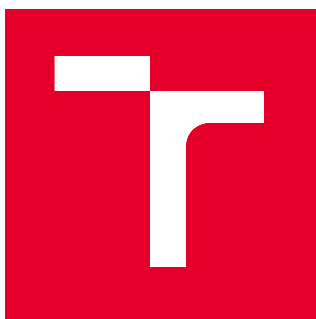


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

DEPARTMENT OF CONTROL AND INSTRUMENTATION

PLÁNOVÁNÍ TRAS PRO MULTIAGENTNÍ ROBOTICKÉ SYSTÉMY

PATH PLANNING FOR MULTI-AGENT ROBOTIC SYSTEMS

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE Libor Macák

AUTHOR

VEDOUCÍ PRÁCE Ing. Tomáš Lázna

SUPERVISOR

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ | FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Automatizační a měřicí technika**

Ústav automatizace a měřicí techniky

Student: Libor Macák ***ID:*** 211158

Ročník: 3 ***Akademický rok:*** 2021/22

NÁZEV TÉMATU:

Plánování tras pro multiagentní robotické systémy

POKYNY PRO VYPRACOVÁNÍ:

Cílem bakalářské práce je seznámit se s existujícími algoritmy pro plánováni tras (path planning) v mobilní robotice s důrazem na metody, které umožňuji bezkolizní provoz více strojů v rámci sdíleného pracovního prostoru. Zvolený algoritmus bude implementován jako balíček pro ROS 2 a v tomto frameworku také otestován.

1. Proveďte rešerší plánovacích algoritmů pro multiagentní systémy.
2. Seznamte se s frameworkem Robot Operating System 2 (ROS 2).
3. Zvolte jeden z plánovacích algoritmů a implementujte jej jako balíček pro ROS 2.
4. Implementujte zjednodušený model robotu s všesměrovým podvozkem a vizualizujte funkcionalitu vytvořeného balíčku pomocí nástroje RViz.

DOPORUČENÁ LITERATURA:

[1] DESARAJU, Vishnu R. a Jonathan P. HOW. Decentralized path planning for multi-agent teams with complex constraints. Autonomous Robots [online]. 2012, 32(4), 385-403 [cit. 2021-9-14]. ISSN 0929-5593. DOI: 10.1007/s10514-012-9275-2

Termín zadání: 7.2.2022 ***Termín odevzdání:*** 23.5.2022

Vedoucí práce: Ing. Tomáš Lázna

doc. Ing. Václav Jirsík, CSc.
předseda rady studijního programu

UPOZORNĚNÍ:

Autor bakalářské práce nesmí při vytváření bakalářské práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

Abstrakt

Práce se zaměřuje na vývoj balíčku Robot Operating System 2 umožňující multiagentní plánování a klade hlavní důraz na bezkolizní provoz více agentů najednou. Dále vysvětluje některé základní pojmy týkající se Robot Operating System 2. Práce také přibližuje problematiku multiagentního plánování, vyčtením základních teoretických pojmů a popisem některých používaných algoritmů.

Klíčová slova

Mapy obsazenosti, multiagentní plánovací algoritmy, A*, ROS, ROS2, ROS2 balíček, Rviz.

Abstract

This work concentrates on developing a Robot Operating System 2 package that enables multiagent path planning and puts main emphasis on non-collision traffic between more agents at the same time. Next it explains some basic concepts about Robot Operating System 2. This work also approaches multiagent path planning problem with listing some basic theoretical concepts and it specifies some used algorithms.

Keywords

Occupancy maps, multiagent path planning algorithms, A*, ROS, ROS2, ROS2 package, Rviz.

Bibliografická citace

MACÁK, Libor. *Plánování tras pro multiagentní robotické systémy* [online]. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedoucí práce: Tomáš Lázna.

Prohlášení autora o původnosti díla

Jméno a příjmení studenta: *Libor Macák*

VUT ID studenta: *211158*

Typ práce: *Bakalářská práce*

Akademický rok: *2021/22*

Téma závěrečné práce: *Plánování pro multiagentní robotické systémy*

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

.
V Brně dne: 22. května 2022 ---------------------------------------

podpis autora

Poděkování

Děkuji vedoucímu bakalářské práce Ing. Tomáši Láznovi za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé diplomové práce.

V Brně dne: 22. května 2022 --

podpis autora

Obsah

SEZNAM OBRÁZKŮ **8**

SEZNAM TABULEK **9**

ÚVOD **10**

1. MULTIAGENTNÍ PLÁNOVÁNÍ **11**

1.1 MAPA 11
1.2 PLÁNOVACÍ ALGORITMY 12
1.2.1 A algoritmus* *14*
1.2.2 Local-Repair A algoritmus* *15*
1.2.3 Cooperative A algoritmus* *15*
1.2.4 Windowed Hierarchical Cooperative A algoritmus* *16*
1.2.5 Operator Decomposition *17*
1.2.6 Distributed Path Consensus algoritmus *18*
1.2.7 Optimal Reciprocal Collision Avoidance algoritmus *18*

2. ROS2 ..... **20**

2.1 ROBOT OPERATING SYSTEM 20
2.2 ZÁKLADNÍ POJMY 20
2.2.1 ROS 2 Nodes *20*
2.2.2 ROS 2 topics *20*
2.2.3 ROS 2 services *21*
2.2.4 ROS 2 parameters *22*
2.2.5 ROS 2 actions *22*
2.2.6 ROS 2 packages *22*
2.2.7 ROS 2 workspace *23*
2.2.8 ROS 2 rqt_console *23*

3. REALIZACE BALÍČKU ROS2 ..... **24**

3.1 AGENT NODE 24
3.1.1 Ragent_node.cpp *24*
3.1.2 Ragent_obj.h *25*
3.1.3 Ragent_obj.cpp *26*
3.2 PLANNER NODE 28
3.2.1 Rmotion_planner_node.cpp *28*
3.2.2 Rmotion_planner_obj.h *29*
3.2.3 Rmotion_planner_obj.cpp *29*
3.3 SERVICE A MESSAGE SOUBORY 33
3.4 LAUNCH SOUBOR 34
3.5 CMAKELIST SOUBOR A PACKAGE SOUBOR 35
3.6 VÝSLEDEK 36

4. ZÁVĚR **39**

LITERATURA **40**

SEZNAM PŘÍLOH **43**

SEZNAM OBRÁZKŮ

1-1 Příklad mapy obsazenosti, kde bílá reprezentuje volný prostor, černá/tmavě šedá obsazený prostor a šedá prostor o kterém nemáme informace [2]. 11
1-2 Příklad upravené mapy obsazenosti, kde bílé body mřížky reprezentují volný prostor a červené body mřížky obsazený prostor [3]. 12
1-3 Ukázka jednoduché mapy [6]. 13
1-4 Roboti se nacházejí v počátečních pozicích [8]. 13
1-5 Roboti se nacházejí v požadovaných cílech [8]. 14
1-6 Stromové vyhledávání A* algoritmu [11]. 15
1-7 Ukázka neřešitelné situace pro LRA* a CA* algoritmy [8]. 16
1-8 Ukázka WHCA* algoritmu [17]. 17
1-9 Ukázka Distributed Path Consensus algoritmu pro 6 agentů [14]. 18
1-10 Ukázka Optimal Reciprocal Collision Avoidance [16]. 19
2-1 Ukázka komunikace node pomocí témat (topic) [19]. 21
2-2 Ukázka komunikace node pomocí služeb (service) [19]. 21
2-3 Ukázka komunikace node pomocí akcí (actions) [19]. 22
2-4 Ukázka rqt_graph [16]. 23
3-1 Ukázka vygenerování volného prostoru. 37
3-2 Ukázka vygenerování prostoru s překážkou. 37
3-3 Ukázka vygenerování dvou agentů. 38
3-4 Možná podoba reálné verze agenta 20. 38

SEZNAM TABULEK

1.1 Pozice ve kterých se roboti nacházejí v jednotlivých krocích [8] 14

ÚVOD

Multiagentní plánování je jedním ze základních problémů mobilní robotiky. Pro jeho řešení je potřeba znát mapu prostředí a zvolit vhodný plánovací algoritmus.

Následující bakalářská práce se zaměřuje na vytvoření multiagentního plánovacího balíčku pro ROS2 (Robot Operating Systém 2) a jeho vizualizaci pomocí nástroje Rviz.

Bakalářská práce se skládá ze tří částí. První část se týká teoretického rozboru multiagentního plánování. Druhá část je o ROS2 a jeho základních pojmech a třetí část popíše samotnou realizaci multiagentního plánovacího balíčku.

1. MULTIAGENTNÍ PLÁNOVÁNÍ

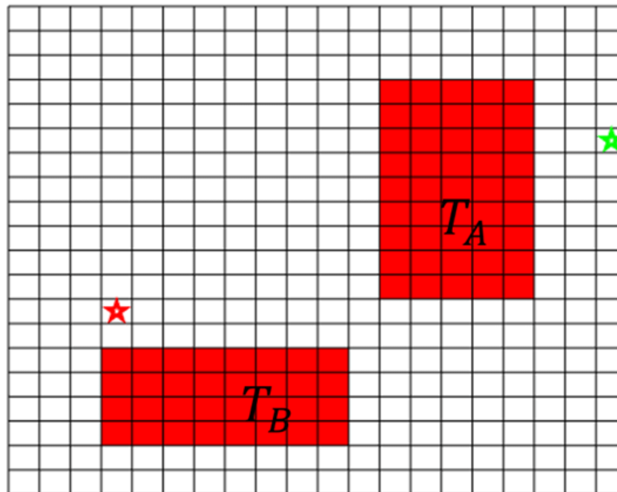
V následující kapitole je seznámení s pojmy pro multiagentní plánování a zároveň popis některých algoritmů pro multiagentní plánování.

1.1 Mapa

Aby bylo možné pro agenta (robota) nalézt optimální trasu do cílové polohy je nutné, aby agent znal mapu prostoru, ve kterém se nachází.

Pro následující bakalářskou práci byla zvolena mapa objemová. Tento typ mapy je vhodnějším, protože umožňuje přesnější reprezentaci 2D mapy vnitřního prostoru. Celý prostor je totiž možné interpretovat jako mřížku bodů obsazenosti a plánovací algoritmus má poté jednoduchý způsob jak zvolit optimální trasu [1].

Obrázek 1-1 Příklad mapy obsazenosti, kde bílá reprezentuje volný prostor, černá/tmavě šedá obsazený prostor a šedá prostor o kterém nemáme informace [2].



Obrázek 1-2 Příklad upravené mapy obsazenosti, kde bílé body mřížky reprezentují volný prostor a červené body mřížky obsazený prostor [3].

