stred` rozsvítí prostřední LED, počká 500 milisekund a spustí plán `okraj`. Ten naopak rozsvítí obě okrajové LED diody, také počká 500 milisekund a spouští původní plán `stred`. Tímto způsobem můžeme definovat cykly. Stačí již jen nastavit záměr agenta, kterým bude spuštění jednoho z plánů, což zapříčíní cyklické blikání LED diod. Jako inicializační zvolíme plán `stred`.

Část	**Obsah**
`PlanBase`	`(stred,($(l,(r,0))$(l,(g,1))$(l,(y,0))$(w,(500))^(okraj)))` `(okraj,($(l,(r,1))$(l,(g,0))$(l,(y,1))$(w,(500))^(stred)))`
`Plan`	`^(stred)`

Tabulka 11.2: **Agent využívající plánů k vytvoření cyklu**

11.3 Vzdálené blikání LED diod

Nyní ilustrujeme naprogramování dvou vzájemně komunikujících agentů. První z nich pouze poslouchá, jakou LED diodu má rozsvítit. Tohoto agenta umístíme na uzel číslo 3. Druhý agent bude řídící a bude instruovat agenta na uzlu 3, jakou diodu má rozsvítit. Řídící agent bude umístěn na uzlu 2. Oba agenty můžeme nalézt v tabulce 11.3 a 11.3. Řídící agent nejprve uloží do své báze znalostí informace o tom, v jakém pořadí budou LED rozsvíceny a po jakou dobu. Poté s využitím unifikace postupně testuje svoji bázi znalostí a vybere jeden údaj. Ten zašle agentu na uzlu číslo 3, on tuto zprávu přijme a informuje o tom řídícího agenta 2 zasláním zpětné zprávy. Poslouchající agent dle pokynů přepne danou LED. Řidící agent čeká na zprávu od agenta č. 3 a jakmile ji obdrží, přepne danou LED a čeká požadovanou dobu. Po uplynutí této doby vybírá další hodnotu z báze znalostí a pokračuje dalším cyklem.

Část	**Obsah**

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

`Plan`	`^(prijem)`

Tabulka 11.3: **Agent umístěný na uzlu číslo 3 - poslouchající agent**

Část	**Obsah**

`PlanBase`	`(odesli,($(a)!(3,&2)&(3)$(s)?(3)))`
	`(napln,(+(led,r,600)+(led,g,700)+(led,r,500)+(led,y,800)`
	`^(blik)#^(napln)))`
	`(blik,(&(1)*(led,_,_)&(2)$(f,&1)-&2^(odesli)&(1)$(r,&2)&(3)`
	`$(f,&1)$(l,&3)&(2)$(r,&1)&(1)$(f,&2)$(w,&1)$(l,&3)^(blik)))`

`Plan`	`^(napln)`

Tabulka 11.4: **Agent umístěný na uzlu číslo 2 - řídící agent**

Kapitola 12

Současný stav a závěr

V současné době je implementováno několik vzorových aplikací, které jak softwarově tak i hardwarově otestovali funkčnost všech komponent. V průběhu příštího roku by měl být k dispozici novější typ mote s větší pamětí RAM, nejspíše 8kB. Tyto mote budou kompatibilní s platformou MICAz a proto by při přechodu na nový typ neměly nastat problémy. Další verze mote bude pravděpodobně obsahovat i jinou senzorovou desku. V příští verzi platformy by tedy bylo vhodné zajistit možnost získávání dat ze všech nabízených senzorů. V případě absence ovladače bude vývoj zaměřen také na napsání vlastních ovladačů.

Hlavním přínosem této práce je ověření, že je skutečně možné interpretovat agenty na platformě bezdrátových senzorových sítích. Budoucí verze již budou mít k dispozici základní stavební bloky a funkčnost aplikace budou spíše jen rozšiřovat. Velmi zajímavou oblastí, která bude v dalších verzích zcela jistě rozšířena bude komunikace uzlů. Příkladem může být vyhledání všech okolních uzlů, implementace pachových stop, směrování či reputace jednotlivých agentů.

Dalším možným rozšířením je platforma umožňující běh více agentů na jednom uzlu. Každý agent může být umístěn ve svém kontejneru a bude sdílet prostor s okolními agenty. Implementace virtualizace však bude vyžadovat větší změny v návrhu platformy.

Závěrem lze říci, že senzorové sítě, ve spojení s agenty tak mají opravdu rozsáhlou oblast využití. Agenti mohou například sledovat mostní konstrukce, hladinu spodních vod, detekovat požár či jinou nebezpečnou událost a informovat o vzniklém nebezpečí pověřené osoby. Naší prací byl tedy položen základní kámen v nové oblasti bádání, která může nalézt skutečné uplatnění.

Literatura

[1] Belief-Desire-Intention software model. April 2009.
URL: <http://en.wikipedia.org/wiki/BDI_software_agent>

[2] *Mote-PC serial communication and SerialForwarder*. May 2009.
URL: <http://docs.tinyos.net/index.php/Mote-PC_serial_communication_and_SerialForwarder>

[3] Haenselmann T.: *An FDL'ed Textbook on Sensor Networks*. 2006.
URL: <http://www.informatik.uni-mannheim.de/~haensel/sn_book/sensor_networks.pdf>

[4] Levis P.: TinyOS Programming. October 2006.
URL: <http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf>

[5] Levis P., et. al.: TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. 2003.
URL: <http://www.cs.berkeley.edu/~pal/pubs/tossim-sensys03.pdf>

[6] M. Wooldridge, N. R. Jennings: Inteligent Agents: theory and practice. In *Knowledge Engineering Review*, 1995, s. 115–152.
URL: <http://www.csc.liv.ac.uk/~mjw/pubs/ker95.pdf>

[7] Martincic F., Schwiebert L.: *Introduction to Wireless Sensor Networking*. Detroit, Michigan: Wayne State University, 2005, 40 s.
URL: <http://media.wiley.com/product_data/excerpt/24/04716847/0471684724.pdf>

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

[9] Zbořil, F.: *Framework for Model-Based Design of Multi-agent Systems*. Vienna, AT: ARGESIM, 2007, ISBN 978-3-901608-32-2, 11 s.

