# Úvod

Napriek tomu, že v súčasnosti existujú multiagentové systémy viac-menej v pozadí verejného záujmu, v praktickom využití nachádzajú svoje miesto. Spomenúť môžeme napríklad tému na vzostupe - „inteligentné domy“, kde navzájom prepojené agenty starajúci sa o rôzne súčasti domu sú schopní konať na základe stavu podmienok vo svojom prostredí.

Ďalším objektom záujmu tejto práce bude decentralizovanosť týchto systémov. Od platformy LCP sa očakáva, že bude multiagentová a taktiež má existovať bez centrálnej autority, v tomto prípade servera. Pre našu komunikačnú platformu, to znamená, že má byť schopná nakonfigurovať sa sama od seba.

## Motivacia

## Cieľ práce

Cieľom tejto práce je rozšírenie Jednoduchej Komunikačnej Platformy (ďalej LCP, podľa „Lightweight Communication Platform“) o možnosť komunikácie agentov v jednej lokálnej sieti s agentami v inej lokálnej sieti. Východiskovým riešením tohto problému je takzvaný „Gateway Agent“, ktorý je schopný posielať správy aj za iných agentov v jeho lokálnej sieti a prakticky sa správa ako virtuálny router.

## Štruktúra práce

Táto práca bude ďalej rozčlenená do siedmich častí. Prvá kapitola začne základnou definíciou pojmu agent a jeho členenie. Za definíciou budú nasledovať teoretické (kapitola 1) a technické (kapitola 2) základy a komunikácia agentov v multiagentových systémoch. V ďalšej časti sa budeme venovať podobným, už existujúcim riešeniam problematiky a príkladom multiagentových alebo decentralizovaných systémov (kapitola 3). Následne popíšeme technológie, ktoré budú použité v riešení ako aj technológie, ktoré budú slúžiť ako inšpirácia pri vytváraní riešení možných problémov (kapitola 4). Tým sa dostaneme k jadru tejto práce a to podrobný popis cieľa, ktorý má byť výsledkom tejto práce (kapitola 5). Táto kapitola bude nasledovaná popisom riešenia hlavného problému a implementáciou tohto riešenia (kapitola 6). V závere v krátkosti zhrnieme výsledok, ku ktorému sme sa dopracovali v tejto práci (kapitola 7).

# Prehľad

Aby sme dokázali plne pochopiť, čo sa budeme snažiť v tejto práci dosiahnuť a pre celkové porozumenie problematiky budeme sa v tejto časti práce venovať prehľadu poznatkov z oblasti agentov a multiagentových systémov. Najskôr si zadefinujeme pojem agent a vysvetlíme teoretické základy multiagentových systémov a ich potenciál, ktorý sa budeme snažiť načrtnúť na príkladoch ich praktického využitia vo svete.

## Inteligentný agent

Klasickú definíciu agenta nám ponúkajú Russel a Norvig[1]:

*„Agent je všetko, na čo sa dá pozerať ako na niečo, čo vníma svoje okolie senzormi a reaguje na toto okolie pomocou aktuátorov.“*

Agent je podľa nich zložený z architektúry a agentového programu, ktorého vytvorenie je úlohou práve umelej inteligencie. Fyzická architektúra nás v tejto práci nebude zaujímať, preto ďalej v tejto kapitole budeme rozumieť pod pojmom agent práve agentový program. Existujú agenty, ktoré sú veľmi jednoduché, ale aj také, ktoré sú zložité. Podľa ich vnemovej inteligencie a schopnosti ich opäť Russel a Norvig rozdelili do týchto piatich kategórií, ktoré vzápätí rozoberieme podrobnejšie:

* **Jednoduchý reflexný agent**
* **Modelovo-založený reflexný agent**
* **Cieľovo-založený agent**
* **Úžitkovo-založený agent**
* **Učenlivý agent**

### *Simple reflex agent*

Tento druh agenta je úplne najjednoduchší. Reaguje totiž len na aktuálny stav jeho prostredia a pritom si neuchováva žiadnu históriu stavov daného prostredia. Tento agent je plne úspešný iba v úplne pozorovateľnom prostredí. Keď sa jedná o čiastočne pozorovateľné prostredie, vo väčšine je tento agent v nekonečnom cykle a jeho správanie je väčšinou podmienené. Ak je splnená podmienka, tak na ňu programovo reaguje (Obrázok č.1).

Obrázok č. 1 Diagram cyklu jednoduchého reflexného agenta v čiastočne pozorovateľnom prostredí

****

### *Model-based agent*

Na rozdiel od jednoduchého reflexného agenta, je v tomto type agenta uložená štruktúra, ktorá má reprezentovať súčasný stav jeho sveta. Ten je ovplyvnený históriou vnímania. Podľa zmien je schopný naučiť sa „ako svet funguje“ a teda je schopný čiastočne zachytiť aj stránky sveta, ktoré nie je schopný vnímať. Táto skutočnosť má za následok, že *model-based* agent je úspešný aj v čiastočne pozorovateľnom prostredí. Svoje reakcie na stav sveta vyberá ako jednoduchý reflexný agent.

### *Goal base agent*

Jeho základ je v podstate rovnaký ako *model-based* agenta, je však rozšírený o informáciu o jeho cieli. Táto informácia v sebe zahŕňa popis požadovaných situácií a stavu sveta. Vďaka tomu je *goal-based* agent schopný z viacerých možností vybrať takú, ktorá ho dostane alebo priblíži k požadovanému stavu. Ako vybrať správnu reakciu je otázka plánovania a prehľadávania. Sú to poddisciplíny umelej inteligencie. Niekedy je tento agent menej efektívny, ale za to je viac flexibilný, pretože jeho cieľ je udaný explicitne a dá sa meniť.

### *Utility based agent*

Na rozdiel od *goal-based* agenta, ktorý je schopný rozoznávať iba medzi stavom, ktorý je požadovaný a ktorý nie je, je *utility-based* agent schopný určiť, nakoľko sú jednotlivé stavy požadované pomocou *úžitkovej funkcie.* Táto funkcia mapuje stav na veľkosť úžitku tohto stavu. Agent zhodnocuje možné stavy podľa toho, koľko ťažkostí alebo úžitku mu daný možný stav prinesie. Takéto zhodnocovanie mu pomáha vybrať najlepšiu postupnosť činností, keďže racionálny agent sa snaží o to, aby dosiahol čo najviac úžitku a teda bol čo najviac „šťastný(spokojný)“.