1.2 Plánovací algoritmy

Plánovací algoritmy mají za úkol nalézt posloupnost akcí, která nalezne cestu mezi bodem, kde se robot aktuálně nachází a zvoleným koncovým bodem. Důraz je kladen na nalezení optimální cesty, dále pak na rychlost hledání a celkovou složitost algoritmu.

Multiagentní plánovací algoritmy patří mezi informované plánovací algoritmy a jejich hlavní výhodou je dodatečná informace ohledně zvoleného cíle, kterou informované algoritmy využívají. Tato informace se nazývá heuristika. Heuristika je odhad ceny mezi aktuálním místem, kde se robot nachází a požadovaným cílem. V Euklidovském prostoru můžeme za dostačující heuristiku považovat přímou vzdálenost vzdušnou čarou mezi aktuální pozicí robota a požadovaného cíle.

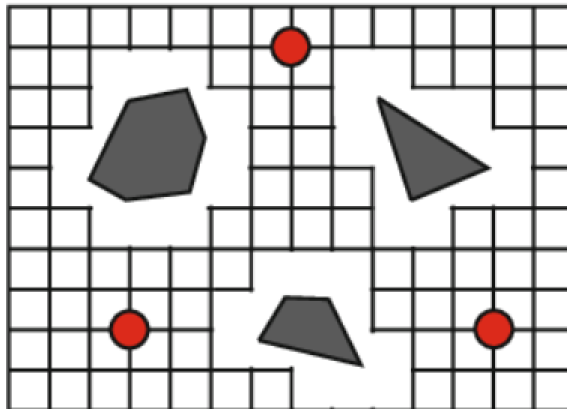
Tato heuristika je poté dána vztahem

$$h = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}\,, \qquad (1.1)$$

kde x_2 a y_2 reprezentují souřadnice požadovaného cíle a x_1 a y_1 reprezentují aktuální souřadnice robota [4].

Agenty, kteří využívají multiagentní plánování, je možné dále dělit na dva typy. První jsou roboti reaktivní, kteří pro rozhodování používají pouze své senzory a druhým typem jsou roboti plánovací, kteří k rozhodování používají mapy. Pro tuto práci jsou významnějšími roboti plánovací, protože jejich navigace bude probíhat v simulovaném prostředí reprezentující uzavřenou místnost. Budou mít přístup ke kompletní mapě prostředí a jejich akce na sobě budou nezávislé (tzn. jejich hlavním cílem bude se bez kolize dostat z počátečního bodu do požadovaného cíle) [5].

Ukázka jednoduché mapy s rozmístěnými roboty je zobrazena na obrázku 1-3.



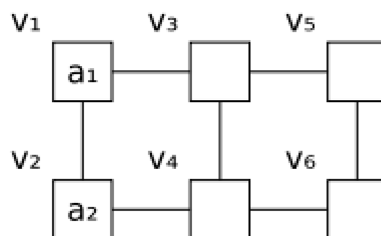
Obrázek 1-3 Ukázka jednoduché mapy [6].

Multiagentní plánování můžeme také nazvat zobecněným jednoagentním plánováním. Na mapě se současně nachází více robotů, kteří nesmějí sdílet stejnou pozici. Každý z nich se současně pohybuje po uzlech mapy, dokud se nedostane do požadovaného cíle. Když si dva roboti potřebují vyměnit pozice, nesmí dojít k následující situaci – Robot 1 se nachází na pozici A a v dalším kroku se přemístí do pozice B a zároveň se Robot 2 nachází na pozici B a přemístí se do pozice A. V takovém případě se robot, pro kterého by to bylo přípustnější, nejdříve přemístí do pozice C a až poté do cílové pozice.

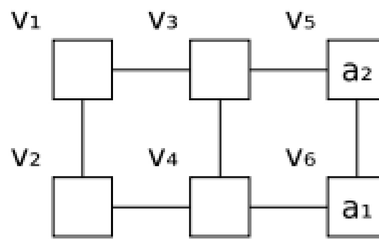
Multiagentní plánování můžeme zapsat následujícím vzorcem

$$\Sigma = (G, A, \lambda_0, \lambda_+) \ , \qquad (1.2)$$

kde G reprezentuje mapu, A je konečný počet robotů, λ_0 je matice počátečních souřadnic robotů a λ_+ je matice souřadnic požadovaných cílů [7]. Příklad řešení multiagentního plánování se dvěma roboty je zobrazen na obrázcích 1-4, 1-5 a tabulce 1.1.



Obrázek 1-4 Roboti se nacházejí v počátečních pozicích [8].



Obrázek 1-5 Roboti se nacházejí v požadovaných cílech [8].

Tabulka 1.1 Pozice ve kterých se roboti nacházejí v jednotlivých krocích [8]

	λ_0	λ_1	λ_2	$\lambda_3 = \lambda_+$
a_1	v_1	v_3	v_5	v_6
a_2	v_2	v_4	v_3	v_5

Při řešení problému multiagentního plánování můžeme v reálném světě narazit na nově nalezené statické a dynamické překážky, které se nenacházejí na původní mapě prostředí. Hlavní otázkou je tedy vždy proveditelnost řešení a dále pak jeho optimalizace. Plánovací algoritmy můžeme rozdělit na spojené a oddělené. Spojené algoritmy řeší plánovací problém jako celek a oddělené algoritmy tento problém dělí na menší podproblémy a následně řeší kolize, pokud nějaké nastanou [9].

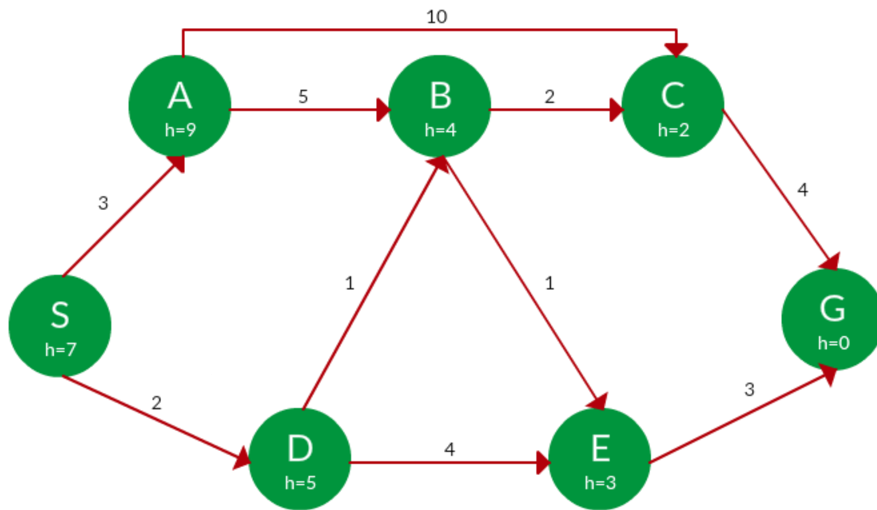
1.2.1 A* algoritmus

Tento algoritmus patří mezi plánovací algoritmy oddělené a ukázka jeho stromového vyhledávání je zobrazena na obrázku 1-6.

Základem A* algoritmu je spojení Uniform Cost Search algoritmu a Greedy Search algoritmu. Tím vznikne funkce f, která je součtem funkce ceny uzlu g a heuristiky h. Z příkladu na obrázku 1-11 je vidět, že algoritmus z počátku (zde S) upřednostní sousední uzel D, protože hodnota jeho funkce f je nižší než pro uzel A. Tento krok by byl stejný i pro Greedy Search algoritmus, ale rozdíl mezi těmito algoritmy je patrný až u druhého kroku algoritmu. Nyní se algoritmus rozhoduje mezi uzly B a E. Greedy Search algoritmus by vybral uzel E jako vhodnější, protože má nižší hodnotu heuristiky h. A* algoritmus však upřednostní uzel B, protože má nižší hodnotu funkce f (f_E = 2+4+3 = 9 > f_B = 2+1+4 = 7). Tímto stylem algoritmus pokračuje, dokud nenalezne požadovaný cíl.

Abychom zaručili, že nalezená cesta bude optimální, musí být odhadovaná hodnota heuristiky h vždy nižší než skutečná hodnota heuristiky h*. Pokud pro výpočet odhadované heuristiky použijeme vzorec (1.1), bude tato podmínka splněna. Pokud

bude h > h*, nebude nalezená cesta optimální, ale bude mnohem rychlejší [4] [10].



Obrázek 1-6 Stromové vyhledávání A* algoritmu [11].

1.2.2 Local-Repair A* algoritmus

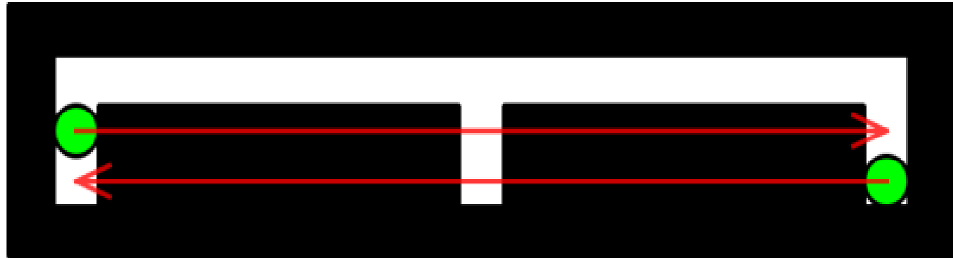
Local-Repair A* algoritmus neboli LRA* algoritmus patří k odděleným plánovacím algoritmům a je také jedním z nejjednodušších.

Jeho princip spočívá v tom, že trasa každého robota je plánována samostatně pomocí A* algoritmu a při samotném plánování jednotlivých tras algoritmus ignoruje ostatní roboty, pokud přímo nesousedí s robotem, jehož trasa je právě plánována. Pokud během hledání tras dojde ke kolizi, LRA* algoritmus se znovu spustí a považuje kolizní místo za překážku, kterou se snaží obejít. Jedním z problémů tohoto algoritmu je nutnost opakovaného běhu A* algoritmu při každé nalezené kolizi a také nulová komunikace mezi roboty. Díky tomu může algoritmus nalézt neřešitelné situace [12].

1.2.3 Cooperative A* algoritmus

Cooperative A* algoritmus neboli CA* algoritmus také patří k odděleným plánovacím algoritmům.

Také plánuje jednotlivé trasy individuálně, ale je rozšířen o rezervační tabulku a čekací funkci. Do rezervační tabulky jsou uloženy pozice robota ve všech krocích naplánované trasy, a při plánovaní trasy pro dalšího robota jsou tyto pozice v daných krocích považovány za překážky. Čekací funkce umožňuje pozastavení robota na *n* kroků. Nevýhodou tohoto algoritmu je striktní požadavek na pořadí, v jakém jsou trasy plánovány. Tomu můžeme částečně zabránit náhodným rozřazením pořadí, v jakém budou jednotlivé trasy plánovány. Stejně jako u LRA* algoritmu mohou nastat neřešitelné situace [12]. Ukázka takové situace je zobrazena na obrázku 1-7.



