VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

MOBILNÍ KAMERA REALIZOVANÁ PROSTŘEDKY ROS2

THESIS TITLE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DANIEL ONDERKA**
AUTHOR

VEDOUCÍ PRÁCE **doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

155048

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Onderka Daniel**
Program:	Informační technologie
Název:	**Mobilní kamera realizovaná prostředky ROS2**
Kategorie:	Vestavěné systémy
Akademický rok:	2023/24

Zadání:

1. Nastudujte problematiku mobilní robotiky a teleprezence. Nastudujte metody tvorby softwaru pro mobilní roboty. Seznamte se s middlewarem ROS2. Seznamte se s demonstrační aplikací pro čtyřkolového robota Adeept AWR 4WD.
2. Navrhněte systém řízení robota Adeept AWR 4WD s využitím ROS2. Systém poběží distribuovaně na RPI4 na robotovi a na stacionárním PC. Systém umožní teleprezenci a přepínání mezi dálkovým ovládáním a autonomním pohybem s vyhýbáním se překážkám.
3. Navržený systém realizujte tak, aby ho bylo možné považovat za demonstraci možností ROS2, včetně dynamické rekonfigurace řídicího software.
4. Realizovaný systém otestujte a diskutujte možnosti dalších rozšíření. Součástí odevzdané práce bude plakát a demonstrační video.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:
Body 1 a 2.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Janoušek Vladimír, doc. Ing., Ph.D.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1.11.2023
Termín pro odevzdání:	9.5.2024
Datum schválení:	6.11.2023

Abstrakt

Tato práce se zabývá middlewarem ROS2 a konkrétně je jejím cílem prozkoumat a demonstrovat možnosti tohoto systému. V práci byl vytvořen systém k ovládání rozšířeného robota Adeept AWR. Kromě čistého ROS2 se práce také zabývá souvisejícími technologiemi jako Gazebo Simulátor do kterého byl přenesen fyziký robot a mapováním s navigací pomocí slam_toolbox a navigation2.

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Klíčová slova

Robot Operating System 2, robotika, Gazebo Simulátor,

Keywords

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

Citace

ONDERKA, Daniel. *Mobilní kamera realizovaná prostředky ROS2*. Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Vladimír Janoušek, Ph.D.

Mobilní kamera realizovaná prostředky ROS2

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana X... Další informace mi poskytli... Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Daniel Onderka
20. dubna 2024

Poděkování

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant apod.).

Obsah

Seznam obrázků

Kapitola 1

Úvod

Tato práce se zabývá tematikou robotiky. Primárním zaměřením je systém pro řízení robotů jménem Robot Operating System 2. ROS2 jak z názvu vyplývá je již druhá verze těchto nástrojů. V minulosti se originální ROS stal de facto standardem pro vývoj softwaru k řízení robotů. Většina práce tedy řeší převážně softwarovou stranu této problematiky. Obor robotiky jako takový se však pohybuje velice blízko hardwaru a tak se tato práce dotýká také některých hardwarových konceptů a principů potřebných k pochopení fungování použitých komponent.

Jak už bylo zmíněno hlavním zaměřením práce je samotný ROS2. Hlavní část práce se tedy snaží demonstrovat funkcionalitu a možnosti tohoto middleware. Obsahem této části je tvorba systému, který bude využívat nástroje ROS2 k ovládání robota. Jako hardware nad kterým bude celá práce implementována byla zvolena stavebnice Adeept AWR 4WD. Kromě čistého ROS2 se práce zaměřuje také na související nástroje a systémy, které nějakým způsobem využívají nebo rozšiřují funkcionalitu ROS2. Prvním z nich je ros2_control knihovna pro řízení robotů. Následně je zasaženo využití lidar seznoru pro mapování a navigaci robota v prostoru. Jako poslední se práce také dotýká Gazebo simulátoru, pro vývoj a testování softwaru před použitím přímo na hardwarovém robotu.

První polovina této dokumentace se zaměřuje na vysvětlení teoretičtějších konceptů souvisejících s danou problematikou. Nejprve se zaměřuje na použitý hardware a principy fungování jednotlivých komponent. Následně přechází na samotný ROS2. Tato část se první podívá na to co ROS2 vlastně je a jak vnitřně funguje. Následně jsou pak vysvětleny koncepty, které používají vývojáři při interakci a vývoje výsledného ROS2 systému. Na konci teoretické části jsou ještě vysvětleny principy související s rozšířeními. Začátek praktické části nejprve ukazuje část výsledného systému která je implementována pouze s využitím ROS2. Zde se projdou uzly pro ovládání hardwarových komponent a vyšší uzly pro řízení celého robota. Dále pak následují jednotlivá rozšíření. Je zde zaměření na řízení motorů pomocí ros2_control, zprovoznění simulátoru a přemostění aby mohly ros2 uzly řídit model v simulátoru. A jako poslední se řeší nastavení a získání potřebných dat ze zbytku ros2 systému pro úspěšné mapování a navigaci.

Kapitola 2

Použitý Hardware

ROS2 je nástroj sloužící k tvorbě robotických aplikací. Z toho důvodu je k demonstraci jeho možností vyžadováno použití nějakého fyzického robota, nad kterým bude aplikace implementována. V této práci byl jako demonstrační robot použit model Adeept AWR 4WD. Z této stavebnice pochází většina komponentů. V pozdější fázi byl tento základ rozšířen o další rozšiřující hardware. Ten byl následně použit k demonstraci pokročilejších konceptů. Mozkem robota je mikropočítač Raspberry Pi 4.

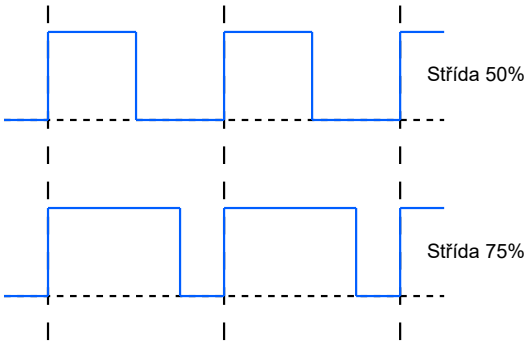
2.1 HW Technologie

V první sekci budou představeny obecné hardwarové technologie. Jedná se o relativně známé koncepty. V souvislosti této práce je využívají některé z použitých komponent.

Pulzně šířková modulace

Umožňuje vytvořit pseudo-analogový výstupní signál na číslicových pinech mikrokontroleru. Mikrokontroléry jsou digitální zařízení a chtěly by tedy s okolním světem komunikovat pomocí jedniček a nul. Reálný svět tak ovšem nefunguje a proto je často potřeba převádět výstup z mikrokontroléru na analogový signál. Problém je v tom, že převod digitálního signálu na analogový je relativně dlouhá a neefektivní operace. Proto vznikla pulzně šířková modulace (PWM), která umožňuje relativně jednoduše simulovat analogový výstup.

PWM využívá toho, že člověk nedokáže rozpoznat rychlé změny, například led dioda blikající na frekvenci 5000 Hz se člověku jeví jako by svítila permanentně. Mechanická zařízení také mívají relativně velkou latenci a dc motoru tedy nevadí, že místo konstantního analogového napětí dostává periodický číslicový signál.

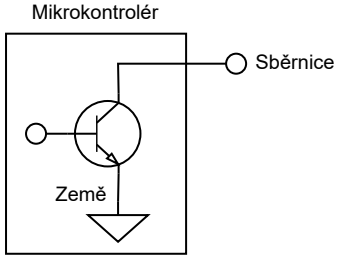


Obrázek 2.1: PWM signál pro různé hodnoty střídy

Při pohledu na klasický digitální signál který rovnoměrně střídá vysokou a nízkou úroveň by šlo říci, že se jedná o PWM signál se střídou 50%. Střída(duty cycle) udává poměr času, kdy je signál v logické jedničce, ku času, kdy je v nule. Součet těchto hodnot se musí rovnat délce jedné periody. Úpravou tohoto poměru lze simulovat analogový signál. [3, str: 116-118]

I2C

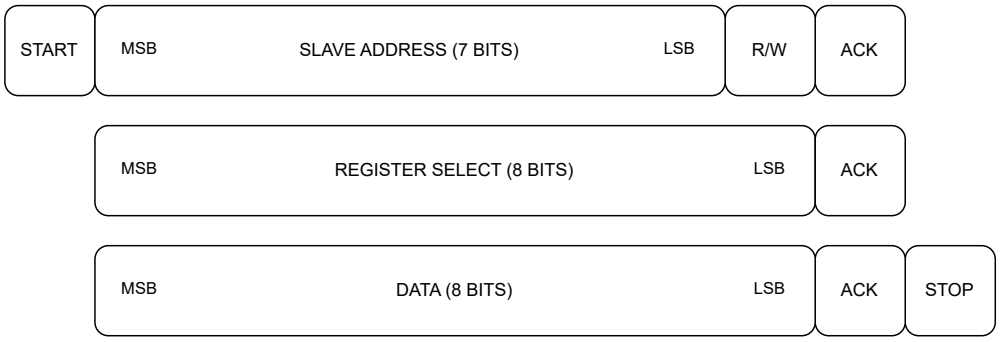
Je synchronní sběrnice, která se vyznačuje svou jednoduchostí a nízkou cenou. Využívá dva vodiče SDA(serial data) a SCL(serial clock). Oba vodiče jsou připojeny k napájecímu napětí pomocí pull-up rezistoru a bez vlivu jiného hardwaru zůstávají v logické jedničce. Zařízení, která jsou na tuto sběrnici připojeny, využívají open drain k úpravě aktuální napěťové úrovně. I2C pracuje s dvěma druhy zařízení, master a slave. Master zahajuje, řídí a ukončuje komunikaci na vodiči SDA. Po dobu průběhu komunikace také generuje hodinový signál na SCL. Typicky se jedná o mikrokontroler. Slave jsou pak ostatní zařízení s nimiž může master komunikovat, typicky různé periferie. [3, str: 88]



Obrázek 2.2: Open drain

Přenos jednoho datového rámce zahájí master zařízení přivedením datové sběrnice do nuly. Následující komunikace se skládá z odeslání rámce o délce osmi bitů a potvrzení o úspěšném přenosu dat od přijímajícího zařízení. Toto potvrzení se nazývá ACK a je provedeno podržením datové sběrnice v hodnotě nula po dobu jednoho taktu. Opačný stav se nazývá NACK a indikuje že nastala chyba. Ukončení přenosu je provedeno navrácením datové sběrnice na hodnotu jedna. [5]

Na obrázku lze vidět, jak může vypadat přenos jednoho datového slova. V prvním rámci je přenesena sedmi bitová adresa, identifikující slave zařízení se kterým chce máster navázat komunikaci. Osmý bit datového rámce indikuje směr, kterým budou posílány data. V dru-



Obrázek 2.3: Datové slovo sběrnice I2C

hém rámci dojde k adresaci konkrétního registru na slave zařízení. A ve třetím, případně dalších, již probíhá samotné posílání dat mezi zařízeními. [10]

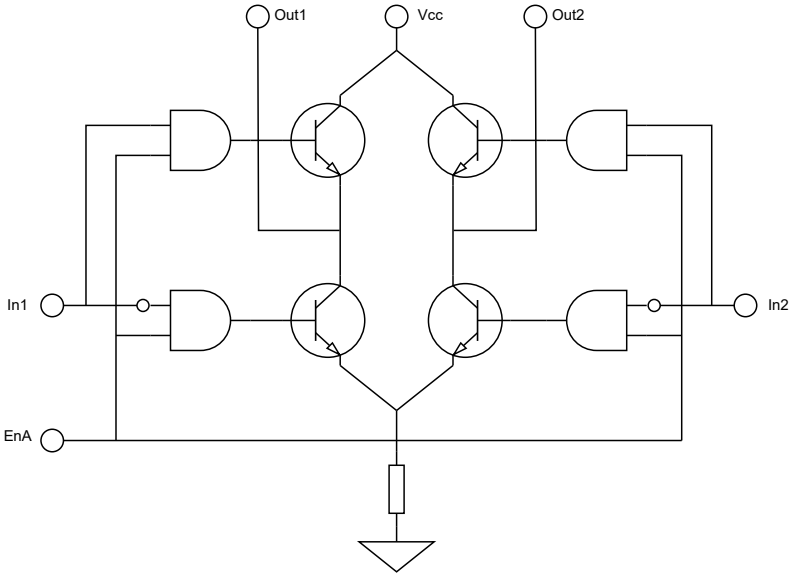
2.2 Adeept AWR 4WD

Tato sekce se již zaměří na použitého robota a konkrétně na komponenty, které jej tvoří. U každé součástky je cílem vysvětlit princip jejího fungování z teoretičtějšího pohledu. A také poukázat její účel na tomto konkrétním robotu.

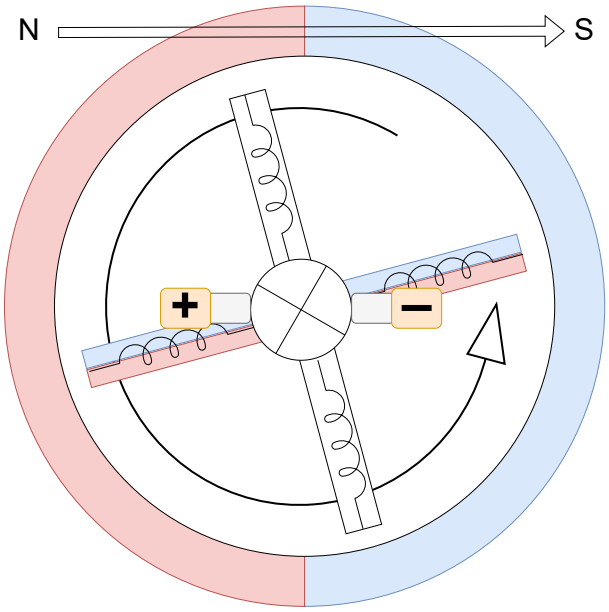
Robot HAT

HAT(hardware attached on top) je hardwarová deska, která slouží k rozšíření funkcionality mikrokontroléru. Tato konkrétní se k Raspberyy Pi připojuje pomocí GPIO(general purpuse input outpu) pinů. Deska jako taková obsahuje rozšiřující čipy a rozhraní sloužící k ovládání připojených periferií.

- PCA9685 [6]
  - generátor PWM signálu
  - 16 kanálů
  - střída s rozlišením 12 bitů(4096 možných hodnot)
  - ovládání přes I2C sběrnici
- L298P [12]
  - ovladač pro řízení dc motoru
  - základem je full bridge obvod
  - umožňuje roztočit motor oběma směry
  - pomocí PWM lze ovládat rychlost motorů
  - připojuje motor na externí napájení
- další rozhraní pro připojení periferií (sledování čáry, ultrazvukový senzor, led)



Obrázek 2.4: Full bridge konfigurace pro ovládání motoru. In1 a In2 určují směr otáčení. EnA je PWM signál určující rychlost otáčení. [12]



Obrázek 2.5: Schéma DC motoru

DC Motor

Pohyb celého autíčka zajišťují čtyři stejnosměrným proudem(direct current) napájené motory. Ovladač motorů L298P je umístěný na Robot HAT.

Elektrický DC motor se skládá ze dvou hlavních částí, stator a rotor. Stator je statická, vnější část, a typicky se jedná o permanentní magnet. Uvnitř statoru se pak nachází rotor, ten se skládá z elektromagnetů, které při zapnutí reagují se statorem(opačné póly se přitahují a stejné odpuzují) a dojde tak k částečnému pootočení. Při správném spínání a vypínání těchto elektromagnetů lze motor rozběhnout. Toto střídání zajišťuje prstenec

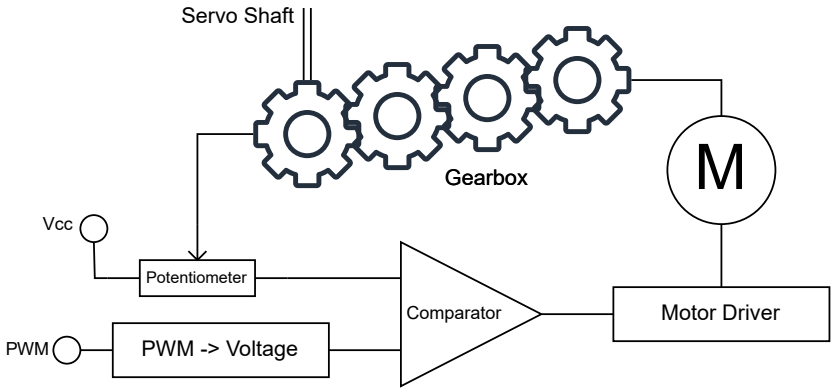
zvaný komutátor. Komutátor je rozdělen na několik od sebe odizolovaných částí, ke kterým jsou připojeny vývody elektromagnetů. S povrchem prstence jsou pomocí pružin v kontaktu dva kartáče. Tyto kartáče se již neotáčí a mohou tak být připojeny na zdroj napájení a zem. Komutátor se otáčí společně s rotorem a při tomto pohybu se kartáče postupně dotýkají různých částí komutátoru a spínají tak jednotlivé elektromagnety, ty zajistí pootočení rotoru a sepnutí následujícího magnetu. [1]

Robot s diferenciálním podvozkem

Tento konkrétní robot disponuje čtyřmi těmito motory. Ty jsou pevně připevněny k tělu robota. Z toho důvodu je zatáčení realizováno diferenciálním způsobem. Tento přístup využívá toho, že motory nejsou na sobě navzájem závislé a můžou se tedy otáčet různými rychlostmi. Pokud se kola na jedné straně robota otáčejí rychleji, urazí větší vzdálenost. Z toho pak vyplývá, že výsledné trajektorie již nejsou dvě rovnoběžky, ale soustředné kružnice. V porovnání s ostatními přístupy pro realizaci pohybu robota se jedná o konstrukčně jednodušší řešení, protože nevyžaduje natáčení kol do stran. Další výhodou je možnost otáčení robota na místě.

