VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

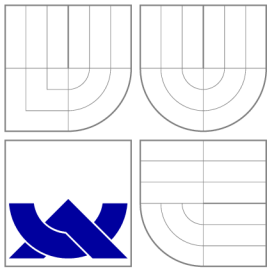
INTELIGENTNÍ AGENTI V BEZDRÁTOVÝCH SÍTÍCH

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

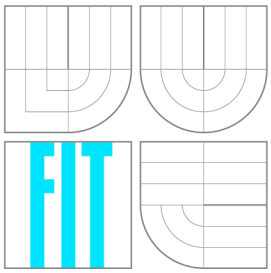
AUTOR PRÁCE
AUTHOR

Bc. MIROSLAV KRUŽLIAK

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

INTELIGENTNÍ AGENTI V BEZDRÁTOVÝCH SÍTÍCH

INTELLIGENT AGENTS IN WIRELESS NETWORKS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MIROSLAV KRUŽLIAK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZBOŘIL FRANTIŠEK, Ph.D.

BRNO 2010

Zadání práce

1. Seznamte se s metodami modelování inteligetních mobilních agentů pro distribuované systémy. Zaměřte se na systémy Jason a Samson.

2. Navrhněte algoritmy pro globální synchronizaci agentů v distribuovaných systémech jako jsou WSN. Zvolte vhodné aplikace, na kterých se dá ukázat správnost navržených algoritmů.

3. Implementujte tyto algoritmy ve výše uvedených systémech včetně prostředí, ve kterém senzorové uzly měří veličiny.

4. Ověřte, že navrženými algoritmy lze dosáhnout uspořádání nezávislých jevů podle dob, ve kterých se udály. Určete případné problematické situace, kdy takové uspořádání určit nelze.

5. Výsledky zhodnoťte a diskutujte zvolený přístup.

6. Vytvořte poster formátu A2, který bude názorně demonstrovat vytvořené dílo, jeho architekturu, principy a použitelnost pro reálné aplikace.

Abstrakt

Práca sa zaoberá synchronizáciou senzorových uzlov v bezdrátovej senzorovej sieti. Využíva sa tu usporiadanie udalostí pomocou logických hodín. Pre synchronizáciu je použitý Lamportov algoritmus, ktorý sa snaží usporiadať udalosti globálne v rámci uvažovaného systému. Práca tiež vyhodnocuje vhodnosť použitia takéhoto princípu pre synchronizáciu. Implentácia je prevedená v agentne-orientovanom jazyku AgentSpeak na platforme Jason. Pre pozorovanie správania takejto synchronizácie a na testovacie účely bolo použité a upravené prostredie Samson.

Abstract

This Master thesis deals with synchronization of sensor nodes in wireless sensor net. It is used event ordering by the implementation of logical clocks . Lamport's algorithm is used here for synchronization, which is trying to order events within the given system. The thesis also evaluates how appropriate this principle for synchronization is. The implementation has been carried out in agent-oriented language AgentSpeak on the Jason platform. Samson environment has been used and modified for observation of this synchronization's behaviour and testing purposes.

Klíčová slova

Lamportov algoritmus, Jason, Samson, AgentSpeak, synchronizácia, logické hodiny, WSN, distribuovaný systém, agent

Keywords

Lamport's algorithm, Jason, Samson, AgentSpeak, synchronization, logical clocks, WSN, distributed system, agent

Citace

Miroslav Kružliak: Inteligentní agenti v bezdrátových sítích, diplomová práce, Brno, FIT VUT v Brně, 2010

Inteligentní agenti v bezdrátových sítích

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Františka Zbořila, Ph.D.

.....
Miroslav Kružliak
26. května 2010

Poděkování

Týmto by som chcel poďakovať Ing. Františkovi Zbořilovi, Ph.D. za jeho pomoc pri konzultáciach a za jeho nápady pri tvorbe tejto diplomovej práce.

Obsah

Kapitola 1

Úvod

Jedným z problémov v efektívnom spracovaní informácií v bezdrátových senzorových sietiach je časová synchronizácia jednotlivých uzlov v tejto sieti. Vzhľadom k tomu, že synchronizácia pomocou fyzických hodín(napríklad pomocou *NTP*[1] protokolu) je zložitá na výpočetný výkon jednotlivých uzlov a pohlcuje priestor na komunikačnom médiu, je nutné uvažovať energeticky efektívne algoritmy pre synchronizáciu uzlov vo WSN[2]. Ďalšou nevýhodou synchronizácie pomocou fyzických hodín je ich nepresnosť(rozdiel časového údaju dvoch nezávislých hodín), preto sa v tejto práci budeme zaoberať hodinami, ktorých jednotkou času je udalosť vyvolaná prvkom distribuovaného systému a nie fyzický čas. Problémom, ktorý sa nám v takomto prípade vynoruje je usporiadanie nezávislých udalostí v celom distribuovanom systéme z rôznych zdrojov. Aj tento problém a spôsoby jeho eliminácie budú diskutované v tejto práci.

1.1 Organizácia práce

Zadaním práce je aj zoznámiť sa s platformou pre agentne orientované programovanie **Jason**, tejto problematike sa venuje celá prvá kapitola, kde sa zaoberáme postupne od architektúry takéhoto programovania až po vysvetlenie syntaxe jazyka AgentSpeak, pre ktorý je táto platforma navrhnutá.

Ďalšia kapitola je venovaná simulátoru pre multi-agentné **SAMSON** prostredie založeného na tejto platforme. Tento simulačný nástroj bude ďalej použitý pre demonštráciu práce algoritmu a pre testovacie účely,

Nasledujúca kapitola sa zaoberá samotnou problematikou synchronizácie.Tu predstavíme možnosti synchronizácie uzlov v distribuovanom systéme, ktoré nie sú závisle od centralizovaného riadenia synchronizácie ani od fyzických hodín. Jedným z mechanizmov pre takúto synchronizáciu je aj implementácia tzv. **Logických hodín**. Prevažne sa tu budeme zaoberať jedným z najznámejších algoritmov, **Lamportovým algoritmom**. No predstavíme si stručne aj algoritmus **vektorových hodín**.

V kapitole 5 si ukážeme návrh agenta, ktorého správanie je podriadené synchronizáciou s ostatnými agentmi. Prvky tohto návrhu sú podriadené princípom fungovania agentne-orientovaného jazyka AgentSpeak.

Kapitola 6 opisuje detailnejšie prvky návrhu spomenuté v predchádzajúcej kapitole. Zaoberá sa detailmi implemntácie a ukazuje spôsob rozhodovania agenta v rôznych situáciach

[1] Network Time Protocol

[2] Wireless Sensor Network - bezdrová senzorová sieť

jeho života. Ďalej tu budú popísané zmeny prevedené v prostredí simulačného nástroja SAMSON a takisto sú tu ukázané výstupy aplikácie, ktoré demonštrujú aktuálnu činnosť agenta.

Nakoniec dané riešenia a jeho vhodnnosť použitia otestujeme z rôznych hľadísk a budeme diskutovať možnosti eliminácie chýb vzniknutých počas synchronizácie.

Kapitola 2

Platforma Jason

Jason je platforma pre vývoj v multi-agentných systémoch, je rozšírením agentne-orientovaného jazyka *AgentSpeak*, tieto rozšírenia však nezvyšujú vyjadrovaciu silu jazyka. Platforma Jason je používaná pre programovanie a simulácie agentov v multi-agentných systémoch, kde sa používa na definovanie správania týchto agentov. Táto platforma je implementovaná v jazyku Java a používa sa ako zásuvný modul do vývojových prostredí ako je jEdit alebo Eclipse.

2.1 BDI architektúra

Úvodom tejto kapitoly si predstavíme filozofický základ pre programovanie agentov akým je BDI[1] architektúra. Táto architektúra nám umožňuje dostatočnú abstrakciu správania agentov v systéme. Následne si predstavíme jednotlivé časti tejto architektúry(využijeme anglické termíny):

Beliefs Informácie o prostedí, v ktorom sa agent nachádza. Tieto informácie mohol agent získať svojím vnímaním, alebo ich mohol získať od iných agentov v prostredí.V tejto práci ich budeme označovať ako predstavy alebo znalosti.

Desires Sú cieľami, ktoré sa agent snaží nasledovať alebo dosiahnúť. Mať takýto cieľ neznamená, že agent musí podľa neho bezpodmienečne konať. Je dokonca žiadúce, aby agent obsahoval túžby, ktoré si odporujú. Túžby sú voľby agenta v rozhodovaní svojho ďalšieho konania.

Intentions Sú stavmi aktuálne vykonávaných cieľov agenta. Agent si vyberie činnosť, ktorú bude vykonávať a pri tomto vykonávaní sa kroky tejto činnosti stanú zámermi agenta.

Viac informácií o BDI architektúre nájdete aj v tejto literatúre [1], [7].

2.1.1 Riadiaci cyklus agenta

Vývoj agenta pomocou vyššie uvedenej architektúry vyžaduje aj znalosť riadiaceho cyklu agenta, v ktorom agent mení svoje ciele na základe informácií získaných z prostredia alebo od iných agentov. Podľa literaúry [1] by pseudokód riadiaceho cyklu agenta mohol vyzerať takto:

[1] Beliefs, Desires, Intentions - voľný preklad: predstavy, túžby a zámery

Algoritmus 2.1.1 Algoritmus riadiaceho cyklu agenta

1: $B \leftarrow B_0$ /*Počiatočné predstavy agenta*/
2: $I \leftarrow I_0$ /*Počiatočné zámery agenta*/
3: **while true do**
4: získaj ďalšiu informáciu p o prostredí zo senzorov
5: $B \leftarrow brf(B, p)$;
6: $D \leftarrow options(B, I)$;
7: $I \leftarrow filter(B, D, I)$;
8: $\pi \leftarrow plan(B, I, Ac)$; /*Ac je množina akcií agenta*/
9: **while** $not(empty(\pi)$ *or* $succeeded(I, B)$ *or* $impossible(I, B))$ **do**
10: $\alpha \leftarrow$ first element of π;
11: $execute(\alpha)$;
12: $\pi \leftarrow$ tail of π;
13: získaj ďalšiu informáciu o prostredí p;
14: $B \leftarrow brf(B, p)$;
15: **if** $reconsider(I, B)$ **then**
16: $D \leftarrow options(B, I)$;
17: $I \leftarrow filter(B, D, I)$;
18: **end if**
19: **if** $sound(\pi, I, B)$ **then**
20: $\pi \leftarrow plan(B, I, Ac)$;
21: **end if**
22: **end while**
23: **end while**

V uvedenom pseudokóde algoritmu 2.1.1 predstavujú premenné B, D, I aktuálne predstavy, túžby a zámery agenta. Ich význam bol už opísaný vyššie.

Hlavný riadiaci cyklus sa v algoritme nachádza medzi riadkami 3 a 23. V tomto cykle agent najprv získa žiadané informácie o prostredí pomocou senzorov, potom na riadku 5 aktualizuje svoje predstavy o prostredí pomocou funkcie brf[2]() a získaných informácií o prostredí.

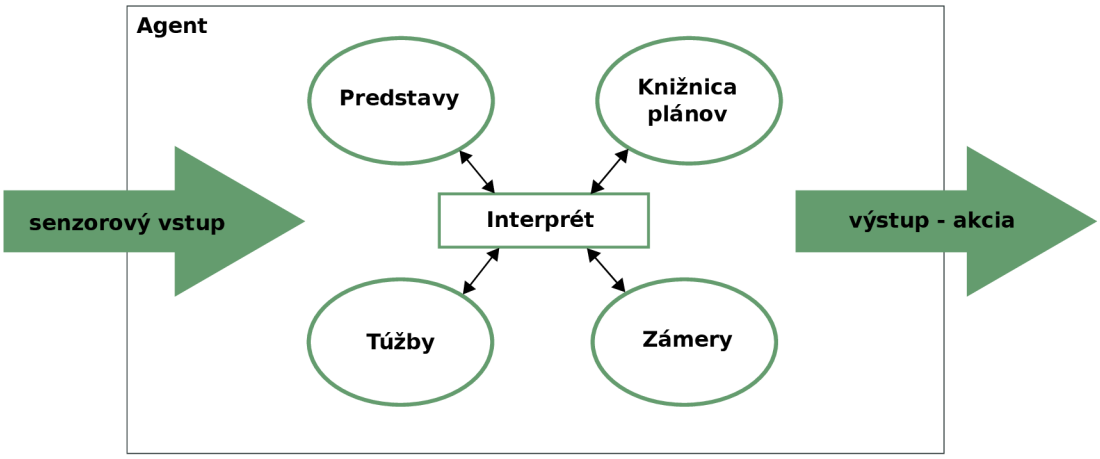
Následne na riadku 6 agent aktualizuje svoje túžby na základe predstáv o prostredí a aktuálnych zámerov agenta. Toto sa prevedie na základe funkcie *options*(). Potom vyberie niektoré z týchto túžob a tie sa stanú zámermi (riadok 7, funkcia $filter()$). Agent teraz zostaví plán na dosiahnutie svojich zaktualizovaných zámerov.

Vnútorný cyklus (riadky 9 - 22) zachytáva realizáciu plánov agentom na dosiahnutie svojich zámerov. Agent skončí štandardne tento cyklus ak je vybratý aktuálny plán činnoti agenta π prázdny. Na riadku 12 agent opäť získava informácie o prostredí, aby aktualizoval svoje predstavy. Následne potom sa agent rozhodne, či nové poznatky stoja za to, aby prehodnotil svoje zámery (pomocou funkcie $reconsider()$). Agent aktualizuje svoje zámery a plány vtedy, ak na základe novozískaných informácií z prostredia zistil, že by sa jeho zámery mali zmeniť. Inak nemá zmysel uvažovať o zmene zámerov. Na konci vnútorného cyklu agent pomocou funkcie *sound*() rozhodne, či aktuálny plán vyhovuje zámerom a predstavám agenta. Ak nevyhovuje, agent vytvorí nový plán činností agenta.

[2]belief revision function

2.1.2 PRS architektúra

Vyššie uvedený algoritmus nám nehovorí o samotnej implementácií jednotlivých funkcií v rozhodovaní agenta. PRS[3] architektúra bola jedna z prvých, ktorá obsahovala prvky architektúry BDI. Hrubú koncepciu tejto architektúry zobrazuje obrázok 2.1.



Obrázok 2.1: PRS architektúra

V PRS architektúre agent nemá na začiatku svojej činnosti žiadne plány, namiesto toho je vybavený knižnicou predkompilovaných plánov, ktoré sú definované programátorom. Každý plán v tejto architektúre pozostáva z týchto komponent:

Cieľ Podmienka, ktorá musí platiť po vykonaní plánu.

Kontext Podmienka, ktorá musí platiť pred vykonaním plánu.

Telo Činnosť, ktorá vedie k splnenniu plánu.

V jazyku AgentSpeak, ktorého rozšírením je platforma Jason je zakomponovaná PRS architektúra tak, aby bola vyditeľnou súčasťou syntaxe tohto jazyka. Cieľom vývoja tohto jazyka bolo poskytnúť jednoduché homogénne programovacie prostredie pre implementáciu rozsiahlych škálovateľných systémov hlavne pre pozorovanie a získavanie teoretických znalostí o distribuovaných multi-agentných systémoch.