### *Learning agent*

Jeho výhodou je, že je schopný fungovať aj v neznámom prostredí. Čím dlhšie sa v prostredí nachádza, tým je schopnejší vyberať lepšie rozhodnutia, pretože sa učí. Skladá sa z troch častí:

* **Učenlivý element** – je zodpovedný za zlepšovanie sa agenta. Odozvu na jeho činy, ktorú získa z prostredia posúva do výkonnostného elementu, aby mu pomohol v budúcnosti konať lepšie rozhodnutia.
* **Výkonnostný element** – má za úlohu vybrať najlepšiu akciu
* **Generátor problémov** – navrhuje akcie, ktoré by mohli viesť k novým skúsenostiam a tým novým poznatkom

## Teoretické základy multiagentových systémov

Aj keď je agent sám o sebe schopný riešiť určité problémy, môže sa stať, že narazí na problém, na ktorého vyriešenie nemá dostatočné prostriedky. Práve preto vznikla nutnosť existencie multiagentových systémov. Sú to také systémy, ktoré v sebe zahŕňajú viacero agentov, ktorí navzájom spolupracujú, a ich prostredie. Katia P. Sycara[2] popísala ich charakter takto:

1. Každý agent má neúplné informácie alebo nedostatočné schopnosti vyriešiť problém a teda má naň limitovaný pohľad
2. Neexistuje žiadna globálna kontrola.
3. Dáta sú decentralizované
4. Výpočty sú asynchrónne

### Typy multiagentových systémov

Klasifikácií multiagentových systémov existuje mnoho. Môžeme ich deliť podľa typu prostredia v ktorom sa agenty nachádzajú, aké komplexné alebo akého typu sú, podľa komunikácie medzi nimi a tak ďalej.

#### Typy multiagentových systémov podľa vzájomnej komunikácie

Je možné rozdeliť multiagentové systémy podľa toho, akým spôsobom agenty medzi sebou komunikujú. Existujú tri typy multiagentových systémov podľa ich vzájomnej komunikácie a to [5]:

* **Bez priamej komunikácie** – tento typ komunikácie môže byť realizovaný napríklad pomocou prostredia
* **Jednoduchá komunikácia** – väčšinou je jednosmerná aj keď kontaktovaný agent môže reagovať na požiadavku
* **Komplexná komunikácia –** taktiež ju môžeme nazývať podmienková, príkladom môže byť „Pohnem sa, ak sa pohneš ty“

#### Typy multiagentových systémov podľa prostredia

V tejto podkapitole popíšeme typy prostredí, v ktorých sa agenti môžu nachádzať. Toto rozdelenie bolo navrhnuté Russelom a Norvigom[1]

* **Plne pozorovateľné a čiastočne pozorovateľnému** – Ak v hocijakom čase môže agent získať celkový stav svojho prostredia, hovoríme o takom prostredí ako o plne pozorovateľnom. Úlohové prostredie je plne pozorovateľné, pokiaľ dokáže agent pomocou svojich senzorov zachytiť všetky informácie, ktoré sú **relevantné** pre jeho rozhodnutie. Čiastočne pozorovateľné prostredie môže byť vtedy, keď je v prostredí vysoký šum, nepresné senzory a teda senzory nedokážu zachytiť presné informácie alebo jednoducho dáta zo senzoru chýbajú.
* **Deterministické a stochastické prostredie –** Deterministické prostredie je také prostredie, v ktorom hocijaký čin má jediný možný výsledok. Naproti tomu v stochastickom prostredí si nemôžeme byť istí, aký účinok náš čin vyvolá.
* **Statické a dynamické prostredie –** Ak prostredie dokáže byť ovplyvnené iba činmi agenta inak zostane nezmenené, také prostredie nazývame statické. Dynamické prostredie sa však mení odhliadnuc od činov agenta.
* **Diskrétne a spojité prostredie –** Napríklad hra šachu má konečný počet pozorovateľných stavov a vnemov. Takémuto prostrediu hovoríme diskrétne. Prostrediu hovoríme spojité ak stavy a vnemy závisia napríklad od času alebo polohy agenta. Príkladom môže byť automatizovaný šofér.
* **Známe a neznáme –** Nejde tu ani tak o prostredie samotné ako o to či agent alebo jeho tvorca vie, ako prostredie funguje. Ak nevie, ide o neznáme prostredie a agent sa musí zákonitosti a fungovanie takéhoto prostredia naučiť. Agent na druhej strane môže vedieť ako prostredie funguje, neznamená to však, že prostredie je plne pozorovateľné. Ako príklad môžeme uviesť hru hľadania mín. Vieme zákonitosti a pravidlá hry, avšak nevieme, kde sa míny nachádzajú. Takéto prostredie je potom čiastočne pozorovateľné.

### Komunikácia v multiagentových systémoch

Komunikácia agentov je jednou z najzákladnejších podmienok v multiagentových systémoch. Je esenciálna k tomu, aby si agenty vedeli vymieňať informácie, koordinovať svoje úlohy a takýmto spôsobom spolupracovať na dosiahnutí ich cieľa. Ak by komunikácie neboli schopní, stratil by sa celý zmysel multiagentového systému. Jazyk, ktorým sa agenty dorozumievajú sa nazýva „Agent Communication Language“, skrátene ACL. Dva najpoužívanejšie agentové jazyky sú FIPA-ACL[3], a KQML[4]. Oba jazyky boli inšpirované teóriou rečových aktov.

### Multiagentové systémy a počítačové siete

Od multiagentových systémov zvyčajne očakávame, že agenty v rámci systému neexistujú na jednom zariadení, ale vo viacerých. Od týchto zariadení už principiálne vyžadujeme, aby boli vzájomne prepojené. Väčšinou sú pripojené do lokálnej siete (od „Local Area Network“) alebo do rozsiahlej siete WAN („Wide Area Network“). Aby však agenty boli schopné po takejto sieti komunikovať, musia byť špecificky naprogramované. Tomuto programovaniu sa tiež hovorí sieťové programovanie.

Jednotlivá funkčnosť siete je rozdelená do sieťových vrstiev v OSI modeli[link] (Open System Interconnection) špecifikovaným organizáciou ISO (International Organization for Standardization). Tento model obsahuje 7 (Obrázok č.2) vrstiev a každá z nich využíva vrstvu pod ňou a slúži vrstve nad ňou. Protokoly na rovnakej vrstve sú schopné spolu komunikovať. [6]



Obrázok č. 2 Vrstvy OSI modelu

#### Routovanie

Routovanie je určovanie cesty paketov v sieti. Zaoberá sa ním tretia, sieťová, vrstva. Existujú mnohé spôsoby ako určiť cestu, ktorou bude paket poslaný. Môže byť vybraná staticky podľa routovacích tabuliek, vopred dohodnutá v rámci jedného dialógu alebo určená dynamicky. Posledný typ prevláda súčasným internetom. Príkladmi protokolov takéhoto routovania je napríklad RIP[7] a OSPF[8].

V dynamickom routovaní, taktiež nazývanom aj adaptatívne, je dôležitá takzvaná routovacia metrika. Je to hodnota, niekedy jej hovoríme aj cena cesty, ktorá je rozhodujúcim faktorom v rozhodovaní, ktorú cestu uprednostniť.