Servo

Servo je komponenta na první pohled velmi podobná DC Motoru. Na rozdíl od něj se však neotáčí donekonečna, ale bývá omezena nějakým úhlem, například 180 stupňů. Hlavní výhodou a důvodem pro použití serva je plná kontrola nad úhlem natočení jeho hřídele.[3, str: 119-121]



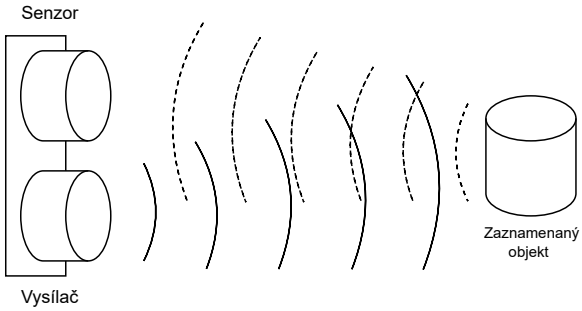
Obrázek 2.6: Vnitřní zapojení serva

Při pohledu na vnitřní zapojení serva lze zjistit, že se prakticky jedná o klasický dc motor připojený na převodovku a rozšířený o elektroniku na jeho řízení. K nastavení úhlu serva se využívá PWM signál. Ten je první přeložen na napěťovou úroveň, která je porovnána s aktuálním natočením serva a výsledek udává směr, kterým se bude otáčet motor. Aktuální natočení serva je získáno využitím potenciometru zapojeného na výstupní hřídel serva.

Ultrazvukový senzor hloubky

V podstatě se jedná o sonar. Slouží k určení vzdálenosti. Senzor zahajuje měření po přijetí pulzu na trig pinu. Následně je odesláno 8 pulzů na frekvenci 40Hz. Výsledkem měření je

časový interval mezi odesláním pulzu a přijetím ozvěny. Tato informace se předává zpět nastavením echo pinu do hodnoty jedna na dobu rovnou výsledku měření. [3, str: 93]



Obrázek 2.7: Ultrazvukový senzor

Pro výpočet vzdálenosti lze využít následující vzorec:

$$S = \frac{(T_2 - T_1) * V_S}{2}$$

Kde T_1 je moment vyslání pulzu, T_2 moment zachycení ozvěny a V_S rychlost šíření zvuku ve vzduchu (cca 340m/s). Výsledek se pak dělí dvěma, protože doba $T_2 - T_1$ je rovna času k překážce a zpět.

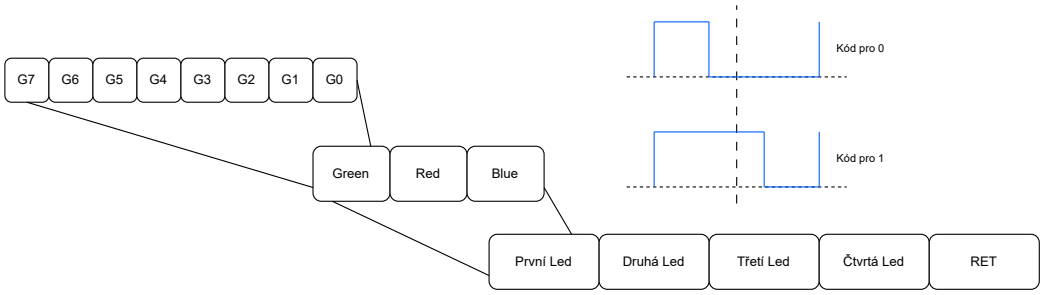
Třícestný senzor pro sledování čáry

Modul využívá fakt, zě intenzita světla odraženého od povrchu je závislá na barvě dané plochy. Například černá barva pohltí téměř veškeré světlo, naopak bílá téměř vše odrazí. Používáno je infračervené záření, protože není ovlivněno okolními zdroji světla, odráží se od velkého množství materiálů a je přesné. Jedná se o třícestný modul a skládá se tedy ze setu tří vysílačů a senzorů. Pokud vysílač svítí a senzor nezaznamenává dostatečnou intenzitu odraženého světla, znamená to, že byla nalezena černá čára. [3]

WS2812 RGB LED

V nejjednodušším případě jsou diody k mikrokontroleru připojeny napřímo pomocí GPIO pinů. Při použití většího počtu ledek se tento přístup stává nepoužitelným. Jedním z řešení je použití WS2812. Jedná se o druh adresovatelných led diod. Tento přistup umožňuje připojit několik set diod pomocí pouze tří vodičů. Konkrétně je použit jeden datový vodič, napájení a země.[14]

Diody jsou na pásku zapojeny sériově. Každá dioda má DIN a DO port. Pokud dioda přijme data, která jí nejsou určená, přeposílá je dále. Komunikace vždy začíná klidovým stavem, datový vodič je v nule. Datové slovo se skládá z 24 bitových bloků. Jeden pro každou diodu. Blok obsahuje tři osmi bitové hodnoty. Jednu pro každou barevnou složku (MSB je posíláno první). Diody pak fungují tak, že přijmou prvních 24 bitů, podle kterých nastaví svou barvu. Tuto část odeberou z datového slova a zbytek přeposílají na výstup. [14]



Obrázek 2.8: Komunikační protokol pro WS2812 led

2.3 Rozšíření Hardware komponent

V minulé sekci byly představeny ty komponenty, které jsou součástí Adeept Kitu. V průběhu vývoje ROS2 systému se objevovaly požadavky na další rozšíření, které by dokázaly vylepšit výsledný produkt. Proto bylo originální hardwarové vybavení doplněno o následující komponenty.

Inertial measurement unit

Jedná se o druh senzoru, jehož úkolem je určování orientace a pozice v prostoru. Použitý IMU disponuje třemi senzory. Pro měření rotace je obsazen tříosý gyroskopem. Pro určení zrychlení, případně rychlosti, disponuje také tříosým akcelerometrem. Poslední funkcí tohoto senzoru je ještě zabudovaný teploměr. Ten však není v této práci použitý.

MEMS

Pod pojmem gyroskop si snad každý představí mechanickou rotující součástku držící si svou orientaci. Tento přístup však nelze použít na miniaturních čipech IMU senzorů. MEMS je zkratka pro micro-electromechanical systems. Jak z názvu vyplývá jedná se o systémy využívající mechanické elementy ve velikostech typických pro elektronické součástky. Běžně se jedná o velikosti pod $100\mu m$.

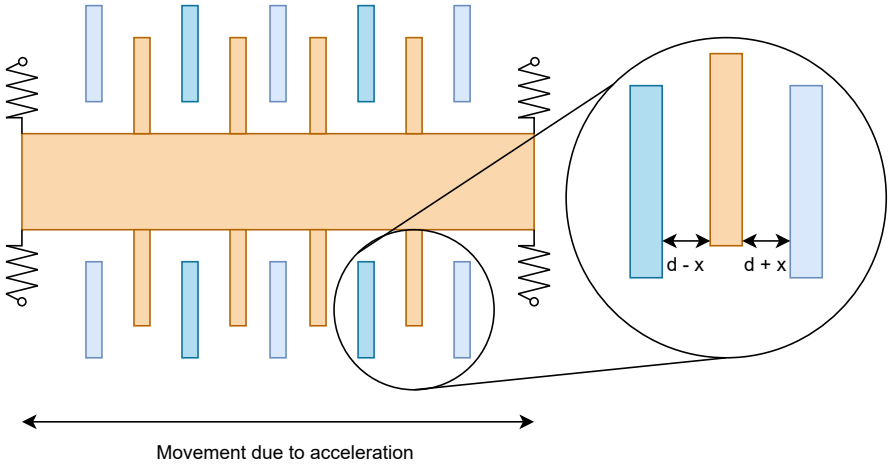
Princip Akcelerometru

Následující obrázek zobrazuje vnitřní strukturu akcelerometru. Senzor se skládá ze dvou hlavních částí. Tou první je pohyblivé oranžové závaží. Protože má nezanedbatelnou hmotnost, je ovlivněno vnějšími vlivy. V případě, že se změní akcelerace působící na senzor, dojde také ke změně relativní pozice mezi tímto závažím a zbytkem senzoru. Tento pohyb je měřen a určuje výslednou akceleraci. Obě části, pohyblivá i statická obsahují desky, které tvoří části kondenzátorů. Při pohybu vnitřní části, dojde také k změně vzdálenosti mezi těmito deskami a tedy i změně kapacity kondenzátorů. [2]

Základní rovnice pro výpočet kapacity je následující:

$$C = \epsilon \frac{A}{d}$$

Kde C je výsledná kapacita, A plocha desek ϵ permitivita prostředí mezi nimi a d jejich vzdálenost.



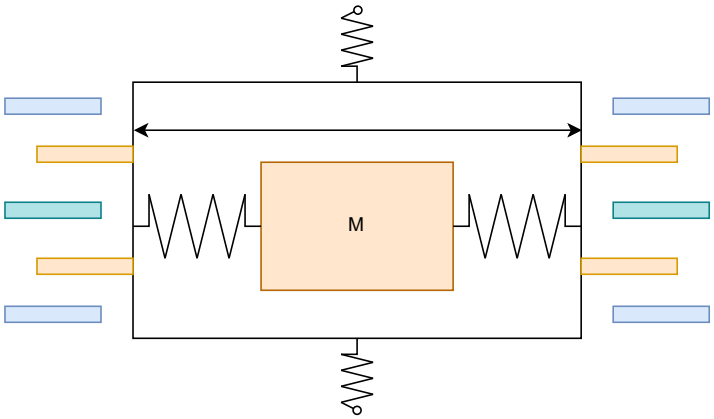
Obrázek 2.9: Vnitřní struktura MEMS akcelerometru

Statické desky jsou zapojeny střídavě a rozdíl mezi jejich kapacitami pak odpovídá posunutí vnitřní části.

$$x \approx d\frac{\Delta C}{C_0}$$

Princip Gyroskopu

MEMS gyroskop využívá podobné principy jako akcelerometr. Skládá se ze dvou částí, vnitřní a vnější(senzorický) rám. Ve vnitřním rámu je umístěno pohyblivé závaží. Na rozdíl od akcelerometru není tato část v klidovém stavu statická ale je rozvibrována do harmonické oscilace podél osy x. Vnitřní rám je celý umístěn pohyblivě, uvnitř senzorického rámu. V momentě kdy dojde k rotaci gyroskopu, bude si vnitřní oscilující část snažit držet svou orientaci. To způsobí pohyb celého vnitřního rámu podél osy y. Vnější a vnitřní rámy jsou stejně jako u akcelerometru obsazeny deskami kondenzátorů. Změna v jejich kapacitě je opět hlídána a přepočítána na rotaci.



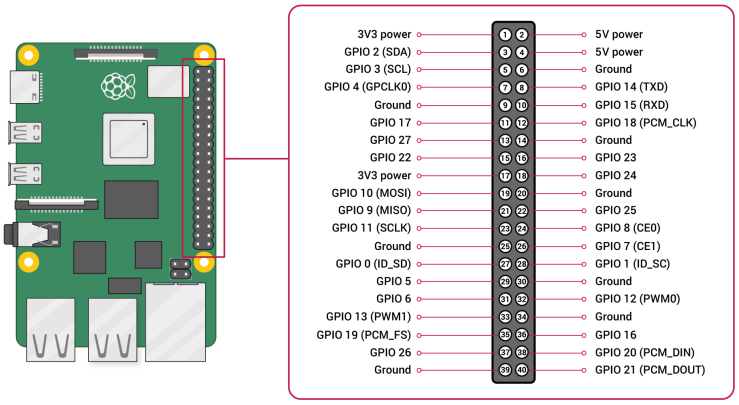
Obrázek 2.10: Vnitřní struktura MEMS gyroskopu

Lidar

Principiálně se jedná o senzor podobný ultrazvukovému. Hlavním rozdílem je, že lidar používá pro měření laser. Data získaná lidarem tak bývají z pravidla přesnější oproti sonaru. Hlavní důvod přidání lidaru však není jeho vyšší přesnost. Tento konkrétní model totiž umožňuje měření v celých 360° okolo robota. Díky tomu lze data získaná z tohoto senzoru použít ka mapování a lokalizaci robota v prostoru.

2.4 Raspberry Pi 4b

Jako mozek celého systému je použit mikropočítač Raspberry Pi. Konkrétně se jedná o verzi 4 model B s operační pamětí o velikosti čtyř gigabajtů. Tato verze disponuje 64bitovým procesorem, který je potřeba pro spuštění 64bitového Ubuntu serveru. Jedná se o doporučený operační systém pro běh ROS2 na platformě Raspberry Pi. Komunikace s většinou použitých periferií je uskutečněna pomocí General Purpuse Input Output(GPIO) pinů. Jedná se o číslicové vývody, které podle potřeby můžou fungovat jako vstup i výstup ze zařízení. Některé z nich pak mají ještě speciální funkce, například GPIO 2 a 3 můžou pracovat jako SDA a SCL připojení pro I2C komunikaci.



Obrázek 2.11: GPIO pinout pro Raspberry Pi

Kamera

Přímo k Rapsberry Pi je připojena oficiální camera module 3. Tento modul dokáže nahrávat video až v rozlišení 2304 × 1296 pixelů a 56 snímcích za vteřinu. [8]

Kapitola 3

Robot Operating System 2

Tato kapitola slouží k představení ROS2. Začátek se zaměří na vnitřní fungování systému. Následně budou představeny koncepty, na kterých je ROS2 postavený a které jsou potřebné pro vývoj aplikací nad tímto middlewarem. Závěr této kapitoly pak projde další části ROS2 jako nástroje příkazové řádky, spouštění uzlů a transformační subsystém.

3.1 Aktuální software

Robot Adeept AWR 4WD je dodáván s ukázkovým softwarem. Ten je implementován v jazyce Python a využívá knihovny třetích stran sloužící k nízkoúrovňovému ovládání hardwarových komponent. Aby byl robot responzivní je celá implementace řešena s použitím python modulů pro multithreading.

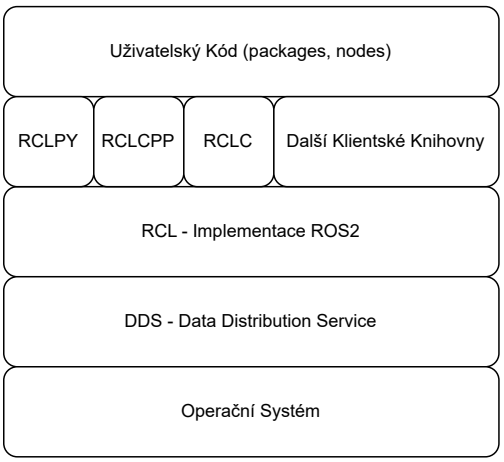
3.2 Seznámení s ROS2

ROS2 je middleware sloužící k vývoji a řízení robotů. Middleware je softwarová vrstva běžící nad operačním systémem. Jejím úkolem je rozšíření operačního systému o další funkcionalitu. Typickou součástí middlewaru bývají knihovny, ovladače, vývojové a monitorovací nástroje. Může také specifikovat doporučené metodologie pro vývoj. ROS2 je již druhá verze tohoto softwaru, která rozšiřuje a opravuje neduhy první verze. Původní ROS1 je považován za de-facto standart pro vývoj robotických aplikací. Tato práce využívá ROS2 distribuci jménem iron. Distribuce v ROS2 lze popsat jako set operačního systému, knihoven a dalších aplikací, které jsou otestovány a je zaručeno, že jsou navzájem kompatibilní. Velkou výhodou ROS je fakt, že se jedná o open source projekt. Díky tomu kolem něj vznikla velká komunita vývojářů, ale i firem a dalších institucí, které tvoří mnoho souvisejícího obsahu. Existuje tedy velké množství knihoven, dokumentací a návodů které usnadňují vývojářům práci. [9]

Vrstvy ROS2

Na nejvyšší úrovni, se nachází programátor, který interaguje s klientskými knihovnami pro vývoj ROS2 aplikací. Tyto knihovny jsou oficiálně dvě a to rclpy pro Python a rclcpp pro C++. Existují také implementace pro další programovací jazyky (rclc, java, C#), které jsou udržovány komunitně. Všechny klientské knihovny pak využívají RCL. To je jádrem celého ROS a obsahuje implementaci všech ROS2 funkcionalit. Je napsáno v jazyce C a

jeho součástí je rozhraní, pomocí kterého poskytuje svou funkcionalitu ostatním klientským knihovnám. Díky tomuto přístupu se uzly implementované v Pythonu budou chovat stejně jako ty implementované v C++. Z toho pak také vyplývá, že uzly implementované na různých klientských knihovnách spolu mohou bez problémů komunikovat. Poslední vrstvou je data distribution service. DDS je komunikační vrstva implementována na UDP protokolu sloužící k předávání informací mezi procesy. Má charakteristiky systémů reálného času, zajišťuje kvalitu a zabezpečení komunikace. Také umožňuje vyhledávání uzlů bez potřeby centralizovaného serveru (vyhledávání je realizováno s využitím multicastové komunikace, zprávy zasílané mezi jednotlivými uzly pak využívají klasický unicast). [9]