Obrázek 1-7 Ukázka neřešitelné situace pro LRA* a CA* algoritmy [8].

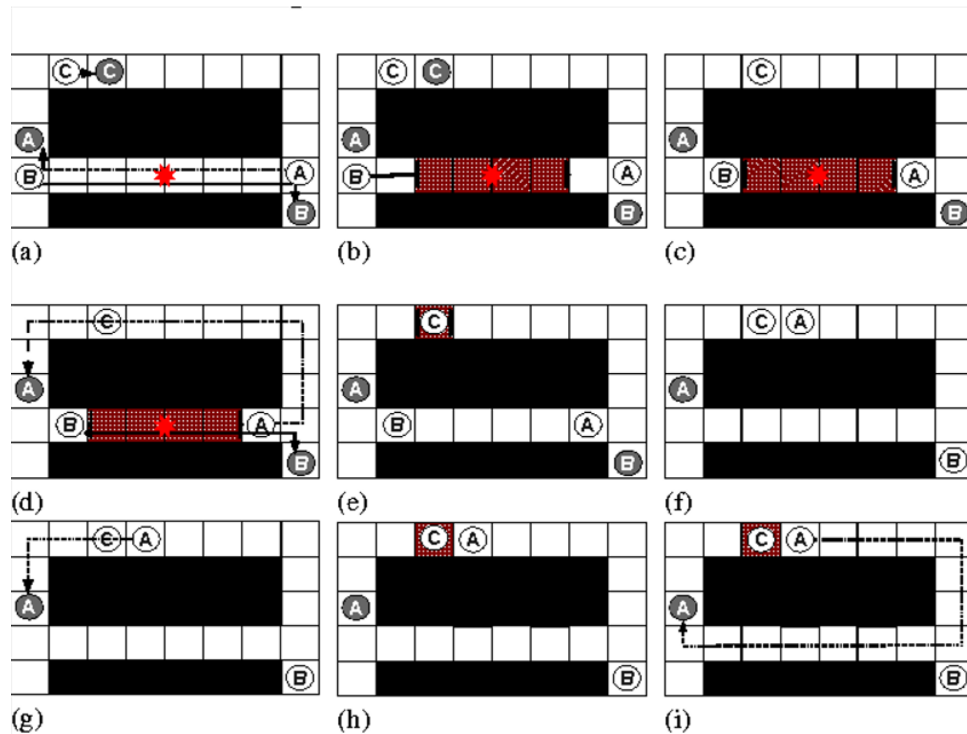
1.2.4 Windowed Hierarchical Cooperative A* algoritmus

Windowed Hierarchical Cooperative A* algoritmus neboli WHCA* algoritmus je rozšířeným CA* algoritmem.

Je rozšířen o okno *w* a hierarchii. Hierarchie je zde tvořena dvěma a více nadřazenými A* algoritmy. Nejnižší vrstva používá nadřazený A* algoritmus pro výpočet nejlepší heuristiky. Pokud je vrstev více, pracuje každá nadřazenější vrstva se zjednodušenější verzí prostředí pro výpočet heuristiky. Abychom zjednodušili výpočet heuristiky a nemuseli při každém volání začínat od začátku, můžeme použít Reverse Resumable A* algoritmus neboli RRA* algoritmus. RRA* algoritmus začíná hledání v cílové pozici robota a hledá jeho počáteční pozici. Algoritmus uchová hledanou trasu v paměti a při novém volání na tuto trasu naváže.

Okno *w* je reprezentováno uzly do vzdálenosti *w* od počáteční pozice robota a všechny uzly se vzdáleností *w* jsou považovány za cílové pozice. WHCA* algoritmus tedy hledá pouze částečná řešení pro všechny roboty. Po jejich nalezení se roboti dají do pohybu, a když dorazí do cílů těchto částečných řešení, vytvoří se nové okno *w* a algoritmus znovu hledá nová částečná řešení, dokud nejsou všichni roboti na hlavních cílových pozicích [12].

Ukázka algoritmu na obrázku 1-8.



Obrázek 1-8 Ukázka WHCA* algoritmu [17].

1.2.5 Operator Decomposition

Operator Decomposition algoritmus neboli OD algoritmus je spojený algoritmus a zlepšený A* algoritmus.

Základním principem OD algoritmu je nalezení cesty pomocí A* algoritmu z počáteční pozice do požadovaného cíle pro všechny roboty. OD algoritmus poté dle daného pořadí přikáže každému robotovi posunout se o jeden krok. Tento proces dělíme na stav standardní a stav přechodný. Standardní stav je takový stav, kdy ani jednomu robotovi nebyl zadán příkaz posunout se o jeden krok a přechodný stav je takový stav, kdy byl aspoň jednomu robotovi zadán příkaz posunout se o jeden krok. Díky tomu se změní složitost hledání trasy z exponenciální na lineární, ale i přesto se jedná o vysoce výpočetně náročný algoritmus.

Výpočetní složitost můžeme snížit využitím okna *w* zmíněného v předchozím algoritmu WHCA* a zavedením detekce nezávislosti (independence detection) neboli ID. ID vytvoří skupiny robotů, u kterých víme, že mezi nimi nedojde ke kolizi a OD algoritmus plánuje jejich trasu najednou. Pokud dojde ke kolizi dvou či více skupin, jsou spojeny do jedné větší skupiny a OD algoritmus přeplánuje trasu [13].

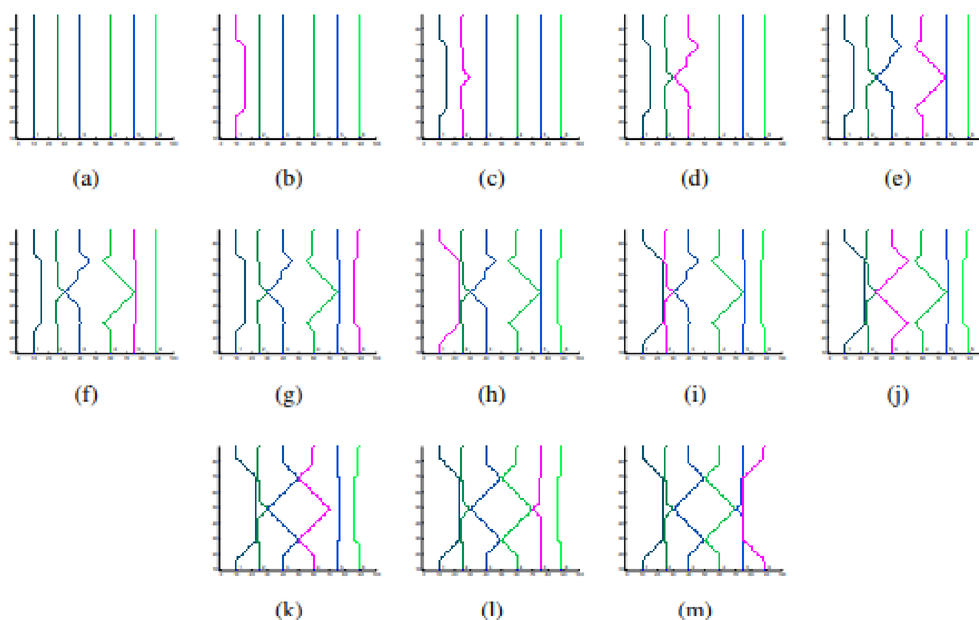
Distributed Path Consensus algoritmus spojuje problémy multiagentního plánování a cílem řízenou navigaci.

1.2.6 Distributed Path Consensus algoritmus

Distributed Path Consensus algoritmus spojuje problémy multiagentního plánování a cílem řízenou navigaci.

Při běžné cílem řízené navigaci je pomocí plánovacího algoritmu zjištěna nejméně náročná cesta grafem, která spojuje počáteční a cílovou pozici.

Tento algoritmus však počítá s tím, že agenti jsou omezeni funkcí v čase, která určuje nutnou maximální vzdálenost mezi agenty. Proto musí být hodnoty jednotlivých bodů v grafu modifikovány dodatečnou časovou proměnou. V 2D prostoru je dostačující hodnota časové proměnné zjištěna jako Euklidovská vzdálenost středů dvou agentů. Agentům je zadána počáteční a cílová pozice a také jejich vzájemná vzdálenost v určitých krocích (například: 1. krok = nedefinováno, 2. krok = 2 kroky, 3. krok = 1 krok, 4. krok nedefinováno atd.). Plánovací algoritmus se poté v jednotlivých iteracích snaží dohledat nejoptimálnější způsob jak tyto podmínky dodržet. Ukázka algoritmu na obrázku 1-9 [14].



Obrázek 1-9 Ukázka Distributed Path Consensus algoritmu pro 6 agentů [14].

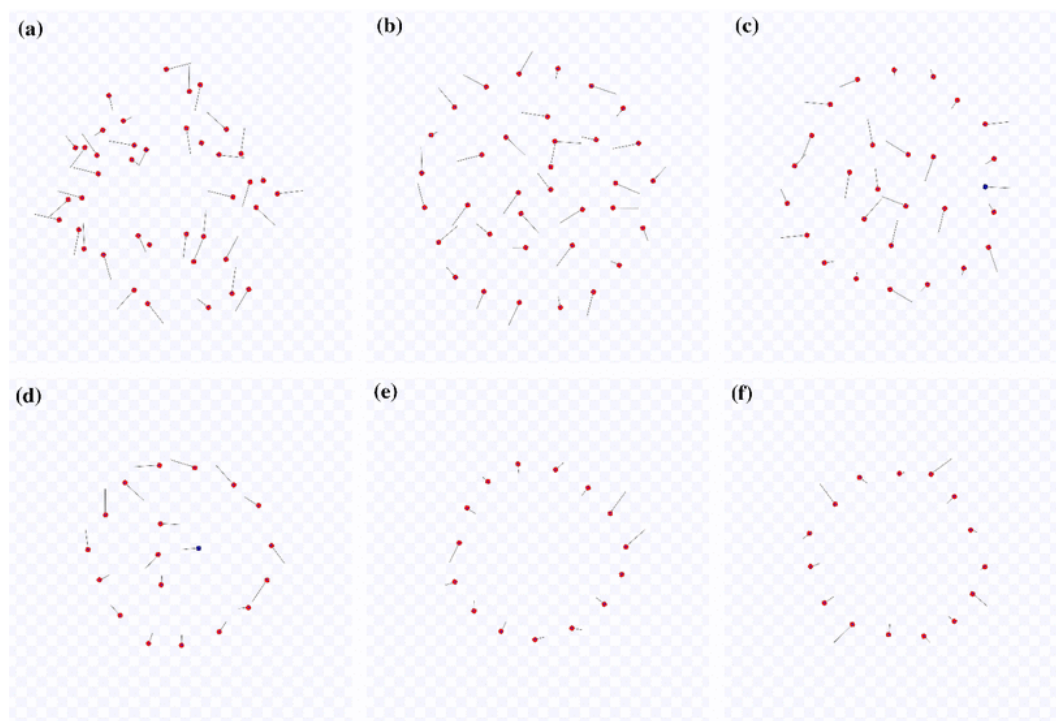
1.2.7 Optimal Reciprocal Collision Avoidance algoritmus

Optimal Reciprocal Collision Avoidance algoritmus poskytuje dostatečné podmínky k tomu, aby byl v multiagentním plánovacím systému zajištěn bezkolizní provoz.