Metrika môže byť [9]:

* **Aditívna –** celková cena cesty je sumou cien jednotlivých úsekov celej cesty
* **Konkávna –** celková cena cesty je minimom cien jednotlivých úsekov celej cesty
* **Multiplikatívna** – celková cena cesty je súčinom cien jednotlivých úsekov celej cesty

Celková hodnota metriky je určená viacerými aspektmi, napríklad počtom hopov, rýchlosťou, odozvou, stratovosťou paketov cesty alebo aj MTU. Spomínaný RIP patrí do rodiny *distance-vector* protokolov. Tie využívajú Bellman-Ford, Ford-Fulkerson alebo DUAL FSM algoritmy pre výpočet najvhodnejšej cesty. Ukladajú si minimálne vzdialenosti do všetkých uzlov v sieti. Vzdialenosťou sa nemyslí fyzická vzdialenosť, ale cena za dosiahnutie určitého uzla. Z toho vyplýva, že čím menšia cena, tým menšia vzdialenosť.

## Existujúce návrhy multiagentových systémov

V súčasnosti existujú mnohé návrhy využitia multiagentových systémov. Sú schopné sa uplatniť v širokej škále oblastí od turizmu, cestnej premávke, opravách, armáde alebo aj v medicíne.

* **PalliaSys[10] –** Návrh multiagentového systému, ktorý má pomôcť pri paliatívnej starostlivosti tým, že pomocou agentov bude monitorovať stav pacienta a bude schopný ponúknuť detailné, aktuálne informácie o pacientovi doktorom.
* **MokSAF –** Systém, ktorý má pomôcť armáde pri kritickom rozhodovaní a ponúka virtuálne prostredie na plánovanie a koordináciu cesty. Spolupracujú v ňom dva typy agentov – *Path Planner* agent*,* ktorý vedie ľudí po ceste, ktorú vyhodnotí ako najkratšiu a berie do úvahy iba fyzické prekážky, ktoré môžu byť zadané aj manuálne do mapy. *Critique Agent* analyzuje nakreslenú cestu a pomáha doladiť detaily, berie pritom do úvahy fyzikálne, ekonomické a sociálne aspekty. Na rozdiel od *Path Planner* agenta nevytvára cestu automaticky, ale oznámi či daná cesta je možná.
* **MAS/LUCC – je** označenie pre multiagentové systémy zaoberajúce sa modelovaním využitia pôdy a jej nasledovné pokrytie.

## Existujúce riešenia

**Java Agent DEvelopment Framework (JADE)**

Middleware, ktorý zjednodušuje implementáciu multiagentových systémov, ktoré sú kompatibilné s FIPA špecifikáciou. Celý software je naprogramovaný v Jave. Obsahuje grafické nástroje, ktoré pomáhajú najmä v štádiách debuggingu a deplymentu. Tento software je v súčasnosti zadarmo. [12]

**FIPA-OS**

Prvý krát vydaný v roku 1999 pre širokú verejnosť *Royalty Free*. Zároveň je prvou Open Source implementáciou FIPA a na jeho vývoji sa podiela mnoho vývojárov. Podarilo sa im vydať cez 10 oficiálnych vydaní. Podporuje väčšinu experimentálnych FIPA špecifikácií. Taktiež je implementovaný v Jave a pre jeho návody je vhodný pre začínajúcich vývojárov FIPA kompatibilných systémov. [13]

## LCP

Jednoduchá komunikačná platforma LCP používa na prenos správ medzi agentami REST TCP/IP požiadavky. Sú založené na FIPA ACL. Oproti zvyčajnej FIPA implemetácii, sú jej cieľom sú heterogénne systémy obsahujúce viacero jednoduchých agentov, ktorí môžu byť vytvorené v hocijakom jazyku a sú schopné komunikovať bez centrálneho manažmentu. [11]

Agenty v takomto systéme majú byť schopné objaviť ostatných agentov na svojej lokálnej sieti, ako aj identifikovať, aké služby tieto agenty ponúkajú. Umožňuje to existenciu viacerých heterogénnych agentov rozmiestnených po lokálnej sieti. Systém má fungovať bez centrálneho manažmentu.

Od LCP očakávame, že aj agenty zo vzdialených lokálnych sietí, FIPA platforiem alebo implemetácií sa budú schopní objavovať a taktiež si medzi sebou posielať správy. Dôsledkom toho vznikol v systéme koncept *Gateway Agenta*, ktorý je za túto funkcionalitu zodpovedný. Virtuálne plní funkciu routera a je rozhraním medzi heterogénnymi časťami systému, ktoré nie sú schopné priamo komunikovať.

//dopln popis DS a MTS

# Špecifikácia zadania práce

Hlavnými časťami LCP, ktorými sa budeme v tejto práci zaoberať sú *Discovery Service* a *Message Transport Service*. Za účelom testovania bude pre nás zaujímavá aj časť *Platform*, ktorá bude obsahovať predchádzajúce komponenty.

## Discovery Service

Ako bolo spomenuté vyššie, súčasťou LCP bude *Discovery Service* (ďalej len DS), ktorý bude zodpovedný za objavovanie agentov, ich služieb a možností ako ich kontaktovať. Okrem objavovania bude mať aj opačnú úlohu a to poskytovať tieto informácie o agentoch, za ktorých je zodpovedný, ostatným DS.

## Message Transport Service

Ďalšia súčasť LCP, *Message Transport Service* (ďalej len MTS)*,* bude na druhej strane zodpovedná za odosielanie správ najlepšou cestou pomocou dostupných informácií od DS. Taktiež sa bude starať o preposielanie správ ich príjemcom, pokiaľ sa nenachádza v konečnom uzle trasy, ktorou je správa poslaná.

## Platform

Táto súčasť obsahuje DS aj MTS a správa sa ako rozhranie medzi nimi. V kontexte tejto práce vznikol tento komponent pre potreby testovania a doplnenia funkčnosti. Tento komponent nevznikol za účelom vytvorenia celého komponentu, ktorý by bol integrovaný do celkového systému, ale ako jeho neúplná simulácia. Jeho úlohou bude spravovať aktuálny zoznam aktívnych agentov ako aj perzistentný zoznam gateway agentov podľa stanovených kritérií a aktuálny zoznam aktívnych gateway agentov.

## Gateway Agent

V kapitole o LCP sme spomenuli pojem GW agenta, ktorého úlohou je premostiť dve heterogénne časti systému. Taktiež virtuálne plní úlohu routera v sieti, pretože jemu sa posielajú správy, ktoré sú určené pre členov mimo lokálnej siete odosielateľa, aby ich poslal smerom k príjemcovi. Taktiež oznamuje agentov zo vzdialených lokálnych sietí na tej svojej. Na jednej lokálne sieti môže byť viac ako jeden a musí byť aspoň jeden GW agent. Dôležitá poznámka: Aby sa predišlo nedorozumeniam, je nutné napísať, že Gateway Agent nie je správne pomenovanie v kontexte tejto práce. Názov pochádza ešte z pôvodného návrhu systému a preto je tento pojem viac menej historický. V našom systéme bude úlohy GW agenta plniť platforma.