2.2 Základy jazyka AgentSpeak pre platformu Jason

Syntax jazyka AgentSpeak vychádza z logického programovacieho jazyka *Prolog*. Syntax jazyka sa skladá z nasledujúcich prvkov, analogicky s architektúrou BDI, podľa [2]:

[3]Procedural Reasoning System

Predstavy: `predikát(termy).`

Plány: `+udalosť : kontext <- telo.`

Telo

Akcia: `akcia(termy).`

Dosahované ciele: `!cieľ(termy)`

Testované ciele: `?predstava(termy)`

Pridanie predstavy: `+predstava(termy)`

Odobranie predstavy: `-predstava(termy)`

V následujúcich podkapitolách si predstavíme jednotlivé časti jazyka detailnejšie. Viac o syntaxy a spôsobe práce tejto platformy nájdete v tejto literatúre [1], odkiaľ bola použitá aj väčšina informácií použitých v tejto kapitole.

2.2.1 Predstavy

Predstavy sú prvou vecou, ktorú je potrebné vedieť pre vývoj v jazyku AgentSpeak. Ako bolo povedané vyššie, každý agent obsahuje štruktúru, v ktorej uchováva svoje predstavy o prostredí v ktorom sa nachádza. Túto štruktúru budeme ďalej nazývať **množina predstáv**.

V logických programovacích jazykoch ju možno predstaviť ako množinu predikátov napríklad:

```
tall(john).
```

Tento predikát priradzuje termu '*john*' vlastnosť '*tall*', čo nám hovorí niečo o objekte 'john' (v uvedenom príklade sa hovorí, že john je vysoký). Takýmito literálmi(predstavami) môžme takisto vyjadriť vzťah medzi viacerými objektmi, napríklad:

```
likes(john, music).
```

Čo nám logicky hovorí, že john má rád hudbu. Agent v obidvoch prípadoch len „verí“, že uvedené predstavy o prostredí sú pravdivé, v skutočnosti pod vplyvom rôznych podmienok nemusia byť pravdivé. Bližšie o logickom programovaní pomocou takýchto predikátov môžme najsť v špecializovanej literatúre a detailný popis takýchto programovacích techník nie je súčasťou tejto práce.

Anotácie

Jedným z rozšírením platformy Jason oproti špecifikácií jazyka AgentSpeak je používanie anotácií. Príklad takejto anotácie je:

```
busy(john)[expires(autumn)].
```

Tento predikát nám hovorí, že objekt john bude zaneprázdnený do jesene. Anotácia nezvyšuje vyjadrovaciu schopnosť jazyka AgentSpeak. Výhoda anotácií je v sprehladnení programovania v jazyku, a tiež ulahčuje správu množiny predstáv. Nami definovaná anotácia `expires(autumn)` nehovorí nič interpretu jazyka o odstránení tejto predstavy akonáhle

nastane jeseň. Avšak programátor môže nadefinovať správanie funkcie pre aktualizáciu množiny predstáv tak, aby táto reagovala na zvolené anotácie podľa potrieb programátora.

Sú však špeciálne anotácie, ktorým interpret jazyka rozumie. Takouto anotáciou je `source`, ktorá hovorí interpretu, čo je zdrojom jednotlivých predstáv, z čoho potom dokáže pomocou funkcie pre aktualizáciu predstáv s touto informáciou pracovať.

Pravidlá

Predstavy je možné taktiež zadávať pomocou pravidiel, čo hlavne pre skúsenejších programátorov môže predstavovať uľahčenie implementácie agenta. Príklad takéhoto plánu môže byť:

```
likes(A, B)
:- A == john & subject(B)[source(S)] & (S == self | S == percept).

likes(A, B)
:- girl(A)[source(mary)] & flower(B).
```

Prvé pravidlo vracia hodnotu pravda, ak prvý term predstavy je `john`, a ak predmet, ktorý má rád je získaný buď zo svojich predstáv alebo vnímaním z prostredia agenta(možnosti použitia špeciálnej anotácie `source`, pojednávanej vyššie). Ak logická hodnota tela tohto pravidla nie je pravda, tak sa vyhodnocuje telo druhého pravidla. Tu je predstava vyhodnotená ako pravda vtedy, ak prvý term predstavy je dievča, a táto predstava je získaná od agenta `mary` a predmet, ktorý má toto dievča rado je kvet.

2.2.2 Ciele

Jednou z najdôležitejších súčastí agenta je stanovenie jeho cieľov. V jazyku AgentSpeak poznáme dva druhy cieľov: *dosahované a testovacie*. Dosahovaný ciel je označovaný operátorom '!'. Napríklad cieľ `!open(door)` hovorí agentovi, že má za cieľ dosiahnúť taký stav, aby bola naplnená predstava `open(door)`. V sekcií 2.2.3 si predstavíme, akým spôsobom agent dosahuje stanovené ciele.

Testovacie ciele slúžia jednoducho na získavanie informácií z predstáv agenta, označujú sa symbolom '?'. Napríklad testovací cieľ `?account_balance(N)`, nám z množiny predstáv agenta zistí aktuálny stav účtu, priradí ho do premennej N a umožní nám s touto hodnotou ďalej pracovať.

Predstavy a ciele agenta sú nedielnou súčasťou pre činnosti agenta. Takouto sú aj plány, ktoré spúšťajú akcie na základe zmien v množine predstáv a cieľov agenta.

2.2.3 Plány

Plán v jazyku AgentSpeak má tri zložky: *spúšťaná udalosť, kontext, telo*. Plán je v tvare:

```
spúšťaná udalosť : kontext <- telo.
```

V nasledujúcich statiach si jednotlivé časti bližšie opíšeme.

Spúšťaná udalosť

Ak sme už povedali, agent v svojej podstate disponuje dvoma typmi znalostí, ktoré riadia jeho činnosť: predstavami a cieľmi. Spúšťacie udalosti reprezentujú zmeny v množine predstáv alebo v množine cieľov agenta. Nasledujúca tabuľka 2.1 zobrazuje prehľad udalostí, ktoré môžu byť prevádzané v tejto časti definície plánu.

Notácia	**Názov**
$+l$	Pridanie predstavy
$-l$	Odstránenie predstavy
$+!g$	Pridanie dosahovaného cieľa
$-!g$	Odstránenie dosahovaného cieľa
$+?g$	Pridanie testovacieho cieľa
$-?g$	Odstránenie testovacieho cieľa

Tabuľka 2.1: Typy spúšťaných udalostí

Udalosti pre pridanie a odobranie predstáv sa vykonávajú, keď agent aktualizuje svoju množinu predstáv, napríklad pri snímaní hodnôt z prostredia, čo sa deje pri každom interpretačnom cykle. Tento cyklus bude diskutovaný v sekcií 2.3.

Pridávanie a odoberanie cieľov sa deje hlavne pri spušťaní iných plánov, ale takisto ako výsledok komunikácie medzi agentmi.

Kontext

Kontext predstavuje logický výraz(literál), ktorý je podmienkou pre aplikovanie daného plánu. Takže pre splnenie daného plánu je potrebné, aby podmienka v kontexte plánu bola vyhodnotená ako pravda. V platforme Jason poznáme niekoľko možností ako vyjadriť obsah podmienky v kontexte. Tieto sú zobrazené v tabuľke 2.2.

Syntax	**Význam**
l	Agent verí, že literál l je pravdivý
$\sim l$	Agent verí, že literál l je nepravdivý
$not\ l$	Agent neverí, že literál l je pravdivý
$not\ \sim l$	Agent neverí, že literál l je nepravdivý

Tabuľka 2.2: Typy literálov v kontexte plánu

Telo

Telo plánu sa spúšťa v prípade, ak nastala udalosť definovaná ako spúšťacia a kontext plánu bol pravdivý vzhľadom na množinu predstáv agenta. Prevádzanie tela je definované ako sekvencia formúl oddelených znakom ';'. Poznáme šesť druhov takýchto udalostí:

Akcie: Predstavujú externé akcie, ktoré sú nadefinované programátorom. Agent pomocou nich mení prostredie v ktorom sa nachádza. Príklad: `rotate(left_arm, 45)`.

Dosahovaný cieľ: Bol diskutovaný vyššie. Ak je nadefinovaný plán pre dosiahnutie tohto cieľa, musí sa spustiť pre dokončenie tela aktuálneho plánu.

Testovací cieľ: Slúži pre získanie informácií z množiny predstáv alebo pre zistenie či, je na základe množiny predstáv nejaká informácia pravdivá. Taktiež môže spustiť iný plán. Jeho syntax bola popísaná v predchádzajúcich statiach.

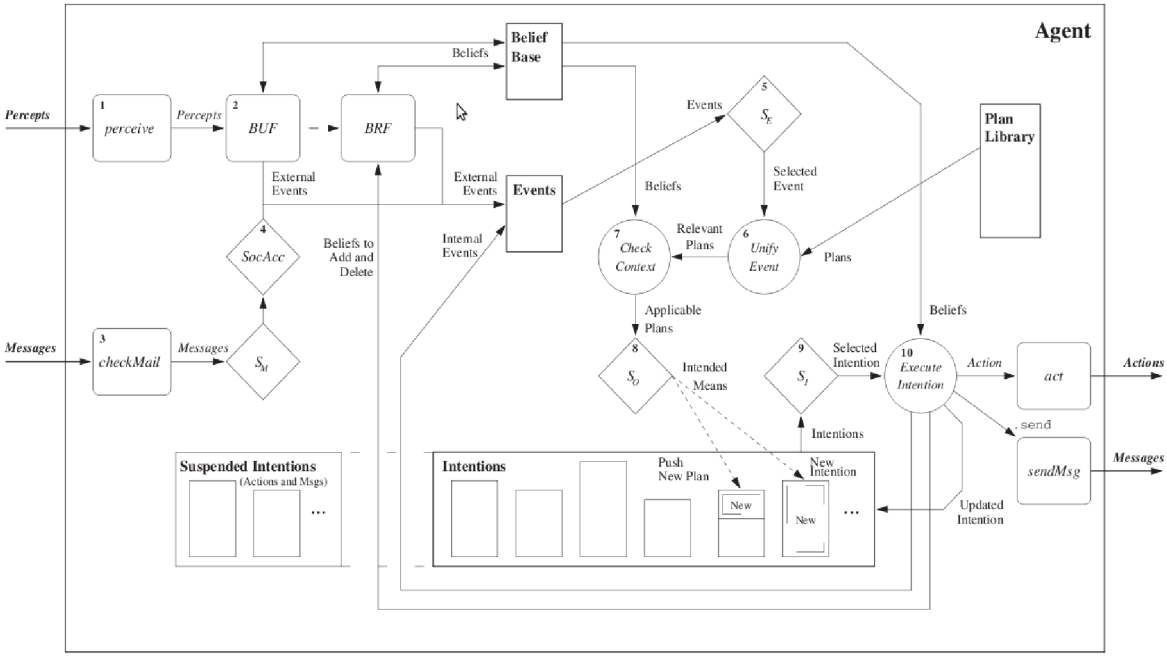
Vlastné záznamy: Tieto záznamy sú pridané ako nové predstavy do množiny predstáv a majú anotáciu $source(self)$.

Interné akcie: Narozdiel od externých akcií, tieto akcie nemenia prostredie agenta vykonávaním nejakej externej činnosti. Ich volanie sa dolišuje tým, že na začiatku volania akcie je symbol '.'. Tieto akcie tiež môžu byť špecifikované programátorom.

Výrazy: Výsledok každej formuly v tele plánu musí mať nejakú pravdivostnú hodnotu. Takouto formulou sú aj výrazy. Príklad: $M < 9$.

2.3 Interpretačný cyklus agenta

Na obrázku 2.2 je schématický popis fungovania interpretačného cyklu agenta. Tento popis je prebratý z literatúry [1]. Interpretačný cyklus je analogický s algoritmom 2.1.1.



Obrázok 2.2: Popis interpretačného cyklu agenta

Jednotlivé kroky v tomto cykle sú na obrázku označené číslami. V interpretačnom cykle sa tak nachádza 10 hlavných krokov. Obdlžníky v schéme predstavujú komponenty zachycujúce aktuálny stav agenta(množina predstáv, množina udalostí, množina zámerov, knižnica plánov, ...).

Kružnice, kosoštvorce a obdlžníky so skosenými hranami predstavujú funkcie. Posledné dve menované môžu byť modifikované programátorom a rozdiel medzi nimi je, že kosoštvorce sú výberovou funkciou a vyberajú jeden prvok, zatiaľ čo funkcie v obdlžníkoch so skosenými hranami môžu vrátiť zoznam prvkov. Činnosť funkcií, predstavenými ako kružnice, nemôže byť modifikovaná programátorom, je daná implicitne v interprete platformy Jason.

Na začiatku života agenta je programátorom tiež nadefinovaná množina predstáv, množina udalostí a knižnica plánov, zatiaľ čo množina zámerov je pri inicializácií prázdna.

Podrobnejšie si jednotlivé kroky interpretačného cyklu nebudeme popisovať, záujemci ich môžu nájsť detailne popísané v tejtto literatúre [1].

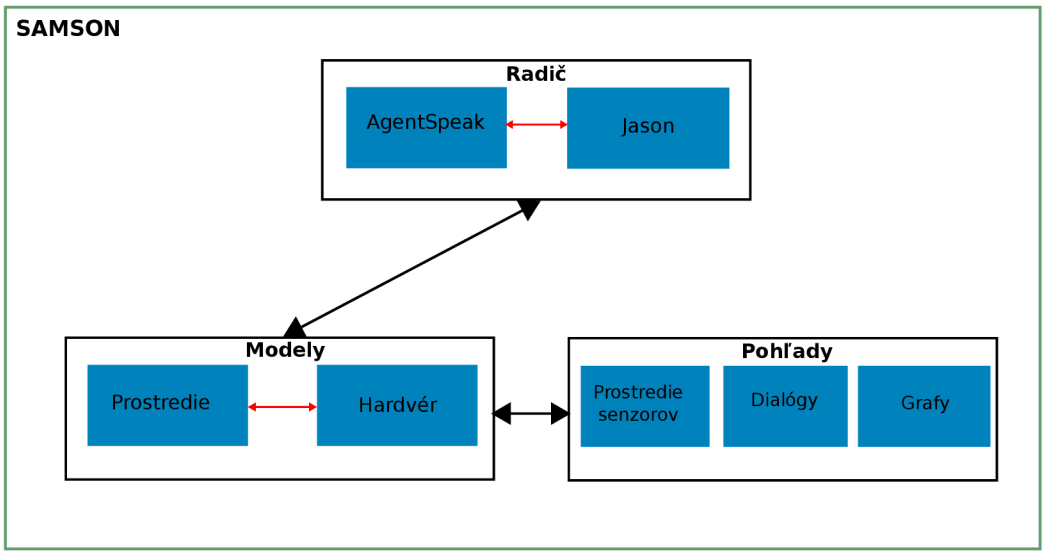
2.4 Zhrnutie

V tejto kapitole sme si predstavili agentne orientovany jazyk AgentSpeak ako aj platformu Jason, pod ktorou je možné v tomto jazyku vyvíjať agentov. Taktiež bola predstavená architektúra BDI a PRS, ktoré sú základom pochopenie vývoja pod platformou Jason. Postupne sme sa zoznámili s jednotlivými prvkami syntaxe jazyka AgentSpeak a tiež sme si vysvetlili princíp fungovania interpreta tohto jazyka a jeho interpretačného cyklu. Tento jazyk bude neskôr použitý pre potreby vývoja synchronizácie medzi uzlami vo WSN. Pre detailnejšie štúdium platformy Jason a jazyka AgentSpeak odporúčam túto literatúru [1], ktorá bola hlavným zdrojom pri tvorbe tejto kapitoly.

Kapitola 3

Simulačný nástroj SAMSON

Pre potreby simulácií a pozorovaní algoritmov pre synchronizáciu vo WSN využijeme simulátor SAMSON[1]. Koncepcia implementácie simulátora vychádza z návrhového vzoru MVC[2] a ako základ pre agentné správanie boli použité silné agentné systémy založené na BDI architektúre. Architektúra simulátora je zobrazená na obrázku 3.1.