Je založen na rychlostních překážkách, které jsou použity při hledání nové rychlosti pro agenty.

Tento algoritmus počítá s tím, že agenti mezi sebou nemohou komunikovat. Algoritmus najde minimální rychlost agentů, pro kterou bude plán bezkolizní a

postupnými iteracemi tuto rychlost zvyšuje, dokud nenalezne tu nejvyšší za podmínky, že každý agent má pouze informace o svém nejbližším okolí. Ukázka algoritmu na obrázku 1-10 [15].



Obrázek 1-10 Ukázka Optimal Reciprocal Collision Avoidance [16].

2. ROS2

V této kapitole jsou představeny základní pojmy ROS2 (Robot Operating Systém 2) a některé rozdíly mezi ROS1 a ROS2.

2.1 Robot Operating System

Pro vývoj robotů a jejich inteligence je potřeba mnoho softwarových nástrojů, které musí například zajistit počítačové vidění, simulaci prostředí, simulaci samotného robota či softwarové ovladače.

Robot Operating System slučuje většinu těchto nástrojů do jedné snadno přístupné struktury. Jeho hlavní výhodou je, že spadá pod vlastnictví Open Source Robotics Foundation, což znamená, že je volně plně k dispozici pro vývoj či distribuci softwaru na něm vyvíjeném. Další výhodou je jeho vysoká dostupnost (je přístupný pro všechny nejpoužívanější operační systémy) a možnost ho nainstalovat i na jednočipové počítače [18].

2.2 Základní pojmy

Základní koncepty struktury Robot Operating System vytváří něco, čemu se říká „ROS (2) graph".

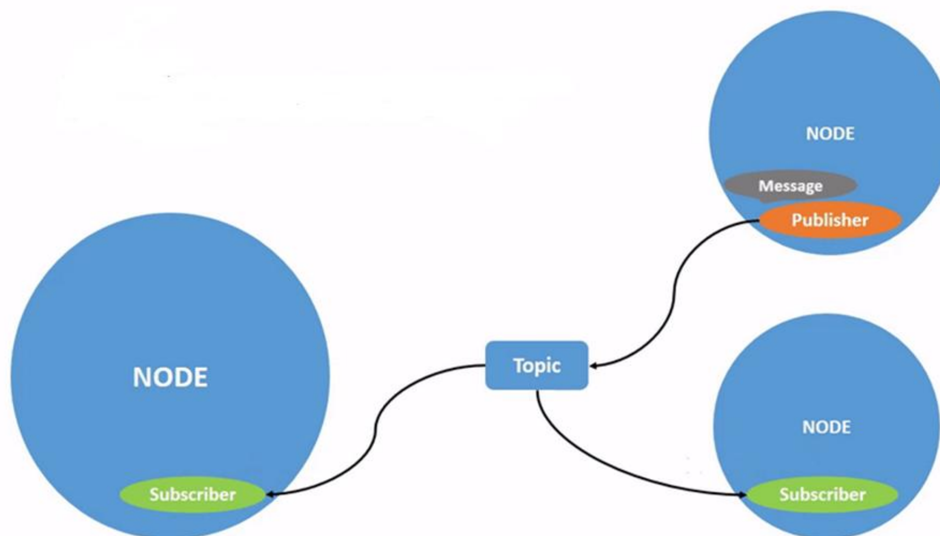
ROS (2) graph je síť ROS 2 prvků, které společně zpracovávají data ve stejný čas.

2.2.1 ROS 2 Nodes

Každá node v ROS by měla být zodpovědná za jeden cíl (například jeden pro ovládání motoru, jeden pro snímače atd.) a zároveň může posílat i přijímat data od jiných node pomocí témat (topics), služeb (services), akcí (actions) nebo parametrů (parameters). Kompletní ROS systém se skládá z mnoha node, které společně komunikují [19].

2.2.2 ROS 2 topics

ROS 2 dělí komplexní systémy na jednotlivé node. Témata (topics) jsou základní složkou ROS 2 grafu, protože slouží jako sběrnice pro jednotlivé node, skrz které si mohou posílat zprávy. Jedna node může vysílat na jakékoliv množství témat a zároveň i příjmat. Témata jsou jedním z hlavních způsobů jak se data pohybují mezi node [19].

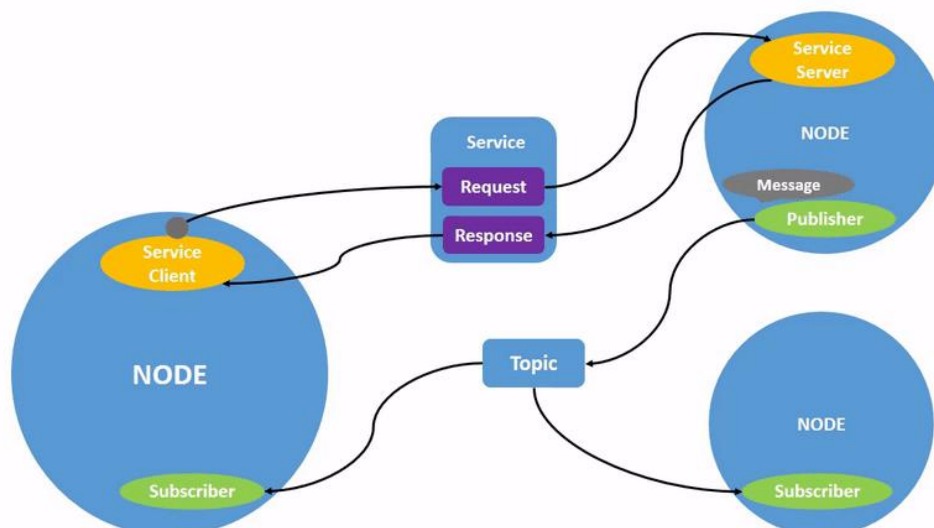


Obrázek 2-1 Ukázka komunikace node pomocí témat (topic) [19].

2.2.3 ROS 2 services

Služby (services) jsou další metodou komunikace mezi node.

Služby jsou založené na modelu volání-a-příjem (call-and-response), který se trochu liší od modelu témat vysílač-přijímač (publisher-subsriber). Témata umožňují node, aby se přihlásili k datům a získávali průběžné aktualizace, ale služby vysílají data pouze v případě, že jsou specificky volány klientem [19].



Obrázek 2-2 Ukázka komunikace node pomocí služeb (service) [19].

2.2.4 ROS 2 parameters

Parametry (parameters) jsou nastavitelnou hodnotou node a každá node může ukládat parametr jako integer, float, booleans, strings nebo lists.

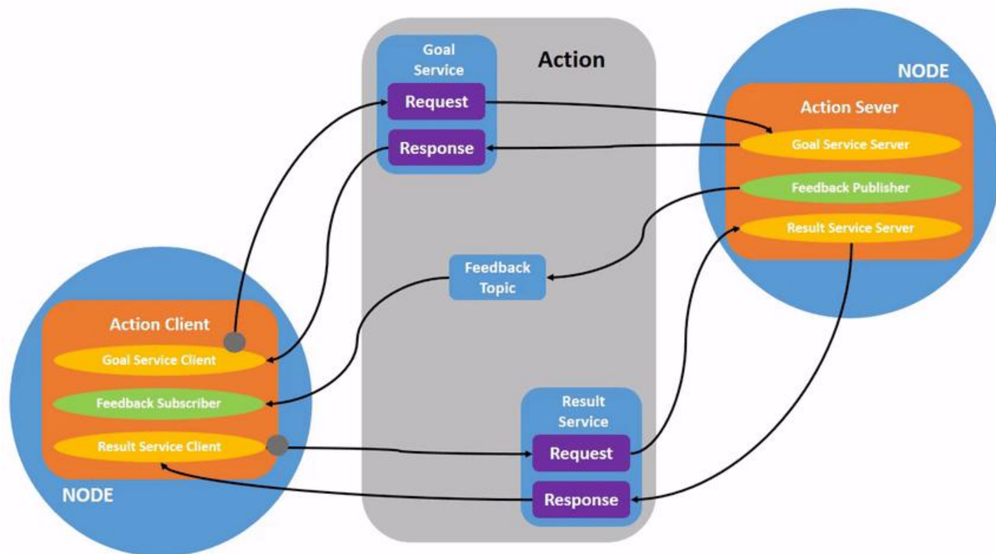
Node musí běžně deklarovat všechny parametry, které bude používat, aby bylo zamezeno špatné konfiguraci. V určitých výjimečných případech není možné znát všechny parametry předem. V takové situaci můžeme node inicializovat s povolením pro nedeklarované parametry [19].

2.2.5 ROS 2 actions

Akce (actions) jsou v ROS 2 komunikačním typem, který se zaměřuje na dlouho běžící úkoly. Skládají se ze tří částí: cíl (goal), zpětná vazba (feedback) a výsledek (reset).

Akce jsou založeny na tématech a službách. Jejich funkčnost se podobá službám, ale na rozdíl od nich je možné akce zrušit během jejich konání a zároveň poskytují stabilní zpětnou vazbu na rozdíl od jedné odpovědi, jakou vrací služby.

Akce využívají model klient-server (client-server). Node klient akce vyšle cíl node serveru akce a ten jako výsledek vrací stabilní zpětnou vazbu [19].



Obrázek 2-3 Ukázka komunikace node pomocí akcí (actions) [19].

2.2.6 ROS 2 packages

Balíčky (packages) je možné považovat za schránky pro ROS kód.

Pro instalaci a sdílení ROS kódu je nutné jej organizovat v balíčku. ROS 2 pro vytvoření balíčku využívá ament jako výrobní systém a colcon jako výrobní nástroj. Oficiálně podporované výrobní systémy jsou ament CMake a ament Python, ale existují

i jiné [19].

2.2.7 ROS 2 workspace

Pracovní plocha (workspace) je adresář obsahující ROS 2 balíčky.

Před používáním ROS 2 je nutné do terminálu, ve kterém se bude pracovat, dodat zdroj ROS 2 instalační pracovní plochy. Tím je v tomto terminálu umožněn přístup k ROS 2 balíčkům.