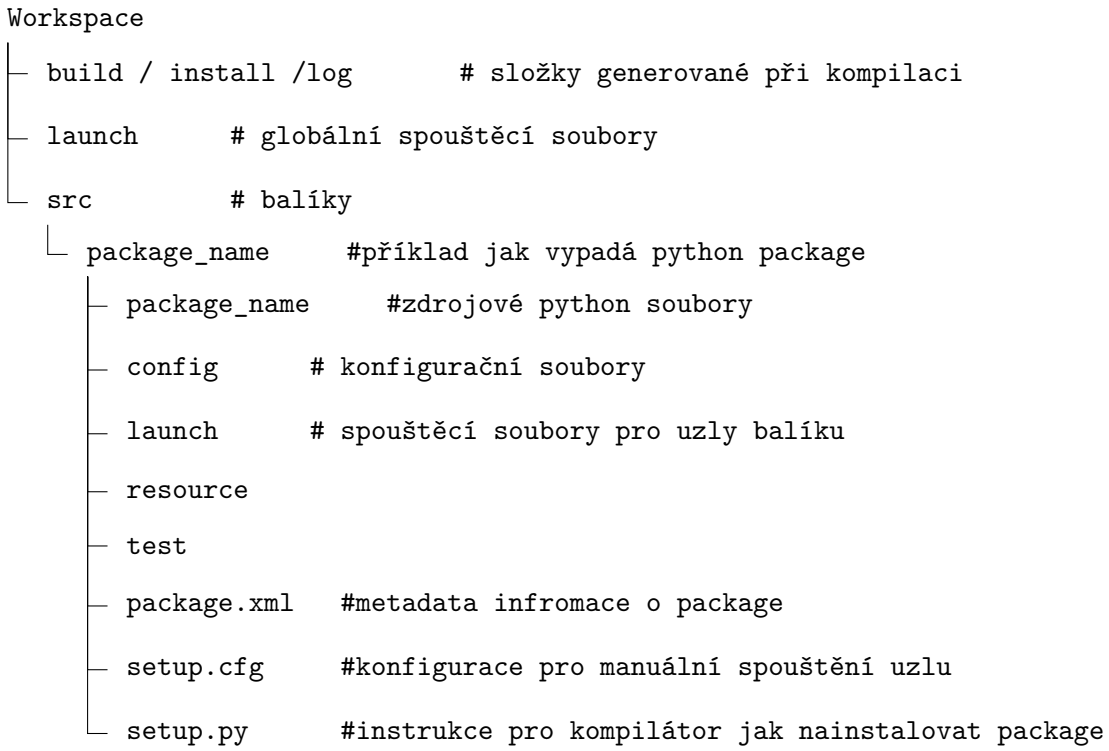
Obrázek 3.1: Vrstvy ROS2 systému

Vývoj v ROS2

Nejvyšší organizační jednotkou v ROS2 je workspace. Jedná se o složku, která slouží k organizaci zdrojových souborů, jejich instalaci a následné spouštění. ROS2 jako takový se také kvalifikuje jako workspace a před použitím musí být nejprve aktivován. K tomu v linuxu slouží příkaz `source`. Aktivace workspace je akumulativní, to znamená, že v jeden moment může být aktivních několik workspace najednou. Typicky se první aktivuje základní ROS2 instalace, která tvoří takzvanou underlay vrstvu. Vývojový workspace aktivovaný jako druhý, se pak nazývá overlay. Pokud má overlay nějaké závislosti, měly by být uspokojeny v underlay. Zdrojové soubory v rámci workspace jsou pak organizovány do balíků(packages). Balík může obsahovat zdrojové soubory, knihovny a definice zpráv. Balíky na sobě můžou navzájem záviset (například balík, který využívá konkrétní rozhraní, závisí na jiném který toto rozhraní definuje). [9]

Node

Celý ROS2 systém je složený z uzlů (node), které mezi sebou navzájem komunikují. Každý uzel je vlastní výpočetní jednotka, která by měla plnit jeden specifický úkol. Tento přístup je podobný objektově orientovanému návrhu. Implementačně je uzel objekt, který dědí ze třídy `Node`. Uzly v ROS2 většinou nepotřebují běžet permanentně, ale pouze v monentě, kdy nastane nějaká událost, kterou je potřeba obsloužit. Z toho důvodu existuje v ROS2 metoda `spin()`, která uspí vykonávání uzlu, dokud jej není potřeba opět využít. Aby ROS2 šetřil výpočetní prostředky, využívá dva přístupy k určení, kdy bude potřeba uzel vzbudit. Prvním



Obrázek 3.2: Struktura ROS2 Workspace

je iterative execution. Ten se používá u uzlů, které vykonávají svou činnost pravidelně, na nějaké předem dané frekvenci. Příkladem může být uzel pro obsluhu senzoru. Ten se v pravidelných intervalech zaktivuje, získá nová data, nějakým způsobem je zpracuje a následně odešle dalším uzlům, které je potřebují pro jejich činnost. Druhým přístupem je event oriented execution. Zde je vyvolání řídícího cyklu důsledkem nějaké události. Typickou událostí je příchod zprávy od ostatních uzlů. Frekvenci spouštění těchto uzlů pak lze odvodit od frekvence příchozích zpráv. Příkladem může být uzel přijímající snímky z kamery na kterých provede výpočet a vrátí odpověď. Frekvence spuštění je dána příchozími snímky, pokud přestanou přicházet, uzel bude neaktivní. [9] [11]

Algoritmus 1: Definice a použití Node objektu

```
1: class CustomNode(Node):
2:     def __init__(self):
3:         super().__init__('node_name')
4: def main(args):
5:     rclpy.init(args=args)
6:     node = CustomNode()
7:     rclpy.spin(node)
8:     node.destroy_node()
9:     rclpy.shutdown()
```

Interface

Jak už bylo řečeno tak uzly ROS2 systému spolu komunikují posíláním zpráv. Aby si uzly navzájem rozuměly, musí mít tyto zprávy stejnou strukturu. K tomuto účelu slouží rozhraní(interface).

ROS2 obsahuje mnoho již vytvořených a vývojáři po celém světě používaných formátů. Tento přístup podporuje znovupoužitelnost vytvořeného kódu a šetří práci. Díky tomu může být software pro ovládání konkrétního kusu hardware implementován pouze jednou s využitím standardního rozhraní a všichni ostatní jej pak mohou využít ve svých systémech. Pokud však standardní interface nevyhovuje potřebám, lze si implementovat vlastní. Více o definici vlastních rozhraní v

V kódu má každé rozhraní vygenerovanou vlastní třídu sloužící k jeho reprezentaci. Instance této třídy jsou používány jako zprávy posílané mezi uzly. Data přenášená těmito zprávami se ukládají do atributů. Atributy těchto objektů mají strukturu danou originálním rozhraním.

Topic

Je základním a také nejčastěji používaným způsobem pomocí kterého spolu ROS2 uzly komunikují. Topic si lze představit jako analogii hardwarové sběrnice. Prakticky se jedná o přesně pojmenované místo, do kterého může n uzlů posílat data (Publish) a m poslouchat co bylo posláno (Subscribe). Zprávy posílané do topicu mají přesný formát a jsou posílány asynchronně. Příkladem použití může být topic, do nějž posílá data uzel ovládající kameru a několik dalších uzlů které tyto data potřebují jej mohou číst. [9]

Algoritmus 2: Subsrciber Node

```
1: self.create_subscription(Interface, "topic_name",
   self.callback_function, qos_profile)
2: def callback_function(self, msg: Interface):
3:     value = msg.item
```

Tento kód ukazuje, jak se může uzel přihlásit k odebírání zpráv z topicku. Nejprve je potřeba (typicky v konstruktoru třídy) zavolat zděděnou metodu sloužící k inicializaci nějaké ROS2 funkcionality. V tomto případě se jedná o `create_subscription`. Jako parametry potřebuje jméno, interface, callback funkci a QoS profil. Callback funkce je volána když uzel příjme zprávu a jako parametr je jí předán objekt reprezentující přijatou zprávu. [11]

Algoritmus 3: Publisher Node

```
1: self.publisher = self.create_publisher(Interface, "topic_name",
   qos_profile)
2: output = Interface()
3: output.item = some_value
4: self.publisher.publish(output)
```

Odesílání zpráv do topicu je velmi podobné poslouchání. Jak bylo řečeno, zprávy jsou instance tříd reprezentující rozhraní. Proto je potřeba tuto instanci vytvořit, naplnit daty a následně předat metodě `publish()` předem vytvořeného publisheru. [11]

Service

ROS2 služby(service) fungují na stejném principu jako klient–server komunikace známá z počítačových sítí. Jedná se tedy o synchronní komunikaci, kde jeden uzel poskytuje nějakou službu a ostatní si na ni mohou poslat požadavek. Od serverového uzlu se předpokládá okamžitá odpověď, aby nedošlo k narušení řídícího cyklu volajícího uzlu. [9]

Algoritmus 4: Service server

```
self.srv = self.create_service(Interface, "service_name",
self.callback_function)
def callback_function(self, request, response):
    value = request.item
    response.item = some_value
    return response
```

Tento kód demonstruje vytvoření služby. Hlavní změnou oproti předchozím příkladům jsou parametry callback funkce. Ty jsou nyní dva, request a response. Principiálně se používají stejně jako u topicků. Jejich atributy tvoří strukturu přijaté / odesílané zprávy. Odpověď se tentokrát neodesílá metodou serveru, ale vrací se jako výsledek funkce. [11]

Algoritmus 5: Service client

```
self.cli = self.create_client(Interface, "service_name")
while not self.cli.wait_for_service(timeout_sec=1.0):
    pass
def send_request(self):
    self.req = Interface.Request()
    self.req.item = some_value
    self.future = self.cli.call_async(self.req)
    rclpy.spin_until_future_complete(self, self.future)
    response = self.future.result()
    value = response.item
```

Klient potřebuje pro své fungování již existující službu. Tato podmínka vyplývá z faktu, že služby jsou určeny k úkolům, které lze vykonat relativně rychle a odpovědět na požadavek v krátkém čase. Klientská strana proto čeká synchronně. V případě že by servis neexistoval, klient by se zasekl v nekonečném čekání. Proto je hned v konstruktoru implementována kontrola, které nedovolí vytvoření uzlu dokud není přítomen server. Čekání na odpověď od serveru, je pak implementována pomocí funkce `spin_until_future_complete()` [11]

Action

Jedná se o rozšířenou verzi služeb. Akce z pravidla vykonává déle trvající požadavek. Například provedení řídícího manévru robota, který je prováděn v reálném světě a z pohledu uzlu se nejedná o krátkodobou záležitost. Akce, na rozdíl od služby, dokáže v průběhu vykonávání své činnosti odesílat průběžné aktualizace o aktuálním stavu zpět volajícímu uzlu. Implementačně akce funguje jako dvě služby a jeden topic. Cílová(goal) služba slouží k zaslání požadavku na server a jeho potvrzení. Výsledková(result) pak vrací výsledek operace. V průběhu akce pak server posílá aktualizace do topicu. [9]

Algoritmus 6: Action server

```
1: self.action_server = ActionServer(self, Interface, "action_name",
   self.execute_callback)
2: def execute_callback(self, goal_handle):
3:     goal_handle.request.item
       // odeslání zpětné vazby volajícímu
4:     feedback = Interface.Feedback()
5:     feedback.item = some_value
6:     goal_handle.publish_feedback(feedback)
       // úspěšné ukončení požadavku
7:     goal_handle.succeed()
8:     result = Interface.Result()
9:     result.item = some_value
10:    return result
```

Implementace akčního serveru kombinuje postupy představené u služeb a topiců dohromady. Jedinou novinkou je parametr `goal\_handle`. Umožňuje interagovat s vnitřní implementací akcí. Obsahuje v sobě přijatý požadavek, publisher zpětné vazby a také ovládání samotného akčního serveru. [11]

Algoritmus 7: Action client - zaslání požadavku

```
1: self.action_client = ActionClient(self, Interface, "action_name")
2: def send_goal(self):
3:     goal_msg = Servo.Goal()
4:     goal_msg.item = some_value
5:     self.action_client.wait_for_server()
6:     self.goal_future = self.action_client.send_goal_async(goal_msg,
       self.feedback_callback_function)
7:     self.goal_future.add_done_callback(self.response_callback_function)
```

V porovnání s klientem služeb je ten akční výrazně složitější. Tento klient interaguje se serverem, který z pohledu uzlů vykonává velmi dlouhé úkoly. Komunikace je proto asynchronní a skládá se z několika callback funkcí. [11]

Prvním krokem je odeslání požadavku, to je podobné jako u služeb. Hlavním rozdílem je předání callback funkce pro zpětnou vazbu a pro odpověď na požadavek.

Algoritmus 8: Action client - reakce na přijmutí nebo zamítnutí požadavku

```
1: def response_callback_function(self, future):
2:     goal_handle = future.result()
3:     if not goal_handle.accepted:
4:         return
5:     self.result_future = goal_handle.get_result_async()
6:     self.result_future.add_done_callback(self.result_callback_function)
```

Zpracování odpovědi bývá vždy téměř totožné. Cílem je zjistit zda byl požadavek přijat. Pokud ano, tak následuje odeslání dotazu na výsledek operace a následné zaregistrování callback funkce. [11]

Algoritmus 9: Action client - callback funkce

```
1: def feedback_callback_function(self, msg):
2:     feedback = msg.feedback
3:     value = feedback.item
4: def result_callback_function(self, future):
5:     result = future.result().result
6:     value = result.item
```

Tyto bloky už pouze ukazují jak vypadají jednoduché funkce pro zpracování zpětné vazby a získání výsledku operace.

Definice vlastních rozhraní

Jak už bylo řečeno, ROS2 disponuje standardními rozhraními. Existují ale případy, kdy tyto rozhraní nevyhovují a je potřeba si vytvořit vlastní. Každý druh komunikace mezi uzly má svou vlastní příponu pro definování rozhraní. [9]

Prvním jsou `.msg` zprávy. Tento formát je využívám topicy. Jedná se o seznam, kde je každá položka definována jako dvojice datový typ a název (případně komentář).

```
int32 angle #comment
string direction
```

Druhým je `.srv`. Slouží pro definici struktury požadavků a odpovědí zasílaných mezi službou a jejími klienty. Tento soubor obsahuje dvě části, požadavek a odpověď, každá je tvořena seznamem položek a jsou odděleny řádkem `---`.

```
int32 a
int32 b
---
int64 sum
```

Poslední je `.action` soubor. Slouží pro komunikaci mezi akčním serverem a klientem. Definice se skládá ze tří seznamů, jeden pro požadavek, druhý pro odpověď a poslední pro zpětnou vazbu.

```
float32 goal_angle
---
bool response
---
float32 current_angle
```

Parametry

ROS2 uzly mohou definovat parametry. V základu fungují jako argumenty funkcí. Umožňují předat dynamické hodnoty při spuštění uzlu. Příkladem může být uzel, sloužící k obsluze periferního zařízení. Parametrem mu jsou předány čísla GPIO pinů, na které je dané zařízení připojeno. Tyto parametry jsou však pokročilejší. Každý uzel zdědí z `Node` třídy set služeb. Ty lze volat za běhu uzlu a umožňují jednak získání aktuální hodnoty parametrů ale také je změnit. Následující kód demonstruje deklaraci a použití parametrů uvnitř uzlu. [9]

Algoritmus 10: PARAMETERS

```
   // deklarace parametru, typicky v konstruktoru
1: self.declare_parameter('parameter_name', 'default_parameter_value')
   // získání hodnoty parametru
2: param =
   self.get_parameter('parameter_name').get_parameter_value().string_value
   // nastavení hodnoty parametru
3: new_param = rclpy.parameter.Parameter(
4:     'parameter_name',
5:     rclpy.Parameter.Type.STRING,
6:     'default_parameter_value'
7: )
8: new_param_list = [new_param]
9: self.set_parameters(new_param_list)
```

Konfigurační soubory

Cílem konfiguračních souborů je usnadnit předávání parametrů spouštěným uzlům. Jejich hlavní výhodou je schopnost nastavit velké množství parametrů

Konfigurační soubory se typicky umisťují do složky `config` v kořenovém adresáři balíku a používají formát `.yaml`.

Následující kód demonstruje strukturu konfiguračních souborů.