Obrázok 3.1: Popis architektúry simulátora, podľa MVC návrhového vzoru

Ako vidíme z obrázka, úlohu radiča v architektúre simulátora prevzala platforma Jason s agentne-programovacím jazykom AgentSpeak, ktoré boli popísané v predchádzajúcej kapitole. Následne si popíšeme ostatné komponenty tejto architektúry.

[1] Strong Multi-Agent Simulation of Wireless Sensor Networks
[2] Model-View-Controller

3.1 Hardvérový model

Model uzlov sa vzťahuje na reálnu hardvérovú architektúru uzlov TMote Sky. Hardvérový model jednotlivých uzlov vo WSN v simulátore SAMSON pozostáva z týchto komponentov: radič, senzory a akčné členy, komunikačné zariadenie a zdroj energie. Správa pamäte aktuálne nie je v simulátore implementovaná.

Implementácia radiča uzlu je založená agentne na platforme Jason. Senzory sú pasívne a všesmerové, jednoducho len zbierajú vzroky z prostredia. V simulátore nie sú implementované žiadne aktívne členy pre pohyb senzorov v prostredí. Paremetry senzorov(rozsah, dosah) sa v simulátore dajú nastviť.

Transportná vrstva komunikačného zariadenia je modelovaná abstraktne, takže o každej správe je predpokladané, že dorazí. Transceiver komunikačného zariadenie je schopný prechádzať niekoľkými stavmi s rôznou spotrebou energie uzlu. V modeli nie sú uvažované komunikačné kanály.

Zdroj energie je modelovaný ako štandardné AA batérie. Spotreba energie je závislá na stave radiča, stave transceiveru komunikačného zariadenia, zápisov/čítaní z/do pamäte a aktuálneho použitia senzorov. Ak kapacita batérií klesne pod určitú hodnotu, zmenší sa hladina napájania zariadenia.

3.2 Model prostredia

Model prostredia v simulátore zahŕňa umiestnenie uzlov, umiestnenie prekážok a nastavanie vlastností senzorov, ktoré sú závislé na aktuálnom umiestnení senzora. Prostredie je v tvare mriežky, kde každá sekcia tejto mriežky obsahuje informáciu o vplyve na vlnové šírenie z/do zariadení. Prekážky v mriežke majú utlmovací účinok na šíriaci sa vlnový signál medzi uzlami. Každá buňka mriežky predstavuje v reálnom svete jednotku o dĺžke približne jeden meter.

3.3 Komponenta zobrazenia simulácie

Táto komponenta predstavuje grafické rozhranie pre zobrazenie stavu simulácie a pre nastavenie jednotlivých prvkov v nej. Obsahuje nasledujúce hlavné prvky pre interakciu s koncovým užívateľom:

Prostredie senzorov Mapa prostredia, kde sa nachádzajú jednotlivé senzorové uzly, dovoľuje nám meniť polohu prekážok, uzlov a slúži aj pre zmenu niektorých parametrov.

Grafy Zobrazujú výsledok merania energie v batériach, atď.

Menu Poskytuje rozhranie pre načítanie, uloženie máp alebo vytvorenie nových.

Konzola Štandardná konzola platformy Jason, používaná na krokovanie simulácie.

Dialógy Používajú sa na komunikáciu s užívateľom a pre ďalšie nastavenia agentného prostredia a uzlov v ňom.

3.4 Zhrnutie

V tejto kapitole sme si stručne opísali simulátor multi-agentného prostredia založený na princípe WSN, zoznámili sme sa s jeho architektúrou a predstavili sme si jednotlivé časti podrobnejšie. Sú tu použité prevažne informácie z literatúry [5], kde je možné nájsť detailnejšie informácie o simulačnom prostredí SAMSON.

Kapitola 4

Algoritmy pre synchronizáciu pomocou logických hodín

V tejto časti si predstavíme synchronizáciu pomocou hodín, ktoré neuvažujú konkrétny absolútny čas, kedy sa udalosť v procese odohrala, ale priraďujú tzv. „časové známky“, ktoré len určujú relatívne poradie udalostí v rámci daného distribuovaného systému, akým synchronizácia vo WSN skutočne je. Jednotlivé udalosti v prvkoch distribuovaného systému spolu komunikujú pomocou zasielania správ. V tejto práci budeme oddelené časti distribuovaného systému (s ohľadom na použitú literatúru) nazývať procesy.

Tu rozlišujeme dva spôsoby usporiadania udalostí v distribuovanom systéme. Prvým je **úplne usporiadanie udalostí** nad distribuovaným systémom, kedy každej udalosti v danom systéme priradíme určitým algoritmom jedinečnú hodnotu časovej známky. Druhým spôsobom je **čiastočné usporiadanie udalostí**, kde uvažujeme len udalosti, ktoré ovplyvňujú ostatné udalosti v iných procesoch distribuovaného systému alebo udalosti v rámci toho istého procesu. Ak sa udalosť deje v rámci jedného procesu a neovplyvňuje ďalšie prvky v systéme, tak je zbytočné tento systém synchronizovať. Viac o porovnaní spôsobov časovej synchronizácie v distribuovaných systémoch môžme nájsť v tejto literatúre [3]. V nasledujúcej časti si predstavíme synchronizáciu pomocou **Lamportových hodín**.

4.1 Lamportov algoritmus

V tejto časti si vysvetlíme princípy Lamportovho algoritmu, kde je ako základ synchronizácie použitá relácia **„happened before“**, ktorá je základom pre usporiadanie udalostí v distribuovanom systéme.

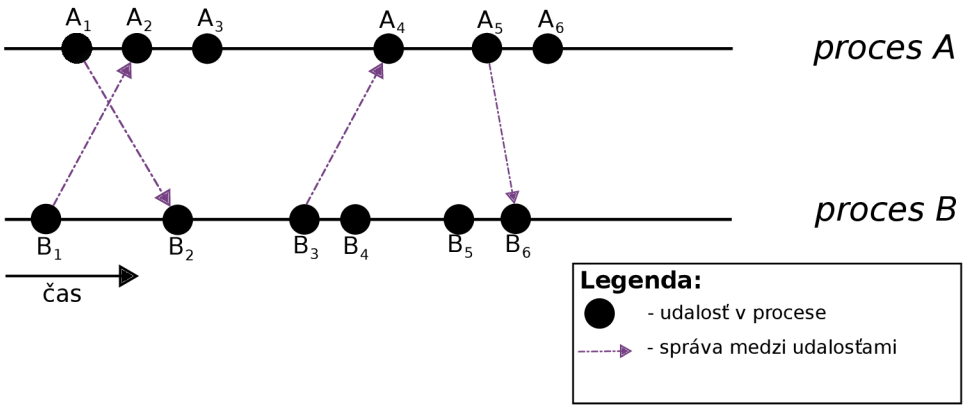
4.1.1 Relácia „happened before“

Túto reláciu označíme „$\rightarrow$“ s ohľadom na ostatnú literatúru.

Definícia. Relácia „$\rightarrow$“ na množine udalostí distribuovaného systému je najmenšia relácia splňujúca jednu z týchto troch podmienok:

1. Ak a a b sú udalosti v rovnakom procese a udalosť a predchádza udalosť b, tak $a \rightarrow b$.
2. Ak a je udalosť posielania správy jedným procesom a b je prijatie tejto správy iným procesom, potom $a \rightarrow b$.
3. Ak $a \rightarrow b$ a zároveň $b \rightarrow c$, potom $a \rightarrow c$.

Dve rozdielne udalosti nazývame *konkurentnými* vtedy, ak $a \not\rightarrow b$ a zároveň $b \not\rightarrow a$, to značí, že tieto udalosti sa kauzálne neovplyvňujú. Zároveň môžme z matematického hľadiska povedať o relácií „$\rightarrow$“, že je *irreflexívna*, *antisymetrická* a *tranzitívna* nad množinou udalostí v distribuovanom systéme.



Obrázok 4.1: Príklad komunikácie jednoduchého distribučného systému

Na obrázku 4.1 môžme vidieť príklad jednoduchého distribuovaného systému o dvoch procesoch, kde sú jednotlivé udalosti v procesoch usporiadané v čase. Tu môžme vidieť, že udalosti A_3 a B_3 sú konkurentné, pretože proces B sa môže dozvedieť, čo proces A urobil v udalosti A_3 až v udalosti B_6.

4.1.2 Funkcia logických hodín v Lamportovom algoritme

Ako už bolo spomenuté vyššie logické hodiny sú úplne osamostatnené od reálneho(fyzického) času. Tieto hodiny bývajú implementované ako jednoduchý čítač udalostí, ktorý pracuje buď nad celým systémom, alebo len nad jednotlivými procesmi systému. Jednou z podmienok dobrého návrhu algoritmu pre synchronizáciu procesov v distribuovanom systéme je, aby hodnota tohto čítača bola globálna, tzn., že pri výskyte udalosti, pre ktorú je treba synchronizácia celého systému je potreba, aby každý z processov vedel o aktuálnej hodnote tohto čítača.

Pre vysvetlenie práce algoritmu si nadefinujeme funkciu logických hodín C_i pracujúcich nad procesom P_i. Táto funkcia priradí každej udalosti daného procesu jedinečnú celočíselnú hodnotu.

Ďalej z pohľadu celého systému definujeme funkciu C, ktorá predstavuje hodiny nad celým systémom vtedy, ak udalosti a priradí nejakú celočíselnú hodnotu. Pre túto platí, že $C(a) = C_i(a)$ vtedy, ak udalosť a patrí do procesu P_i.
Pre správnu prácu logických hodín musia naše definície byť postavené na poradí udalostí v akom prebiehali.

Najsilnejšou v takomto systéme je podmienka, že ak sa udalosť a objaví pred udalosťou b, tak by mal byť čas udalosti a nižší ako čas udalosti b.

Táto podmienka je nazývaná *Podmienka hodín*[1].
Zapísané formálne:

$\forall a, \forall b : a \rightarrow b \Rightarrow C(a) < C(b)$, kde a a b sú nejaké udalosti v systéme.
Z definície relácie „$\rightarrow$“ môžme potom povedať, že *podmienka hodín* je splnená v nasledujúcich dvoch podmienkach:

Podmienka $C1$: Ak a a b sú udalosťami v procese P_i a a predchádza b, potom $C_i(a) < C_i(b)$.
Podmienka $C2$: Ak a je odosielateľom správy v procese P_i a b je prijímateľom tejto správy v procese P_j, potom $C_i(a) < C_j(b)$.

Následne si ukážeme ako implementovať funkciu hodín C_i do procesu P_i tak, aby vyhovovala základnej *podmienke hodín*. Uvažujme implementáciu hodín v procese ako jednoduchý čítač. Tu si stanovíme prvé implementačné pravidlo pre vyjadrenie hodín v procese:

Implmentačné pravidlo $IR1$: Každý proces P_i inkrementuje hodnotu čítača hodín C_i medzi dvoma za sebou idúcimi udalosťami.

Pravidlo $IR1$ zaručuje podmienku $C1$.
Pre pravidlo splňujúce podmienku $C2$ predpokladáme, že každá správa m obsahuje hodnotu T_m rovnú hodnote čítača hodín odosielajúceho procesu. Hodnota T_m je teda časovou zna-čkou. Po prijatí správy procesom si tento musí inkrementovať svoj čítač hodín, tak aby bol vyšší ako hodnota T_m. Teda pravidlo definujeme takto:

Implmentačné pravidlo $IR2$:

(a) Ak udalosť a je odoslanie správy m procesom P_i, potom správa m obsahuje časovú známku $T_m = C_i(a)$.

(b) Po prijatí správy m procesom P_j tento nastaví hodnotu svojho hodinového čítača C_j na hodnotu vyššiu alebo rovnú ako je jeho aktuálna hodnota a zároveň hodnotu vyššiu ako je hodnota T_m obsiahnutá v správe.

4.1.3 Úplne usporiadanie udalostí

Pre úplne usporiadanie udalostí, tzn. nad celým distribuovaným systémom, môžme použiť systém hodín jednotlivých uzlov definovaný vyššie. Stačí jednoducho usporiadať udalosti podľa časov ich výskytu. Pre tieto potreby definujeme ľubovoľný spôsob usporiadania procesov a definujeme ho relačným operátorom „$\prec$“. Ďalej si definujeme reláciu „$\Longrightarrow$“:

Definícia „$\Longrightarrow$“: Ak a je udalosť v procese P_i a b je udalosť v procese P_j tak, $a \Longrightarrow b$ vtedy a len vtedy, ak buď $C_i(a) < C_j(b)$ alebo $C_i(a) = C_j(b)$ a zároveň $P_i \prec P_j$.

Pre uvedené môžme povedať, že je to *úplne usporiadanie*. Potom pre splnenie podmienky hodín môžme napísať, že ak $a \rightarrow b$ tak $a \Longrightarrow b$. Inými slovami relácia „$\Longrightarrow$“ vedie k usporiadaniu systému z čiastočného usporiadania „happened before“ k úplnemu usporiadaniu udalostí systému.

[1] Preklad z literatúry, ktorého anglický ekvivalent je *Clock Condition*

Úplne usporiadanie pomocou relácie „$\Longrightarrow$“ závisí od implementácie usporiadania jednotlivých procesov a hlavne od samotnej implemnetácie jednotlivých hodín v týchto procesoch. Rôzne logické hodiny, ktoré spĺňajú silnú podmienku hodín, môžu viesť k rôznemu úplnemu usporiadaniu udalostí v systéme.

Úplne usporiadanie udalostí v rámci distribuovaného systému je dôležitým elementom pre synchronizáciu prvkov v takomto systéme. V distribuovaných systémoch je dôležitá takáto synchronizácia v prípadoch, kedy jeden alebo viacero prvkov chcú zmeniť prostredie daného systému alebo chcú upozorniť ostatné prvky na vykonávanie určitej činnosti dôležitej pre budúce rozhodovanie ostatných častí distribuovaného systému.

V nasledujúcej časti si predstavíme pôvodný Lamportov algoritmus pre vzájomné vylúčenie procesov, ktorý ilustruje synchronizáciu nezávislých procesov pri vstupe do kritickej sekcie.

4.1.4 Lamportov algoritmus vzájomného vylúčenia

Uvažujme distribuovaný systém zložený z vopred známeho pevného počtu procesov, ktoré zdieľajú určitý prostriedok tak, že práve jeden z týchto procesov môže využívať daný prostriedok v určitom čase. Procesy sa teda pred využitím tohto prostriedku musia synchronizovať, aby vstupovali do kritickej sekcie(využívali tento prostriedok) po jednom. Algoritmus musí spĺňať nasledujúce tri podmienky:

(I) Proces, ktorému bol povolený vstup do kritickej sekcie, musí uvoľnit zdieľaný prostriedok predtým, ako môže byť využitý iným procesom.

(II) Rôzne požiadavky procesov pre vstup do kritickej sekcie musia byť odbavené v poradí v akom prišli.

(III) Predpokladáme, že každý proces, ktorý vstúpi do kritickej sekcie ju aj uvoľní. Potom predpokladáme, že každý požiadavok pre vstup do kritickej sekcie je niekedy povolený.

Ďalej predpokladáme, že poslané správy nejakými procesmi sú prijaté inými procesmi v rovnakom poradí ako boli odoslané a zároveň každá odoslaná správa bude prijatá. Dôležitý je mechanizmus komunikácie v takomto systéme, preto musíme zaručiť, že každý proces systému môže poslať správu všetkým ostatným procesom v tomto systéme.

Taktiež súčasťou každého procesu je dátova štruktúra s názvom *fronta požiadaviek*, ktorá je viditeľná len procesu, ktorý ju spravuje. Táto fronta obsahuje záznamy v tvare $T_j : P_j$, kde T_j je hodnota hodín procesu P_j. Na začiatku predpokladáme, že fronta je prázdna.