[10] Zbořil, F.: *Simulation for Wireless Sensor Networks with Intelligent Nodes*. Cambridge, GB: IEEE Computer Society, 2008, ISBN 0-7695-3114-8, 6 s.
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4489026>

[11] Zbořil, F.: Úvod do agentních a multiagentích systémů, 2008, podklady k přednáškám kurzu AGS.

Seznam příloh

Příloha A: Soubor "`TestSerial.java`"
Příloha B: Soubor "`TestAMAppC.nc`" a "`TestAMC.nc`"
Příloha C: CD obsahující zdrojové texty

Příloha A

Využití Java třídy MessageListener

Soubor ‘‘TestSerial.java’’

```
1  import java.io.IOException;
2
3  import net.tinyos.message.*;
4  import net.tinyos.packet.*;
5  import net.tinyos.util.*;
6
7  public class TestSerial implements MessageListener {
8
9    private MoteIF moteIF;
10
11   public TestSerial(MoteIF moteIF) {
12     this.moteIF = moteIF;
13     this.moteIF.registerListener(new TestSerialMsgIn(), this);
14   }
15
16   public void sendPackets() {
17     int counter = 0;
18     TestSerialMsgOut payload = new TestSerialMsgOut();
19     try {
20       while (true) {
21         System.out.println("Sending packet " + counter);
22         payload.set_counter(counter);
23         moteIF.send(2118, payload);
24         counter++;
25         try {Thread.sleep(1000);}
26         catch (InterruptedException exception) {}
27       }
28     }
29     catch (IOException exception) {
30       System.err.println("Exception thrown when sending packets. Exiting.");
31       System.err.println(exception);
32     }
33   }
34
35   public void messageReceived(int to, Message message) {
36     TestSerialMsgIn msg = (TestSerialMsgIn)message;
37     System.out.println(msg.get_source_adress()+"->"+
38       message.getSerialPacket().get_header_dest() +
39       ": Received packet sequence number " + msg.get_counter());
```

```
40    }
41
42    private static void usage() {
43      System.err.println("usage: TestSerial [-comm <source>]");
44    }
45
46    public static void main(String[] args) throws Exception {
47      String source = null;
48      if (args.length == 2) {
49        if (!args[0].equals("-comm")) {
50          usage();
51          System.exit(1);
52        }
53        source = args[1];
54      }
55      else if (args.length != 0) {
56        usage();
57        System.exit(1);
58      }
59      PhoenixSource phoenix;
60      if (source == null) {
61        phoenix = BuildSource.makePhoenix(PrintStreamMessenger.err);
62      }
63      else {
64        phoenix = BuildSource.makePhoenix(source, PrintStreamMessenger.err);
65      }
66      MoteIF mif = new MoteIF(phoenix);
67      TestSerial serial = new TestSerial(mif);
68      serial.sendPackets();
69    }
70  }
```

Příloha B

Využití komunikačního rozhraní

Soubor ‘‘TestAMAppC.nc’’

```
configuration TestAMAppC {}
implementation {
  components MainC, TestAMC as App, LedsC;
  components ActiveMessageC;
  components new TimerMilliC();

  App.Boot -> MainC.Boot;
  App.Receive -> ActiveMessageC.Receive[6];
  App.AMSend -> ActiveMessageC.AMSend[6];
  App.SplitControl -> ActiveMessageC;
  App.Leds -> LedsC;
  App.MilliTimer -> TimerMilliC;
}
```

Soubor ‘‘TestAMC.nc’’

```
#include "Timer.h"
module TestAMC {
  uses {
    interface Leds;
    interface Boot;
    interface Receive;
    interface AMSend;
    interface Timer<TMilli> as MilliTimer;
    interface SplitControl;
  }
}
implementation {
  message_t packet;
  bool locked;

  event void Boot.booted() {
    call SplitControl.start();
  }
  event void SplitControl.startDone(error_t err) {
    if (err == SUCCESS) {
      call MilliTimer.startPeriodic(1000);
    }
    else {
      call SplitControl.start();
    }
```

```
26    }
27    event void SplitControl.stopDone(error_t err) {
28    }
29    event void MilliTimer.fired() {
30      if (locked) {
31        return;
32       }
33      else if (call AMSend.send(AM_BROADCAST_ADDR, &packet, 0) == SUCCESS) {
34        call Leds.led0On();
35        locked = TRUE;
36      }
37    }
38    event message_t* Receive.receive(message_t* bufPtr,
39                                     void* payload, uint8_t len) {
40      call Leds.led1Toggle();
41      return bufPtr;
42    }
43    event void AMSend.sendDone(message_t* bufPtr, error_t error) {
44      if (&packet == bufPtr) {
45        locked = FALSE;
46        call Leds.led0Off();
47      }
48    }
49  }
```