## Správy

Agenty budú medzi sebou komunikovať pomocou správ. Tieto správy delíme na tri skupiny:

1. **Status Messages –** správy typu *Hello* a *Bye*.
2. **Notify Message –** správa, ktorá obsahuje informácie o agentoch
3. **Standard Message –** správa pre agenta

Prvé dva typy sú správy pre DS, posledná sa odovzdá agentovi na spracovanie.

# Návrh riešenia

Vzhľadom na cieľ tejto práce a návrh komponentov systému v tejto kapitole špecifikujeme požiadavky na ich funkčnosť.

## Platform

Medzi úlohy platformy bude okrem úloh rozhrania medzi DS a MTS patriť udržiavanie zoznamu všetkých agentov v systéme a informácií o nich. V tomto ohľade spolupracuje s DS, ktorý jej aktuálne informácie poskytuje. Ak sú nejaké informácie o agentovi neaktuálne, aktualizuje ich. Pokiaľ ide o transport adresy, aktualizuje ich alebo ich vymaže. Ak agent nemá informáciu o žiadnej, platforma ho považuje za neexistujúceho a zo zoznamu ho vymaže. Platforma kontroluje platnosť transport adries v pravidelných intervaloch.

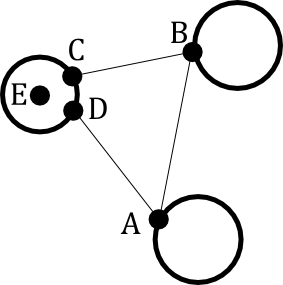
V súvislosti s MTS bude mať platforma na starosti odovzdávanie správ od agentov na nej bežiacich MTS spolu s agentami, ktorým majú byť tieto správy doručené. Adresy a týchto agentov mu bude poskytovať zo svojho zoznamu. Táto úloha platí aj spätne. Platforma prevezme správu od MTS pre svojich agentov a odovzdá im ju na spracovanie.

## Discovery Service

DS ihneď po spustení oznámi svoju prítomnosť na lokálnej sieti pomocou správy *Hello* čím u ostatných DS spustí odoslanie správy *Notify* (viď nižšie). DS oznamujúca svoju prítomnosť týmto krokom zistí existenciu agentov na svojej lokálnej sieti. V prípade, že sa mu nepodarí nikoho kontaktovať a nedostane žiadny zoznam aktívnych agentov, predpokladá, že sa v systéme nachádza sám. V takomto prípade musí počkať, kým sa do systému pripojí pre neho viditeľný agent, ktorý by mu bol schopný tieto informácie poskytnúť alebo sa premiestni do lokálnej siete s aktívnymi členmi.

Počas svojho behu bude v pravidelných, pevne stanovených intervaloch posielať správu *Notify*, aby informoval ostatné DS o súčasnom stave agentov. Položky, ktoré bude o agentoch treba vedieť sú meno, ktoré je unikátne, služby, ktoré ponúka, príznaky (tzv. *flags)* a zoznam transport adries, t.j. na aké adresy možno agentovi poslať správu tak, aby sa k nemu dostala. Od DS očakávame, že zistí všetky možné cesty ku všetkým agentom bez toho, aby cesta bola výsledkom cyklu (podrobnejšie sa týmto budeme zaoberať v kapitole implementácia, sekcia XML message. Táto požiadavka vznikla preto, aby keď nastane situácia, že zanikne cesta k nejakému agentovi a reálne existuje iná, tak by mala byť ihneď použiteľná, nie až po ďalšej *Notify* správe. Viď (obrázok č.3) Práve preto sa v nej oznamujú všetky možné cesty.

Príklad: Nech kružnice vyjadrujú lokálne siete. Body na jej obvode nech sú GW agenty a body vnútri kružnice, nech sú štandardné agenty. Úsečky medzi GW agentami nech vyjadrujú schopnosť navzájom sa kontaktovať. Predpokladáme, že GW agent vie kontaktovať štandardného agenta na svojej lokálnej sieti a preto odpadáva potreba úsečiek. Nech agent A chce kontaktovať agenta E. Vyberie cestu cez agenta s najlepšov metrikou. Nech je to počet hopov. Agent A teda vyberie cestu cez GW agenta D (metrika má hodnotu 2). Teraz predpokladajme, že agent D sa odpojí a teda zanikne pôvodná cesta A-D-E. Nech A chce znovu poslať správu E. Ak by si ukladal iba práve jednu cestu k agentovi, tak by musel čakať, kým sa mu nejaká nová cesta oznámi, ale keďže si ukladá všetky možné cesty, vie, že existuje cesta cez B a C a preto správu ihneď odošle po tejto ceste.



Transport adresa bude definovaná pomocou URL. Pre potreby udržiavania aktuálnych informácií bude obsahovať aj informáciu o tom, do kedy je platná. Za účelom vyberania vhodnej cesty bude obsahovať aj informáciu o metrike danej adresy. Úlohou DS je samozrejme takéto správy aj spracovávať. Ak je v zozname oznámených agentov taký, ktorý v zozname aktívnych agentov ešte nie je, pridá ho do zoznamu agentov s prislúchajúcimi informáciami. Naopak, ak sa už daný agent v zozname nachádza, aktualizuje informácie o ňom.

Keď sa DS zo systému odpája, pošle ostatným DS správu typu *Bye*. Podľa nej vedia, že transport adresy, ktoré tento DS oznamuje už nebudú platné a preto ich platforma náležite spracuje.

## Message Transport Service

MTS je zodpovedné za spracovávanie štandardných správ v prípadoch: prijímanie, odosielanie a preposielanie[[1]](#footnote-1). Najprv preberieme odosielanie správ. MTS prevezme od platformy správu na odoslanie, k nej dostane informácie o tom, kto ju posiela a mená agentov, ktorým ju posiela. MTS vyberie transport adresu agenta podľa najlepšej metriky. Pre každú rôznu transport adresu sa vytvorí obálka, ktorá obsahuje informáciu o odosielateľovi a príjemcoch tejto obálky. Ak MTS vyberie pre rôznych agentov rovnakú transport adresu, príjemcovia tejto obálky budú všetky agenty s touto adresou.

Príklad: Pre správu majme príjemcov A, B, C. MTS vyberie pre A,B adresu T1 a pre C adresu T2. Potom pre adresu T1 vytvorí obálku, kde príjemcovia sú A a B. Pre adresu T2 vytvorí druhú obálku, kde príjemca je iba C.

Obálku, ktorá bude obsahovať taktiež aj správu odošle na príslušnú adresu. Na druhej strane sa táto obálka spracuje podľa príjemcov a rozhodne sa či správa patrí nejakému z platformových agentov alebo treba správu preposlať ďalej. Ak ide o prípad preposielania, správu vložíme do obálky s upravenými príjemcami a pošleme na adresu, ktorá je vyberaná obdobne ako v prvom prípade. Ak ide o prípad prijímania, MTS odovzdá správu pre platformového agenta platforme, ktorá ho odovzdá agentovi, pre ktorého je správa určená.