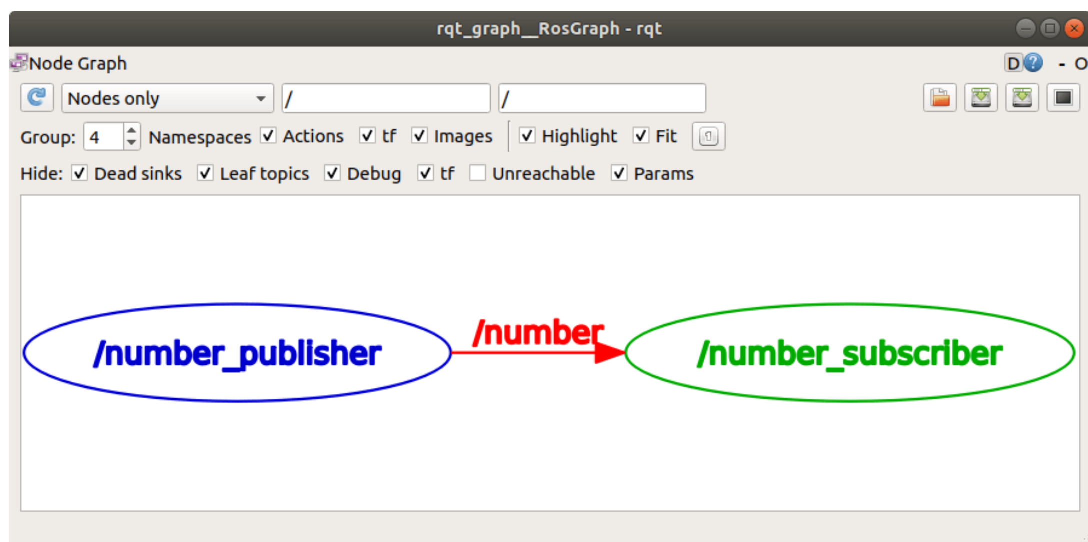
U pracovních ploch je také možné dodat zdroj k „překrytí“ (overlay) či „podkrytí“ (underlay). Jedná se o vedlejší pracovní plochu, do které je možné přidávat balíčky, aniž by to jakkoliv ovlivnilo primární pracovní plochu. Podmínkou je, že „podkrytí“ musí obsahovat závislosti všech balíčků v „překrytí“ [19].

2.2.8 ROS 2 rqt_console

Rqt_console je nástroj grafického uživatelského rozhraní (GUI), který je možné využít pro sledování log zpráv v ROS 2.

Běžně jsou tyto zprávy zobrazeny v terminálu, ale rqt_console umožňuje tyto zprávy ukládat, filtrovat a nebo se na ně blíže a přesněji podívat [19].

Ukázka rqt_console grafu je na obrázku 2-5.



Obrázek 2-4 Ukázka rqt_graph [16].

3. REALIZACE BALÍČKU ROS2

V následující kapitole bude popsána samotná realizace balíčku ROS 2 včetně útržků kódu, jejich vysvětlení a případné řečení rozdílu mezi ROS 1 a ROS 2.

Samotný kód se skládá ze dvou .h souborů, čtyř .cpp souborů, jednoho .msg souboru, dvou .srv souborů, jednoho .rviz souboru a jednoho .launch.py souboru.

Většina zobrazených kódů bude reprezentovat pouze část celku a jejich kompletní podoba se nachází v digitální příloze.

3.1 Agent Node

Agent node se skládá ze tří souborů *ragent_node.cpp*, *ragent_obj.cpp* a *ragent_obj.h*.

Tento node má za úkol vytvořit agenta a aktualizovat jeho pozici.

3.1.1 Ragent_node.cpp

```
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  auto n = rclcpp::Node::make_shared("agent");

  Agent_Robot agent(n, my_s_id, start_pose, 10.0, 30.0);

  rclcpp::executors::MultiThreadedExecutor exec;
  exec.add_node(n);
  exec.spin();
  rclcpp::shutdown();
  return 0;
}
```

Rclcpp::init plní v ROS2 stejnou funkci kterou v ROS1 plní *ros::init*. Novým rozdílem jsou však jiné vstupní argumenty. Pro ros::init je potřeba dodat i název node (zde „agent“), ale v ROS2 se tento název udává až při vytváření samotného node.

Toto vytváření probíhá pomocí *rclcpp::Node::make_shared()*. Tento příkaz vytvoří ukazatel na node a vloží ho do proměnné n.

Agent_Robot agent() vytvoří class Agent_Robot, která inicializuje veškeré služby, a témata proměnné n.

Proměnná *exec* umožňuje paralelní spuštění služeb na více vláknech, aby nedošlo ke zaseknutí programu.

Příkaz *exec.spin()* aktivuje proměnnou n a tím spustí její služby a témata.

3.1.2 Ragent_obj.h

```
#include <ros2_agent/msg/ros_agent_info.hpp>
#include "ros2_agent/srv/ros_get_plan.hpp"
#include "ros2_agent/srv/update_goal.hpp"
```

Kromě běžných #include, které se v programu nachází kvůli správnému chodu programu, jsou zde ještě tyto tři unikátní #include. Jedná se o vlastní zprávu a dvě služby, které jsou v programu využívány pro přenos souřadnic agenta a jeho jména.

```
class Agent_Robot
{
    public:
            explicit          Agent_Robot(std::shared_ptr<rclcpp::Node>
node_handle,    std::string    serial_id,    geometry_msgs::msg::Pose2D
start_pose, const double period, const double timer_hz);
```

Ve veřejné části třídy Agent_Robot se nachází konstruktor jehož parametry jsou zadávány *ragent_node.cpp*. Proměnné *period* a *timer_hz* se využívají pro výpočet rychlosti agentů.

```
std::shared_ptr<rclcpp::Publisher<ros2_agent::msg::RosAgentInfo>>
pub_agent_feedback;

std::shared_ptr<rclcpp::Publisher<visualization_msgs::msg::Marker>>
pub_agent_marker;

            rclcpp::Service<ros2_agent::srv::UpdateGoal>::SharedPtr
srv_update_goal;

            rclcpp:Client<ros2_agent::srv::RosGetPlan>::SharedPtr
srv_get_plan;
            rclcpp::TimerBase::SharedPtr tmr_odom;
            rclcpp::callback_group::CallbackGroup::SharedPtr
callback_group_;
```

V privátní části třídy *Agent_Robot* jsou deklarovány všechny služby, klienti, vysílači a příjemci potřebné pro daný node. Rozdíl mezi ROS2 a ROS1 je zde ten, že vše musí být deklarováno jako ukazatel. Ukazatel *callback_group_* je využíván pro *MultiThreadedExecutor exec*.

V souboru *ragent_obj.h* jsou dále deklarovány hlavičky jednotlivých funkcí.

3.1.3 Ragent_obj.cpp

```
    this->node->declare_parameter("serial_id");
    this->node->declare_parameter("x_c");
    this->node->declare_parameter("y_c");
    this->node->declare_parameter("theta_c");
```

V konstruktoru třídy *Agent_Robot* je potřeba deklarovat parametry, jejichž hodnota je uložena ve spouštěcím souboru programu.

```
    int a = serial_id.at(6);
    srand (time(NULL)+ a);
    agent_color[0] = (rand() % 100)*0.01;
    agent_color[1] = (rand() % 100)*0.01;
    agent_color[2] = (rand() % 100)*0.01;
```

Pro zajištění unikátních barev agentů je použita funkce *rand()* a číslo agenta (*serial_id* = „*agent_x*“, kde x je číslo agenta) vytvoří jedinečnou kombinaci barev.

```
    this->srv_update_goal                    =                    this->node-
>create_service<ros2_agent::srv::UpdateGoal>("update_goal",
std::bind(&Agent_Robot::agent_update_goal,      this,      _1,      _2),
::rmw_qos_profile_default, callback_group_);
```

Tímto způsobem jsou inicializovány služby. *Ros2_agent::srv::UpdateGoal* je jméno souboru, ve kterém je uložen předpis služby, *„update_goal“* je téma, na které bude služba vysílat, *agent_update_goal* je funkce, která přijme hodnoty ze žádosti a *callback_group_* umožní proměnné *MultiThreadedExecutor exec* spustit jinou službu paralelně na jiném vlákně.

```
void                         Agent_Robot::agent_build_path_marker(const
vector<geometry_msgs::msg::Point> &vect)
{
    visualization_msgs::msg::Marker marker;
    pub_path_marker->publish(marker);
}
```

Funkce *agent_build_path_marker()* vytváří uživatelem viditelný objekt. V tomto případě se jedná o čáru, která reprezentuje agentovu cestu z počáteční do cílové pozice. Po nastavení velikosti, tvaru, barev a času jak dlouho má značka existovat, odešle ukazatel *pub_path_maker* na téma */visualization/path* značku *marker*. Na stejném tématu přijímá Rviz, který na obrazovce cestu vykreslí.

```
void                          Agent_Robot::agent_update_transform(const
geometry_msgs::msg::Pose2D &pose)
{
    geometry_msgs::msg::TransformStamped transform;
    tf_broadcaster_->sendTransform(transform);
}
```

Funkce *agent_update_transform()* má za úkol změnit agentovu polohu vzhledem ke stavu světa. Tím je myšleno viditelně pohybovat s agentem, aby bylo možné na obrazovce vidět, jestli se blíží k cíli. Toho je dosáhnuto vysíláním na téma */agent_x/base_link*, na kterém zároveň přijímá Rviz.

```
void Agent_Robot::agent_update_pose()
{
    ros2_agent::msg::RosAgentInfo msg;
    msg.serial_id = serial_id;
    msg.start_pose = pose;
    pub_agent_feedback->publish(msg);
    agent_update_transform(pose);
    agent_build_agent_marker();
}
```

Funkce *agent_update_pose()* je spouštěna časovačem (v tomto případě se jedná o 1 vteřinu). Jejím úkolem je aktualizovat pozici agenta a zároveň kontrolovat, jestli se agent pouze otáčí, pohybuje se, nebo jestli už dorazil do cíle. Zjištěnou polohu poté vysílá na téma */agent_feedback* a zároveň volá funkce *agent_update_transform()* a *agent_build_agent_marker()*.

```
void Agent_Robot::agent_build_agent_marker()
{
    visualization_msgs::msg::Marker marker;
    pub_agent_marker->publish(marker);
}
```

Funkce *agent_build_agent_marker()* vytváří vzhled agenta jako takového. Do proměnné *marker* uloží jeho hodnoty a ty následně vyšle na téma */visualization/base_link*. Na tomto tématu přijmá Rviz, který již agenta jako takového zobrazí na obrazovce.

```
bool                              Agent_Robot::agent_update_goal(const
std::shared_ptr<ros2_agent::srv::UpdateGoal::Request> req,

std::shared_ptr<ros2_agent::srv::UpdateGoal::Response> res)
{
        auto srv3 = srv_get_plan->async_send_request(srv1);

        bool success = srv3.valid();

    if (success == true)
        {
            Path = point_list;
            agent_build_path_marker(point_list);
        }
    return true;
```

Funkce *agent_update_goal()* je službou volanou pomocí terminálu. Uživatel zvolí cílovou pozici a funkce nejdříve vyhodnotí, jestli je cíl právoplatný, či zda se agent bude pouze otáčet. Pokud je cíl v pořádku je uvnitř služby zavolána služba *srv_get_plan->async_send_request()*, která pomocí souřadnic cílové pozice naplánuje optimální cestu z počáteční pozice do cílové. Pokud je plán nalezen, je pomocí *agent_build_path_marker()* nakreslena cesta a agent se dá do pohybu, dokud nedorazí do cíle.