Launch File

ROS2 systém se skládá z velkého množství navzájem komunikujících uzlů. Spouštění každého uzlu zvlášť by bylo pracné a zdlouhavé. Proto existují spouštěcí(launch) soubory, které mají za úkol nastartovat ROS2 systém. Možnými formáty pro psaní těchto souborů jsou `Python`, `yaml` a `xml`. V této práci jsou použité `Python` spouštěcí soubory. Základní struktura těchto souborů je vždy stejná. Funkce, kterou musí každý definovat se jmenuje `generate_launch_description`. Návratovou hodnotou je objekt `LaunchDescription` jehož obsahem jsou jednotlivé cíle, které mají být vykonány. Následující kód obsahuje příklad

Algoritmus 11: CONFIG FILE

```
  // parametry pro konkrétní uzel
1: node_name:
2:   ros__parameters:
3:     int_param: 16
4:     double_param: 3.14 #comment
5:     string_param: "radians"
  // wildcard, parametry pro všechny uzly
6: /**: ...
```

jednoduchého spouštěcího souboru s dvěma cíli. Jedním z nich je klasické spuštění uzlu. Je mu předán parametr a jeden z jeho topiců je přemapován na jiné jméno. Přemapování je užitečná funkcionalita, která pomáhá s kompatibilitou uzlů. Druhý cíl pak volá jiný spouštěcí soubor. Běžně se spouštěcí soubory používají minimálně na dvou úrovních. Vyšší se nacházejí na úrovni workspace. Jejich cílem je spuštění větší části ROS2 systému. Používají k tomu volání nižších spouštěcích souborů. Ty se nacházejí na úrovni balíků. Mají za úkol spustit a nakonfigurovat konkrétní uzel. [9]

Nástroje příkazové řádky

Jak už bylo řečeno tak ROS2 systémy se skládají z velkého množství navzájem komunikujících uzlů. Taková struktura je vhodná na organizaci a modularitu. Naopak pro ladění chyb v rozsáhlejším systému by mohla být problematická. ROS2 proto disponuje sadou nástrojů, které mají za úkol pomoci vývojářům zjišťovat informace o právě běžících uzlech. Obecný formát volání je:

```
ros2 [command] [sub_command]
```

Názvy příkazů jsou odvozené od jednotlivých ROS2 konceptů. Například `node`, `topic`, `service` a `param`. Podpříkazy pak reprezentují akce, které budou nad daným konceptem provedeny. Typicky se jedná o `list` a `info`, případně další, specifické pro konkrétní příkaz. Díky této struktuře a dobrému pojmenování jsou velice intuitivní a příjemné na používání. Zde je několik ukázkových a často používaných příkazů.

```
#prints all active topics
ros2 topic list

#subsribes to specific topic and prints out received messages
ros2 topic echo /topic_name

#prints structure of specified interface
ros2 interface show /path_to_interface

#prints node and all of its publishers, subscribers, services, actions
ros2 node info /node_name
```

Algoritmus 12: Launch file

```
def generate_launch_description():
    // spuštění konkrétního uzlu
    goal = Node(
        package='package_name',
        executable='node_name',
        namespace='namespace_name',
        parameters=[{
            'param_name' : param_value,
        }],
        remappings=[
            ('topic_name', 'different_topic_name'),
        ]
    )
    // zavolání dalšího launch souboru
    other_goal = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([
            PathJoinSubstitution([
                FindPackageShare('package_name'), 'launch',
                'node_name.py'
            ])
        ]),
    )
    return LaunchDescription([
        goal,
        other_goal
    ])
```

Geometric Transformation Subsystem

Jedná se o subsystém ROS2, který realizuje geometrické transformace mezi jednotlivými částmi robota. Vztah mezi dvěma objekty lze definovat pomocí posunu (translation) a otočení (rotation). Matematicky jsou tyto složky reprezentovány maticemi, které po spojení vytváří výslednou transformační matici. Roboti se většinou skládají z velkého množství částí. Ty na sebe bývají navzájem zavěšeny a zároveň se jejich relativní pozice neustále mění. Není proto vhodné počítat tyto vztahy manuálně. A přesně z toho důvodu existuje TF.

$$\begin{pmatrix} x_B \\ y_B \\ z_B \\ 1 \end{pmatrix} = \begin{pmatrix} R^{xx}_{A\to B} & R^{xy}_{A\to B} & R^{xz}_{A\to B} & T^{x}_{A\to B} \\ R^{yx}_{A\to B} & R^{yy}_{A\to B} & R^{yz}_{A\to B} & T^{y}_{A\to B} \\ R^{zx}_{A\to B} & R^{zy}_{A\to B} & R^{zz}_{A\to B} & T^{z}_{A\to B} \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x_A \\ y_A \\ z_A \\ 1 \end{pmatrix}$$

Matice pro manuální výpočet tranformace

Základním prvkem se kterým TF pracuje je takzvaný frame, neboli rám. Rám reprezentuje nějakou část robota, jako senzor, kolo a podobně. Tyto rámy jsou uspořádány do stromové struktury, kde má každý uzel vždy jednoho předka. Podle konvence je kořenem

rámem `base_link`. Ten se fyzicky nachází ve středu robota. Dalším běžným rámem je `odom`. Ten reprezentuje vztah mezi aktuální pozicí robota a počátkem souřadného systému.

Pro předávání dat mezi uzly jsou využívány dva topicy.

- `/tf` - dynamické transformace, komponenty připojené přes serva, motory
- `/tf_static` - statické transformace, komponenty navzájem pevně spojené

Pro interakci s transportním subsystémem se nepoužívají běžné subsribery a publishery ale speciální objekty k tomu určené. Pro zasílání nových dat do TF systému slouží `TransformBroadcaster`. Ten se používá vesměs stejně jako klasický publisher.

Algoritmus 13: Transform Broadcaster

```
self.tf_broadcaster = tf2_ros.TransformBroadcaster(self)

transform = TransformStamped()
transform.header.stamp = current_time.to_msg()
transform.header.frame_id = parent_frame
transform.child_frame_id = child_frame
transform.transform.translation.x = translation
transform.transform.rotation.w = quaternion
self.tf_broadcaster.sendTransform(transform)
```

Pro získání dat z tf systému pak slouží `TransformListener`. Ten neslouží k jednoduchému čtení zpráv posílaných v topicu, ale umožňuje dotazování se tf systému na konkrétní transformace. Dotaz se skládá ze specifikování dvou rámců, mezi kterými je transformace hledána. Tyto dva rámce nemusí být v tf stromu přímí potomci. Cesta od jednoho k druhému může vést přes několik uzlů, ale dokud jsou navzájem dosažitelné, tak tf vrátí výslednou transformaci. Druhou částí dotazu je čas, ten je potřebný protože, pokud je cesta mezi rámy delší, může se i během několika milisekund výrazně změnit výsledná transformace a použití aktuálního času tak není vhodné. Specifikování času také umožňuje získávat pozice z minulosti.

Algoritmus 14: Transform Listener

```
self.tf_buffer = Buffer()
self.tf_listener = TransformListener(self.tf_buffer, self)

t = self.tf_buffer.lookup_transform(
    to_frame_rel,
    from_frame_rel,
    rclpy.time.Time()
)
```

3.3 Formáty pro popis Robotů

V robotice se často objevuje potřeba definovat strukturu a fyzikální vlastnosti robotů, objektů případně prostředí, ve kterém se nachází. Nejzjevnějším případem jsou simulátory

ale existují i jiné použití kde programy potřebují znát aktuální stav, odometrii robota, a k tomu je nejprve potřeba znát jeho stukturu.

Unified Robotics Description Format

URDF je primární formát, který ROS2 používá k popisu robotů. Samotný kód je složený primárně z `<link>` a `<joint>` prvků. Link(článek) reprezentuje fyzické části robota jako jeho tělo či kola. Články jsou složené ze tří složek. Těmi jsou fyzikální `inertial` vizuální `visual` a kolizní `colision`. Vizuální a kolizní složky bývají většinou stejné. Oddělené definice umožňují použít u komplexních objektů jednodušší kolizní složku a šetřit tak výkon. Fyzická složka pak definuje hmotnost a matici setrvačnosti. Joint(kloub) pak tvoří nějaký vztah mezi linky. Jointy můžou být fixní nebo různými způsoby pohyblivé. Pod pohyblivostí je samozřejmě možná rotace kolem některé z os. Ale také je umožněno klouzání podlé některé z os.

XML Macro

Xacro lze použít na jakékoli XML soubory, ale primárně se používá k generování URDF souborů. Cílem tohoto formátu je zjednodušení URDF souborů, které se při popisu složitějších robotů stávají dlouhé a náchylné na chyby. Xcarco tedy přidává funkce sloužící k eliminaci těchto nedostatků. První z nich je možnost definovat konstanty, v urdf se často opakují stejné hodnoty na několika místech, jednoduchým případem může být definice vizuální a kolizní části <link> elementů které bývají často totožné.

```
<xacro:property name="wheel_radius" value="0.035" />
```

Na konstanty pak navazuje vkládání matematických výrazů, místo statických hodnot.

```
<cylinder radius="${wheel_diameter/2}" length="0.1"/>
```

Hlavní funkcionalitou tohoto formátu jsou však makra. Ty umožňují zaobalit blok kódu a přiřadit mu identifikátor pomocí kterého lze takové makro vkládat na další místa v kódu. Makra mohou brát na vstupu také parametry. V těle makra lze pak tyto parametry vkládat a v kombinaci s matematickými výrazy vytvářet komplexní definice.

```
<xacro:macro name="identificator" params="name mass:=default_value">
```

Posledním rozšířením které xacro oproti urdf prináší je možnost rozdělení definice robota do více souborů.

```
<xacro:include filename="components.xacro"/>
```

Simulation Description Format

SDF je formát založený na XML. Jak z názvu vyplývá, slouží k popisu robotů, ale také světů ve kterých se budou následně pohybovat. Jedná se o primární formát využívaný Gazebo simulátorem.

3.4 Mapování a Navigace

Problém mapování a navigace je komplexní problematika, jejichž kompletní porozumění výrazně přesahuje rozsah této práce. Nástroje jako `slam_toolbox` a `Navigation 2` umožňují provádět mapování a navigaci na téměř jakémkoli robotovi. Jejich velkou výhodou je fakt, že je lze používat i bez hloubkového porozumění této problematiky.

Simultaneous Localization And Mapping

SLAM řeší problém tvorby mapy neznámého prostředí a současné lokalizace robota na této mapě. Vysokoúrovňový princip fungování SLAM zobrazuje následující algoritmus. První krok, observation, získává data ze senzorů a extrahuje z nich významné vlastnosti(features) okolí. Druhým krokem je data association. Jeho úkolem je asociovat získané vlastnosti okolí s významnými body(landmarks) mapy. Další tři kroky se použijí v případě, kdy algoritmus ztratí informaci o aktuální pozici robota. Využitím kombinace dat ze senzorů, odometrie a dalších algoritmů mají za úkol lokalizovat aktuální pozici robota. Poslední dvojice pak slouží k zpřesnění výsledné mapy. Při pohybu robota vznikají malé odchylky v odometrii i mapě. Tyto odchylky se postupně sčítají a delší trasy tak můžou mít relativně velkou chybu. Loop closure má za úkol zaznamenat moment, kdy se robot vrátí do pozice, ve které již v minulosti byl. S touto informací může následně upravit mapu tak, aby byla tato odchylka eliminována. [7]

Algoritmus 15: SLAM

```
repeat:
    (1)Observation
    (2)Data association
    if tracking failure detected then
        (3)Relocalitation
        (4)Motion estimation
        (5)Optimization
    if loop closure detected then
        (6)Loop closure correction
        (7)New landmark initialization
```

Tento algoritmus pochází z [7]

Navigation 2

Cílem knihovny Navigation 2 je pokročilé řízení robotů v prostředí. Pomocí mapy a transformací robota zvládá navigovat roboty skrze rozsáhlá prostředí. K rozhodování tato knihovna používá Behaviorální stromy. Požadavky na navigaci jsou přijímány pomocí akcí. [4]

V ROS2 systému je Nav2 tvořeno několika navzájem provázanými a spolupracujícími uzly. Dalo by se říct, že se jedná o svůj vlastní podsystém. Hlavními uzly, které Nav2 používá jsou `Planner`, `Controller`, `Smoother` a `Recovery`. Všechny tyto uzly jsou modulární. Jejich chování tak lze měnit použitím různých pluginů. Planner má za úkol vytvořit trajektorii pro navigaci. Úkolem Controlleru je řízení robota podél vypočítané trajektorie do cíle. Recovery pak řeší případy, kdy se robot odchýlí od plánované trajektorie. [4]

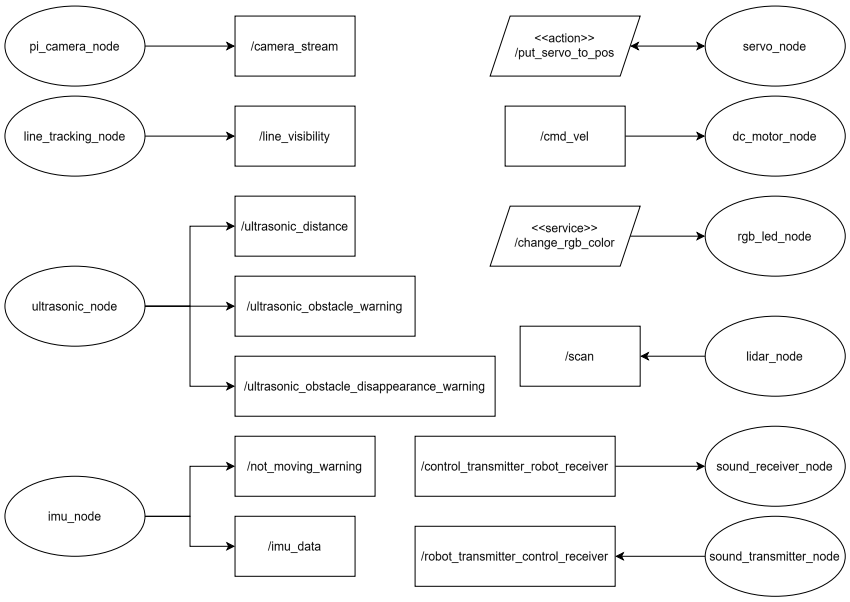
Kapitola 4

Implementace ROS2 systému

Zaměřením této kapitoly je vytvořený ROS2 systém. Následující strany popisují principy použité k implementaci uzlů, ale také demonstrují praktické příklady použití ROS2 funkcionalit.

4.1 Uzly pro řízení komponent

Z pohledu vzdálenosti od hardwaru se jedná o nejnižší uzly. Úkolem těchto uzlů je využít rozhraní poskytnuté danou komponentou k jejímu ovládání a následnému zpřístupnění její funkcionality zbytku ROS2 systému.



Obrázek 4.1: RQT Graf všech hardwarových uzlů a jejich rozhraní

Komponenty součástí adeept kitu

První skupinou probraných uzlů budou komponenty pocházející z Adeept Kitu.

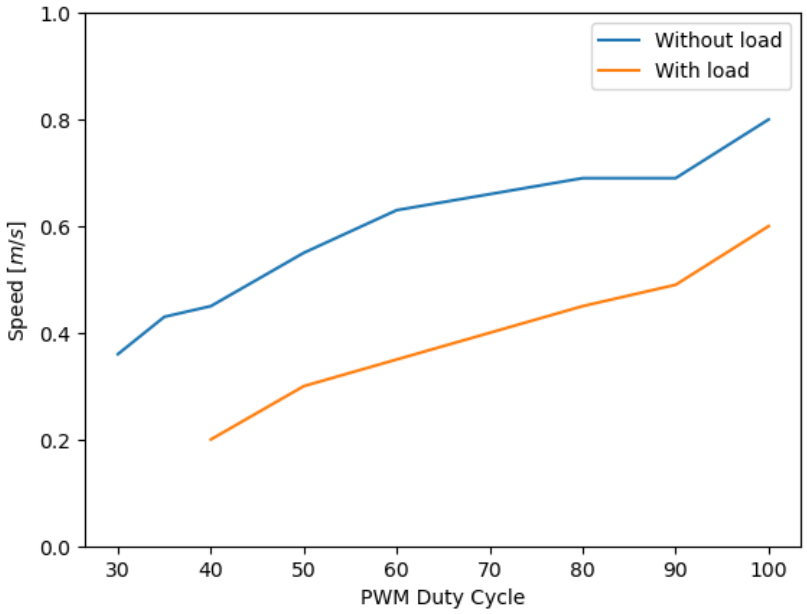
Motory

Jak už bylo řečeno v teoretické části, součástí Robot HAT je také full-bridge ovladač. Řídící uzel `dc_motor_node` s tímto obvodem interaguje přímo pomocí GPIO pinů. K ovládání GPIO pinů byl použit Python modul `RPi.GPIO`. Zbytek řízení motorů už realizuje přímo uzel nastavováním hodnot vývodů `In` a úpravou PWM frekvence na vývodu `En`.

Od zbytku ROS2 systému pak uzel přijímá příkazy pomocí `cmd_vel`(command velocity) topicu. Jedná se o běžně používané jméno pro zasílání příkazů na pohyb robota. Zprávy v tomto topicu jsou typu `geometry_msgs/msg/Twist`. Robot s diferenciální nápravou se může pohybovat pouze v přímo vpřed / vzad, nebo otáčet do stran. Z Twist zprávy jsou pro něj tedy důležité pouze dvě složky `linear.x` a `angular.z`. Pomocí nich lze spočítat výsledná rychlost otáčení kol na jedné a druhé straně.

$$\omega_L = \frac{V - \omega * b/2}{r}$$
$$\omega_R = \frac{V + \omega * b/2}{r}$$

Kde ω_L a ω_R jsou výsledné rychlosti motorů v $[rad/s]$. V je zadaná lineární a ω angulární rychlost v $[m/s]a[rad/s]$. b je rozpětí mezi koly a r poloměr kola v $[m]$. [13]



Obrázek 4.2: Lineární rychlost robota pro různé hodnoty střídy motorů

Tento graf zobrazuje lineární rychlost robota pro různé hodnoty PWM střídy motorů. Měření pod zátěží bylo zaznamenáno jako vzdálenost, kterou robot ujede za jednu vteřinu. Hodnoty bez zátěže byly získány pomocí změření času jedné otáčky hřídele motoru. Získaný čas byl následně přepočítán na vzdálenost. Je zde vidět, že hodnoty střídy menší, než 30-40% nedokážou překonat fyzikální jevy(setrvačnost, tření, ...) aby motory vůbec roztočily. Zároveň růst není zcela lineární. Pro potřeby řízení motoru jsou hodnoty dostatečně blízko aby je řídící uzel za lineární považoval.

Problém s motory, který se pořádně projevil až u jednoho z rozšíření je jízda po oblouku (současný pohyb vpřed a zatáčení). Nejsem si zcela jistý co je příčinou tohoto jevu, ale

hodnoty vypočítané využitím předchozího vzorce nevedou na předpokládaný oblouk. K ujištění, že není chyba na mé straně, byl v pozdější fázi využit framework ros2_control k řízení motorů. Ten realizuje výpočet kinematiky vlastním vestavěným ovladačem a výsledek byl stejný. Zde je důležité zmínit, že robot se po oblouku pohybovat dokáže, pouze na to vyžaduje větší rozdíl mezi rychlostmi kol. Předpokládanou příčinou je použití levných motorů, protože pokud je na jeden z nich vyvinuta větší váha(stačí aby se motor na opační straně dostal z důvodu nerovnosti povrchu do vzduchu) často se přestane otáčet a naopak když ostatní motory jedou vyšší rychlostí můžou být ty na druhé straně „taženy“ za nimi i přes to že by se měly otáčet pomaleji.

Servo

Adeept AWD 4WD je vybaven pouze jedním servem, které slouží k ovládání úhlu natočení kamery. Konkrétní použitý model je Adeept AD002. Úhel serva se nastavuje pomocí PWM signálu. Jeho generování zajišťuje čip PCA9685, který je součástí Robot HAT. V softwaru je k jeho řízení využita knihovna `Adafruit_PCA9685`. Řídící uzel serva se jmenuje `servo_node` a z pohledu ROS2 systému se jedná o demonstraci jednoduchého action serveru. Server příjme požadavek a zkontroluje limity. Rozsah pohybu serva je větší než prostor pro kameru, řídící uzel proto omezuje maximální a minimální hodnoty natočení serva. Následně pomalu otáčí servem a v průběhu odesílá feedback zprávy s aktuálním úhlem.