## Gateway Agent

Zodpovedá za to, aby boli agenty na vzdialených lokálnych sieťach objavené no lokálnej sieti tohto agenta a aby agenty z jeho lokálnej siete boli objavené na tých vzdialených. Navyše sa stará o posielanie a doručovanie správ medzi navzájom vzdialenými lokálnymi sieťami. Kvôli tomu, že agenty na lokálnej sieti nie sú schopné kontaktovať vzdialené lokálne siete, vzniká potreba využívať routovanie. A pretože jednotlivé agenty a hlavne gateway agenty nemusia byť stále pripojené do systému, je nutné, aby toto routovanie bolo dynamické.

# Implementácia

Obsahom tejto kapitoly bude podrobnejší opis nášho riešenia. Najprv uvedieme použité technológie a v krátkosti ich popíšeme, potom popíšeme problémy, ktoré vyplývajú z požiadaviek na systém. Na koniec rozoberieme jednotlivé riešenia spomenutých problémov a ukážky kódu metód, ktoré sú zaujímavé pre cieľ tejto práce.

## Použité technológie

Napriek pôvodným plánom nezávislosti systému od vonkajších knižníc bolo riešenie tejto práce implementované za pomoci frameworku Qt, ktorý pracuje nad programovacím jazykom C++11. Z dôvodu zjednodušenia práce sme použili vývojové prostredie Qt Creator pod linuxovou distribúciou Ubuntu 14.04. Pre potreby použitia funkcionality HTTP servera sme navyše použili webový C++ framework Tufão, ktorý využíva funkcie Qt. Komunikáciu medzi agentami sprostredkováva buď UDP/IP, ktoré posiela správy v JSON dokumente, alebo HTTP POST, ktorý využíval XML. V krátkom prehľade predstavíme iba Qt a Tufão, pretože predpokladáme, že ostatné technológie sú známe.

### Qt

Qt je využiteľné v grafických, ale aj konzolových aplikáciách. Tento framework je cross-platformový[14] a vybrali sme si ho preto, lebo ponúka funkcionalitu, ktorá uľahčila programovanie inak zdĺhavých sieťových metód. Jeho jednoduché použitie pre pripojenie na multicastovú skupinu, posielanie a prijímanie datagramov z neho robilo dokonalého kandidáta aj kvôli asynchrónnosti jeho sieťového API. Bohužiaľ priamo neimplementoval funkcionalitu HTTP servera. Ďalšou výhodou Qt je používanie signal-slot mechanizmu, ktorý je jednou z jeho hlavných vlastností. Slúži na komunikáciu medzi objektmi. Použitie je veľmi jednoduché. Pripojíme signál jedného objektu na slot toho istého alebo iného objektu. Pokiaľ objekt tento signál vyšle, príslušný objekt ho spracuje svojím slotom, kde slot je normálna funkcia C++, ktorá sa ale navyše dá pripojiť na signál. Aj toto si vyžadovalo vznik takzvaného *Meta-Object Compiler*-u, v skratke *moc*. Ten spracováva hlavičkové súbory C++ a keď nájde triedu, ktorá obsahuje makro Q\_Object, vytvorí pre ňu *meta-object* kód.

### Tufão

Hlavným dôvodom, prečo použiť tento ďalší cross-platformový framework bola pre nás práve jeho implementácia HTTP servera a jednoduché spracovávanie požiadaviek a automatizovanie niektorých častí odpovedí servera pomocou signal-slot mechanizmu Qt a implementácie HTTP request a HTTP response. Výrazným plusom tohto frameworku je aj prehľadná dokumentácia [15].

## Problémy plynúce z požiadaviek

V tejto časti sa budeme venovať problémom, ktoré boli priamym následkom požiadaviek a vo väčšine boli vyriešené priamo v implementácii. Budú spomenuté aj také problémy, ktoré si uvedomujeme, ale z dôsledku časovej tiesne neboli implementované. Problémy, plynúce zo samotnej implementácie budú riešené v nasledujúcej sekcii (6.3)

### Problém prepojenia vzdialených lokálnych sietí

Ak máme dve vzdialené siete, v ktorých ani jeden z agentov z jednej siete nemá žiadny kontakt na hocijakého agenta z druhej siete, potom nie je možné, aby sa agenti vzájomne objavili cez internet bez nejakého centralizovaného prvku. V decentralizovaných systémoch ako implementácia DHT(*Distributed Hash Table*)[16] - Kademlia sa používa proces nazývaný *bootstrap*. Nový člen, ktorý sa pripojí do systému kontaktuje vopred určený, známy *bootstrap node*, ktorý mu odovzdá adresu na aspoň jedného člena siete, cez ktorého potom postupne získa adresy na ostatných členov. Napriek tomu, že DHT sa považuje za decentralizovaný systém, my v *bootstrap node* vidíme centralizovaný prvok. V takomto prípade pre nás existuje málo možností ako naše dve vzdialené siete prepojiť. Jednou z možností je premiestniť agenta z jednej siete do siete druhého agenta. Tým pádom bude mať vo svojom zozname adresy agentov zo svojej predchádzajúcej siete spolu s adresami na Gateway agentov v systéme v čase jeho odchodu. Objavenie ostatných účastníkov na jeho súčasnej lokálnej sieti nebude problém, pretože samotný DS tieto prostriedky ponúka.

### Problém neaktívneho agenta

Napriek tomu, že tento problém už sme načrtli v predchádzajúcich kapitolách, je vhodné ho zvýrazniť. V ideálnych podmienkach agent pri odchode zo systému oznámi svoj odchod správou typu *Bye.* Po prijatí správy sa informácie súvisiace s agentom odstránia. Čo ak agent odíde zo systému nečakane následkom výpadku prúdu alebo zlyhania techniky? Riešenie tohto problému je jednoduché. Určíme dobu platnosti adries na agenta. Táto doba závisí od toho či je to agent zo vzdialenej lokálnej siete alebo „domácej“. V určitých intervaloch platforma spustí kontrolu platnosti adries agentov, adresy, ktoré sú neplatné sa zo zoznamu odstránia. Ak agent nemá žiadne platné adresy, odstráni sa zo zoznamu aktívnych agentov.