3.2 Planner Node

Planner node se také skládá ze tří souborů *rmotion_planner_obj.cpp*, *rmotion_planner_node.cpp* a *rmotion_planner_obj.h*.

Tento node má za úkol plánování tras pro agenty.

3.2.1 Rmotion_planner_node.cpp

```
int main( int argc, char** argv ){
    rclcpp::init(argc, argv);
    auto n = rclcpp::Node::make_shared("nodes");
    Motion_Planner mp(n, 10, 10, 10, 10);

    rclcpp::spin(n);
    return 0;
}
```

Rmotion_planner_node.cpp je skoro totožný s *ragent_node.cpp*. Jediným rozdílem je absence *MultiThreadedExecutor exec*, protože služby Planner node nevolají žádné další služby.

3.2.2 Rmotion_planner_obj.h

```
struct Path
{
    std::string serial_id;
    double time_of_plan;
    vector<geometry_msgs::msg::Point> point_list;
};
enum Status {FREE, OCCUPIED, START, GOAL};

struct Grid_node
{
    bool operator<(Grid_node other) const
};
```

Rmotion_planner_obj.h obsahuje navíc dvě struktury. Struktura *Path* v sobě obsahuje id agenta a zároveň list bodů, které tvoří naplánovanou agentovu cestu. Struktura *Grid* v sobě obsahuje celou mapu, po které se agent pohybuje. Pomocí výčtu *Status* určí u jednotlivých bodů na mapě zda-li je možné je použít pro plánování trasy. Dále se v ní nachází funkce *operator<*, která seřadí body mapy podle ceny určené plánovacím algoritmem.

```
rclcpp::Subscription<ros2_agent::msg::RosAgentInfo>::SharedPtr
sub_agent_pose;
```

V *rmotion_planner_obj.h* je také vytvořen příjemce (subscriber), který přijímá zprávy od *Agent node* na tématu */agent_feedback*.

V hlavičkovém souboru jsou dále deklarovány velikosti mapy, ceny přechodu mezi body na mapě a také hlavičky jednotlivých funkcí.

3.2.3 Rmotion_planner_obj.cpp

```
    planner_draw_rviz_nodes();

void Motion_Planner::planner_draw_rviz_nodes()
{
    visualization_msgs::msg::Marker marker_free, marker_occupied;

    for (int i {0}; i <= X_MAX; i++)
        for (int j{0}; j <= Y_MAX; j++)
        {
            geometry_msgs::msg::Point p;
            p.x = i;
            p.y = j;
```

```
            // optional - draw occupied nodes as a different color
            // if ((i == 2 || i == 3) && (j == 2 || j == 3 || j == 4
|| j == 5 || j == 6))
            //     marker_occupied.points.push_back(p);
            // else
            marker_free.points.push_back(p);
        }
   while (pub_grid_nodes_free->get_subscription_count() < 1)
     {
     }
    pub_grid_nodes_free->publish(marker_free);
    pub_grid_nodes_occupied->publish(marker_occupied);
}
```

Při inicializaci Planner node je ihned volána funkce pro vykreslení volných a okupovaných bodů na mapě.

Funkce *planner_draw_rviz_nodes*() vytvoří dva markery *marker_free* a *marker_occupied*. Dále vznikne matice MxM (zde se jedná o matici 10x10) *p*, která je bez dodatečných podmínek celá zkopírována do proměnné *marker_free*. Uživatel si poté může pomocí podmínky zvolit, zda chce mít na mapě překážky. Souřadnice zvolené jako překážky jsou následně uloženy do proměnné *marker_occupied*. Obě proměnné (*marker_free marker_occupied*) jsou poté pomocí *pub_grid_nodes_free* a *pud_grid_nodes_occupied* odeslány na témata *visualization/grid_nodes_free* a *visualization/grid_nodes_occupied*. Z obou těchto témat přijímá Rviz a ten je vykreslí na obrazovku.

```
struct Path Motion_Planner::planner_plan_path(const
geometry_msgs::msg::Point start_point, const geometry_msgs::msg::Point
goal_point, const std::string serial_id, const
vector<geometry_msgs::msg::Point> collisions)
```

Hlavní funkcí *rmotion_planner_obj.cpp* je funkce *planner_plan_path*(). Jejím úkolem je pomocí plánovacího algoritmu nalézt optimální cestu z počáteční pozice agenta do cílové pozice.

```
    vector<Grid_node> open;
    Grid_node grid[X_MAX+1][Y_MAX+1] = {};
    for (size_t i{0}; i <= X_MAX; i++)
    {
        for (size_t j{0}; j <= Y_MAX; j++)
        {
            Grid_node n = {};
            // Optional - include an OCCUPIED region
            // if ((i == 2 || i == 3) && (j == 2 || j == 3 || j == 4
|| j == 5 || j == 6))
            //     n.stat = OCCUPIED;
            // else
            n.stat = FREE;
            n.pos[0] = i;
            n.pos[1] = j;
```

```
            n.past_cost = INT_MAX;
            grid[i][j] = n;
        }
    }
```

Nejdříve si funkce vytvoří proměnnou *grid*, do které uloží všechna volná pole a všechny překážky bez ohledu na jiné agenty (překážky jsou znovu voleny uživatelem). Dále všem bodům na mapě nastaví maximální cenu přechodu kvůli následnému zjištění počáteční pozice.

```
for (auto &p : collisions)
    {
        grid[(int)p.x][(int)p.y].stat = OCCUPIED;
    }

    int start[] = {(int)start_point.x, (int)start_point.y};
    int goal[] = {(int)goal_point.x, (int)goal_point.y};
    grid[start[0]][start[1]].stat = START;
    grid[start[0]][start[1]].past_cost = 0;
    grid[goal[0]][goal[1]].stat = GOAL;
    open.push_back(grid[start[0]][start[1]]);
```

Následně podle předaného vektoru *collisions* funkce musí nastavit kolizní body s jinými agenty na mapě jako okupované a předá algoritmu informace o počáteční a cílové poloze.

```
    while (open.size() != 0)
    {
        open.erase(open.begin());
        grid[x_curr][y_curr].is_closed = true;
        if (grid[x_curr][y_curr].stat == GOAL)
        {
            geometry_msgs::msg::Point goal;
            final_path.point_list.push_back(goal);
```

Funkce nyní prohledá mapu, dokud nenalezne cíl, který následně uloží na konec finální cesty.

```
            while (x_curr != start[0] || y_curr != start[1])
            {
final_path.point_list.insert(final_path.point_list.begin(),
grid[x_curr][y_curr].parent);
            }
            final_path.time_of_plan = node->now().seconds();
            break;        }
```

Po nalezení cíle, funkce postupně prohledává sousední body, dokud nenalezne počátek. Po jeho nalezení přestane prohledávat a spočítá cenu přechodů mezi jednotlivými body. Nakonec je použita funkce *operator<*, která srovná body do správného pořadí a funkce vrátí hodnotu *final_path*, která obsahuje cestu pro agenta.

```
geometry_msgs::msg::Point
Motion_Planner::planner_check_collision(const        struct        Path
current_path)
{
if    (path_obj.point_list.back().x    ==    current_path_point.x    &&
path_obj.point_list.back().y == current_path_point.y)
                    {
                        return current_path_point;
                    }
```

Funkce *planner_check_collision()* zjišťuje, zda dojde během cesty ke kolizi dvou agentů. Nejdříve je vyhodnoceno, zda ke kolizi dojde za jízdy. Pokud ne, znamená to, že jeden z agentů dorazí do cíle dříve, než by došlo ke kolizi. V takovém případě stačí pouze zjistit, zda je cílová pozice jednoho agenta na cestě agenta druhého. Pokud ano vyhodnotí funkce cílovou pozici prvního agenta jako překážku pro agenta druhého.

Pokud by však mělo dojít ke kolizi za jízdy, musí funkce nejdříve zjistit, v jakém bodě ke kolizi dojde. Následně potom musí vyhodnotit, zda je kolize problematická. Jeden z agentů už by totiž mohl být v čase, kdy ke kolizi dojde dávno pryč. Když funkce vyhodnotí kolizi jako problematickou, vrátí tento bod plánovači a ten jej vyhodnotí jako překážku.

```
bool                             Motion_Planner::planner_get_plan(const
std::shared_ptr<ros2_agent::srv::RosGetPlan::Request>             req,
std::shared_ptr<ros2_agent::srv::RosGetPlan::Response> res)

        current_path = planner_plan_path(start_point, goal_point, req-
>serial_id, collisions);
        collision_location = planner_check_collision(current_path);
        collisions.push_back(collision_location);
        res->path = current_path.point_list;
        return true;
    }
```

Funkce *planner_get_plan()* je službou volanou uvnitř Agent node. Jejím úkolem je pomocí vnitřních funkcí vypočítat a zpětně odeslat naplánovanou trasu zpět do Agent node, aby z ní bylo možné vykreslit plánovanou trasu a započít agentovu cestu.

Služba získá cílové souřadnice agenta a pomocí funkcí *banner_plan_path()* a *banner_check_collision()* vypočítá nejoptimálnější trasu, kterou agent musí zvolit a tu následně pomocí odpovědi (response) odešle nazpět.

```
void
Motion_Planner::planner_agent_pose_callback(ros2_agent::msg::RosAgentI
nfo::SharedPtr msg)
{
    for (size_t i{0}; i < agent_start_poses.size(); i++)
    {
        if (agent_start_poses.at(i)->serial_id == msg->serial_id)
        {
            agent_start_poses.at(i) = msg;
        }
    }
    if (!found)
    {
        archived_paths.push_back(new_agent_path);
    }
}
```

Funkce *planner_agent_pose_callback()* přijímá data z Agent node je do proměnné *archived_paths* aktualizovat hodnoty stávajícího agenta, nebo uložit hodnoty úplně nového agenta.

3.3 Service a message soubory

Při programování je pro určité funkce třeba volat vlastní konkrétní zprávu či požadavek.