Kamera

Zachytávání snímků kamery je realizováno pomocí knihovny OpenCV. Aby bylo možno dosáhnout relativně krátké odezvy při přehrávání videa, není obraz zaznamenáván v nativním rozlišení a snímkovací frekvenci kamery, ale byly použity snížené hodnoty. V základním konfiguraci je použito rozlišení 960 na 540 pixelů a snímkovací frekvence 20hz. Pro přenos snímků v ROS2 systému je použita zpráva vestavěného typu `CompressedImage`. Před odesláním jsou ještě data zakódována do `jpeg` formátu pomocí funkce z `OpenCV` knihovny.

Formát	Base64	Image	CompressedImage
Velikost zprávy:	0,24 MB	1,56 MB	0,17 MB

Hodnoty v této tabulce pocházejí z utility `$ros2 topic bw`

Před finálním rozhodnutím o použití `CompressedImage` formátu bylo experimentováno také se zprávami typu `Image`. Tento typ obsahuje více rozšiřujících informací o přenášeném obrázku jako jeho rozměry, použité kódování a podobně. Na první pohled se tedy zdá jako vhodnější formát. Snímky kamery jsou ale přenášené mezi dvěma různými zařízeními pomocí wifi připojení. Tento objemově větší formát tak přináší příliš velké zpoždění, které překonává použitelné hranice pro streamování. Na počátku vývoje byl použit ještě třetí možný přístup. Ten však nevyužívá ROS2 funkcionality a proto byl později změněn. Tímto přístupem je zakódování přenášených snímků do `base64` formátu a následné přenesení jako jednoduchou `string` zprávu. Base64 kódování se ukázalo jako zcela funkční možnost a dokonce dosahuje podobných výsledků ve zpoždění jako má `CompressedImage`.

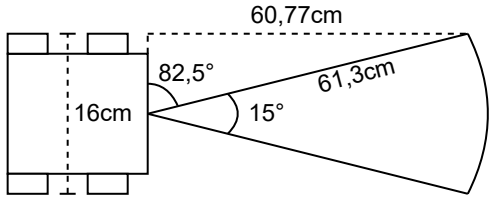
Ultrazvukový senzor

Jedná o model hc-sr04. Ten dokáže měřit vzdálenost od 2cm do 400cm s přesností na 3mm. Získávání dat z ultrazvukového senzoru je relativně jednoduché a nevyžaduje tedy externí

knihovnu. Interakce se senzorem je zajištěna pomocí GPIO pinů. K jejich ovládání je opět použit modul `RPi.GPIO`.

Uzel zahajuje komunikaci vysláním pulzu na trig vodiči. V ten moment začne senzor měřit a uzel jen čeká na odpověď. Ta přijde na echo pinu. Její začátek je signalizován tím, že senzor nastaví echo pin na hodnotu jedna. V ten moment si uzel uloží časovou značku. Poté počká, než se echo pin vrátí zpět do nuly. V ten moment získává druhou značku. Pomocí těchto dvou momentů lze vypočítat výslednou vzdálenost. Výpočet vzdálenosti je podrobněji popsán v teoretické části (str: 8).

Do ROS2 systému tento uzel odesílá kromě pravidelné informace o aktuální vzdálenosti také dvě další zprávy. Tou první je jednoduché varování o detekci překážky. Uzel varuje pokud je naměřená vzdálenost menší než zadaná hranice. Toto varování však zaznamená pouze překážky přímo před robotem. Druhá zpráva tak doplňuje tu první. Snaží se detekovat překážky, které můžou stále vést ke kolizi, ale senzor je již nedetekuje.



$$\cos(82.5°) = \frac{8\text{cm}}{\text{prepona}} \Rightarrow 61.3\text{cm}$$

$$\sin(82.5°) = \frac{\text{protilehla}}{61.3\text{cm}} \Rightarrow 60.77\text{cm}$$

Obrázek 4.3: Překážky přehlédnutelné ultrazvukovým senzorem

Tento obrázek demonstruje případy, které se snaží druhé varování zachytit. Je na něm vidět, že existuje meziprostor, ve kterém může překážka zmizet ze senzoru, ale i přes to vést ke kolizi s robotem. Tento meziprostor je na jedné straně ohraničen vzdáleností pro klasické detekování překážek, například 20cm. Na druhé straně lze hranici vypočítat pomocí úhlu ve kterém senzor detekuje překážky a šířky robota. V tomto případě se jedná o 61cm. Druhá zpráva tedy informuje o tom, že nějaká překážka zmizela z radaru v této potenciálně nebezpečné zóně.

Sledování čáry

Posledním praktickým senzorem kterým robot Adeept AWR disponuje je třícestný senzor na sledování čáry. Komunikace je v tomto případě ještě jednodušší. Senzor má tři výstupy připojené na tři GPIO piny. Každá dvojice vysílače a senzoru má svůj vlastní vývod. Hodnoty těchto výstupů obsahují aktuální stav detekování čáry. Z pohledu uzlu tak stačí pravidelně číst hodnoty a odesílat je dále do ROS2 systému. K práci s GPIO piny tento uzel také používá modul `RPi.GPIO`.

Led

Součástí Adeept sady je také několik adresovatelných led diod. Jedná se o hardware, který je pro hlavní funkcionalitu naprosto nepotřebný. Ponechat však na robotu nevyužitý a neovládatelný hardware nedává smysl. Proto byl nakonec implementován uzel na jejich řízení. Z pohledu ROS2 se jedná o klasický service server. Požadavky obsahují RGB barevnou hodnotu, pro nastavení všech led zároveň. Řízení tohoto typu led pásku je komplexnější záležitost a proto je k jeho ovládání využit python modul `rpi_ws281x`. Aby tento modul mohl správně pracovat, vyžaduje spuštění s vyššími oprávněními. Důvodem k tomuto požadavku je fakt, že vnitřně využívá kód v jazyce C, který pracuje s GPIO piny přímo přes

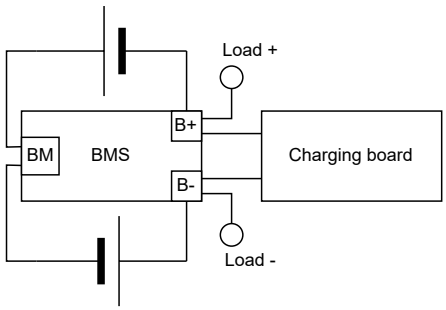
/dev/mem. Problém nastává v tom, že ROS2 při svém běhu nevyžaduje a tedy ani nepoužívá zvýšená oprávnění. Z toho důvodu na tyto případy není připravený a prosté přidání `sudo` před spouštěcí příkaz nefunguje. Toto lze obejít a spustit uzel s oprávněními podobnými příkazu `sudo`. Avšak uzly spuštěné tímto způsobem nedokážou komunikovat s klasicky spuštěnými uzly. Problém je ve výsledku vyřešen použitím druhého skriptu. Ten obsahuje jen minimum kódu pro ovládání zmíněného modulu. V `sudoers` je tomuto konkrétnímu souboru umožněno spuštění se `sudo` oprávněními, aniž by bylo vyžadováno heslo. Tento skript je následně volán z hlavního uzlu použitím `subprocess` modulu. Jedná se o velmi neelegantní řešení. Vzhledem k tomu, že se svícení LED diodami není důležitá funkcionalita a zbytečně upravuje systémově důležitý soubor `sudoers`, je tento krok v instalačním skriptu separátně volitelný.

Další hardware nad rámec Adeept Kitu

V tento moment byly probrány všechny uzly, které řídí komponenty stavebnice Adeept Awr 4WD. Následuje popis rozšíření původního hardwaru o další komponenty, které buď rozšiřují funkcionalitu robota, nebo usnadňují jeho použití.

Nabíjení

V originální konfiguraci napájí robota dvě sériově zapojené 18650 baterie. V tomto zapojení poskytují napětí 8V s maximálním proudem 4A. Součástí Adeept Kitu jsou kolébky na baterie. Ty jsou přímo připojené k zátěži(robot HAT). Nabíjení baterií musí být realizováno externě. Tento způsob je nepohodlný hlavně proto, že přístup ke kolébkám vyžaduje sundání kol robota. Prvním rozšířením je tedy přidání BMS a nabíjecí desky. Nejrozšířenějším způsobem nabíjení baterií 18650 jsou desky s čipem TP4056. Jedná se o hojně dostupný produkt, který lze koupit v různých kombinacích ochran a vstupů pro napájení. Problémem je fakt, že tyto desky jsou určeny pro nabíjení jedné 18650 baterie. Potenciálním řešením pro paralelní zapojení je použití více těchto desek. Pro baterie zapojené sériově, které používá tento robot, jsou tyto desky nevhodné. Lepší přístup je využití Battery Management System. Jak z názvu vyplývá, jedná se o desku pro správu baterií. BMS se vyrábí specificky pro konkrétní zapojení, které bude monitorovat. Použitý model je typu 2S, pro dvě sériově zapojené baterie. Kromě kladného a záporného pólu disponuje také připojením pro bod mezi bateriemi. Tento bod umožňuje desce monitorovat stav každé z baterií zvlášť. V kombinaci s BMS je dále využita samotná nabíjecí deska. Jedná se o model, který na rozdíl od TP4056 nabíjí dvakrát vyšším nabíjecím napětím.

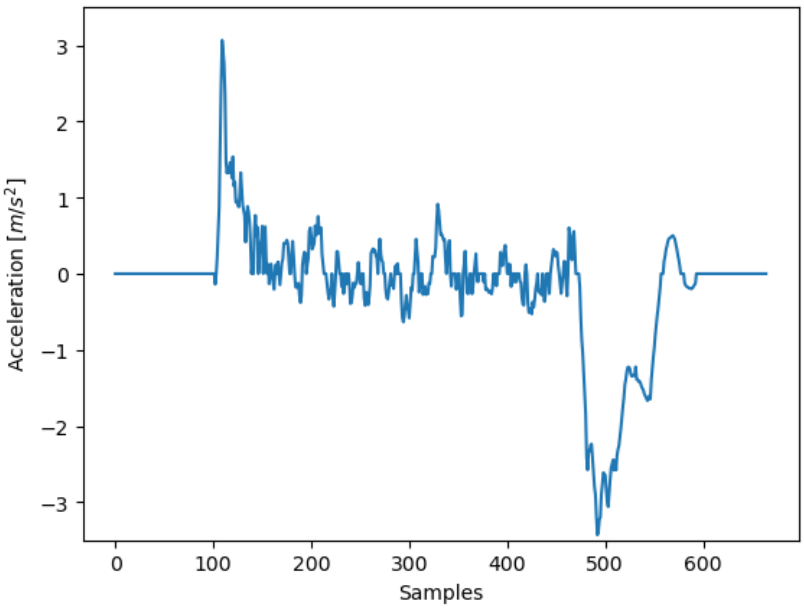


Obrázek 4.4: Zapojení BMS a nabíjecí desky

Inertial Mesurement Unit - IMU

Druhým rozšířením oproti originální stavebnici je přidání IMU senzoru. Konkrétně se jedná o model MPU5060. Ten disponuje tříosým akcelerometrem a gyroskopem. Uzel na jeho ovládání, využívá knihovnu třetí strany `mpu6050-raspberrypi`. Získaná data je nejprve potřeba vyčistit. Přímo v hardwaru je implementována dolní propusť, sloužící k eliminaci vibrací a jiných nechtěných vlivů. Nastavuje se zápisem do konfiguračního registru na adrese `0x1A`, což lze provést využitím knihovní funkce `set_filter_range`. Získaná data se nadále čistí softwarově. Protože se jedná o levný senzor, tak není zcela přesný a čtené hodnoty obsahují malé odchylky i v době, kdy by měly být nula. Proto se v softwaru aplikuje offset, který posune výsledky blíže k reálným hodnotám. Dalším krokem může být získaná data průměrovat mezi více vzorky a dosáhnout tak ještě většího utlumení výkyvů.

Do ROS2 systému tento uzel nejprve odesílá aktuální hodnoty zaznamenané senzorem a to pomocí zpráv typu Twist. Následně také posílá varování o kolizi, to detekuje v momentě, kdy `cmd_vel` topic obsahuje příkaz pohybu, ale imu senzor čte výrazně odlišná data. Jako poslední tento uzel ještě odesílá odometrii. Odometrie udává transformaci aktuální pozice robota vůči počátku. Uzel tedy pravidelně čte hodnoty senzoru a integruje je, aby získal absolutní pozici robota. Největším problémem u získávání odometrie byl vliv gravitace na akcelerometr a fakt, že stejné zrychlení a zpomalení se né vždy rovná. Gravitace vede na špatné hodnoty zrychlení v případě, že je robot nakloněn v nebo proti směru pohybu. Různé hodnoty akcelerace a decelerace je pravděpodobně způsobená rychlou změnou, kterou senzor nestihne vzorkovat dostatečně rychle.



Obrázek 4.5: Výsledná rychlost po zintegrování tohoto grafu je -0.1496m/s

Tento graf zobrazuje případ kdy došlo k špatnému zaznamenání zpomalení a výsledná rychlost je následně záporná. Důležité je zmínit že případy kdy dochází k špatnému měření nejsou časté (cca každé 10). Z grafu je také vidět, že když se robot pohybuje je měření ovlivněno vibracemi motorů. Při testování se tento jev neprojevil dostatečně aby ovlivnil funkcionalitu.

Ve finální verzi jsou tyto vlivy převážně eliminovány využitím dat z `cmd_vel` topicu. Tento přistup umožňuje ignorovat výkyvy dat získaných z imu senzoru v momentech, kdy by podle příkazů z `cmd_vel` měl být robot statický.

Light Detection And Ranging - Lidar

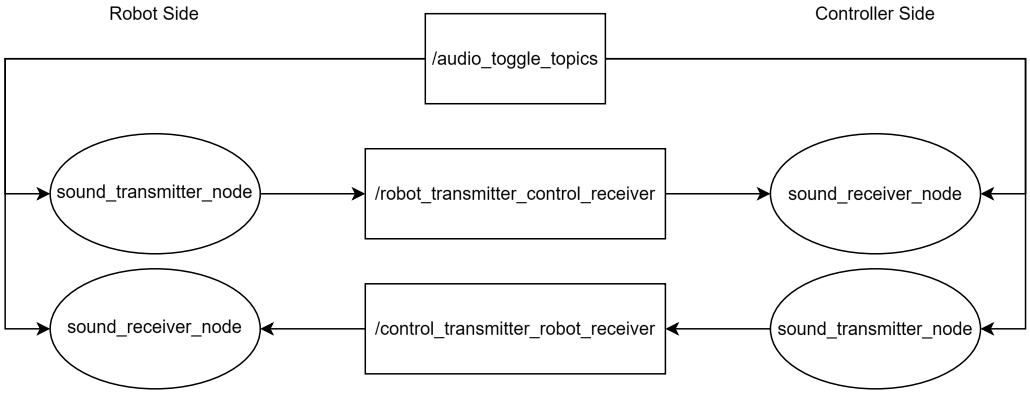
Posledním a zároveň nejkomplexnějším přidaným senzorem je lidar. Konkrétním použitým modelem je `LD 19 D 300`. K jeho ovládání je využit již existující ROS2 balík `ld19_lidar`. Výstupem tohoto uzlu jsou zprávy typu `LaserScan` na topicu `/scan`. Pro zajištění lepší kompatibility s `slam_toolbox` byly v tomto uzlu provedeny menší změny v hlavičce odesílaných zpráv a byly odebrány nepotřebné výpisy, které zpomalovaly odesílání.

Přenos zvuku

V zadání práce je zmíněno také téma teleprezence. Již probrané komponenty zajistí záznam videa a pohyb robota. K plnohodnotnější teleprezenci je však potřeba zajistit také přenos zvuku. A na rozdíl od videa ideálně obousměrně. Mozkem robota je mikropočítač Raspberry Pi 4, na kterém běží plnohodnotný operační systém. Připojení externích periferií jako mikrofon a reproduktory proto není problém. Řízení komponentů v tomto případě zajistí operační systém. ROS2 uzly se díky tomu můžou zaobírat pouze záznamem, přenosem a přehráváním dat. Pro tento účel byly vytvořeny dva uzly.

První slouží k nahrávání a odesílání audio dat. Pro záznam zvukového toku v reálném čase je použit Python modul `sounddevice`. Tato knihovna kromě jiného disponuje také `Stream` třídami. `InputStream` umožňuje kontinuální záznam zvuku. Při inicializaci se jí(kromě dalších parametrů) předá callback funkce. Ta je volána vždy, když je potřeba zpracovat nasnímaný blok dat. ROS2 nedisponuje standardním typem zprávy pro přenos zvukových dat. Je zde využit vlastní typ složený ze dvou polí o prvcích typu `float32`(každé pro jeden kanál).

Druhý uzel pak realizuje příjem a přehrávání získaných dat. Jedná se o obrácenou verzi předchozího uzlu. Přijímaná data jsou před přehráváním ukládána do bufferu. Důvodem je fakt, že přenos mezi dvěma uzly není zcela spolehlivý a datové bloky se mohou zpozdit. Pro přehrávání získaných dat je využita `OutputStream` třída.