### Problém nedoručených správ

Tento problém v implementácii nie je vyriešený. Plynie z predchádzajúceho problému. Medzi tým, ako agent vypadne zo systému a vypršaním platností jeho adries môže v najhoršom prípade vzniknúť hluché miesto až niekoľko minút. Správy sa mu budú napriek tomu stále odosielať na platné adresy, ale nebude ich mať kto prijať. Jedným z možných riešení je ukladanie správ, ktoré sa nepodarilo doručiť, aby mu boli odoslané po znovupripojení do systému. Pri tom musíme brať do úvahy aj to, že daný agent sa nemusí pripojiť na tej istej lokálnej sieti a tak isto v tej chvíli nemusí byť agent, ktorý ma správu preňho uloženú, aktívny. Ďalším zlepšením situácie by mohol byť takzvaný „*Proxy Bye“*, ktorý vychádza z toho, že platnosť adresy vypadnutého agenta, ktorú ohlasuje na svojej lokálnej sieti vyprší skôr. Ak vyprší platnosť adresy na lokálnej sieti, môžeme predpokladať, že ani žiadna iná adresa na tohto agenta nebude platná a preto by mohlo byť úlohou GW agenta danej siete ohlásiť neprítomnosť toho spadnutého ostatným. Toto riešenie môže v sebe niesť isté riziká, ktoré bude treba doriešiť, ako napríklad čo ak sa vzápätí spadnutý agent pripojí, ohlási svoju prítomnosť správou *Hello* a na to ostatným príde správa *Proxy Bye,*  ale riešenie tohto by nemalo byť také komplikované. Samotná riešenie *Proxy Bye* vie zmenšiť hluché miesto až na pol minúty.

## Riešenia jednotlivých komponentov

Táto sekcia obsahuje podrobnejší opis implementácie komponentov, popísané zaujímavé metódy, poprípade ukážky z nich. Budeme sa venovať aj algoritmom, ktoré boli vytvorené pre toto riešenie.

### AgentInfo

Je dátový typ, ktorý obsahuje dva druhy informácií o agentovi. Dátový typ AgentDescription a hashmapu z reťazca na TransportAddressProperties.

#### AgentDescription

Obsahuje informácie, ktoré definujú agenta. Obsahuje tieto položky:

* Reťazec name – jednoznačný identifikátor agenta, používame ho pri odosielaní štandardných správ
* Zoznam reťazcov services – je to pole reťazcov, ktoré obsahuje informáciu o tom, aké služby vie tento agent ponúknuť
* Zoznam reťazcovt flags – pole reťazcov, ktoré obsahujú príznaky (flags) agenta

#### TransportAddressProperties

Tento dátový typ ukladá informácie o jednej adrese, ktorou možno kontaktovať agenta. Keďže ich môže byť viac, ukladajú sa v AgentInfo v hashovacej tabuľke. Jeho položky sú:

* celé číslo metric – metrika danej adresy. Adresy, ktoré pochádzajú z lokálnej siete majú metriku 0, aby sa dalo identifikovať, že z nej pochádzajú. Agent si neukladá hocijakú inú adresu na daného agenta, pokiaľ už existuje adresa s metrikou 0. Je to zabezpečenie proti prípadu, že na jednej lokálnej sieti je viac ako jeden GW agent. V takom prípade si oznamujú aj agentov, ktorí patria do ich spoločnej lokálnej siete.
* čas validUntil – informácia o tom, do kedy je táto adresa platná
* DiscoveryService \*sourceDs – toto je jediná informácia, ktorá sa neoznamuje v *Notify*. Slúži na to, aby DiscoveryService na lokálnej sieti neohlasoval naspať adresu, ktorú sám zistil. Je to spôsob, ako sa vyhnúť cyklu s nepravdivými a zbytočnými informáciami. Keď rozparsuje *Notify* do sourceDs uloží adresu na seba. Je to smerník a nie príznak kvôli budúcim plánom so systémom, keď bude existovať viac druhov DS a MTS na jednej platforme.
* Zoznam reťazcov origins – obsahuje zoznam GW agentov, ktorý ohlasujú túto adresu. Význam tejto položky bude vysvetlený v sekcii XML Message.

### Message

Pri správach je dôležité rozlišovať či prišli z DS alebo MTS. Hlavný rozdiel medzi nimi je, že správy v DS sa posielajú v JSON a správy z MTS v XML. Napriek tomu, že tieto správy majú niektoré položky podobné, ich štruktúry sú viac rozdielne.

#### JSON správy

Odosielajú sa priamo príjemcovi a preto obsah správy je priamo prístupný a netreba k správe pridávať informácie o jej ceste, odosielateľovi a príjemcoch, pretože sú určené pre všetkých v multicastovej skupine. Správy, ktoré sa odosielajú v tomto formáte sú *Notify, Hello* a *Bye.*

Štruktúra *Notify* obsahuje položky:

* type - typ správy
* agents – tu sa nachádzajú agenti, ktorých oznamujeme. V tomto prípade sú to platformové agenty. V prípade, že je odosielateľ aj Gateway agent, v tejto položke oznamuje aj agentov, o ktorých preposielanie sa stará. Položka obsahuje samozrejme aj informácie o všetkých agentoch.
* gwAgents – je to len zoznam adries na Gateway agentov, ktorí sú aktívni v systéme. Odosielajú sa všetkým a všetci si túto informáciu uložia, ale aktívne ju využívajú len ostatní GW agenti. Potreba pre všetkých, aby táto informácia bol uložená je v prípade, že sa z bežného agenta stane GW agent.

Štruktúra *Hello* a *Bye* je rovnaká:

* type - ako u *Notify*
* address – adresa platformy. Podľa typu správy sa pridá do zoznamu alebo sa zo zoznamu odstránia všetky položky s ňou súvisiace (okrem prípadu zoznamu GW agentov).

#### XML správy

Z dôvodu putovania týchto správ po internete, vznikla potreba komplikovanejšieho spôsobu posielania týchto správ. V prvom rade sa tieto správy posielajú v obálke (*Envelope)*, ktorá je tiež v XML. Pre úplnosť popíšeme štruktúru oboch.

#### Envelope

Štruktúra obálky je pre všetky typy správ rovnaká až na jednu položku. Keďže v prípade XML správ, sa všetky správy okrem *Standard Message* posielajú iba GW agentom, líšia sa tieto správy v tom, že *Standard Message* má navyše položku *recipients*. Štruktúra obálky vyzerá nasledovne:

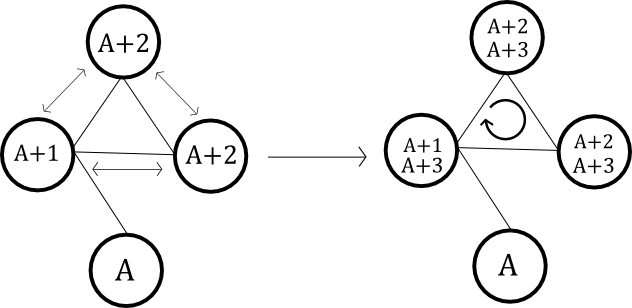
* messageType - to isté, ako type u JSON správy.
* sender – odosielateľ správy. V prípade *Standard Message* je to identifikátor agenta (meno), u ostatných je to adresa platformy
* recipients – mená agentov, ktorým má byť doručená táto obálka. Tento zoznam nie je ekvivalentný s menami agentov, ktorým má byť doručená správa. //pridaj obrázok
* message – do tejto položky sa ukladá samotná XML správa.

#### XML Message

V prípade samotnej správy v XML tvare opäť záleží na tom, akého typu je. Keďže správy, ktoré MTS dostáva od platformy už očakávame v takom tvare, aby sa jednoducho vložili do obálky, v tejto práci sa jej štruktúre nevenujeme, pretože to je mimo rámca tejto práce. *Hello* a *Bye* sú obdobné s JSON správami a preto spomenieme iba rozdiel *Notify* oproti JSON variantu.