Algoritmus

Algoritmus je definovaný nasledujúcimi piatimi pravidlami(každé pravidlo formuje nejakú elementárnu udalosť):

1. Pre požiadanie o pridelenie vstupu do kritickej sekcie proces P_i pošle správu s požiadavkou v tvare $T_m : P_i$ každému procesu v distribuovanom systéme a uloží túto správu do svojej fronty požiadavkou, kde T_m je aktuálna hodnota logických hodín tohto procesu.

2. Keď proces P_j prijme správu $T_m : P_i$ s požiadavkou o vstup do kritickej sekcie, uloží ju do svojej fronty požiadaviek a pošle potvrdzujúcu správu(obsahujúcu časovú známku) procesu P_i.

3. Pri odchode z kritickej sekcie proces P_i odstráni zo svojej fronty požiadaviek všetky správy tvaru $T_m : P_i$ a pošle správu s časovou známkou a s príznakom opustenia kritickej sekcie ostatným procesom v systéme.

4. Keď proces P_j dostane správu od procesu P_i s príznakom opustenia kritickej sekcie, tak odstráni zo svojej fronty všetky požiadavky procesu P_i v tvare $T_m : P_i$.

5. Proces P_i má povolený vstup do kritickej sekcie, keď sú splnené nasledujúce podmienky:

   (a) Ak je v jeho fronte požiadaviek správa $T_m : P_i$, ktorá je usporiadaná pred akoukoľvek inou správou so žiadosťou o vstup do kritickej sekcie reláciou „$\Longrightarrow$“.

   (b) Proces P_i dostal správu s požiadavkou o pridelenie kritickej sekcie od všetkých ostatných procesov s časovou známkou vyššou ako je časová známka správy T_m procesu P_i.

Pozn.: Podmienky 5(a) a 5(b) sú testované lokálne každým jedným procesom v distribuovanom systéme.

Tento algoritmus je distribuovaný. Každý proces sa nezávisle správa podľa daných pravidiel a systém nemá žiadny centrálny synchronizačný prvok. Preto tento prístup môžme využit pri ľubovoľnej synchronizácií v takomto distribuovanom systéme. Príklad takéhoto algoritmu je zobrazený na tomto obrázku 4.3.

Stavový automat

Algoritmus opísaný podmienkami vyššie sa dá popísať stavovým automatom. Predpokladajme množinu C možných príkazov v algoritme a množinu S ako množinu stavov algoritmu. Ďalej pre prechod medzi stavmi automatu s definujeme funkciu e takto:

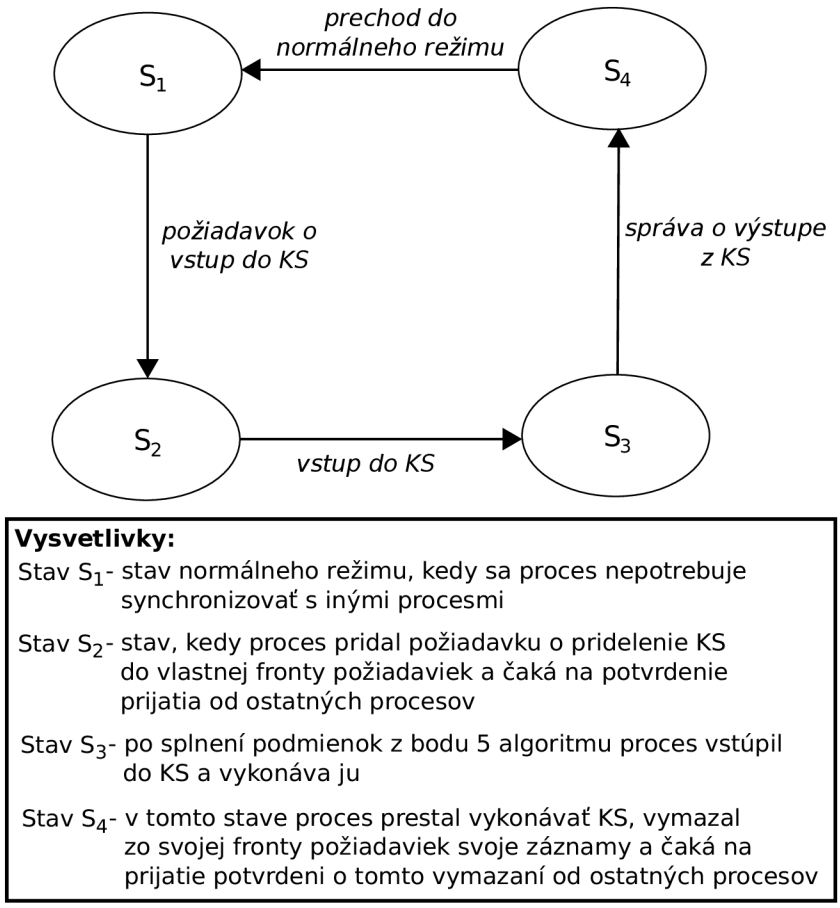
$$e : C \times S \rightarrow S$$

Táto relácia popisuje prácu stavového automatu. Inak povedané, prevedenie príkazu z množiny C v aktuálnom stave z množiny S zmení aktuálny stav automatu do stavu taktiež z množiny S. Práca každého procesu v danom systéme sa dá opísať takýmto stavovým strojom. Na nasledujúcom obrázku znázornime takúto činnosť procesu pri vstupe do kritickej sekcie 4.2.

Uvedený príklad je len ilustratívny, pre jednoduchosť tam nie sú všetky hrany vedúce medzi prostredím(ostatnými procesmi) a procesom. Taktiež tam nie sú zobrazené spätné hrany pri nedoručení potvrdení o prijatí správ. Obrázok zobrazuje vstup do kritickej sekcie jedného procesu.

Nežiadúce správanie

Pôvodny Lamportov algoritmus taký, aký sme ho predstavili vyššie, môže viesť k situáciam, ktoré sú v reálnom distribuovanom systéme nežiadúce. Predstavme si situáciu, kedy proces

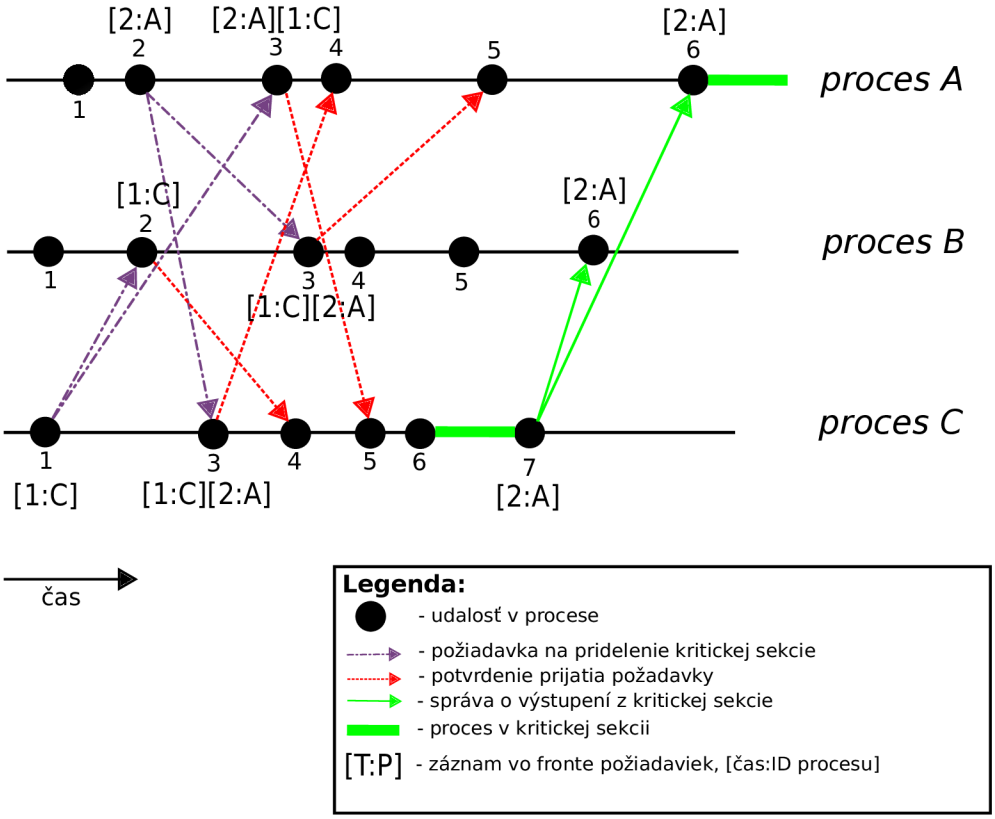


Obrázok 4.2: Jednoduchý príklad stavového automatu pre synchronizáciu procesov

A sa chce synchronizovať s procesom B. To isté chce v neskoršom čase urobiť proces B. Tu môže nastať situácia, kedy má časová známka požiadavky procesu B nižšiu hodnotu ako časová známka požiadavky procesu A, čo nevyhovuje podmienkam stanoveným v algoritme pre vzájomné vylúčenie. V predstavenom algoritme nie je žiadna možnosť, ako zistiť, že požiadavka procesu A bola pred požiadavkou procesu B. Táto situácia je v rozpore k definícií algoritmu, kedy prvý vstupuje do kritickej sekcie proces, ktorý požiadal o jej pridelenie ako prvý.

Táto situácia môže byť spôsobená rôznymi reálnymi situáciami v distribuovanom systéme. Napríklad veľkým množstvom prvkov v systéme, čo môže spôsobiť hustú komunikáciu medzi prvkami tohto systému(proces môže dostať viac požiadaviek ako iný, všetky požiadavky sa ešte nedostali ku všetkým procesom), čo môže spôsobiť inkonzistenciu.

Riešením tu môže byť zosilnenie podmienky hodín na všetky udalosti systému, čo značí, že usporiadnie udalostí bude globálne nad celým distribuovaným systémom. To je však v reálnom svete ťažko implementovateľné. Ďalším spôsobom je predstavenie funkcie fyzických hodín v distribuovanom systéme. Tento spôsob však naráža na problémy s nimi spojené, ako sú ich samotná synchronizácia a nepresný chod hodín medzi procesmi.



Obrázok 4.3: Demonštrácia synchronizácie pred vstupom do kritickej sekcie

Uvedený algoritmus má časovú zložitosť $O(3(N-1))$ na jednu požiadavku o pridelenie kritickej sekcie. Kde N je počet prvkov v distribuovanom systéme. Viac informácií o uvedenom algoritme nájedete v tejto literatúre [4], odkiaľ bolo čerpané pri písaní tejto kapitoly, informácie o tejto problematike môžte nájsť takisto tu [8] a tu [9].

4.2 Vektorové hodiny

V tejto sekcií si okrajovo predstavíme algoritmus, ktorý je modifikáciou Lamportovho algoritmu. Algoritmus vektorových hodín funguje taktiež nad už predstavenou reláciou „happened before“. Rozdiel je v tom, že miesto odoslania len aktuálneho stavu logických hodín procesu, tento odošle aj logický čas udalostí, na ktorých kauzálne závisí. To znamená, že v každej správe odosiela nielen aktuálny stav lokálnych hodín, ale aj hodín procesov, od ktorých už prijal nejakú správu.

Takýto prístup sa nazýva kauzálne doručovanie. Zvyšovanie logických hodín pri prijatí

správy je rovnaké ako v Lamportovom algoritme. Výhodou tohto algoritmu je, že pomerne spoľahlivo zaručuje úplne usporiadanie udalostí v systéme. Naviac tento algoritmus dokáže spoľahlivo určiť, ktoré udalosti na sebe závisia a ktoré nie. Viac sa o princípe fungovania vektorových hodín môžme dozvedieť v tejto [3] alebo tejto [10] literatúre.

4.3 Zhrnutie

V tejto časti sme si predstavili matematický základ pre synchronizáciu v distribuovaných systémoch pomocou logických hodín, predstavili sme si detailne Lamportov algoritmus a poukázali na jeho základné princípy, ale takisto sme si povedali aj o chybách tohto algoritmu. Na konci práce sme si okrajovo predstavili algoritmus vektorových hodín.

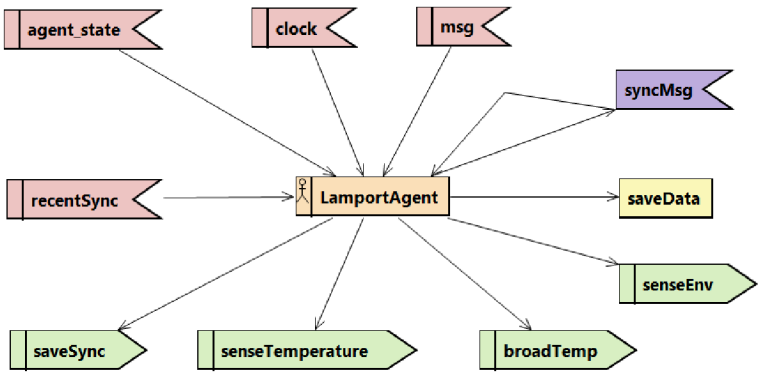
V závere môžme tiež poukázať na jednu z nevýhod v algoritme vektorových hodín, čo je nárast vo veľkosti správ medzi prvkami systému, čo môže mať nechcený efekt hlavne vo WSN systémoch, ktoré sú energeticky veľmi citlivé. Naproti tomu je viac výhodnejší, ak je nutná silná konzistencia logických hodín systému, ako Lamportov algoritmus.

Na záver je nutné povedať, že algoritmus pre synchronizáciu v distribuovaných systémoch závisí od počiatočnej špecifikácie tohto systému, kde treba brať ohľad na komunikačnú sieť tohto systému, požiadavkach na pamäť, či energetickú náročnosť jednotlivých prvkov systému.

Kapitola 5

Návrh aplikácie

V tejto časti sa budeme venovať jadru aplikácie pre agentov multi-agentného systému simulújuceho správanie bezdrátových senzorov pri ich synchronizácii. Použitý bol vyššie spomínaný Lamportov algoritmus. Návrh agenta vychádza z konceptu metodológie Prometheus, o ktorej sa môžme dočítať viac v tejto literatúre [6]. V nasledújucom texte si predstavíme základné prvky, ktoré sa podieľajú pri rozhodovaní v činnosti agenta. Návrh je zobrazený nasledujúcim obrázkom 5.1.



Obrázok 5.1: Návrh práce agenta podľa metodológie Prometheus

5.1 Prvky návrhu

V tejto podkapitole si predstavíme jednotlivé funkčné prvky návrhu činnosti agenta tak, ako sú zobrazené na na obrázku 5.1.

Červenými opačnými šípkami sú zobrazené predstavy agenta alebo informácie, ktoré agent získal svojím vnímaním z okolia. Zelené šípky predstavujú ciele agenta, ktoré sa snaží agent dosiahnúť. Modrou opačnou šípkou je zobrazená správa, ktorou agent komunikuje s ostatnými agentmi v prostredí. Žltý obdĺžnik predstavuje uloženie dát agentom.

Následne si predstavíme jednotlivé funkčné bloky konkrétnejšie. Tu je treba poznamenať, že agenti pracujú ako senzory vo WSN, teda tomu zodpovedá aj spôsob ich práce.