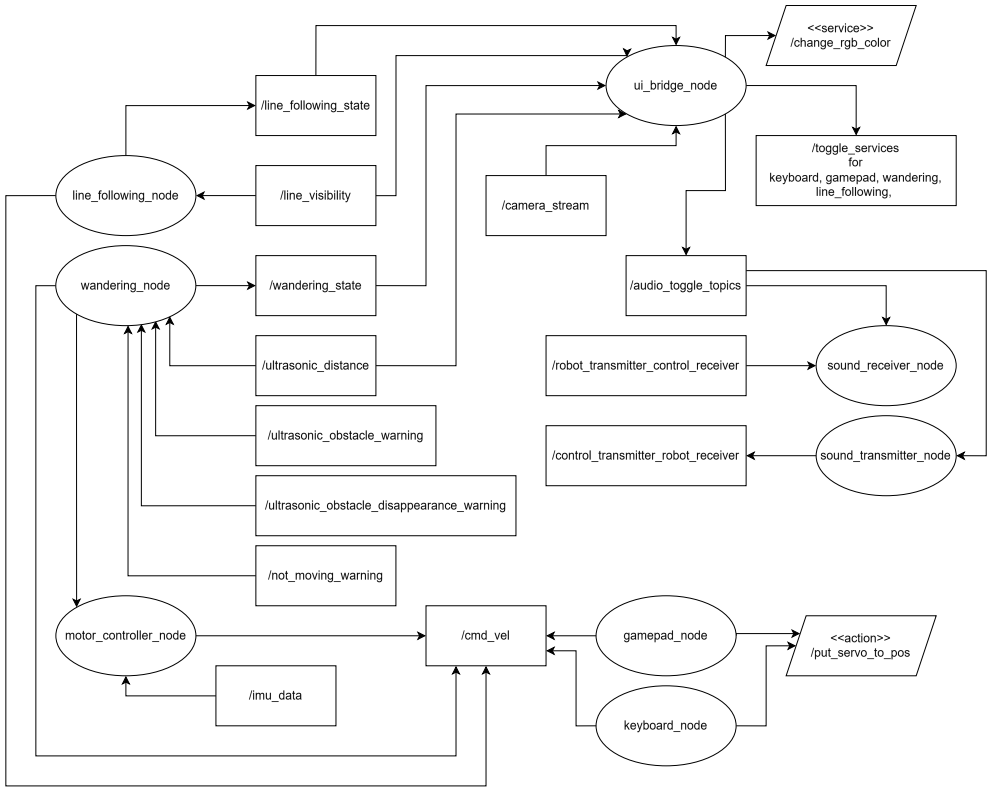


Obrázek 4.6: RQT Graf obousměrného přenášení zvuku

Pro obousměrný přenos dat mezi robotem a stacionárním zařízením jsou použity dvě dvojice těchto uzlů. Oba uzly disponují mechanizmem pro pozastavení záznamu / přehrávání.

4.2 Řízení robota na vyšší úrovni

V tento moment jsou implementovány všechny uzly pro ovládání komponent. Následuje sekce zabývající se řídícími uzly. Ty se nachází v balíku `controllers`. Jejich úkolem je zpracovávat data z několika hardwarových uzlů současně a v podle získaných dat řídit robota jako celek.



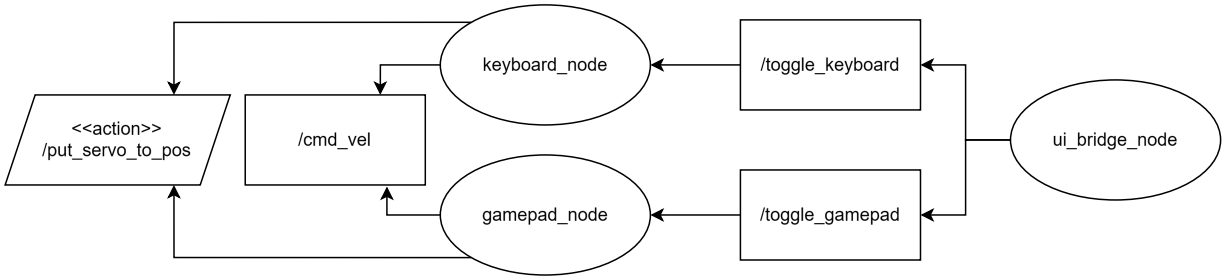
Obrázek 4.7: RQT Graph struktura řídícího systému

Distribuovanost

Řídící jednotky robotů bývají z pohledu výpočetního výkonu relativně slabá zařízení. Výpočetně náročné uzly mohou mít problémy s během na těchto zařízeních. Z toho, a dalších důvodů umožňuje ROS2 distribuovat uzly mezi více fyzických zařízení. Z vývojářského pohledu je distribuovanost zcela v režii ROS2. Jediný předpoklad pro její fungování je vzájemná dosažitelnost zařízení přes počítačovou síť. Druhým požadavkem je nastavení stejného DOMAIN_ID na obě zařízení, což je ve výchozím nastavení splněno. Ve správně fungujícím distribuovaném systému mohou uzly běžící na různých zařízeních navzájem komunikovat a interagovat stejně jako by běžely na jednom. V této práci se tedy předpokládá, že následující uzly poběží na druhém stacionárním zařízení.

Manuální řízení

Pro manuální řízení je systém vybaven dvěma uzly. První čte vstupy z klávesnice. Využívá k tomu modul `pynput`. Druhý pak s pomocí `pygame` knihovny získává vstupy ovladače. Oba uzly ovládají motory a servo pomocí jejich specifických rozhraní. Na obou uzlech se také nachází service servery pro příjem příkazů sloužících k zastavení nebo spuštění smyček, které zajišťují zachytávání vstupu od uživatele.



Obrázek 4.8: RQT Graf uzlů pro manuální řízení

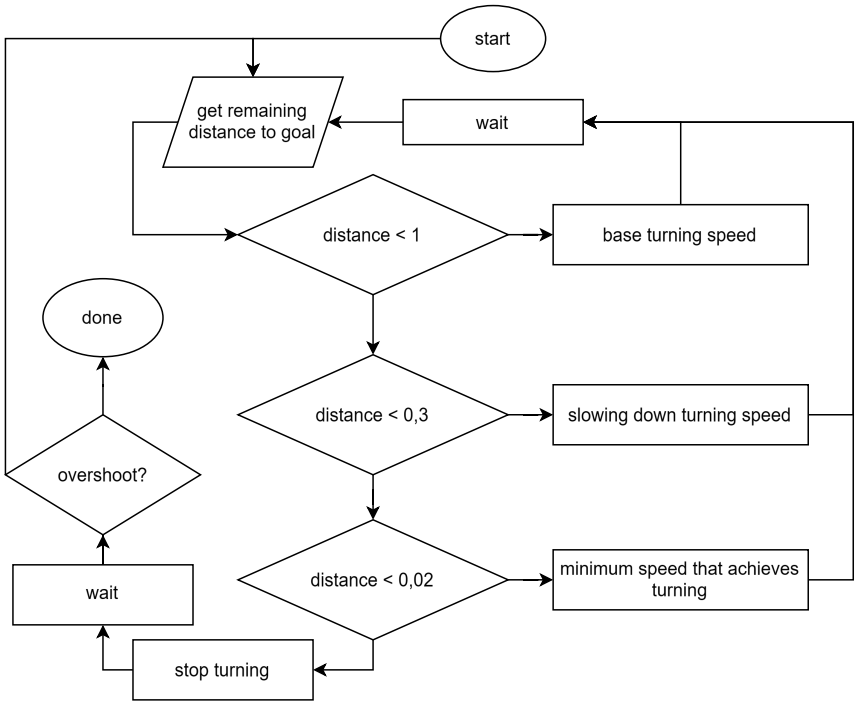
Pomocný uzel pro přesné otáčení robota

Vzhledem k tomu, že použité motory nejsou opatřeny enkodéry je, bez využití dalších senzorů, jakékoli otáčení velice nepřesné. Motorům lze samozřejmě zadat konkrétní rychlost na určitý časový interval a dostat se tak na přibližně správnou orientaci. Ale opravdu se jedná jen o přibližný úhel z důvodů externích vlivů a nepřesností. Například setrvačnost hraje velkou roli při otáčení o úhly menší než 90° nebo při vyšším využití prostředků mikrokontroléru, můžou být reakce na příkazy zpožděné. Proto byl vytvořen tento uzel, který využívá gyroskopická data z imu senzoru, aby zajistil, že se robot dokáže v případě potřeby otáčet o přesně dané úhly. V ROS2 systému se jedná o action server. Přijímá požadavky a následně zasílá příkazy na `cmd\_vel` topic. Podle hodnot získaných z imu pak postupně snižuje rychlost otáčení v závislosti na vzdálenosti od cílového úhlu.

Z pohledu demonstrace ROS2 funkcionality tento uzel využívá pokročilejší možnosti action serveru. Konkrétně se jedná o schopnost definování a předání vlastních callback funkcí, pro jednotlivé události související s akcemi. V případě standardního použití action serveru se pracuje pouze s execute callback funkcí. V jejím těle by mělo proběhnou celé zpracování požadavku.

V případě tohoto uzlu tento přístup nelze použít. Každý spuštěný ROS2 uzel má svůj vlastní execution loop, funkce spin(). Když se uzel točí, reaguje na události a volá callback funkce. V průběhu vykonávání takové funkce se však uzel netočí. A zde nastává problém. V průběhu vykonávání execute callbacku by tento uzel potřeboval průběžná data z imu senzoru. Ty se však získávají v subscriber callback funkci, která se nezavolá v průběhu vykonávání execute callbacku. Proto bylo potřeba použít další callback funkce action serveru k zprovoznění tohoto uzlu.

- goal_callback
  - vyvolá se při příchodu požadavku, rozhoduje o jeho přijmutí nebo zamítnutí
  - v případě tohoto uzlu: pokud již probíhá obsluha nějakého požadavku, tak jsou všechny nově příchozí zamítnuty, důvodem je fakt, že po dokončení aktuálního cíle



Obrázek 4.9: Algoritmus přesného otáčení

bude výsledná orientace robota jiná, než když byly tyto požadavky zaslány, což s největší pravděpodobností znamená, že již nejsou platné

- handle_accepted_callback
  - vyvolá se po přijmutí cíle
  - v tomto uzlu zahajuje zpracování požadavku
- execute_callback
  - měl by obsahovat hlavní funkcionalitu action serveru, rozhoduje o úspěchu požadavku
  - v tomto uzlu je volán aby ukončil vykonávání požadavku
- cancel_callback
  - vyvolán když klient zažádá o zrušení vykonávání cíle, rozhoduje o jeho přijmutí nebo zamítnutí

Callback funkce mají dané rozhraní, které musí následovat. Nejvýraznějším požadavkem je návratová hodnota.

Bloudění

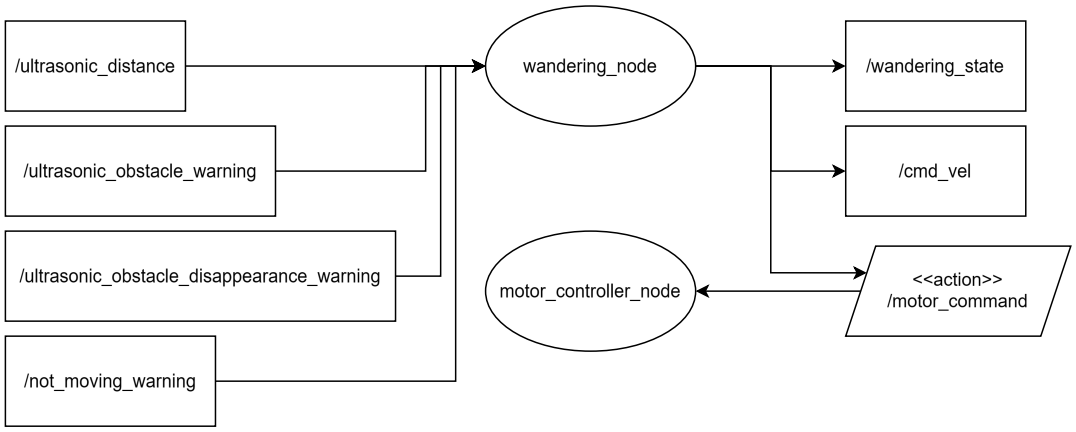
Náhodné bloudění robota po místnosti je základním autonomním pohybem. Cílem tohoto režimu je náhodný pohyb robota v prostoru, s cílem vyhýbat se překážkám.

K získávání informací o svém okolí používá tento uzel data ze dvou senzorů. Prvním je ultrazvukový senzor vzdálenosti. Ten je umístěn na přední straně robota a detekuje překážky v

Algoritmus 16: Action Server Callbacks

```
1: def goal_callback(self, goal_request):
2:     if self.goal_handle is not None and self.goal_handle.is_active:
3:         return GoalResponse.REJECT
4:     else:
5:         return GoalResponse.ACCEPT
```

15° úhlu před robotem. Druhým použitým senzorem je IMU. Přímo z něj získává pouze varování o kolizi. Nepřímo jej využívá při přesném otáčení s využitím `motor_controller_node` uzlu.



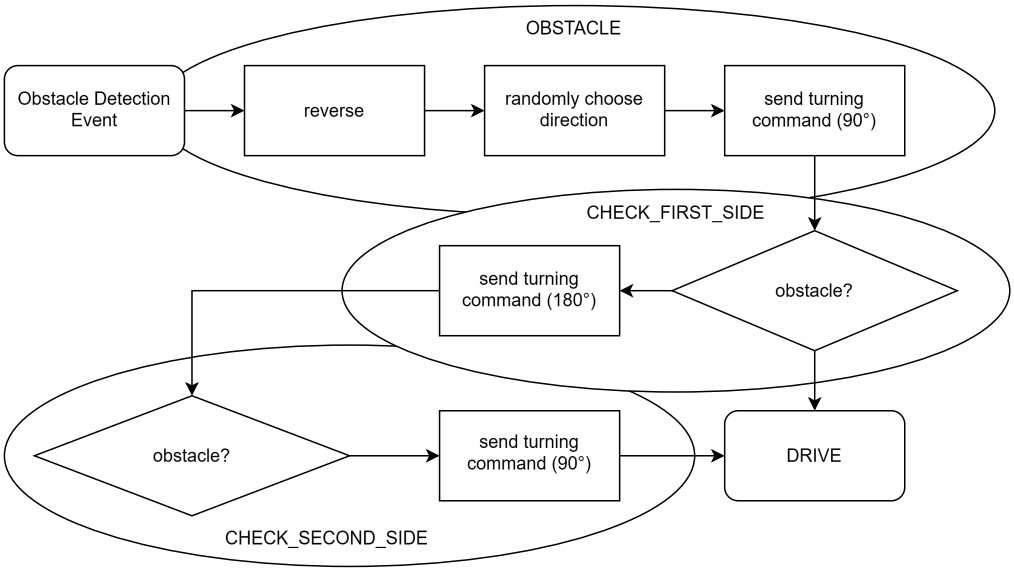
Obrázek 4.10: RQT Graf Bloudění

Jádrem implementace je konečný automat. Ten byl v průběhu vývoje obohacen o další funkcionalitu. Primárně bylo potřeba reagovat na externí události a volat funkce jiných uzlů. Výsledný kód tedy není čistým konečným automatem. Z abstraktnějšího pohledu tento režim provádí tři hlavní akce. Tou první je jednoduchý pohyb vpřed. Další dvě zajímavější akce pak realizují vyhýbání překážkám. Úvodními stavy těchto akcí je `OBSTACLE` a `SCAN_START`.

`OBSTACLE` má za úkol reagovat a vyřešit překážky nacházející se přímo před robotem. Detekování a přechod do tohoto stavu zajišťuje buď zpráva od ultrazvukového senzoru o překážce nacházející se příliš blízko, nebo zpráva od imu senzoru varující o tom, že `cmd_vel` vyslala příkaz o pohybu vpřed avšak akcelerometr změnu nezaznamenal, což implikuje, že se robot opírá o nějakou překážku.

Algoritmus hledání volné cesty je zobrazen na následujícím diagramu. Ve zkratce funguje následovně. V náhodném pořadí zkontroluje obě strany robota. Pokud na jedné z nich najde volný prostor ihned tudy pokračuje v jízdě. Pokud je v obou směrech překážka, vrací se zpět odkud přijel.

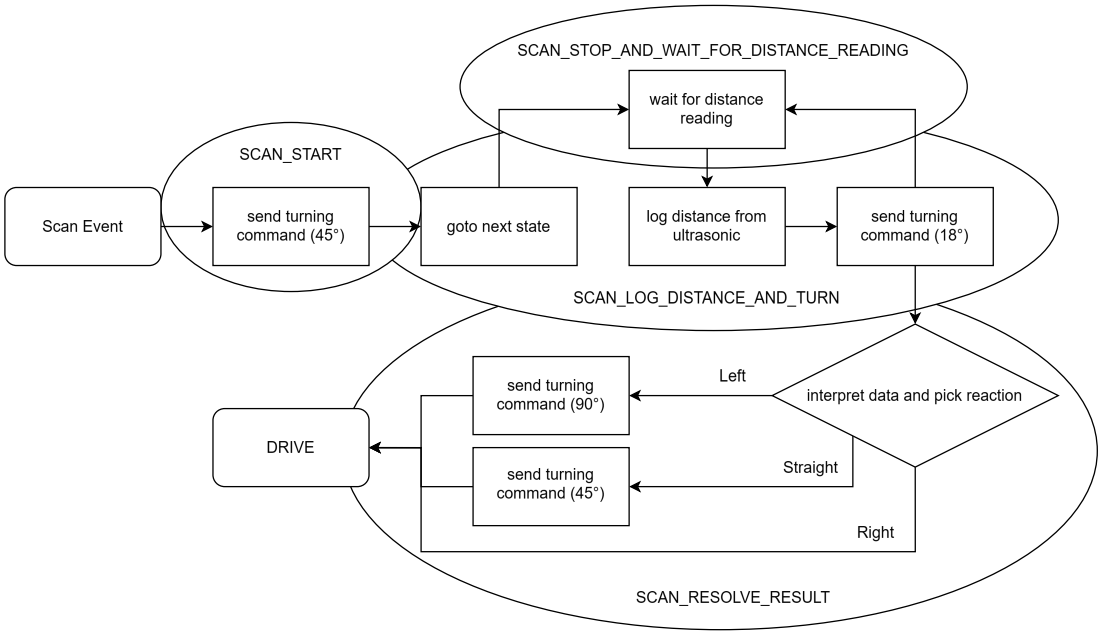
`SCAN_START` je oproti předchozímu chování více obezřetné. Jeho originálním záměrem bylo hledat překážky, které zmizely z ultrazvukového senzoru (více o tomto jevu v kapitole x). Tato funkcionalita byla následně rozšířena a ve finální verzi probíhá pravidelné skenování prostoru před robotem. Vzhledem k tomu, že se jedná o preventivní akci, nesnaží se vyhýbat překážkám přímo před robotem. Jeho cílem je nalézt překážky v blízkosti aktuální trajektorie robota. A v reakci na ně odklonit směr pohybu dále od nich. Pokud skenování na-



Obrázek 4.11: Algoritmus vyhnutí se překážky přímo před robotem.

lezne překážku přímo před robotem, pokračuje v pohybu vpřed. Tím efektivně přenechává vyřešení tohoto problému `OBSTACLE` stavu. Možné přechody do úvodního stavu skenování lze vyvodit z předchozího textu. Prvním je přijmutí varování o detekci zmizelé překážky od ultrazvukového senzoru. Druhým je pak pravidelný přechod ze stavu `DRIVE`.