Správy sme sa snažili vytvoriť jednotné pre JSON a XML, ale keďže XML *Notify* musí v sebe zahŕňať dodatočné informácie o routovaní, bola doňho pridaná položka *origins* ku každej transport adrese. Je určená iba GW agentom a hovorí o tom, aké túto adresu ohlasovali. Je dôležitá preto, aby sme sa vyhli cyklom v routovaní, ktoré by mali za následok, že by sa adresa ohlasovala vždy s o jedno väčšou metrikou a musela by byť uložená, lebo by sa považovala za úplne novú adresu (viď. obrázok 4). Takto je každá adresa jednoznačne definovaná pomocou zoznamu *origins*. Adresu neohlasujeme GW agentovi, ktorý sa nachádza v tomto zozname. Týmto zabezpečíme, že sa ostatní dozvedia všetky možné cesty k agentovi a zároveň v nich nevznikne cyklus.

Príklad cyklu: Po tom, ako agent, ktorý oznámi ostatným adresu s metrikou A+1, bude stav vyzerať ako na ľavej strane obrázku č.4. V ďalšom kole *Notify*, ak by prebiehali súčasne, bude vyzerať stav ako na pravej strane. Tam môžeme pozorovať, že jeden agent má adresy s metrikami A+1 a A+3. To je chybné, pretože adresa A+3 je dôsledkom spomínaného cyklu.



Celý obsah tohto typu je obalený v XML elemente *notifyInfo*.

### DiscoveryService

Jeho dôležité členské premenné sú smerník na platformu ktorej sú súčasťou, kvôli zoznamu agentov a smerník na UDP socket využívaný na obsluhu UDP multicast. Jeho metódy slúžia na odosielanie, prijímanie a spracovávanie správ typu *Hello, Bye*  a *Notify.*

Vo svojom konštruktore najskôr spracuje adresu na platformu. Potom sa UDP socket menom m\_udpSocket pripojí na nami zvolený port, kde počúva na všetky IPv4 adresy. UDP socket je typu QUdpSocket, ktorý ponúka Qt. Následne sa pripojí do multicastovej skupiny, na ktorú chodia vyššie spomenuté správy. Po tom, ako je pripojený na túto skupinu, pošle na ňu správu *Hello*. Keď na náš socket príde nejaký datagram, vyšle signál readyRead. Na spracovanie tohto signálu využívame slot processPendingDatagrams. Úlohou tohto slotu je spracovať prichádzajúce dáta do podoby, s ktorou sa pohodlne pracuje a bola pre nás kľúčová – QByteArray. Toto bajtové pole sa odošle metóde handleDatagram. Ten z bajtového poľa spraví, QJsonDocument, v ktorého tvare bola odoslaná správa a zistí z nej správu. V tejto metóde sa potom rozhodne, ako sa správa spracuje podľa jej typu. Správu *Notify* spracováva samotný DS, ale *Hello, Bye* spracováva platoforma, pretože tieto správy sa spracovávajú rovnako pri DS aj MTS, kým spracovanie *Notify* prebieha na nich inak.

Spracovávanie *Notify* prebieha tak, že z nášho QByteArray-u spravíme QVariantMap. Čo je mapa s kľúčom QString a hodnotou QVariant. Kľuče sú *type* (typ správy), *agents*(informácie o agentoch, ktorí sú v tejto správe ohlásení) a *gwAgents* (adresy súčasných GW agentov v systéme) o ktorých sme písali v sekcii 6.3.1. Informácie z týchto sekcií sa spracujú tak, že nové informácie sa pridajú a staré sa aktualizujú. O spracovávaní správ *Hello* a *Bye* budeme hovoriť v sekcii *Platform*.

Pri odosielaní *Notify* najprv odošleme informácie spomenuté v sekcii AgentInfo o platformových agentoch, tie sa posielajú vždy. Ak je ale odosielateľ GW agent, v tom prípade oznámi aj agentov, ktorých DS nezískal z lokálnej siete a teda sú zo vzdialených lokálnych sietí. Ďalšou úlohou DS je ohlásiť ostatných GW agentov v systéme. Jednotlivé položky, ktoré sa posielajú sú popísané v odseku o spracovávaní tejto správy. Ukážka časti kódu oznamovania vzdialených agentov je v ukážke č.1.

for (it = m\_platform->m\_forwardedAgents.constBegin(); it !=

m\_platform->m\_forwardedAgents.constEnd(); ++it){

            QVariantMap addresses;

            auto it2 = it.value().transportAddresses.constBegin();

            while (it2 != it.value().transportAddresses.constEnd()){

                if (it2.value().sourceDs != this)

                    addresses[MY\_ADDRESS] = fromProperties(it2.value(), false).toVariantMap();

                ++it2;

            }

            if (addresses.empty())

                continue;

            agent[NAME] = QVariant(it.value().desription.name);

            agent[SERVICES] = QVariant(it.value().desription.services);

            agent[FLAGS] = QVariant(it.value().desription.flags);

            agent[TRANSPORT\_ADDRESSES] =  addresses;

            agentInfo[it.value().desription.name] = agent;

}

O posielanie *Hello* a *Bye* sa stará metóda writeStatusMessage, ktorá má za vstupné parametre QString typ správy, keďže tieto správy sa líšia iba typom.

### MessageTransportService

Má štyri členské premenné. Prvou je smerník na platformu m\_platform z rovnakého dôvodu ako pri DiscoveryService. Druhá je HTTP server – m\_server, ktorý nám ponúka Tufão. Tretiu a štvrtú premennú nám taktiež ponúka Tufão, sú to HTTP *request* a *response – m\_request, m\_response*, teda žiadosť a odpoveď. Jeho funkčnosť sa orientuje na prijímanie a odosielanie správ pomocou HTTP. Okrem toho však virtuálne ponúka funkčnosť DS, keďže práve v MTS sa riešia správy medzi GW agentami *Hello, Bye* a *Notify* Štandardné správy sa posielajú cez MTS aj obyčajným agentom.

Jediné, čo MTS vo svojom konštruktore spraví je, že si inicializuje premennú m\_platform. pripojí signál m\_server-u requestReady na jeho slot handleRequest a zapne server, aby počúval na porte 22222 na hocijakú adresu. Na rozdiel od DS, vo svojom konštruktore nepošle *Hello*, pretože GW agenty sa načítavajú až v konštruktore platformy, takže by sa táto správa nikam neposlala, pretože by nemala prijímateľov.