5.1.1 Získavané znalosti agenta

msg

Túto znalosť získava agent zo svojho rádia, ktorým komunikuje s ostatnými senzormi v prostredí, v ktorom sa nachádzajú. Znalosť sa pridáva do množiny znalostí agenta a spustí sa prípadná reakcia na prijatie správy agentom. Samozrejme agent musí byť v dosahu rádiového vysielača agenta, ktorý správu vysiela.

clock

Znalosť **clock** je udžiavaná agentom a predstavuje aktuálnu hodnotu logických hodín agenta. Táto znalosť je udržiavaná priamo agentom. V popise implementácie bude popísané akým spôsobom agent túto informáciu udržiava.

agentState

Informácia udržiavaná pomocou tejto znalosti rozdeľuje činnosť agenta na tri stavy. Tieto tri stavy môžme zhrnúť nasledovne:

- Prvým z týchto stavov je stav, kedy agent sníma veličinu pomocou senzorov z okolia a podľa hodnoty sa rozhoduje, či bude svoj logický čas synchronizovať s ostatnými agentmi v systéme.
- Ďalším stavom je stav, do ktorého sa prechádza z predchádzajúceho stavu a to v prípade, že meraná veličina dosiahla určitú hraničnú hodnotu a vyšle pomocou svojho rádia synchronizačnú správu ostatným agentom.
- Posledným zo stavov je stav, kedy agent prijíma synchronizačnú správu a vyvoláva reakciu na túto udalosť podľa obsahu prijatej správy.

recentSync

Touto znalosťou si agent udržuje informáciu o pôvodcovi synchronizácie a hodnote jeho logických hodín. Táto informácia slúži pre potreby kontroly synchronizácie medzi agentmi. S detailnejším popisom spôsobu práce s touto znalosťou sa zoznámime v časti 6.1.6.

5.1.2 Ciele agenta

Tu sa budeme zaoberať konkrétnymi cieľmi, ktoré sa agent počas svojej činnosti snaží dosiahnúť na základe získaných znalostí.

senseEnv

Tento cieľ slúži ako životný cyklus agenta. Stará sa o periodické snímanie veličín z prostredia pomocou senzorov a následne vykonáva činnosti na základe získanej hodnoty meranej veličiny.

senseTemperature

Cieľ len zaobaľuje snímanie veličiny z prostredia. Reakciou na jeho spustenie je volanie internej akcie agenta a práca s výsledkom tejto akcie.

broadTemp

V reakcii na spustenie tohto cieľa sa agent rozhoduje na základe nameranej veličiny, či spustí synchronizáciu s ostatnými agentmi. Ak sa tak rozhodne, tak podnikne akcie potrebné pre zaiatok samotnej synchronizácie v agentnom systéme.

saveSync

Cieľ slúži len pre ukladanie informácií o synchronizácií v multi-agentnom systéme. Táto informácia by mala byť v ideálnom prípade rovnako usporiadaná v každom agentovi. Takto získané informácie slúžia na vyhodnotenie úspešnosti synchronizácie v celom systéme.

5.1.3 Ostatné prvky návrhu

saveData

Tento prvok predstavuje dátovy sklad agenta, tzn. že agent má prostriedky na to, aby v určitej forme ukladal informácie získané zo svojej činnosti. Je evidentné, že tento sklad informácií je obhospodarovaný v reakcií na cieľ **saveSync**.

syncMsg

Prvok je len abstraktný, predstavuje komunikáciu medzi agentami. V našom prípade neprebieha komunikácia medzi agentmi pomocou vopred pripravených protokolov. Prijímanie správ agentmi prebieha v podstate na základe vnímania okolia, kedy rádioprijímač príjme správu len od agenta, v ktorého rádiovom dosahu sa nachádza.

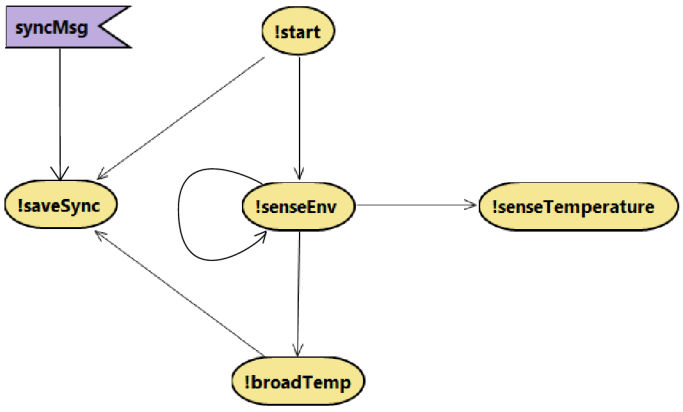
5.1.4 Vzťahy medzi činnosťami agenta

V tejto časti si ukážeme aké sú vzťahy medzi činnosťami agenta. Činnosť agenta je ovplyvnená zmenou v množine predstáv alebo spustením cieľa agenta na základe vykonávania určitých externých akcií agenta. Diagram 5.2 zobrazuje vzťahy medzi jednotlivými cieľmi zobrazenými na obrázku 5.1, kde sú len jednotlivé činnosti vymenované a zaradené do kategorií, podľa spôsobu, akým agent s týmito prvkami pracuje.

Cieľ **!start** na uvedenom obrázku je inicializačným cieľom agenta. Agent začína práve týmto cieľom, kde inicializuje svoje synchronizačné údaje a pokračuje cieľom **!senseEnv**. Ako už bolo povedané vyššie, tento cieľ predstavuje životný cyklus agenta a z obrázka vyplýva, že sa vola rekurzívne. Z tohoto cieľa sa meria náhodná veličina (v našom prípade teplota) a následne sa za určitých splnených podmienok synchronizuje s ostatnými agentmi pomocou cieľa **!broadTemp**. O meranie teploty sa postará cieľ **!senseTemperature**. V prípade synchronizácie s ostatnými agentmi sa volá z cieľa **!broadTemp** taktiež cieľ **!saveSync**. V uvedenom obrázku je tiež zobrazené použitie cieľu **!saveSync** v prípade, že spĺňuje Lamportové podmienky pre synchronizáciu logických hodín, a že táto synchronizácia je podmienená iným agentom, teda prijatím jeho synchronizačnej správy.

5.2 Zhrnutie

V tejto kapitole sme si predstavili návrh samotnej aplikácie. Na začiatku sme si predstavili základné stavebné prvky, ktoré riadia činnosť agenta, ako aj jeho interakciu s okolím. Ďalej



Obrázok 5.2: Zobrazenie vzťahov medzi jednotlivými cieľmi

boli spomenuté vzťahy medzi jednotlivými cieľmi agenta a bola nám tak priblížená jeho činnosť. Pre návrh bol použitý softvér PDT[1], ktorý využíva špecifikáciu už vyššie spomínanej metodológie Prometheus.

[1]Prometheus Design Tool

Kapitola 6

Implementácia

Táto kapitola sa zaoberá implementáciou prvkov popísaných v návrhu aplikácie. Ďalej tu budú popísané zmeny prevedené v prostredí SAMSON, do ktorého sú posadení agenti a nakoniec budú stručne predstavené vlastné funkcie pre platformu Jason.

Implementácia samotného agenta je rozdelená do dvoch súborov. Prvým z nich je `lamport_agent.asl`, kde je implementovaná logika merania teploty, prijatia synchronizačnej správy samotná synchronizácia logických hodín agenta.

Druhým súborom je `lamport_goals.asl`, kde sú implementované pomocné ciele pre všesmerové vysielanie synchronizačnej správy, meranie teploty prostredia a ukladanie synchronizačných dát. Kód v týchto súboroch je v prílohe A.

6.1 Časti implementácie

Najdôležitejšiou časťou aplikácie sú ciele agenta implementované v agentnom jazyku AgentSpeak. Tento jazyk výhodne riadi činnosť agenta pomocou svojej syntaxe logického programovacieho jazyka. Nedieľnou súčasťou sú však aj získané znalosti z prostredia a od ostatných agentov. V jednotlivých prvkoch si vymenujeme a vysvetlíme externé akcie agenta a dôležité funkcie platformy Jason. Pridanie znalostí a a vytvorenie cieľov slúžia ako spúšťací mechanizmus pre vykonávanie činnosti agenta. Začneme s cieľom **start**.

6.1.1 start

1. Po vytvorení agenta sa vygeneruje náhodná hodnota, ktorá bude použitá ako perióda merania náhodnej veličiny agentom.
2. Ďalej sa táto hodnota zaokrúhli a vypíše sa na agentnú konzolu informácia o štarte agenta a o jeho perióde snímania náhodnej veličiny.
3. V treťom kroku sa inicializuje hodnota logických hodín agenta.
4. Následne sa nastaví hardvér senzoru a to nasledovne:
   (a) Nastaví sa stav mikrokontroléru na stav zapnutý.
   (b) Stav rádia bude v stave zapnutý pre príjimanie.
   (c) Nastaví sa sila vysielania rádia senzora.

5. Do množiny znalostí agenta sa pridá začiatočná hodnota synchronizácie a potom sa uloží do súboru pomocou cieľa **saveSync**.

6. Nakoniec sa pridá cieľ **senseEnv**, ktorého parametrom je perióda snímania veličiny z prostredia.

6.1.2 senseEnv

Telo tohto cieľa sa na základe kontextu vytvorenia cieľa spustí len v prípade ak sa agent nesynchronizuje s ostatnými agentmi, alebo nie je synchronizovaný iným agentom.

1. V prvom kroku si agent pridá na začiatok zoznamu cieľov **senseTemperature**, ktorá zistí teplotu v prostredí, v ktorom sa agent nachádza.

2. Ďalším krokom je zistenie hodnoty logických hodín z množiny znalostí agenta, nasleduje vymazanie všetkých výskytov tejto znalosti z tejto množiny a nastavenie znalosti s hodnotou inkrementovanou o jednotku. Zmeranie teploty predstavuje interná udalosť agenta, preto nastáva inkrementácia logických hodín podľa špecifikácie Lamportovho algoritmu.

3. V tomto kroku sa pridá na začiatok zoznamu cieľov agenta **broadTemp**, ktorý sa na základe nameranej teploty rozhodne, či synchronizuje svoje logické hodiny s ostatnými agentmi.

4. Posledným krokom je rekurzívne spustenie cieľa **senseEnv** s prerušením spustenia na dobu určenú pri spustení agenta.

6.1.3 broadTemp

V kontexte spustenia tohto cieľa je zistenie teploty zo znalosti pridanej po zmerani teploty prostredia. Následne sa táto teplota porovná s nami zvolenou hraničnou teplotou. Ak je väčšia, tak sa spustí telo tohto cieľa. Jediným parametrom tohto cieľa je hodnota logických hodín, ktorá bude prípadne použitá pre synchronizáciu. Telo cieľa prejde nasledovnými krokmi:

1. Nastaví sa hodnota stavu agenta na stav, ktorý indikuje synchronizáciu s ostatnými agentmi systému.

2. Následne sa uložia synchronizačné údaje do súboru.

3. Potom sa prepne rádio zo stavu spustené len pre prijímanie na stav spustené pre prijímanie aj vysielanie.

4. Tu agent vystaví na svoje rádio synchronizačnú správu, ktorú dostanú len agenti v dosahu rádia. Vysiela sa pomocou externej akcie agenta s názvom **sendTX**. Táto akcia má tieto tri parametre:

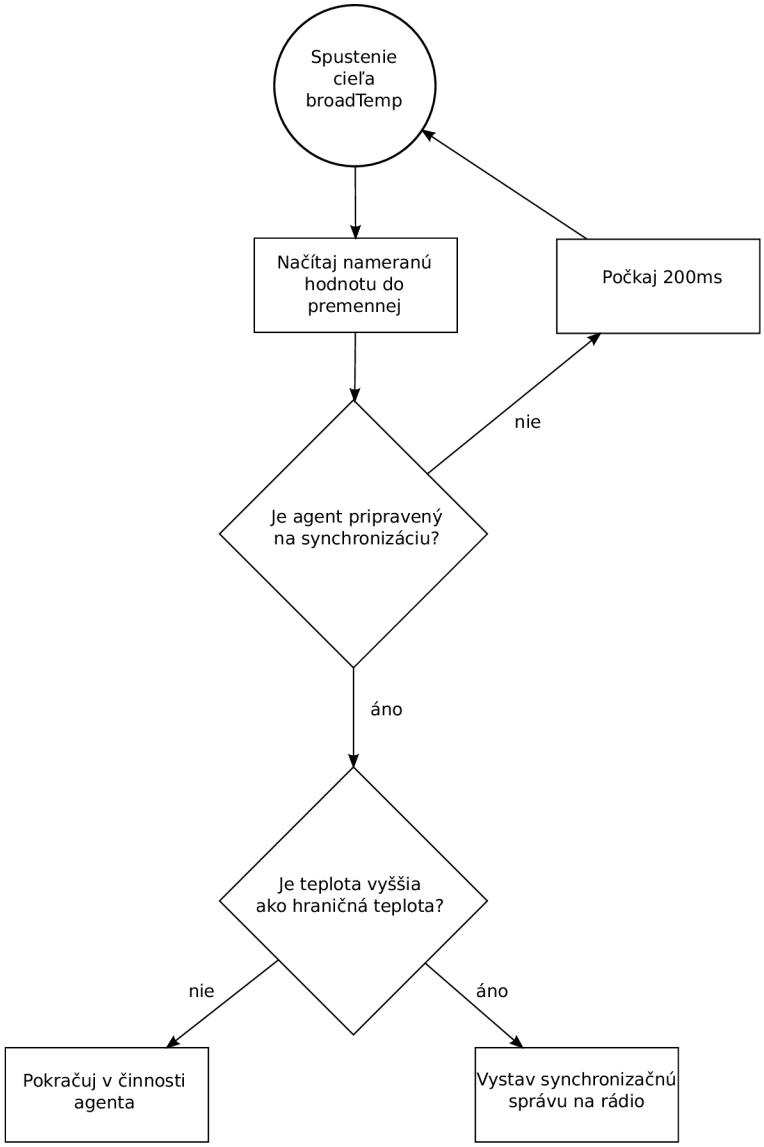
   (a) Jedinečné meno agenta.

   (b) Hodnotu logických hodín získanú ako parameter tohto cieľa.

5. Rádio sa opäť prepne do stavu len pre prijímanie.

6. Na konci sa nastaví stav agenta na základný stav, kedy sa nesynchronizuje s ostatnými agentmi, a ani nie je jedným z týchto agentov synchronizovaný.

Práca a tohto cieľa je zobrazená na nasledujúcom obrázku 6.1.



Obrázok 6.1: Spracovanie nameranej hodnoty agentom

6.1.4 senseTemperature

Ako už bolo povedané vyššie tento cieľ iba zaobaľuje snímanie veličiny senzormi.

1. V prvom kroku sa z množiny znalostí agenta vymažú všetky výskyty **sensorVal**. Táto znalosť je vždy pridávaná pomocou externej akcie, takže sa v jazyku AgentSpeak nedá využiť zmena znalosti pomocou predpony $-+$ pred cieľom. Preto sa toto vykonáva pred každým novým meraním.

2. Tu sa volá už spomenutá externá funkcia agenta **sense**, ktorej parametrom je identifikátor senzora podľa typu hodnoty, ktorú ideme merať. Podľa identifikátora môžme merať svietivosť, vlhkosť a teplotu prostredia. Špecialným prípadom je meranie napätia v batériach senzora. V našom prípade však používame len meranie teploty prostredia.

3. Po zmeraní teploty sa vypíše jej hodnota na agentnú konzolu.

6.1.5 saveSync

Tento cieľ sa stará o dátové úložište agenta. Vstupom tohto cieľa je pôvodca synchronizácie medzi agentmi a hodnota jeho logických hodín.

1. Na začiatku sa identifikátor pôvodcu synchronizácie pomocou funkcií platformy Jason skonkatenuje s hodnotou logických hodín.

2. Následne sa pomocou už spomínanej internej akcie agenta **saveData** uloží synchronizácia tohto agenta v tvare:
`<Identifikátor pôvodcu>,<Hodnota logických hodín pôvodcu>` Tieto dáta sa ukladajú do súboru s názvom tvaru `[Identifikátor agenta].dat` do zložky Data koreňového adresára projektu.