Algoritmu skenování je opět zobrazen na následujícím diagramu. Úhel hledání překážek je 90°. Celkem je provedeno šest čtení začínající na úhlu -45° a končící na +45°. Získaná data jsou následně interpretovány a je učiněna adekvátní reakce.



Obrázek 4.12: Algoritmus preventivního hledání překážek.

Sledování čáry

Sledování čáry je v robotice známý úkol a na vyšších úrovních se v této disciplíně pořádají závody. Avšak v základním provedení se jedná o jednoduchý úkol. Tato funkcionalita je implementována hlavně z toho důvodu, že robot disponuje senzorem, který je k sledování čáry určený.

Implementace je provedena použitím konečného automatu. Použitý senzor je třícestný. Automat tedy bude obsahovat s 2^3 stavů. Z tohoto počtu jsou dva stavy koncové a to 101, protože se jedná o nevalidní stav a 000 indikující ztrátu čáry. Mezi všemi ostatními lze navzájem přecházet. Následující stav se volí podle aktuálních hodnot senzorů. Přechodová logika byla následně obohacena o další podmínky s cílem reagovat na ztrátu čáry. To umožnilo následovat čáru s ostrými zatáčkami (pravý úhel a více). A také překonávat krátké přerušení v čáře. Zde je důležité zmínit, že tato funkcionalita byla testována hlavně v simulátoru a i tam má poměrně hodně nedostatků.

4.3 Spouštěcí Soubory (Launch files)

Každý uzel ve výsledném systému má k sobě vytvořený spouštěcí soubor. Ve většině případů se jedná o jednoduché soubory jejichž úkolem je načtení a předání konfiguračního souboru spouštěnému uzlu. Některé parametry je vhodné změnit také při spouštění uzlu. Spouštěcí soubory, které toto umožňují, musí tento parametr explicitně definovat.

```
ros2 launch lauch.py param_name:=value
```

Ve složce `ros2_ws/launch` se nachází hromadné spouštěcí soubory. Ty mají za úkol využít nižších spouštěcích souborů k nastartování větší části systému zároveň. Díky této hierarchické struktuře už není potřeba řešit předávání všech parametrů a lze se zaměřit jen na to důležité. Pokud jsou volány soubory, které parametry explicitně definují, lze je v tento moment přepsat, stejně jako by to udělal uživatel z příkazové řádky. Příkladem využití této funkcionality je řídící uzel `wandering_node`. Pokud je tento uzel spuštěn manuálně, předpokládá se, že v systému žádný jiný řídící uzel neběží. Může tedy zahájit vykonávání hned po inicializaci, aniž by jeho příkazy kolidovaly s dalšími řídícími uzly. V případě, že je však volán jako součást hromadného spouštění, přepíše se výchozí hodnota parametru `start_right_away` a všechny řídící uzly tak budou po inicializaci čekat na další příkaz.

```
IncludeLaunchDescription(
    launch_goal,
    launch_arguments={'start_right_away': 'false'}.items()
)
```

Hromadné spouštěcí soubory startují mnoho uzlů zároveň. Běžným požadavkem tak je výměna několika málo uzlů za jiné. V takovém případě by bylo potřeba vytvořit druhý, převážně totožný soubor. K eliminaci tohoto problému slouží podmíněné spouštění. Jeho jednoduchou verzi využívá `adeept_robot_launch.py`. V tomto případě slouží k rozhodnutí, který ze dvou možných uzlů k řízení motorů bude použit. Cílům spouštěcího souboru se do proměnné `condition` přidá podmínka. Jedná se o speciální třídy (`Ifcondition`, `Unlesscondition`, ...). V konstruktoru se jim předává true / false výraz. Jeho hodnotu lze získat například z parametru. Podmíněné spouštění může být i komplexnější a to v kombinací s `PythonExpression`. Tato třída je použita v `gazebo_simulation_launch.py` souboru, kde se porovnáním hodnoty parametru určuje, který svět bude spuštěn.

```
IncludeLaunchDescription(
    launch_goal,
    condition=IfCondition(
        PythonExpression([
            '"', world_select_val, '"', ' == "wandering"'
        ])
    )
)
```

Komplexnější spouštěcí soubory často potřebují větší kontrolu nad tím, kdy dojde ke provedení jednotlivých cílů. V případě, že je potřeba pouze opozdit provedení některého z cílů lze využít `TimerAction`. Jedná se o třídu, která spustí daný cíl až po předem stanoveném časovém intervalu. Tato funkce je využita při spouštění `ros2_control` v souboru `diffdrive_launch.py`, kde slouží k zpoždění „spawn“ ovladače až po dokončení inicializace pluginu.

```
TimerAction(
    period=10.0,
    actions=[
        Node()
    ]
)
```

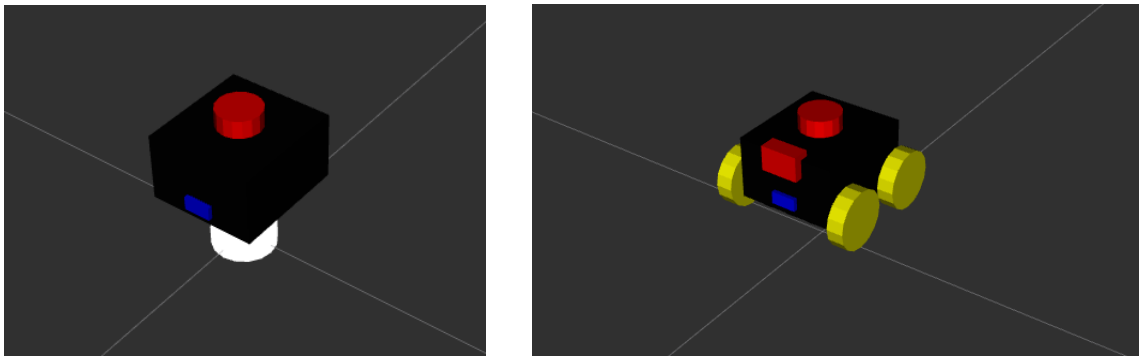
K přesnějšímu řízení pak slouží event handlery. Ty umožňují přesnější kontrolu nad tím, kdy dojde k vykonání jednotlivých cílů. Typickými událostmi jsou spuštění a ukončení procesu, případně, pokud se jedná o lifecycle uzly, také reakce na přechody do konkrétních stavů. Obsluha událostí je využita přímo v balíku gazebo simulace v souboru `gazebo_world_launch.py`. Je zde celý řetěz těchto obsluh, které postupně spustí simulaci, přeloží `xacro` model na `urdf`, „spawnou“ jej do simulace a po jejím ukončení ještě uklidí vygenerovaný `urdf`.

```
RegisterEventHandler(
    OnProcessStart(
        target_action=simulator,
        on_start=convert_xacro
    )
)
```

4.4 Model Robota

Model robota slouží k předání informace o jeho struktuře, vzhledu a fyzikálních vlastnostech dalším částem ROS2 systému. Formát pro definici modelu, který ROS2 používá je `urdf`. Většina nástrojů však nepracuje přímo s definicí v `urdf` formátu, ale získává tyto informace z `/tf` subsystému. Zpracování, interpretaci a následné odesílání informací o aktuálním stavu robota zajišťuje `robot_state_publisher`. Jedná se o oficiální ROS2 uzel. Vstupem tohoto uzlu je `urdf` model robota. Prvním výstupem je odesílání statických i dynamických transformací do `/tf` systému. Aby mohl odesílat dynamické tranformace musí mít informaci o aktuálních natočení kloubů. Ty získává posloucháním `joint_states` topicu.

Druhým výstupem tohoto uzlu je topic `robot_description`. Jeho obsahem je celý `urdf` popis, odeslaný jakožto string zpráva.



Obrázek 4.13: Zobrazení dat z robot_state_publisheru v nástroji RVIZ. Levý obrázek zobrazuje pouze statické tranformace (joint_states topic je prázdný), a pravý pak celý model

Tvorba Modelu

Jak už bylo řečeno, základní formát používaný k popisu modelů je `urdf`. ROS2 využívá také druhý, rozšiřující formát `xacro`. Cílem `xacra` je vyřešit některé neduhy čistého `urdf` a usnadnit tak vývojářům psaní popisů robotů. Tato práce používá k popisu robota `xacro` formát.

Následující blok kódu demonstruje využití téměř všech možností `xacra` na jednom místě. V čistém `urdf` se velmi často opakuje téměř totožná definice `link` elementu. Jedná se o zdlouhavý zápis s minimálními změnami pouze v hodnotách. Následující blok kódu vytváří makro, které umožní zkrátit tuto nepřehlednou a na chyby náchylnou sekci na jednořádkové zavolání makra. Pro zvýšení znovupoužitelnosti, využívá toto makro také parametry. Ty jsou předány v místě použití makra. Díky nim může být makro obecnější a tedy použitelné na více místech. Poslední parametr ukazuje také možnost nastavení výchozí hodnoty parametru. Parametry lze využít také v matematických výrazech. Výpočet matice setrvačnosti pro každý `<link>` element je zbytečně složité a opět náchylné na chyby. Použití matematického výrazu, který se sám vyhodnotí podle hodnot předaných parametry je výrazně lepší a pohodlnější přístup.

Popis robota v `xacro` formátu nelze použít jako přímou náhradu `urdf`, ale musí být nejprve přeložen. K tomu slouží následující příkaz.

```
xacro in.xacro > out.urdf
```

Při překladu dojde k spojení případných více souborů dohromady, nahrazení a vyhodnocení maker, matematických výrazů a parametrů.

4.5 Uživatelské rozhraní

Uživatelské rozhraní na straně stacionárního zařízení využívá knihovnu Qt, konkrétně její python verzi PyQt5. Rozhraní jako takové je odděleno od ostatních řídících funkcionalit a pouze zajišťuje zobrazování informací získatelných z topiců a odesílání příkazů ostatním uzlům. Zbytek ROS2 systému je tedy plně ovladatelný z prostředí příkazové řádky. Co se týče implementace tak ROS2 i qt mají svůj vlastní execution loop, což vytváří problém, protože pokud jeden z nich neběží, tak daná část nefunguje. Jednou možností řešení je použití více procesů. To však vede na problémy se synchronizací. Lepší a také použitý přístup tedy je řídit oba cykly manuálně.

Algoritmus 17: Macro with params

```
1: <xacro:macro name="box_link"params="name mass x y z material:=red»
2:     <link name="$name»
3:         <inertial>
4:             <mass value="$mass"/>
5:             <inertia ixx="$(1/12) * mass * (y*y+z*z)"ixy="0.0"ixz="0.0"
6:                 iyy="$(1/12) * mass * (x*x+z*z)"iyz="0.0"
7:                 izz="$(1/12) * mass * (x*x+y*y)"/>
8:         </inertial>
9:         <visual>
10:            <geometry>
11:                <box size="$x $y $z"/>
12:            </geometry>
13:            <material name="$material"/>
14:        </visual>
                                    ⋮
```

```
while controllers.user_interface.global_variables.executeEventLoop:
rclpy.spin_once(node, timeout_sec=0.001)
app.processEvents()
```

ROS2 část zajišťuje komunikaci se zbytkem systému. Obsahuje subscribery na topicy ze kterých volá qt funkce na zobrazení získaných dat. Kromě těch pak obsahuje také funkce sloužící k zasílání příkazů na servery či odesílání dat do topicků. Ty jsou volané z qt části. Qt část se pak stará čistě o zobrazování získaných dat nebo volání ROS části v reakci na uživatelské akce.

Kapitola 5

Nástroje související s ROS2

5.1 Gazebo Simulátor

Použití simulátoru při vývoji softwaru na řízení robotů je časté a užitečné. Umožňuje vyvíjet software i bez fyzického robota, případně testovat funkcionalitu bez vlivů reálného světa. Tato práce využívá novou `ignition` větev Gazebo simulátoru.

Před tím, než lze začít využívat výhody simulátoru, je nejprve potřeba vytvořit model robota a světy ve kterých se bude pohybovat. Nativním formátem který Gazebo využívá je `sdf`. V této práci je `sdf` použito k definici světů.

Definice světa

Definice světů se skládá ze dvou základních částí. Nejprve se zpravidla definují obecné vlastnosti jako parametry simulace, fyzikální charakteristiky světa a pluginy. Druhou částí je pak samotná definice objektů, které se budou ve světě nacházet. Ty můžou být složené z jednoduchých tvarů, které lze definovat přímo v souboru.

```
<box>
 <size>8 0.1 0.2</size>
</box>
```

Druhou možností je využít `<mesh>` tag pro vložení komplexnějších objektů. Umožňuje totiž vložit soubor obsahující model vytvořený například v Bleneru. Poslední možností je využití oficiální fuel knihovny, která obsahuje mnoho uživateli vytvořených modelů, které lze jednoduše vložit do světa.

Definice modelu

Pro definice modelů existuje několik použitelných formátů. První možností je využít stejně jako pro světy formát `sdf`. Tímto způsobem lze zapsat definici modelu ve stejném souboru jako zbytek světa. Pokud se jedná o komplexnější definici lze ji zapsat externě a vložit od stvěta pomocí `<include>` tagu. Z důvodu kompatibility se zbytkem ROS2 systému umožňuje gazebo použít také `urdf` potažmo `xacro` soubory. Modely definované v `urdf` se do světa nevkládají přímo, ale „spawnují“ se až po spuštění simulace. Slouží k tomu gazebo service jménem `/world/world_name/create` a příkaz `ign service`.

Klasický, již probraný, model definuje vizuální a fyzikální vlastnosti robota. Tato definice struktury je pro většinu nástrojů dostačující. Simulátor, který má za úkol simulovat chování

robota, potřebuje dodatečné informace o tom, co jednotlivé elementy modelu reprezentují z pohledu funkcionality. K tomu slouží `<gazebo>` tagy. Zde by bylo dobré zmínit, že do definic modelů lze vkládat aplikačně specifické tagy. Příkladem je právě `<gazebo>`, nebo `<ros2_contorl>` tag. Ostatní nástroje zpracovávající takovou definici tyto neznámé tagy ignorují.

Možným obsahem `<gazebo>` tagu je buď `<senzor>` nebo `<plugin>`. Senzory se vždy přiřazují k nějakému existujícímu `link` elementu. Tento `link` pak reprezentuje fyzické vlastnosti a vzhled daného senzoru. Argumentem `senzor` tagu je určení konkrétního typu snímače. V jeho těle se pak definují parametry. Ty jsou specifické pro jednotlivé typy senzorů. Gazebem podporované senzory jsou typicky komplexnější modely jako lidar a hloubkové kamery. Naopak jednodušší snímače použité na demonstračním robotovi nejsou nativně implementovány. Ultrazvukový senzor vzdálenosti je tedy ve výsledném modelu nahrazen lidarem. Aby se jeho fungování více blížilo referenčnímu senzoru, byl omezen úhel měření na 15° a počet vysílaných paprsků na tři. Modul pro sledování čáry je pak realizován pomocí tří kamer. Každá s rozlišením jeden pixel. Získanou RGB hodnotu následně zpracovává pomocný uzel.

Pluginy jsou kusy kódu, umožňující rozšířit simulátor o další funkcionalitu. Jejich použití v definici modelu slouží převážně k simulaci motorů a serv. Základní instalace Gazebo Simulátoru obsahuje mnoho užitečných pluginů. Příkladem může být `DiffDrive` nebo `JointController`. První zmíněný simuluje řízení robota s diferenciálním podvozkem. Druhý pak umožňuje simulovat chování podobné servu.

ROS Gazebo Bridge

Krása Gazebo Simulátoru je jeho provázání s ROS2 systémem. Pomocí oficiálního nástroje `ros_gz_bridge` lze přemostit komunikaci mezi ROS2 systémem a Gazebo Simulátorem. Tímto způsobem lze použít stejné řídící uzly pro reálného i simulovaného robota. Gazebo vnitřně používá podobný systém topiců a zpráv jako má ROS2. Tento most pak funguje tak, že vytvoří ROS2 uzel, který zajišťuje překlad mezi Gazebo a ROS2 zprávami.

Spouští se pomocí příkazu:

```
ros2 run ros_gz_bridge parameter_bridge /topic_name@gazebo_msg_type@ignition_msg_type
```

Pro efektivnější použití lze předat bridge uzlu config soubor s definicí více topiců které budou přemostěny.

```
- ros_topic_name: "ros_chatter"
gz_topic_name: "gz_chatter"
ros_type_name: "std_msgs/msg/String"
gz_type_name: "gz.msgs.StringMsg"
direction: IGN_TO_ROS  # BIDIRECTIONAL or ROS_TO_IGN
```

Zařízení kompatibility

Ve většině případů stačí k zajištění kompatibility čisté přemostění topiců mezi Gazebem a ROSem. Ale existují také speciální případy, kdy je potřeba data nějakým způsobem upravit aby blíže odpovídaly reálnému světu.

- Kamera

Most pro přenos obrazových dat využívá na ROS2 straně zprávy typu Image. Ale jak bylo podrobněji probráno v sekci o fyzické kameře. Výstup z uzlu pro její ovládání je ve formátu CompressedImage. Data vycházející ze simulátoru musí proto být přetypovány aby byla zajištěna kompatibilita.