K tomu slouží pomocné *srv* a *msg* složky, ve kterých jsou uloženy uživatelem definované zprávy a služby.

```
string serial_id
geometry_msgs/Pose2D goal_pose
---
geometry_msgs/Point[] path
```

Toto je příklad služby s názvem *RosGetPlan.srv*, která jako žádost přijímá řetězec reprezentující agentovo jméno a proměnou *goal_pose*, která v sobě obsahuje souřadnice cíle.

```
string serial_id
geometry_msgs/Pose2D start_pose
```

Zde je vidět příklad zprávy s názvem *RosAgentInfo.msg*. Formát této zprávy vysílá Agent node a následně ho přijímá Planner node, který z ní vyhodnocuje a do archivu ukládá počáteční polohu agenta.

3.4 Launch soubor

Při spouštění více node naráz, je výhodnější všechny sloučit do jednoho spouštěcího odkazu.

V ROS2 k tomuto účelu slouží soubor .launch.py.

```
def generate_launch_description():

    serial_id_1_launch_arg = DeclareLaunchArgument(
        "serial_id_1", default_value=TextSubstitution(text="agent_1")
    )
    x_1_launch_arg = DeclareLaunchArgument(
        "x_1", default_value=TextSubstitution(text="2")
    )
    y_1_launch_arg = DeclareLaunchArgument(
        "y_1", default_value=TextSubstitution(text="0")
    )
    theta_1_launch_arg = DeclareLaunchArgument(
        "theta_1", default_value=TextSubstitution(text="0")
    )
```

Protože agenti využívají argumenty při jejich vytváření, musí být deklarovány jak v jejich konstruktoru, tak v jejich spouštěcím souboru. Z kódu je možné vidět, že požadované argumenty jsou jméno agenta a jeho počáteční pozice. Následují argumenty využívá agent_1. Pro ostatní agenty jsou deklarace totožné, jen je 1 nahrazena požadovaným číslem.

```
agent1_node = Node(
            package='ros2_agent',
            namespace ='agent1',
            executable='agent_node',
            output ='screen',
            parameters=[{
                "serial_id": LaunchConfiguration('serial_id_1'),
                "x_c": LaunchConfiguration('x_1'),
                "y_c": LaunchConfiguration('y_1'),
                "theta_c": LaunchConfiguration('theta_1'),
            }]
        )
```

Výše uvedený kód slouží ke spuštění jednoho z agentů. Je potřeba definovat balíček, ve kterém se node nachází, spouštěč, který je deklarovaný v souboru CMakeList.txt a výstup. Doplňující definice je namespace, který umožňuje spuštění více node se stejnými tématy a službami a další definicí jsou parametry, jejichž deklarace se nachází výše. Název nacházející se nalevo od příkazu *LaunchConfiguration* je jméno parametru, který je potřeba deklarovat v konstruktoru node.

```
    return LaunchDescription([
        serial_id_1_launch_arg,
        x_1_launch_arg,
        y_1_launch_arg,
        theta_1_launch_arg,
        agent1_node,
    ])
```

Na konci launch souboru je potřeba napsat názvy všech deklarovaných parametrů a spouštěcích proměnných.

3.5 CMakeList soubor a package soubor

Při sestavování programu pomocí nástroje colcon je nutné deklarovat všechny závislosti v souborech CMakeList.txt a package.xml.

```
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
find_package(builtin_interfaces REQUIRED)
find_package(rosidl_default_generators REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(visualization_msgs REQUIRED)
find_package(tf2 REQUIRED)
find_package(tf2_ros REQUIRED)
find_package(rviz2 REQUIRED)
```

Pomocí příkazu *find_package* je nutné v souboru CMakeList.txt uvést všechny balíčky, které jsou pro chod programu potřeba.

```
rosidl_generate_interfaces(ros2_agent
  "msg/RosAgentInfo.msg"
  "srv/RosGetPlan.srv"
  "srv/UpdateGoal.srv"
  DEPENDENCIES builtin_interfaces geometry_msgs
)
```

Pokud se v programu nachází vlastní zprávy a služby je nutné je vygenerovat i s potřebnými náležitostmi.

```
add_executable(agent_node     src/ragent_obj.h     src/ragent_node.cpp
src/ragent_obj.cpp)
ament_target_dependencies(agent_node  rclcpp  std_msgs  geometry_msgs
visualization_msgs tf2 tf2_ros rviz2)

install(TARGETS
   agent_node
   DESTINATION lib/${PROJECT_NAME})

install(DIRECTORY
  launch rviz
  DESTINATION share/${PROJECT_NAME}/)
```

Nakonec CMakeList.txt souboru je potřeba napsat spouštěče, které budou vytvářet node, jejich náležitosti a nainstalovat tyto spouštěče i jejich launch soubor.

```
  <depend>rclcpp</depend>
  <depend>std_msgs</depend>
  <depend>geometry_msgs</depend>
  <depend>visualization_msgs</depend>
  <depend>tf2</depend>
  <depend>tf2_ros</depend>
  <depend>rviz2</depend>
  <build_depend>builtin_interfaces</build_depend>
  <build_depend>rosidl_default_generators</build_depend>
  <exec_depend>builtin_interfaces</exec_depend>
  <exec_depend>rosidl_default_runtime</exec_depend>
```

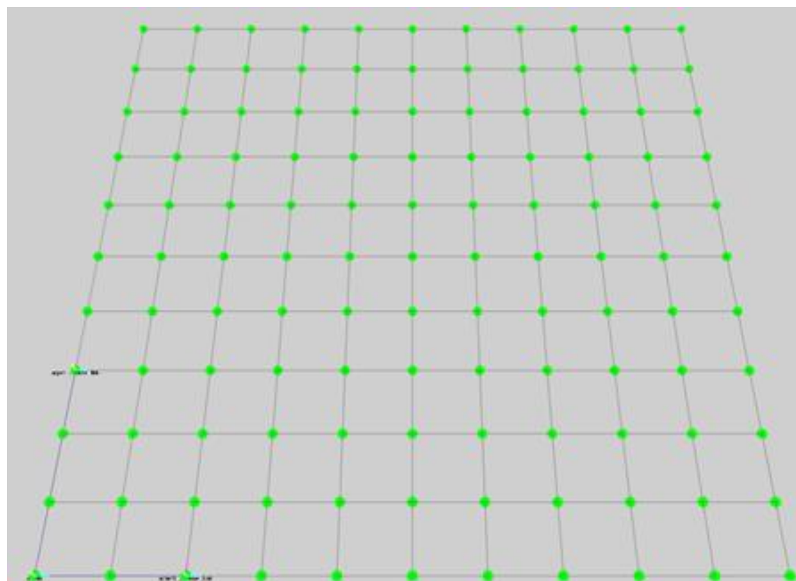
V souboru package.xml je stejně jako v souboru CMakeList.txt nutné napsat všechny náležitosti které program potřebuje ke svému chodu.

3.6 Výsledek

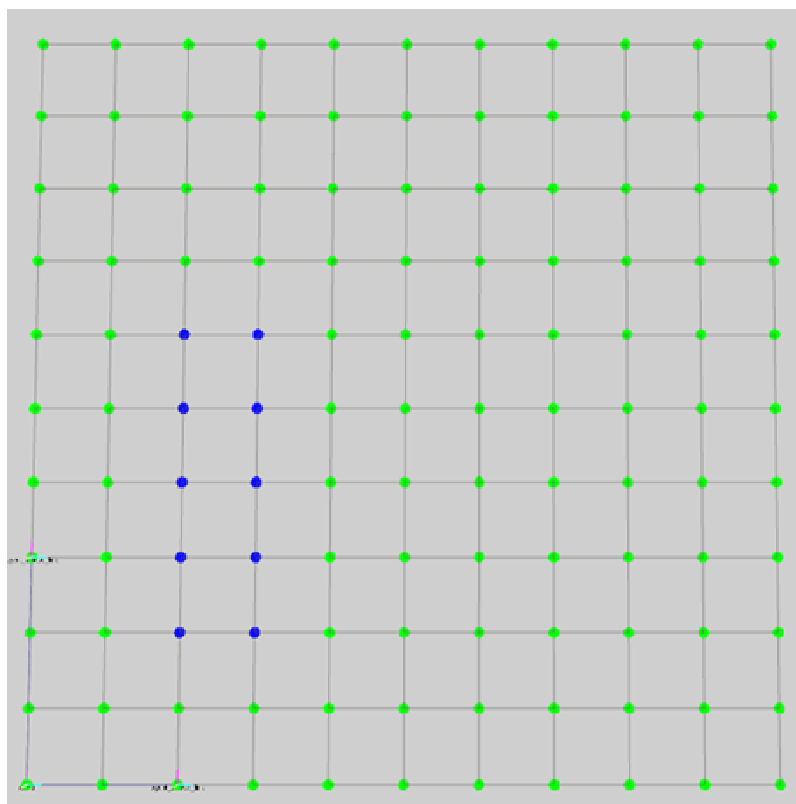
Před spuštěním programu je nutné v terminálu s otevřenou pracovní plochou dodat zdroj pomocí příkazu *source install/setup.bash*.

Následně je možné program spustit pomocí příkazu *ros2 launch ros2_agent ros2_agent_start.launch.py*. *Ros2_agent* je názvem balíčku s kódem a *ros2_agent_start.launch.py* je launch soubor. Různé verze simulovaného prostředí zobrazené pomocí nástroje Rviz je možné vidět na obrázcích 3-1, 3-2, 3-3.

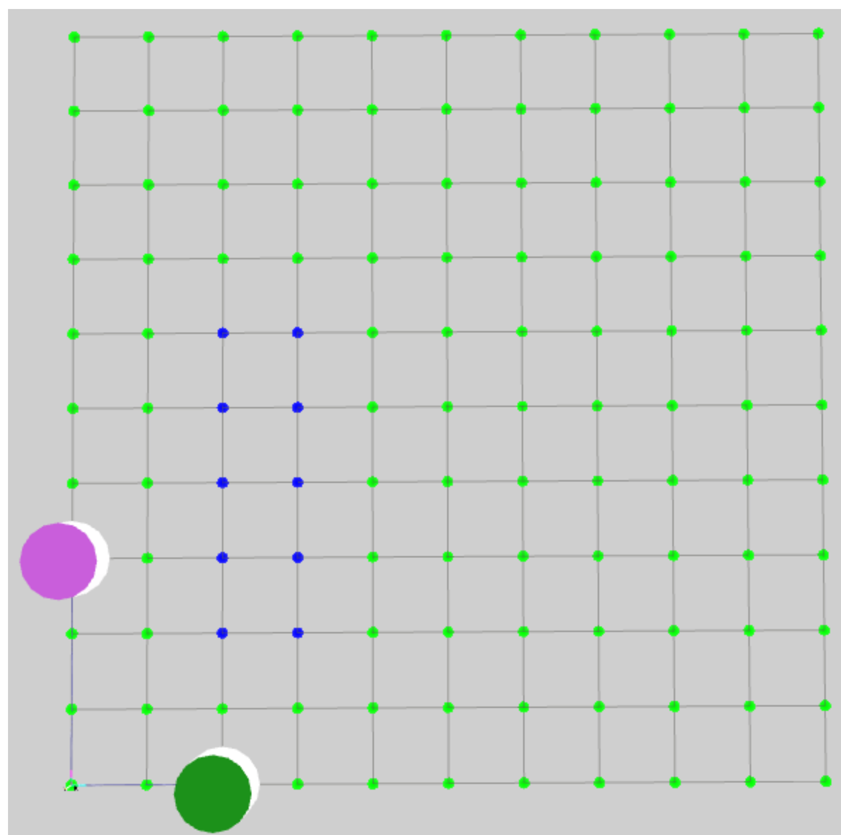
Pro spuštění plánovacího algoritmu je nutné otevřít druhý terminál s otevřenou pracovní plochou se zdrojem a zadat příkaz *ros2 service call /agentx/update_goal ros2_agent/srv/UpdateGoal "{goal_x: '4.0', goal_y: '6.0', goal_theta: '0.0'}"*, kde x reprezentuje číslo agenta.