6.1.6 msg

V tejto sekcií si predstavíme reakciu na prijatie správy agentom. Vyššie už bolo povedané, že prijatie správy sa prejaví ako pridanie znalosti agentovi. Toto je spúšťacia udalosť, na ktorú musí agent nejako reagovať. Táto znalosť má tri parametre. Prvým je pôvodca správy, tzn. identifikátor agenta, ktorý chce svoju hodnotu logických hodín synchronizovať s ostatnými agentmi. Druhým parametrom je identifikátor agenta, ktorý preposlal synchronizačnú správu. Nemusí to byť nutne len pôvodca synchronizačnej správy. Posledným paremetrom je hodnota logických hodín, podľa ktorých sa má agent synchronizovať.

Kontextom pridania tejto znalosti je kontrola na obsah správy, tj. či obsahom je číslo, ďalej sa kontroluje hodnota hodín poslaná v správe. Podľa tejto kontroly sa telo tejto udalosti rozdeľuje na dve vetvy. O tomto rozdelení budeme diskutovať nižšie. Ďalšou podstatnou podmienkou v kontexte je kontrola, či už agent bol synchronizovaný dvojicou údajov obsiahnutých v prijatej správe. Táto kontrola prebieha pomocou prítomnosti znalosti **recentSync** v množine znalostí agenta. Hodnotami tejto znalosti sú pôvodca synchronizačnej správy a hodnota jeho logických hodín. Ak sa táto informácia už nachádza v množine znalostí, tak sa neprevedie telo udalosti.

Poslednou kontrolou v kontexte je kontrola na stav agenta. Telo tejto udalosti (pridanie znalosti **msg**) je spustené iba v prípade, ak je agent v stave merania veličiny z prostredia, čiže sa nesynchronizuje, ani nie je synchronizovaný. Ak je táto kontrola neúspešná, tak agent počká určitý čas a snaží sa znovu o synchronizáciu pomocou tej istej správy.

Telo tejto udalosti prechádza nasledujúcimi krokmi:

1. V prvom kroku sa nastaví stav agenta na stav, kedy je agent synchronizovaný iným agentom.

2. Tu sa líšia jednotlivé telá spustených udalostí. V prípade, že hodnota hodín prijatá v správe je menšia alebo rovná ako hodnota hodín agenta, tak sa len inkrementuje

hodnota hodín o jednotku. Ak je hodnota väčšia, tak sa táto nastaví ako logický čas agenta.

3. Následne sa táto synchronizácia uloží pomocou cieľu **saveSync**.

4. Ďalej sa nastaví stav rádia agenta na stav pre vysielanie aj prijímanie.

5. Správa sa vystaví na rádio agenta, zmení sa však odosielateľ správy, pôvodca ostáva. Toto zaisťuje rozosielanie ku každému dostupnému agentovi v multi-agentnom systéme.

6. Potom sa rádio opäť prepne do stavu len pre pijímanie a na konzolu sa vytlačí správa o synchronizácií s informáciou o agentovi, od ktorého správa prišla, o pôvodcovi synchronizačnej správy a hodnote jeho logických hodín. Na to sa vymažú z množiny znalostí všetky záznamy **msg**.

7. Na konci agent prejde do pôvodného stavu, ktorý indikuje meranie veličiny z prostredia.

Celú situáciu okolo prijímania synchronizačných správ môžme vidieť na obrázku 6.2.

6.2 Úprava prostredia

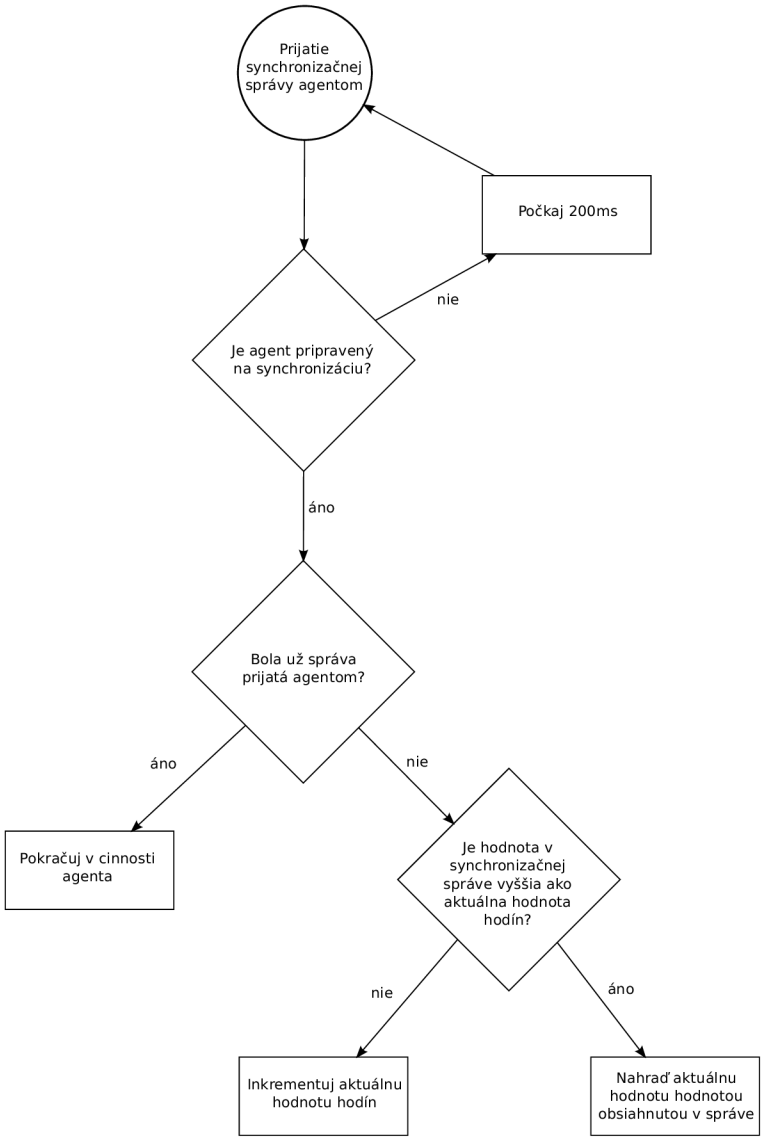
V tejto časti opíšeme úpravy prevedené do prostredia SAMSON. Prostredie muselo byť modifikované, pretože nevyhovovalo všetkým požiadavkám tejto práce.

Prvou úpravou je doplnenie časovača do triedy *NetEnvModel*, ktorý sa stará o zmenu náhodných veličín. V pôvodnej implementácií sa nastavili hodnoty všetkých veličín pri spustení aplikácie a ostali nezmenené počas celého behu programu. Časovač sa spúšťa každých päť sekúnd počas celého behu programu a nastavuje náhodne veličiny vo všetkých dielikoch pracovného prostredia.

Ďalšou úpravou bolo doplnenie externých akcií agenta, tzn. takých, ktoré vykonajú nejakú činnosť agenta vedúcu k modifikácií stavu prostredia, zmene správania agenta, zmene jeho stavov, atď. Tieto akcie sa dajú volať zo zdrojových kódov napísaných v jazyku AgentSpeak. V prostredí SAMSON boli doimplementované reakcie pre tieto akcie. Rozlíšenie týchto akcií je naimplementované v triede *NetEnviroment*, ktorá slúži pre komunikáciu prostredia s platformou Jason. Vstupom vykonávania týchto interných funkcií je metóda *executeAction*. Vyššie menované funkcie sú nasledovné:

sendTX Táto externá akcia bola už naimplementovaná, len bola prispôsobená na iný typ problému. Nová implemnetácia počíta s dvoma parametrami, ktorými sú identifikácia pôvodcu správy a obsah správy. Stará implementácia počítala so štyrmi parametrami a bola zachovaná na všetkých vrstvách implementácie prostredia. Efektom tejto akcie je aj vytvorenie znalosti v množine znalostí agenta, ktorý správu poslanú pomocou akcie **sendTX** príjme. Touto znalosťou je **msg** a bola popísaná vyššie v tejto práci.

setClock Menovaná interná akcia nastavuje hodnotu logických hodín agenta. Hodnota je udržiavaná v triede *SensorNode*. Získava sa pomocou znalosti **clock**. Táto znalosť je pridávaná pri každom volaní metódy *updatePercepts*. V tejto metóde sú pridávané všetky znalosti vytvorené činnosťou agenta v prostredí.



Obrázok 6.2: Spôsob spracovania prijatej synchronizačnej správy agentom

Azda poslednou významnou zmenou prostredí je zmena vo vykresľovaní prostredia. V pôvodnej implementácií sa pri agentovi modrou farbou vykresľuje hodnota vzdialenosti k bodu presmerovania správ v multi-agentnom systéme. V našej implementácií sa modrou farbou nad agentom zobrazuje aktuálna hodnota logických hodín agenta. Táto zmena bola prevedená v triede *NetView*.

6.3 Doplňky do jazyka JASON

Tu si predstavíme interné akcie agenta, ktoré sú užívateľským rozšírením platformy Jason. Tieto akcie sa implementujú v jazyku Java a vyvolávajú sa zo zdrojových kódov agenta napísaných v jazyku AgentSpeak. Týmto sa môže ľahko rozšíriť sada funkcií poskytnutých platformou Jason.

Doprogramované akcie boli rozdelené do dvoch Java balíkov, podľa funkčnosti a to nasledujúco:

Balík debug_package - Tento balík slúžil len na účely vývoja aplikácie agenta a nie je nevyhnutný pre samotnú funkčnosť aplikácie. Je v ňom implementovaná len nasledujúca akcia:

list_bels Táto akcia vypíše do agentnej konzoly aktuálny obsah množiny znalostí agenta, v ktorom bola spustená. Akcia je bez parametrov.

Balík myp - V tomto balíku sa nachádzajú interné akcie, ktoré boli využité pri implementácií riadenia agenta. Rozšírujú množinu štandardných interných akcií platformy Jason.

round Akcia len zaokrúhli reálne číslo na vstupe na celé číslo. Parametrami tejto akcie, sú term, ktorý sa má zaokrúhliť a term, do ktorého sa uloží zaokrúhlená hodnota.

toString Funkciou tejto akcie je previesť hodnotu termu na reťazec znakov. Tiež táto akcia ma dva vstupné parametre. Prvým je už spomínaný term a druhý je term, kam sa výsledok práce akcie uloží.

Ako už bolo spomenuté, akcie sa spúšťajú priamo z kódu agenta napísanom v jazyku AgentSpeak. Ich volanie má tvar *balík.akcia(term1, term2, ...)* (napr. myp.toString(AgentName, Name)).

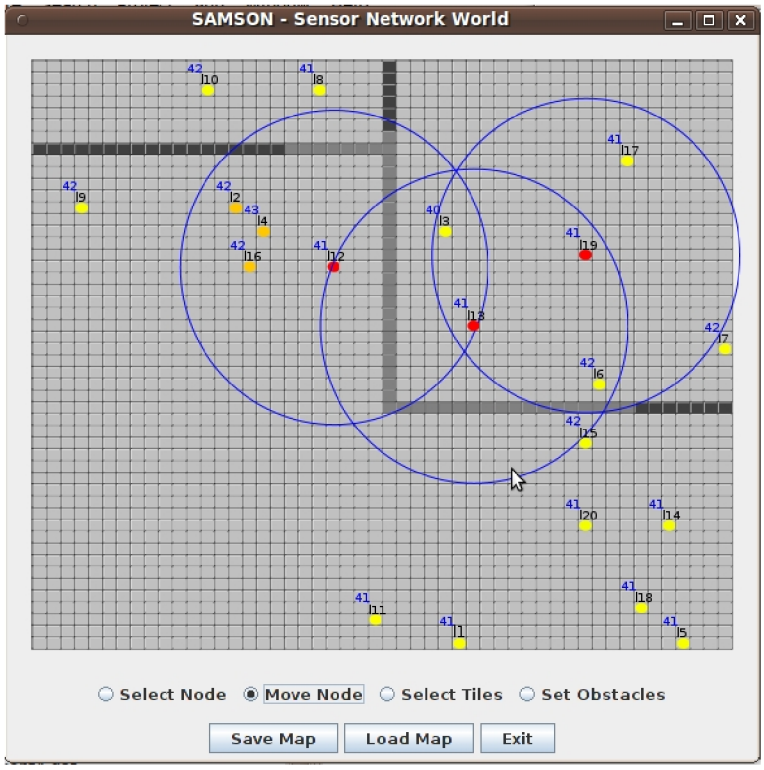
6.4 Výstupy aplikácie

Táto časť nám bude slúžiť pre oboznámenie sa so simulačným prostredím a jeho jednotlivými časťami. Začneme oknom aplikácie SAMSON, kde je zobrazená simulácia synchronizácie agentov a postupne sa zoznámime s jednotlivými jeho časťami.

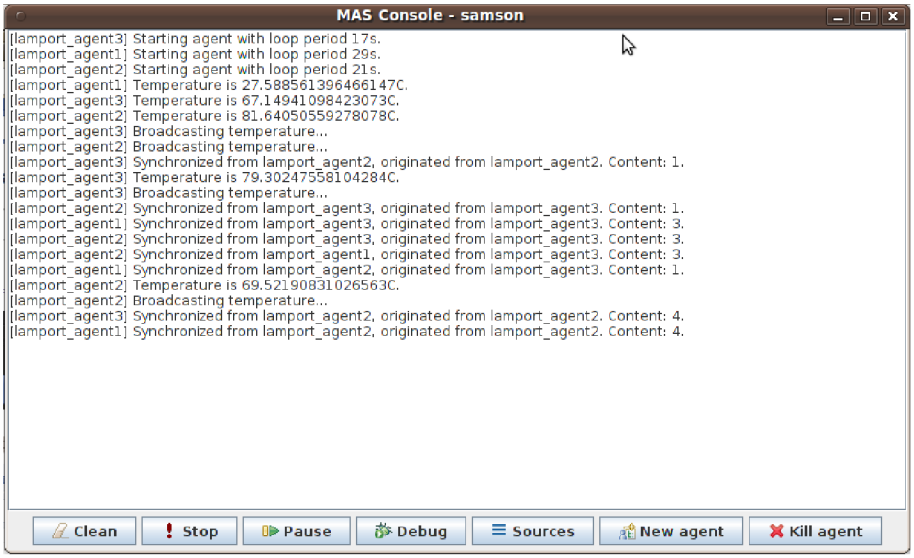
Na obrázku 6.3 môžme vidieť simulačné prostredie, kde strana každého dieliku mriežky zodpovedá asi metru v reálnom prostredí. Žlté bodky v prostredí predstavujú pozíciu agentov. Keď je agent sfarbený do červena tak vysiela správu svojmu okoliu. Dosah tejto správy je naznačený modrou kružnicou. Ak sa agent sfarbil do oranžova aktuálne prijíma správu od iného agenta. Na čierno sfarbený agent má slabé baterky a preto je neaktívny a ďalej sa s ostatnými nesynchronizuje. Modré čísla v ľavom hornom rohu od značky agenta znamenajú aktuálnu hodnotu jeho logických hodín. Tmavošedou farbou sú označené prekážky v šírení komunikačnej vlny. V týchto oblastiach má komunikácia medzi agentmi väčšie oneskorenie ako pri normálnych dieloch prostredia, vďaka nasteveným odlišným fyzikálnym vlastnostiam prekážky. Bledšou šedou farbou je zorbrazené aktuálne šírenie komunikčnej vlny cez takúto prekážku. O spôsobe používania prostredia SAMSON sa môžme dočítať tu [5].