Ako sme rozoberali v sekcii o XML správach, každá takáto správa sa odosiela v XML obálke. writeHttpMessage má na starosti jej zloženie podľa typu správy, ktorá sa v tejto metóde do obálky aj vkladá. Správa je jeden zo vstupných parametrov. Na konci tejto metódy sa obálka odošle správnym agentom cez HTTP POST, podľa adries spĺňajúc podmienku o jej odoslaní podľa návrhu riešenia v predchádzajúcej kapitole. Samotné správy, ktoré sa v nej skladajú pochádzajú buď od platformy už v konečnom tvare(štandardné správy), alebo ich zloží samotné MTS (*Hello, Bye, Notify*).

*Hello* a *Bye* vytvára metóda sendHttpStatusMessage podľa ich štruktúry, ktorá potom na konci zavolá writeHttpMessage a odovzdá jej vytvorenú správu pre GW agentov.

*Notify* sa vytvára v sendHttpNotify taktiež podľa predchádzajúcich požiadaviek a štruktúry a ako predtým, správa sa odovzdá writeHttpMessage, kde príjemcami sú opäť GW agenty.

Keď príde HTTP Request na náš server, reaguje naň vyslaním sigálu requestReady, na ktorý sme v konštruktore napojili slot handleRequest. V tomto slote si uložíme request aj response do príslušných členských premenných MTS a napojíme signál „end“ requestu na slot processHttpMessage. Signál end sa vyšle po tom, ako je request kompletný. Podľa typu a informácií v obálke, ktorú získa z m\_request sa processHttpMessage rozhodne ako so správou naloží:

* *Standard Message* – ak je správa podľa obálky určená pre nejakého platformového agenta, vyšle signál *messageReady*, že je správa preňho pripravená a odstráni ho zo zoznamu príjemcov. Ak zoznam príjemcov nie je potom prázdny, ostávajúcich príjemcov a správu odovzdá vyššie spomenutej metóde writeHttpMessage, ktorá s ňou korektne naloží.
* *Hello/Bye –* správa sa posunie platforme, aby si upravila zoznam agentov alebo aj GW agentov podľa jej obsahu
* *Notify* – telo správy sa odovzdá metóde processXmlNotify, ktorý rozparsuje správu podľa jej štruktúry a aktualizuje alebo pridá agentov či GW agentov, podľa jej obsahu

Na obrázku č.6 ukazujeme, ako sa ukladá transport adresa agenta. Najprv sa zistí či už táto adresa existuje, ak áno, tak sa aktualizuje, inak sa vloží nová. Práve tu môžeme vidieť využitie *origins*, ktoré je jednoznačný identifikátor cesty.

agents[info.desription.name].desription = info.desription;

        auto it = info.transportAddresses.begin();

        bool metricExists = false;

        while (it != info.transportAddresses.end()){

            if (agents[info.desription.name].transportAddresses[it.key()].origins == it.value().origins){

                agents[info.desription.name].transportAddresses[it.key()] = it.value();

                metricExists = true;

            }

            if (it == info.transportAddresses.constEnd() && !metricExists){

                agents[info.desription.name].transportAddresses.insertMulti(it.key(), it.value());

            }

            ++it;

        }

Na obrázku č.7 môžeme opäť vidieť úlohu *origins* v kontexte vyhnutiu sa cyklu oznamovania adresy agenta tým, že adresu neposielame agentovi, ktorý už je s touto adresou oboznámený.

writer->writeStartElement(TRANSPORT\_ADDRESSES);

    for (auto it2 = info.transportAddresses.constBegin();

         it2 != info.transportAddresses.constEnd(); ++it2){

        if (it2.value().route.contains(recipient))

            continue;

        writer->writeStartElement("route");

        writer->writeTextElement(METRIC, QString::number(it2.value().metric));

        writer->writeTextElement(VALID\_UNTIL, it2.value().validUntil.toString());

        writer->writeTextElement("transportAddress", MY\_ADDRESS + "/forwardedAgents");

        writer->writeTextElement("origins", it2.value().origins.join(" "));

        writer->writeEndElement(); // route

    }

    writer->writeEndElement(); // transportAddresses

### Platform

Platforma bola naprogramovaná kvôli testovaniu DS a MTS, nie je to jej konečná implementácia, ale iba jej simulácia. Okrem toho, že obsahuje samotné DS a MTS, medzi jej členské premenné patria tri časovače, ktoré podrobnejšie rozoberieme nižšie. Ďalej v sebe zahŕňa príznak či je GW a štyri zoznamy agentov. Prvé dva zoznamy sú typu QHash z QStringu na AgentInfo a sú to zoznamy m\_platformAgents(platformové agenty) a m\_forwardedAgents (ostatné agenty). Tretí zoznam je QStringList m\_knownGatewayAgent, ktorý obsahuje adresy gateway agentov za určité časové obdobie a je ich len obmedzený počet. Tieto agenty nemusia byť aktívne. Posledný zoznam je QStringList m\_gatewayAgents, kde sú uložené adresy aktívnych GW agentov v danej dobe.

Platforma vo svojom konštruktore inicializuje premenné m\_ds a m\_mts. V tomto kroku získajú adresu pre ich smerníky na ňu. Po inicializácii načíta zo súboru zoznam m\_gatewayAgents

# Citacie

[1] Artificial Intelligence, Russel a Norvig, 2009

[2] Multiagent Systems, Katia P. Sycara, 1998

[3] Foundation For Physical Intelligent Agents. FIPA ACL Message Structure Specification. Technical report, FIPA, 2002

[4] Yannis Labrou and Tim Finin. A Proposal for a new KQML Specification A

Proposal for a new KQML Specification. Discourse, (TR CS-97-03), 1997.

[5] Types and Priorities of Multiagent System Interactions, Martin Ngobye1, Wouter T. de Groot, and Theo P. van der Weide, 2009

[6] Computer Networks (5th Edition), Tannenbaum and Wetherall

[7] RIP Version 2, G. Malkin, 1998

[8] OSPF Version 2, J. Moy, 1991

[9] Distributed dynamis QoS-aware routing in WDM optical networks, S. Dharma Rao, C. Siva Ram Murthy, 2005

[10] PalliaSys: agent-based proactive monitoring of palliative patients, A.Moreno, A.Valls, D.Riaňo,

[11] <http://ii.fmph.uniba.sk/~siska/lcp>

[12] http://jade.tilab.com/

[13] <http://fipa.org/resources/livesystems.html>

[14] <http://qt-project.org/doc/qt-4.8/qmake-manual.html>

(15) <http://vinipsmaker.github.io/tufao/ref/1.x/>

[16] http://bittorrent.org/beps/bep\_0005.html

(<http://www.fipa.org/repository/aclspecs.html>)

# 9.Distributed dynamic QoS-aware routing in WDM optical networks

10 <http://deim.urv.cat/~itaka/Publicacions/iwpaams05.pdf>

11 <http://tools.ietf.org/html/rfc2453>

12 http://tools.ietf.org/search/rfc1247

<http://www.sciencedirect.com/science/article/pii/S1389128604003123>

http://www.aaai.org/ojs/index.php/aimagazine/article/view/1370/1270

1. Pozn.: Preposielanie nastane v prípade, že MTS je súčasťou GW platformy [↑](#footnote-ref-1)