Obrázek 3-1 Ukázka vygenerování volného prostoru.



Obrázek 3-2 Ukázka vygenerování prostoru s překážkou.



Obrázek 3-3 Ukázka vygenerování dvou agentů.

Vizualizace má představovat abstrakci reálného světa. Agenti jsou abstrakcí robota se všesměrovým podvozkem a modré tečky reprezentují stěny či jiné překážky.

Přibližná reálná podoba agenta je vyobrazena na obrázku 3-4.

Obrázek 3-4 Možná podoba reálné verze agenta 20.

4. ZÁVĚR

V bakalářské práci se mi podařilo vytvořit balíček pro ROS 2, který v sobě obsahuje plánovací algoritmus pro multiagentní systémy. Výhodou tohoto balíčku je jeho upravitelnost. Je volně možné měnit vzhled robota či prostředí, nebo je možné změnit i celý plánovací algoritmus. Jednou z nevýhod je poměrně pomalá simulace samotného pohybu robota. To však může být způsobeno slabším vývojovým hardwarem.

Stávající verze programu by mohla být upravena, aby došlo ke zlepšení asynchronního servisního volání.

Mezi možná vylepšení může patřit generace nových dynamických překážek, kromě samotných agentů. Dále propojení s vizualizačním a simulačním nástrojem Gazeebo pro zajištění co nejvěrnější simulace, nebo také možnost měnit cílovou pozici za běhu programu.

LITERATURA

[1] *STACHNISS, Cyrill a Nived CHEBROLU. Occupancy Grid Maps (Cyrill Stachniss). https://www.youtube.com* [online]. Bonn: Cyrill Stachniss, 2020, 28.8.2020 [cit. 2022-04-10]. Dostupné z: https://www.youtube.com/watch?v=v-Rm9TUG9LA&list=PLgnQpQtFTOGSeTU35ojkOdsscnenP2Cqx&index=4

[2] *HOWARD, Andrew, Lynne PARKER a Gaurav S. SUKHATME. The SDR Experience: Experiments with a Large-Scale Heterogeneous Mobile Robot Team. Https://www.researchgate.net [online].* Tennessee: Lynne Parker, 2004, June 2004 [cit. 2022-04-10]. Dostupné z: https://www.researchgate.net/figure/Occupancy-grid-map-produced-during-a-multiple-robot-multiple-entry-trial-two-robots_fig2_2873269

[3] *XIE, Yangmin, Rui ZHOU a Yusheng YANG. Improved Distorted Configuration Space Path Planning and Its Application to Robot Manipulators. Www.researchgate.net [online]. Shanghai: Shanghai University, 2020, Oct 2020 [cit. 2022-04-10]. Dostupné z: https://www.researchgate.net/figure/The-process-of-path-planning-in-DC-space-a-C-space-map-Grid-nodes-can-be-categorized_fig3_346411752*

[4] *STACHNISS, Cyrill. Robot Motion Planning using A* (Cyrill Stachniss). Https://www.youtube.com/ [online].* Bonn: Cyrill Stachniss, 2020, 9.10.2020 [cit. 2022-04-10]. Dostupné z: https://www.youtube.com/watch?v=HR1TNa8Lp7w

[5] DE WEERDT, Mathijs a Bradley CLEMENT. Introduction to planning in multiagent systems. *Https://www.researchgate.net/* [online]. USA: Mathijs De Weerdt, 2009, December 2009 [cit. 2022-04-10]. Dostupné z: https://www.researchgate.net/publication/220535412_Introduction_to_planning_in_multiagent_systems

[6] YU, Jingjin a Steven M. LAVALLE. Multi-agent Path Planning and Network Flow. *Https://www.semanticscholar.org/* [online]. USA: Jingjin Yu, 2012, 25.4.2012 [cit. 2022-04-10]. Dostupné z: https://www.semanticscholar.org/paper/Multi-agent-Path-Planning-and-Network-Flow-Yu-LaValle/6f95cd234860b369b661f479afe75ca8fe133692

[7] SILVER, David. Cooperative Pathfinding. *Https://www.aaai.org/* [online]. Edmonton: David Silver, 2005, 1.6.2005 [cit. 2022-04-10]. Dostupné z: https://www.aaai.org/Papers/AIIDE/2005/AIIDE05-020.pdf

[8] MAJERECH, Ondřej. Introduction to planning in multiagent systems. *Https://dspace.cuni.cz/* [online]. Prague: Ondřej Majerech, 2017, 18.7.2017 [cit. 2022-04-10]. Dostupné z: https://dspace.cuni.cz/bitstream/handle/20.500.11956/90574/DPTX_2014_2_11320_0_443220_0_164071.pdf?sequence=1&isAllowed=y

[9] SAJID, Qandeel, Ryan LUNA a Kostas E. BEKRIS. Multi-Agent Pathfinding with Simultaneous Execution of Single-Agent

Primitives. *Https://ojs.aaai.org/* [online]. USA: David Silver, 2012, 2012 [cit. 2022-04-10]. Dostupné z: https://www.google.cz/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjjq5a57pL1AhUR3aQKHayPBi8QFnoECAQQAQ&url=https%3A%2F%2Fojs.aaai.org%2Findex.php%2FSOCS%2Farticle%2Fdownload%2F18243%2F18034&usg=AOvVaw0Ih4uTfqagdhKUKfS9VM5C
[10] *LAVALLE, Steven M. PLANNING ALGORITHMS [online].* Cambridge: Cambridge University Press, 2006 [cit. 2022-04-10]. ISBN 978-0521862059. Dostupné z: http://lavalle.pl/planning/
[11] *AKHTAR, Rafi. Search Algorithms in AI. Https://www.geeksforgeeks.org/* [online]. Kalkata: Rafi Akhtar, 2021, 20.8.2021 [cit. 2022-04-10]. Dostupné z: https://www.geeksforgeeks.org/search-algorithms-in-ai/
[12] SILVER, David. Cooperative Pathfinding. *Https://www.aaai.org/* [online]. Edmonton: David Silver, 2005, 1.6.2005 [cit. 2021-12-06]. Dostupné z: https://www.aaai.org/Papers/AIIDE/2005/AIIDE05-020.pdf
[13] STANDLEY, Trevor S. Finding Optimal Solutions to Cooperative Pathfinding Problems. *Https://ojs.aaai.org/* [online]. Los Angeles: Trevor S. Standley, 2010, 2010 [cit. 2021-12-06]. Dostupné z: https://www.google.cz/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwimy8yYrZP1AhV9SPEDHb0OCwgQFnoECAcQAQ&url=https%3A%2F%2Fojs.aaai.org%2Findex.php%2FAAAI%2Farticle%2Fview%2F7564%2F7425&usg=AOvVaw02cSM15WvPwONVPp-OWm3z
[14] BHATTACHARYA, Subhrajit, Maxim LIKHACHEV a Vijay KUMAR. Distributed Path Consensus Algorithm. *Www.lehigh.edu* [online]. Pennsylvania: University of Pennsylvania, 2010, 2010 [cit. 2022-04-10]. Dostupné z: https://www.lehigh.edu/~sub216/local-files/tech_report_FULL_25932.pdf
[15] VAN DEN BERG, Jur, Stephen J. GUY, Jamie SNAPE, Ming C. LIN a Dinesh MANOCHA. Optimal Reciprocal Collision Avoidance. *Https://gamma.cs.unc.edu* [online]. North Carolina: University of North Carolina, 2011, May 2011 [cit. 2022-04-10]. Dostupné z: https://gamma.cs.unc.edu/ORCA/
[16] SORIANO MARCOLINO, Leandro, Yuri TAVARES DOS PASSOS, Alvaro SOUZA a Luiz CHAIMOWICZ. Avoiding target congestion on the navigation of robotic swarms. *Www.researchgate.net* [online]. New York: Autonomous Robots, 2017, Aug 2017 [cit. 2022-04-10]. Dostupné z: https://www.researchgate.net/figure/Execution-screenshots-of-the-ORCA-algorithm-video-available-at_fig27_303905805
[17] BNAYA, Zahy a Ariel FELNER. Conflict-Oriented Windowed Hierarchical Cooperative A∗. *Www.semanticscholar.org* [online]. Washington: Zahy Bnaya, 2014, 29 September 2014 [cit. 2022-04-10]. Dostupné z: https://www.semanticscholar.org/paper/Conflict-Oriented-Windowed-

Hierarchical-Cooperative-Bnaya-Felner/922a36b25c8be6115cb7f82c684590aa9c3978cf

[18] TAWIL, Yahya. An Introduction to Robot Operating System (ROS). *Www.allaboutcircuits.com* [online]. Idaho: All about circuits, 2017, 26 June 2017 [cit. 2022-05-10]. Dostupné z: https://www.allaboutcircuits.com/technical-articles/an-introduction-to-robot-operating-system-ros/

[19] LALANCETTE, Chris Lalancette. Tutorials. *Docs.ros.org* [online]. California: ros.org, 2022, 24 Mar 2022 [cit. 2022-05-10]. Dostupné z: https://docs.ros.org/en/foxy/Tutorials.html

[20] LOVEDALE, Antony. 3WD 48mm Omni Wheel Robot Kit 10019. *Www.active-robots.com* [online]. UK: Active Robots [cit. 2022-05-10]. Dostupné z: https://www.active-robots.com/3wd-48mm-omni-wheel-robot-kit-10019.html

SEZNAM PŘÍLOH

PŘÍLOHA A - ZDROJOVÝ KÓD PROGRAMU JE ULOŽEN NA PŘILOŽENÉM CD 44

Příloha A - Zdrojový kód programu je uložen na přiloženém CD