Ďalším výstupom, ktorý aplikácia poskytuje sú výpisy agentov na konzolu. Tu sa vypisuje aktuálna činnosť, ktorá prebieha na agentovi. Príklad takéhoto výstupu je zobrazený tu 6.4.

Na uvedenom obrázku vidíme všetky možné výstupy agentov. Takto môžme sledovať prebieh činnosti agenta. Obrázok zobrazuje začiatok činnosti agentov, výber intervalu pre meranie teploty, samotné meranie teploty, indikácia všesmerového vysielania ostatným agentom a vypisuje sa takisto informácia o prijatej synchronizačnej správe.



Obrázok 6.3: Okno prostredia agentov



Obrázok 6.4: Agentná konzola

6.5 Zhrnutie

Na záver by bolo vhodné spomenúť, že pre implementáciu bolo použité prostredie Eclipse. V kapitole popisujúcej implementáciu sme si na začiatku predstavili obsah cieľov riadiacich

životný cyklus agenta. Podrobne sme si popísali jednotlivé kroky prevedené v tele spustenia týchto cieľov ako aj podmmienky, za ktorých sú ciele spustené. V ďalšej časti boli popísané úpravy v prostredí SAMSON a doplnené externé akcie. Následne sme v stručnosti predstavili interné akcie doimplementované do platformy Jason. Nakoniec sme sa oboznámili s výstupmi aplikácie, ktoré budú použité v nasledujúcej kapitole pre vyhodnotenie použitého riešenia implementácie Lamportovho algoritmu pre problém synchronizácie v distribuovanom systéme.

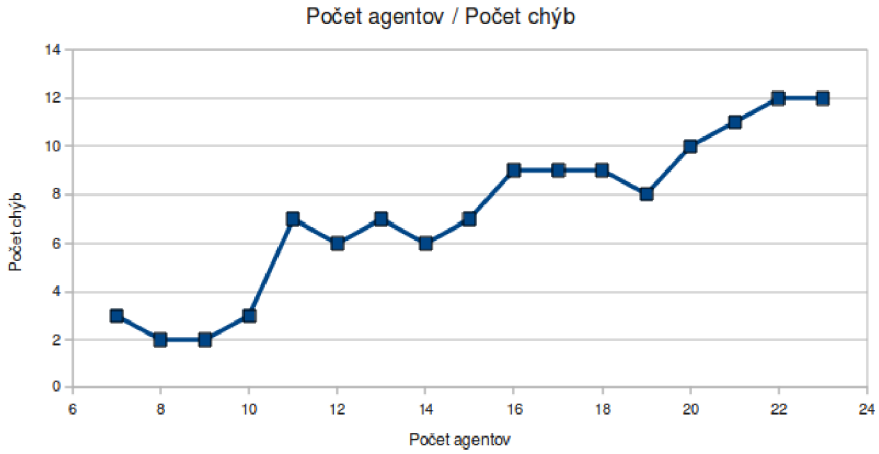
Kapitola 7

Testovacie scenáre

Tu sa budeme hlavne venovať skúmaniu použitého riešenia na chyby v synchronizácií. Pod týmto pojmom rozumieme situáciu, kedy je v distribuovanom systéme nemožné určiť, presné poradie udalostí. Táto situácia nastáva v prípade, ak dva alebo viac prvkov tohto sytému sa rozhodne synchronizovať s ostatnými vtedy, keď jedna alebo viac synchronizácií na týchto prvkoch ešte neprebehla, ale prebehla v ostatných uzloch systému. Inak povedané nemožno zistiť, ktorá udalosť bola skôr a ktorá neskôr, pretože synchronizácia nestihla prebehnúť medzi všetkými agentmi distribuovaného systému.

7.1 Vplyv počtu agentov na chybu v synchronizácií

V tejto podkapitole si ukážeme ako počet agentov vplýva na chyby v synchronizácií. Je zrejmé, že tento vplyv bude mať stúpajúcu tendenciu vzhľadom na tento počet, pretože v systéme s vyšším počtom agentov je aj vyššia pravdepodobnosť, že viac agentov sa bude chcieť synchronizovať v skoro rovnaký čas. Nasledujúci graf 7.1 zobrazuje vplyv rôzneho počtu agentov na počet chýb v synchronizácií.



Obrázok 7.1: Závislosť počtu chýb na počte agentov

Na grafe vidíme výsledky pre počet 7-23 agentov. Týto agenti merali teplotu miesta, kde ležali v intrevale od 5 do 20 sekúnd. Pri nameraní teploty rovnej alebo vyššej ako 65

stupňov celzia agent synchronizuje svoju aktuálnu hodnotu logických hodín s ostatnými agentami v prostredí, v ktorom sa nachádza. Pre umiestnenie agentov platí, že je pseudonáhodné. Jedinou podmienkou je, že agenti vytvárajú jednu „broadcastovú doménu“, čo implikuje, že každý agent sa môže synchronizovať s každým agentom v danom prostredí. Ďalším parametrom merania je počet lokálnych udalostí, ktoré prebehli na jednom agentovi systému. V našom prípade ich bolo 50.

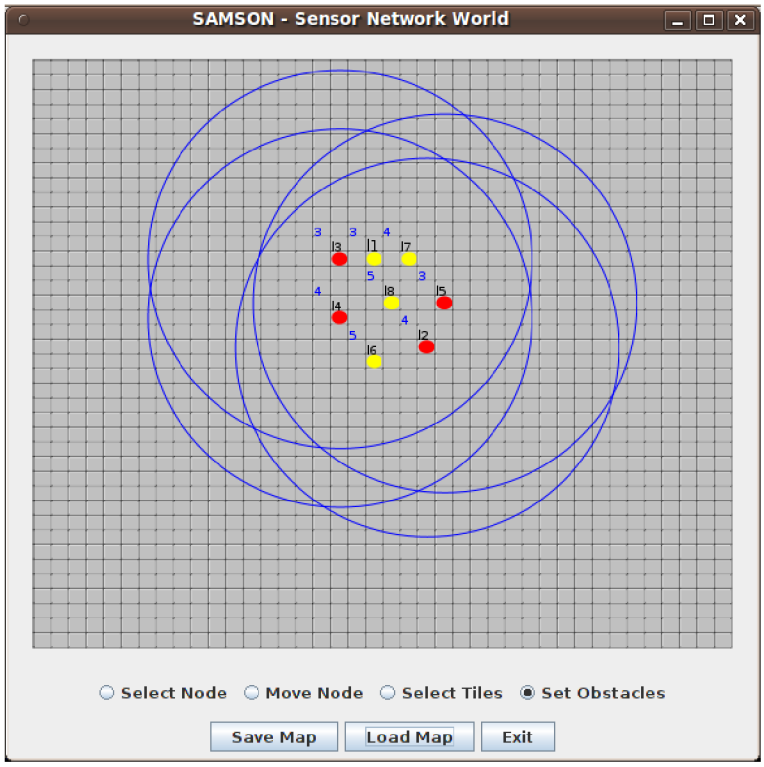
Ako môžeme vypozorovať z grafu, tak krivka závislosti má stúpajúcu tendenciu. Odchylky sú spôsobené hlavne topológiou náhodného rozloženia agentov a samozrejme vygenerovaným intervalom merania teploty.

7.2 Vplyv topológie na chybu v synchronizácií

Chyby v synchronizácií samozrejme závisia na relatívnej polohe agentov. V tejto časti sa pokúsime ukázať, akým spôsobom sa dá ovplyvniť počet chýb v synchronizácií ich polohou voči sebe.

7.2.1 Každý agent má v dosahu rádia všetkých agentov

Takáto situácia je zobrazená na obrázku 7.2.



Obrázok 7.2: Rozloženie agentov tak, že každý agent je v dosahu každého

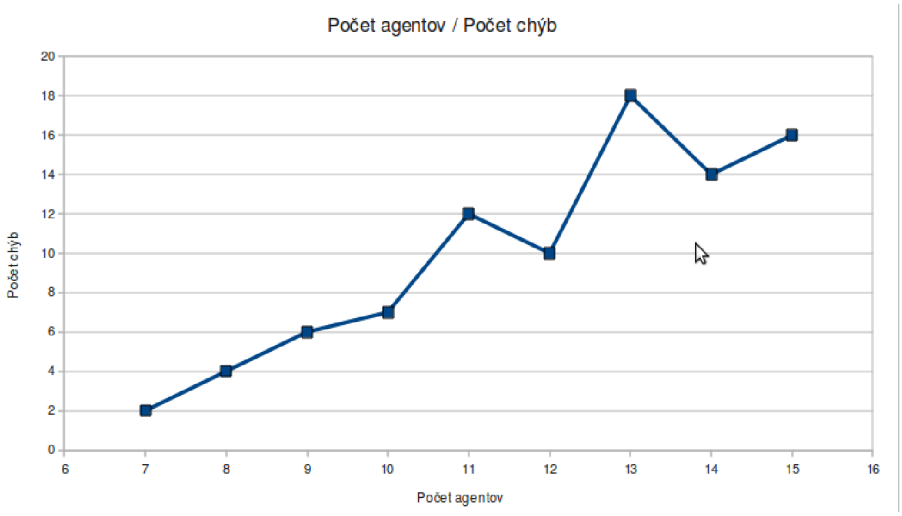
Tento obrázok zobrazuje rozloženie agentov v prostredí takým spôsobom, že každý agent dokáže priamo komunikovať so všetkými agentami v prostredí. Pre takúto situáciu je zvolenie synchronizácie pomocou logických hodín ideálnym riešením, pretože po nameraní teploty

pre synchronizáciu sa agent v ideálnom prípade zosynchronizuje so všetkými agentmi v prostredí. Pri tejto topológií môže chyba synchronizácie nastať iba v prípade, ak sa momenty meraní dvoch alebo viacerých agentov stretnú a začnú sa naraz synchronizovať. Samozrejme nemusia sa tieto momenty stretnúť úplne, stačia len na takú dobu, aby synchronizácia nestihla byť spracovaná všetkými agentmi.

Táto topológia je však použiteľná len v obmedzených prípadoch, čo vychádza z obmedzeného dosahu prenosu dát medzi agentmi. Dokonca môžme povedať, že s problémami riešiteľnými touto topológiou by sme sa stretli len ojedinele.

7.2.2 V dosahu agenta sú maximálne dvaja agenti

Tento prístup hovorí o usporiadaní agentov v rade za sebou, pričom každý agent má maximálne dvoch susedov, ktorý sú v dosahu jeho rádia. Na grafe 7.3 si ukážeme, aká bude chyba synchronizácie v takejto topológií pre rôzny počet agentov.



Obrázok 7.3: Závislosť počtu chýb na počte agentov. Každý agent má maximálne dvoch agentov v dosahu.

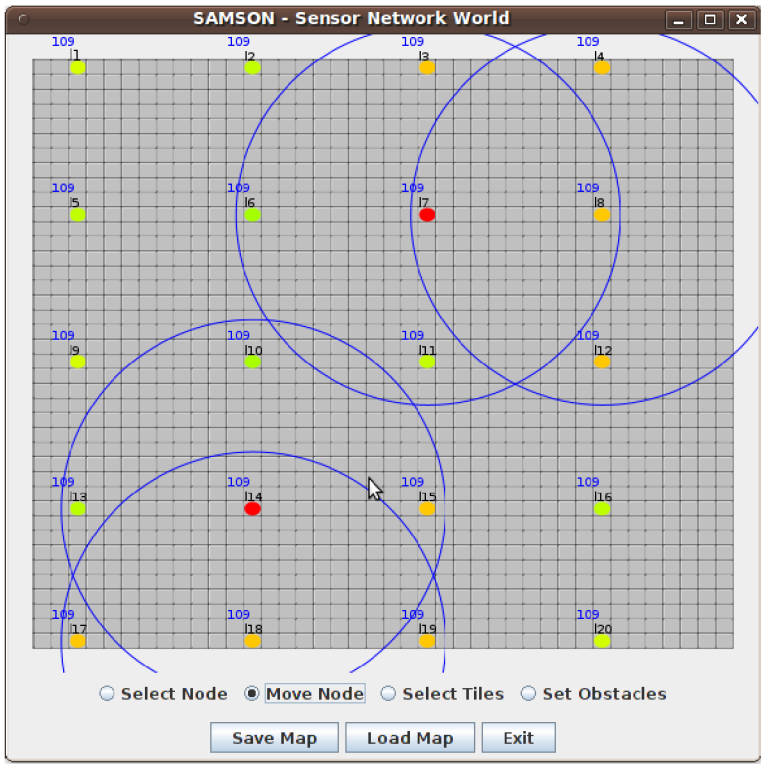
Meranie, ktorého výsledkom je graf na obrázku 7.3 bolo spustené za podobných podmienok ako to predchádzajúce. Rozloženie agentov však zodpovedá topológií spomínanej vyššie. Z grafu je vidno, že počet chýb je omnoho vyšší, pretože rozloženie agentov je len také, aké je potrebné pre všesmerové vysielanie medzi agentmi. Toto rozloženie viac zodpovedá reálnemu rozloženiu. Dá sa povedať, že táto topológia a topólogia, kedy je každý agent v dosahu sú dvoma extrémami, ktoré ohraničujú použiteľnosť synchronizácie pomocou logických hodín. Na krivke je vidno, že pri 50-tich udalostiach sa chyba, pri synchronizácií môže blížiť až k 40% a reálne to môže byť aj viac[1].

V ďalšej časti si predstavíme prístupy kombinujúce poznakty, ktoré boli predstavené vyššie.

[1]Pozn.: Záleží to od usporiadania náhodných udalostí v rámci agenta a celého sytému.

7.2.3 V dosahu agenta sú maximálne štyria agenti

Tu si ukážeme topológiu, ktorá je akýmsi rozšírením tej predchádzajúcej. Ako nadpis hovorí v dosahu rádia každého agenta sú maximálne štyria ďalší. Toto rozloženie ilustruje obrázok 7.4.



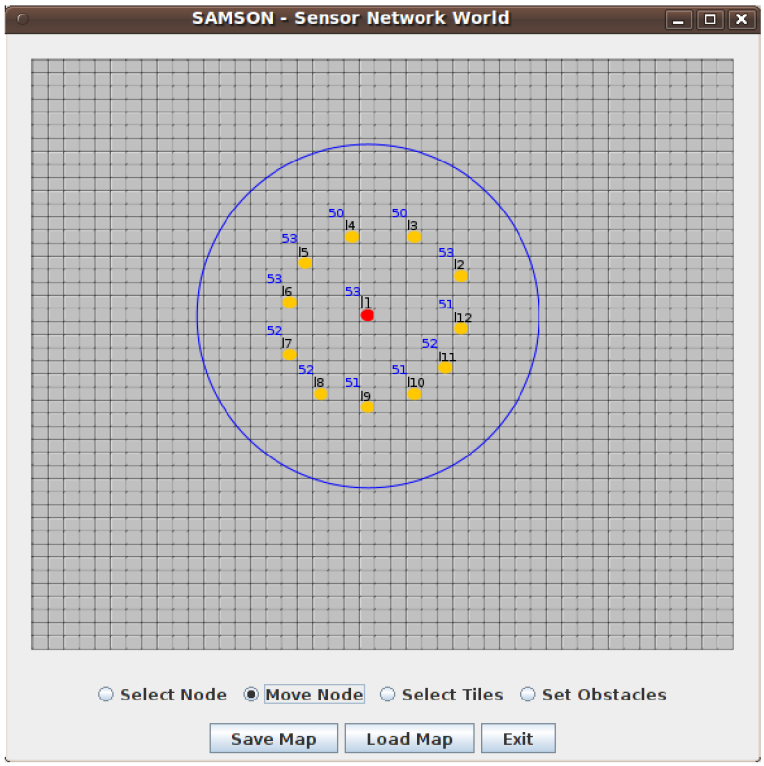
Obrázok 7.4: Rozloženie agentov tak, že každý agent má v dosahu maximálne štyroch agentov

Výhodou tohto riešenia je rýchle šírenie synchronizácie a to aj vertikálne aj horizontálne. Pre uvedenú situáciu, kde bolo dvadsať agentov, nám po viacerých meraniach vyššiel podiel synchronizačných chýb k celkovému počtu udalostí okolo 25-30%. Táto topológia má ešte jednu výhodu, a tou je odolnosť voči výpadkom jednotlivých agentov, napr. vplyvom stavu batérie agenta. Topológia sa javí ako dobrý kompromis medzi dvoma už spomínanými topológiami a je vhodná hlavne na plošné merania náhodnej veličiny agentom.

7.2.4 Topológia „hviezda“

Situácia tejto topoógie je zobrazená na orázku 7.5.

V tomto sa stará stredný prvok o rozdistribuovanie synchronizačnej správy v maximálne dvoch skokoch. Toto nám zaručuje relatívne rýchlu synchronizáciu medzi agentmi. Nevýhodou však je energetická náročnosť pre agenta v strede. Viacerými meraniami bol percentuálny počet chýb v riešení do 10%. Uloženie agentov do sústredných kružníc okolo stredného prvku by sa dalo zvýšiť o viac vrstiev. Týmto by sme získali pomerne rýchlu synchronizáciu. S počtom agentov by sa však zvyšovala aj náročnosť na stredový prvok systému.



Obrázok 7.5: Topológia „hviezda“

7.3 Vplyv intervalu merania náhodnej veličiny na chybu synchronizácie

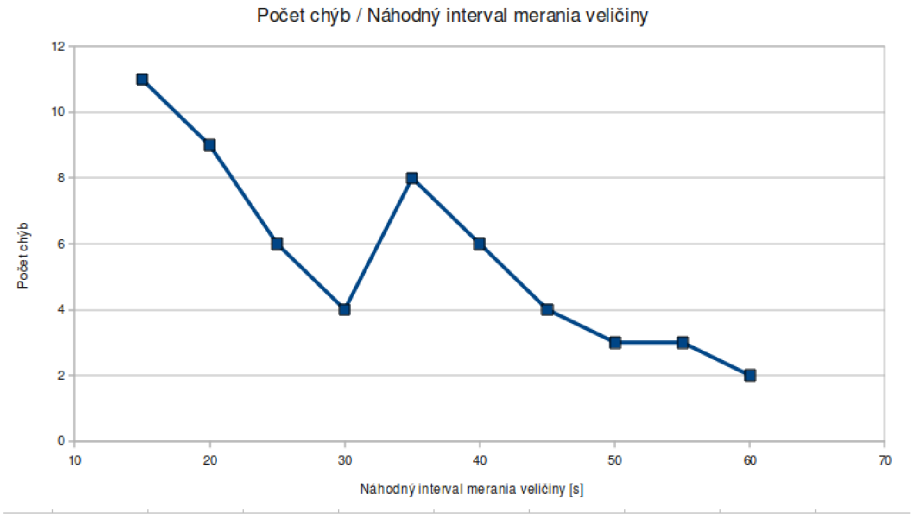
Zatiaľ sme pracovali s 15 sekundovým intervalom merania veličiny. Táto hodnota bola náhodne vygenerovaná s ohľadom na päť sekundový interval zmeny náhodnej veličiny v prostredí. Teraz si ukážeme ako môže vyšší rozdiel medzi intervalmi jednotlivých agentov ovplyvniť počet chýb v synchronizácií. Pre testovanie použijeme jedenásť agentov rozložených tak, aby v dosahu jedného boli maximálne dvaja ďalší. Výsledok merania je zobrazený na obrázku 7.6.

Z grafu je vidno, že krivka má tendenciu klesať. To znamená, že čím má náhodný interval väčší rozsah, tým je vyššia pravdepodobnosť, že sa agenti nesynchronizujú v rovnaký čas. Intervaly náhodného merania boli zvyšované a merané po piatich sekundách.

7.4 Zhrnutie

V tejto kapitole sme sa zamerali na vhodnosť implementácie z rôznych pohľadov. Najprv sme si ukázali ako počet agentov náhodne umiestnených v prostredí vplýva na usporiadanie udalostí v celom systéme. Bol tu tiež vysvetlený pojem synchronizačná chyba, ktorá má za následok nemožnosť určiť v akom poradí sa nejaké udalosti vyskytli v systéme.

Potom sme preberali vplyv topológie uloženia agentov v prostredí na spomínanú synchronizačnú chybu. Ako najlepšie riešenie je tu uvažované také, ktorého uzly majú v dosahu, čo najviac iných uzlov pre komunikáciu. Tu však musíme uvažovať aj reálne riešenie uloženia



Obrázok 7.6: Závislosť počtu chýb na náhodnom intervale merania veličiny jedenástich agentov.

agentov podľa požiadavkov na prostredie.

Nakoniec sme si ukázali závislosť od rôznych intervalov merania náhodnej veličiny, z čoho vyplynul záver, že čím sa tento interval od agenta k agentovia viac líši, tým menšia je pravdepodobnosť synchronizačnej chyby a nesprávneho usporiadania udalostí v celom systéme.

Chyby boli merané porovnaním jednotlivých súborov, kde každý agent ukladá záznamy o synchronizácií.

Kapitola 8

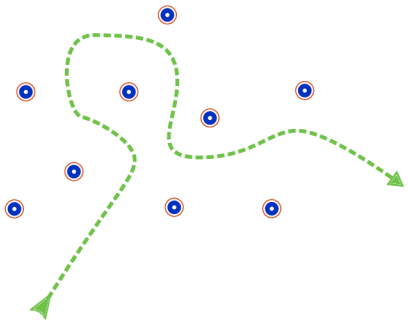
Záver

Zadaním diplomovej práce bolo zoznámiť sa s metódami pre modelovanie mobilných inteligentných agentov v distribuovaných systémoch, tu sme si predstavili platformu Jason a simulátor práce takýchto systýmov akým je SAMSON. Ten svojou koncepciou predstavuje vhodné prostredie pre simulácie WSN, pretože sám vychádza z hardvérovej platformy pre mobilné bezdrôtové senzory (TMote Sky). V práci boli predstavené len základy týchto systémov, potrebné pre pochopenie práce agentov a analogicky práce zariadení vo WSN.

Nasledujúca časť sa venuje synchronizácií v distribuovaných systémoch, tu bol detailne popísaný princíp práce logických hodín a boli predstavené dva spôsoby práce s nimi v distribuovaných systémoch. Detailne tu bol opísaný Lamportov prístup pre globálnu synchronizáciu, z ktorého sa vychádzalo pri implementácií synchronizácie.

Návrh aplikácie bol urobený s ohľadom na jej funkcionalitu. Bola pri ňom preto zvolená metodológia modelujúca vzťahy agentov v agentných systémoch.

Cieľom aplikácie je implementácia synchronizácie vo WSN, ktorá slúži na zistenie poradia meraných veličín v prostredí a to globálne v rámci siete bezdrátových senzorových uzlov. Detaily tejto implementácie môžme nájsť v kapitole 6, kde sme si vysvetlili jednotlivé postupy použité pri implementácií takéhoto agentného systému. Na nasledujúcom obrázku je zobrazený modelový príklad použitia synchronizácie na usporiadanie udalostí v distribuovanom systéme 8.1.



Obrázok 8.1: Modelový príklad merania veličiny v bezdrátovej senzorovej sieti

Uvedený obrázok predstavuje bezdrôtovú senzorovú sieť, kde modrou farbou sú označené jednotlivé senzorové uzly, trajektória meranej veličiny je označená zelenou farbou a

sústredné kruhy okolo senzorových uzlov predstavujú komunikačný dosah senzorov. Nami implementovaná synchronizácia má riešiť zmenu tejto veličiny v čase, tzn., že každá jedna meraná vzorka od uzlov, ktoré zaznamenali danú veličinu, alebo jej zmenu, by mala dostať časovú známku tak, ako sa v skutočnosti daná meraná veličina vyskytla v sytéme. Uvedený príklad je len ilustratívny. Cieľom je však vedieť poradie meraných vzoriek nezávisle na mieste, kde sa vyskytli v danom prostredí.

Pri testovaní implementácie, sme si ukázali vplyvy na synchronizáciu celého systému, ako je topológia, intervaly merania jednotlivých agentov alebo počet agentov v danom systéme. Z uvedených výsledkov môžme vyvodiť záver, že synchronizácia pomocou Lamportovho algoritmu je vhodná len v určitom počte prípadov, kedy daný systém môže tolerovať chybu v usporiadaní udalostí v rámci neho. Taktiež závisí aj od topológie senzorov, ktoré merajú veličinu, pretože niektoré topológie vykazujú vysokú chybovosť v synchonizácií.

V budúcnosti by sa mohla skúmať súvislosť implementovaného riešenia na spotrebe energie jednotlivých uzlov v systéme. Taktiež spotreba prvkov v závislosti na topológií, pretože je zrejmé, že niektoré topológie vyťažujú určité uzly viac ako tie druhé. Ďalej by mohlo byť navrhnuté riešenie, ktoré by eliminovalo chybu v synchronizácií alebo by aspoň zamedzilo nesprávnemu usporiadaniu udalostí v distribuovanom systéme.

Literatura

[1] Bordini, R. H.; Wooldridge, M.; Hübner, J. F.: *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007, ISBN 0470029005.

[2] Hübner, J. F.; Bordini, R. H.: Programming AgentSpeak agents with Jason. 2005, [Online, navštívené 3.1.2010].
URL http://www.inf.furb.br/~jomi/poa/slides/slides-Jason-bas-05.pdf

[3] Krzyzanowski, P.: Clock Synchronization.
URL http://www.krzyzanowski.org/rutgers/notes/content/06-clocks.pdf

[4] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, ročník 21, č. 7, 1978: s. 558–565, ISSN 0001-0782, doi:http://doi.acm.org/10.1145/359545.359563.

[5] Morris, A.: SAMSON: Strong Multi-Agent Simulation of Wireless Sensor Networks. 2008.

[6] Padgham, L.; Winikoff, M.: The Prometheus Methodology. 2004, [Online, navštívené 30.3.2010].
URL http://goanna.cs.rmit.edu.au/~linpa/Papers/bookchB.pdf

[7] Wikipedia: Belief-Desire-Intention software model — Wikipedia, The Free Encyclopedia. 2009, [Online, navštívené 3.1.2010].
URL
http://en.wikipedia.org/wiki/Belief-Desire-Intention_software_model

[8] Wikipedia: Happened-before — Wikipedia, The Free Encyclopedia. 2009, [Online, navštívené 12.12.2009].
URL http://en.wikipedia.org/wiki/Lamport_ordering

[9] Wikipedia: Lamport timestamps — Wikipedia, The Free Encyclopedia. 2009, [Online, navštívené 12.12.2009].
URL http://en.wikipedia.org/wiki/Lamport_timestamps

[10] Wikipedia: Vector clock — Wikipedia, The Free Encyclopedia. 2009, [Online, navštívené 17.12.2009].
URL http://en.wikipedia.org/wiki/Vector_clocks

Dodatek A

Kód agenta pre synchronizáciu pomocou Lamportových hodín

A.1 lamport_agent.asl

```
//------------------------------------------------------
// Initial Beliefs
//------------------------------------------------------
agent_state(0).  //0... sensing
                 //1... synchronizing
                 //2... being synchronized


//------------------------------------------------------
// Including goals
//------------------------------------------------------

{ include(,,lamport\_goals.asl‘‘) }

//------------------------------------------------------
// Initialize Agent
//------------------------------------------------------

!start.

+!start : true
    <- .random(Tmp);
       myp.round(Tmp*60 + 5, Time);
       .print(,,Starting agent with loop period ‘‘, Time, ,,s.‘‘);
.my_name(Name);
        setClock(0);
        setMCUState(0);
        setRadioState(0);
        setTXPower(-24);
        +recentSync(Name, 0);
```

```
        !saveSync(Name, 0);
        !senseEnv(Time).

//------------------------------------------------------
// Acquired beliefs
//------------------------------------------------------

//------------------------------------------------------
// Synchronizing logical clocks
//------------------------------------------------------
+msg(Src, From, Content) : .number(Content) & clock(MyClock)
               & MyClock < Content & not recentSync(Src, Content) & agent_state(0)
    <-  -+agent_state(2);
        setClock(Content);
        +recentSync(Src, Content);
        !saveSync(Src, Content);
        setRadioState(3);
        sendTX(Src, Content);
        setRadioState(0);
        .print(,,Synchronized from ‘‘, From, ,,, originated from ‘‘, Src,
                                          ,,. Content: ‘‘, Content, ,,.‘‘);
        .abolish(msg(_,_,_));
        -+agent_state(0).

+msg(Src, From, Content) : .number(Content) & clock(MyClock)
               & MyClock >= Content & not recentSync(Src, Content) & agent_state(0)
    <- -+agent_state(2);
        setClock(MyClock+1);
        +recentSync(Src, Content);
        !saveSync(Src, Content);
        setRadioState(3);
        sendTX(Src, Content);
        setRadioState(0);
        .print(,,Synchronized from ‘‘, From, ,,, originated from ‘‘, Src,
                                          ,,. Content: ‘‘, Content, ,,.‘‘);
        .abolish(msg(_,_,_));
        -+agent_state(0).

+msg(Src, From, Content) : .number(Content) & clock(MyClock)
                 & not recentSync(Src, Content) & (agent_state(1) | agent_state(2))
    <-  .wait(200);
        +msg(Src, From, Content).

+msg(Src, From, Content) : true
    <- .abolish(msg(_,_,_)).
```

```
//------------------------------------------------------
// Perception loop
//------------------------------------------------------

-!senseEnv(Period) : true
    <-  .concat(,,now + ``, Period, ,, s``, TimeFromNow);
        .concat(,,+!senseEnv(``, Period, ,,)``, Goal);
        .at(TimeFromNow, Goal).

+!senseEnv(Period) : agent_state(0)
     <- .concat(,,now + ``, Period, ,, s``, TimeFromNow);
        .concat(,,+!senseEnv(``, Period, ,,)``, Goal);
        !senseTemperature;
        ?clock(N);
        .abolish(clock(_));
        setClock(N+1);
        !broadTemp(N+1);
        .at(TimeFromNow, Goal).
```

A.2 lamport_goals.asl

```
//------------------------------------------------------
// Sensing temperature from environment
//------------------------------------------------------
+!senseTemperature : true
    <- .abolish(sensorVal(_));
       sense(0);
       ?sensorVal(Temp);
       .print(,,Temperature is ``, Temp, ,,C.``).

//------------------------------------------------------
// Save synchronizing data
//------------------------------------------------------
+!saveSync(Originator, Clock) : true
    <-  myp.toString(Originator, Orig);
        myp.toString(Clock, Clk);
        .concat(Orig, ,,,``, Clk, Str);
        saveData(Str).

-!saveSync : true
    <- true.

//------------------------------------------------------
// Broadcasting clock value
//------------------------------------------------------
-!broadTemp(Clk) : true
    <-  .wait(200);
        !broadTemp(Clk).
```

```
+!broadTemp(Clk) : sensorVal(Temp) & Temp < 70
    <-  true.

+!broadTemp(Clk) : sensorVal(Temp) & Temp >= 70 & agent_state(0)
     <- -+agent_state(1);
        .my_name(MyName);
        .print(,,Broadcasting temperature...‘‘);
        +recentSync(MyName, Clk);
        !saveSync(MyName, Clk);
        setRadioState(3);
        sendTX(MyName, Clk);
        setRadioState(0);
        -+agent_state(0).
```