- Sledování Čáry

Jak bylo řečeno výše, Gazebo nemá nativní podporu senzoru pro sledování čáry. Výstup ze simulátoru je tedy ve formátu tří Image zpráv, každá obsahující jeden RGB pixel. Vzhledem k tomu, že se jedná o tři samostatné zprávy, musí být nejprve seskupeny podle časových značek. Poté jsou získaná data převedena na binární hodnotu reprezentující viditelnost čáry. To je provedeno zkombinováním RGB složek pixelu na jedu grayscale hodnotu. Na ni lze jednoduše aplikovat thresholding a získat tak výsledné rozhodnutí o viditelnosti čáry.

- Měření vzdálenosti

Ultrazvukový senzor je v simulátoru proveden pomocí lidaru. Je tedy potřeba zkombinovat data ze všech tří paprsků do výsledné vzdálenost. Uzel ovládající fyzický senzor, také odesílá dodatečné varování ohledně detekování překážek. Je tedy potřeba doplnit také tuto funkcionalitu.

- Servo

Uzel pro ovládání fyzického serva používá action server. Tento simulovaný musí z důvodu kompatibility dělat totéž.

- Motory, Lidar, Odometrie

U těchto uzlů probíhá pouze jednoduché přemapování jmen. U odometrie je rozdíl v tom, že výstupem není klasický topic ale `/tf` rám.

5.2 ROS2 Control

Ros2 control je framework implementující teorii řízení. V sekci o ovládání komponent byl představen uzel, který ovládá dc motory s cílem realizovat diferenciální pohyb robota. Tvorba vlastního uzlu k tomuto účelu je zcela validní přístup. Problém je v tom, že podobný uzel bude potřebovat každý mobilní robot. A proto existuje ros2 control, který má za cíl zjednodušit tvorbu řídících systémů.

Controller Manager

Je hlavní řídící jednotkou, která zajišťuje navázání ovladačů(controllers) a hardwarových pluginů (drivers). Manager se spouští pomocí uzlu `ros2_control_node`. Jako první krok po spuštění potřebuje získat informace o robotu, kterého bude ovládat. Ty hledá v `robot_description` topicu. Konkrétně z něj získá seznam kloubů a hardwarových rozhraní, kterými tyto klouby disponují. Příkazové rozhraní `command_interface` slouží k posílání dat směrem k hardwaru, například nastavení rychlosti(velocity) otáčení motoru. Stavové rozhraní `state_interface` pak slouží k získávání informací z dané komponenty zpět do ROS2 systému, může se jednat například o výstup enkodéru motoru. Další inicializačním krokem je zpracování konfiguračního souboru. Ten obsahuje seznam použitelných ovladačů a

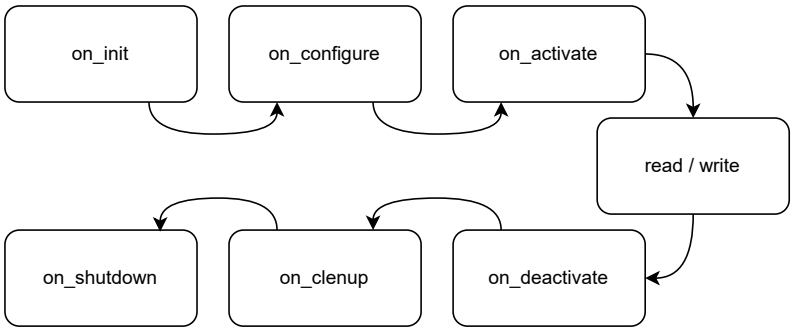
jejich nastavení. V tento moment bude manager čekat na spuštění některého z ovladačů definovaných v konfiguračním souboru, aby mohlo dojít k jeho navázání na kompatibilní hardwarové rozhraní. Tímto krokem je vše připraveno a může započít ovládání robota. U příkladu ovládání robota s diferenciálním podvozkem je využit, již implementovaný ovladač, který zajišťuje výpočty týkající se kinematiky. Hardware plugin se pak stará pouze o ovládání motorů pomocí GPIO rozhraní.

Model

Aby Control Manager věděl, se kterými částmi robota bude pracovat, musí být upravena jeho `urdf` definice. Konkrétně je potřeba přidat `<ros2_control>` tag. V jeho obsahu je jako první vybrán konkrétní hardware plugin, který bude zajišťovat řízení komponent. Jsou zde také zapsány parametry, které budou při spuštění předány danému pluginu. Dále jsou vybrány klouby a jejich rozhraní, které bude tento plugin a potažmo ros2 control ovládat.

Hw Plugin

Hardware Pluginy jak z názvu vyplývá jsou části kódu které budou v ros2 control ekosystému zajišťovat komunikaci s hardware komponentami. Na rozdíl od controlerů, u kterých lze často využít existující implementace, protože se požadované funkcionality často opakují. U hardware pluginů existuje mnoho různých komponent s různými rozhraními a, tak je často potřeba si napsat vlastní. Pro psaní pluginů se používá jazyk C++. Struktura jako taková je podobná lifecycle uzlům ROS2. Znamená to, že třída musí definovat speciální metody, které se volají v průběhu inicializace / destrukce objektu.



Obrázek 5.1: Základní postup volání lifecycle funkcí

- `on_init`
  - načtení parametrů definovaných v urdf modelu
  - kontrola, že klouby zadané v urdf odpovídají očekávání
- `on_configure / on_cleanup`
  - připravení a nastavení hardwaru
  - například nastavení gpio pinů, nastavení jejich směru, inicializace pwm
- `on_activate / on_deactivate`
  -

- `export_state_interfaces` / `export_command_interfaces`
  - nabídne rozhraní definované v urdf a inicializované v pluginu k spárování s ovladači
- `read`
  - získává hodnoty z hardwaru a ukládá je do vnitřních proměnných aby se jejich hodnoty mohly dostat k ovladačům
- `write`
  - podle hodnot z vnitřních proměnných zasílá příkazy hardwarovým komponentům

Controllers

Ovladače se definují pomocí konfiguračního `yaml` souboru. Ten následuje stejná pravidla, jako konfigurace pro kterýkoli jiný ros2 uzel. Nejprve se zadávají parametry pro samotný `controller_manager`. Zde se volí ovladače, které bude možné načíst. Dále pak následují konfigurace specifické pro jednotlivé ovladače.

Integrace ros2 control s Gazebo simulátorem

Slouží k tomu balíček `gz_ros2_control`. Struktura ros2 control zůstává i pro simulátor stále stejná. Hlavní změnou je použití jiného hardware pluginu. Ten místo ovládání fyzické komponenty zajišťuje řízení modelu v Gazebo Simulátoru. Aby však mohl ovládat simulační prostředí, musí být `urdf` popis robota rozšířen o načtení dalšího Gazebo pluginu. Posledním rozdílem či zjednodušením je to, že spouštění Controller Manageru je součástí zmíněného pluginu a není tedy potřeba jej zapínat externě.

Algoritmus 18: PLUGIN LOAD

```
1: <plugin filename="control-system.so"name="GazeboSimROS2ControlPlugin»
2:   <parameters>$(find package_name)/config/controllers.yaml</parameters>
```

5.3 Navigace a mapování

Před zahájením mapování je nejprve potřeba vytvořit několik uzlů, které budou poskytovat informace o aktuální pozici robota, jeho struktuře a podobně.

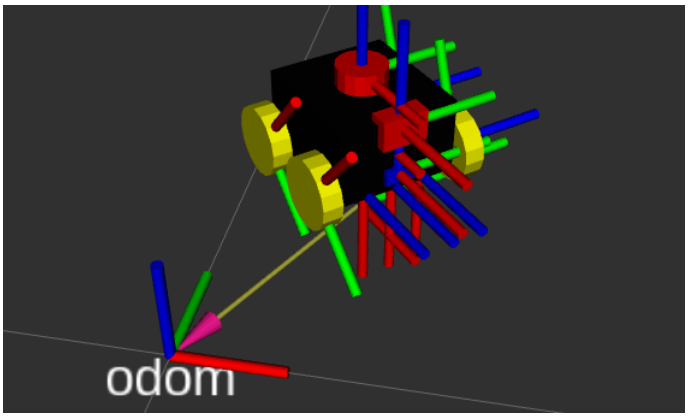
Model

Jak mapování tak navigace hojně využívají `/tf` subsystém. Z něj získávají informace o struktuře a transformacích robota. Podrobněji se o `/tf` mluví v [] a o modelech v [].

Odometrie

Druhým krokem k mapování a lokalizaci je získání aktuálních souřadnic, orientace a rychlosti robota. K tomu slouží odometrie. Odometrická dala lze získat několika způsoby. Jedním z často používaných a také poměrně přesných přístupů je využití dat získaných z enkodérů motorů kol a následný výpočet vzdálenosti, kterou urazily jednotlivá kola diferenciálního podvozku. Vzhledem k tomu, že použitý robot nedisponuje motory s enkodéry byl v této

práci využit druhý přístup. Tím je zpracování dat získaných z imu senzoru. Integrací dat z akcelerometru a gyroskopu lze získat aktuální pozici a natočení vůči počátku. Nevýhodou tohoto přístupu je její menší přesnost a akumulace chyb vedoucí k postupnému vzdalování těchto dat od reality. Postupná akumulace chyby u odometrie vzniká u všech přístupů a ostatní části ROS2 systému realizující mapování a navigaci tak s touto skutečností počítají. V této práci tuto funkcionalitu zajišťuje uzel `imu_node`, který výsledná data odesílá jako odom frame do tf subsystému.



Obrázek 5.2: Zobrazení robota včetně jeho transformačních rámů v Rviz, fixed_frame je nastaven na odom a robot se tedy může pohybovat oproti počátku souřadného systému

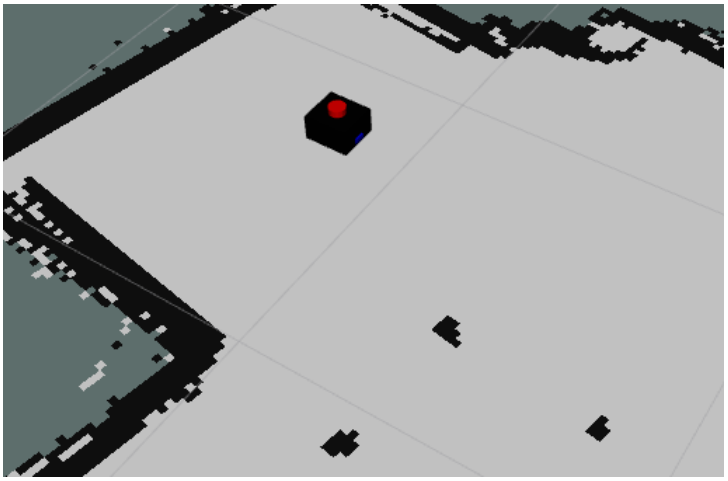
Mapování

Posledním krokem k úspěšnému mapování je získání dat o okolním prostředí. To zajišťuje lidar senzor umístěný na robotu. Aby bylo možné data získané z lidaru správně vizualizovat v rviz musí existovat transformace mezi rámem lidaru a base frame robota. Pokud je lidar součástí definice modelu, postará se o tuto transformaci `robot state publisher`.

Mapování jako takové pak zajišťuje perfektní knihovna `slam_toolbox`. Jedná se o komplexní soubor nástrojů souvisejících se SLAM. Tato práce využívá `async_slam_toolbox_node`. Jedná se o online, async mapování. Online znamená, že uzel pracuje nad aktuálními daty, knihovna totiž umožňuje také tvorbu mapy z předem zaznamenané historie. Async pak zajišťuje zpracování vždy nejnovějších dat, což zlepšuje latenci, ale může vést k přeskočení některých scanů. Vzhledem k tomu, že slam je komplexní problém, existuje mnoho nastavitelných parametrů. Ty se předávají při spouštění pomocí konfiguračního souboru. V demo příkladech se nahází ukázkový konfigurační soubor, který byl jen s menšími modifikacemi využit i v této práci. Hlavní změnou je zvýšení rozlišení vytvářené mapy. Vývojáři knihovny předpokládají její využití v průmyslu a základní hodnoty tedy pracují s předpokladem, že robot je větší a pohybuje se ve velkých halách. Uzel pak odesílá výslednou mapu ve standardním formátu `nav_msgs/OccupancyGrid` do `/map` topicu.

Navigace

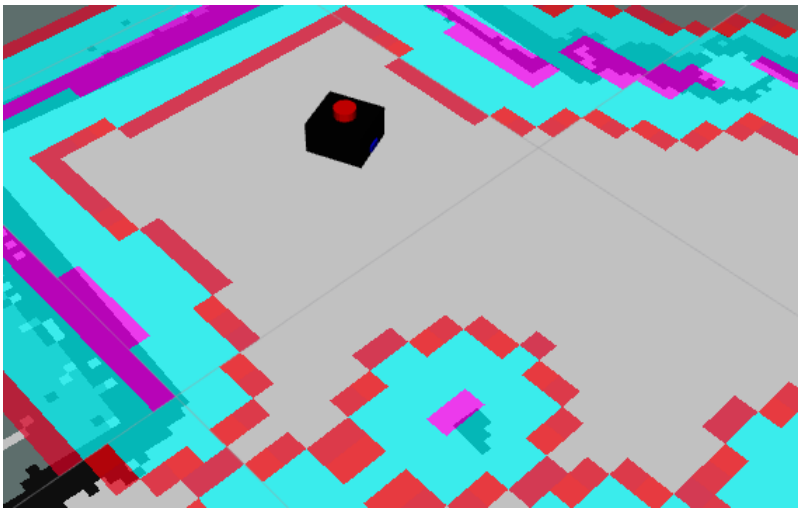
Pro navigaci byla využita knihovna Navigation 2. Nav2 je rozsáhlý nástroj, který tvoří v podstatě samostatný ROS2 systém skládající se z několika navzájem spolupracujících uzlů. Principiálně je Nav2 založen na behaviorálních stromech.



Obrázek 5.3: Mapa vytvořená slam toolboxem zobrazená v nástroji Rviz

Z vnějšího pohledu uzly Nav2 poslouchají topicy `/scan`(lidar data) `map`(slam mapa) a transformace z tf systému, primárně ty související s `odom` rámem. Zobrazitelným výstupem je několik costmap. Jedná se o upravenou mapu z `map` topicu obohacenou o ceny jednotlivých polí. Tyto ceny jsou používány plánovacím serverem (jeden z uzlů Nav2) k určení optimální cesty k cíli. Příkaz pro zahájení navigace je přijímán action serverem se jménem `/navigate_to_pose`.

Uživatel může ovlivnit chování Nav2 systému pomocí konfiguračního souboru. Ten je poměrně komplexní a umožňuje nastavovat velké množství parametrů a dokonce vyměňovat řídící pluginy. V porovnání s slam konfigurací zde bylo potřeba změnit hodně parametrů. Použité motory nedokážou vyvinout menší rychlost než 0.2m/s byly proto upraveny minimální rychlosti. Stejně jako u slam konfigurace je robot menší a pohybuje se v menších prostorách, byly proto zmenšeny inflation vzdálenosti. Ty jsou okolo překážek a ovlivňují ceny polí při plánování cesty.



Obrázek 5.4: Slam mapa překrytá lokální navigation 2 costmapou v nástroji Rviz

Instalace

Výsledný systém je komplexní a obsahuje mnoho závislostí které vyžaduje pro jeho správnou funkcionalitu.

Kapitola 6

Závěr

Cílem práce bylo vytvořit ROS2 systém pro ovládání robota Adeept AWR 4WD a demonstrovat na něm možnosti ROS2. Tohoto cíle bylo dosaženo a implementovaný systém tak může fungovat jako ukázka možností ROS2. V rámci implementace bylo využito všech důležitých funkcionalit a konceptů používaných v ROS2. Dokumentace pak funguje jako vysvětlení jednotlivých funkcionalit případně jako rozcestník při hledání příkladu konkrétní funkcionality.

Práce pak nad rámec původního cíle demonstruje také použití dalších souvisejících nástrojů. Jedním z nich je Gazebo Simulátor, díky kterému lze jednoduše vyvíjet a testovat ROS2 uzly i bez fyzického robota. Dále pak demonstruje použití frameworku ros2_control pro řízení motorů robota. Nakonec pak bylo hardwarové vybavení robota rozšířeno o další senzory jako IMU a lidar, které umožnili zaměřit se na problematiku mapování a navigace. V tomto ohledu byly využity nástroje slam_toolbox a navigation2.

V dlouhodobějším horizontu by šlo implementovat rozšíření systému na multirobotickou aplikaci, kde by dva roboti navzájem komunikovali a společně vykonávali nějakou činnost.

Literatura

[1] Al-Naib, Ahmed. DC Motors. *Researchgate* [online]. Květen 2019. Dostupné z: https://www.researchgate.net/publication/332835517.

[2] Andrejašic, M. *MEMS ACCELEROMETERS* [online]. Ljubljana: University of Ljubljana, březen 2008 [cit. 2024-04-17]. Dostupné z: https://public.websites.umich.edu/~bkerkez/courses/cee575/Handouts/5_adxl_theory.pdf.

[3] Braunl, T. *Embedded Robotics: From Mobile Robots to Autonomous Vehicles with Raspberry Pi and Arduino*. Fourth edition. Singapore: Springer, 2022. ISBN 9811608032.

[4] Macenski, S., Martin, F., White, R. a Ginés Clavero, J. *Nav 2 Documentation* [online].

[5] *I2C-bus specification and user manual* [online]. 7.0. NXP Semiconductors, 1982, 2021-10-01 [cit. 2023-11-12]. Dostupné z: https://www.nxp.com/docs/en/user-guide/UM10204.pdf.

[6] NXP Semiconductors. *PCA9685: 16-channel, 12-bit PWM Fm+ I2C-bus LED controller* [online]. 2009 [cit. 2023-11-12]. Dostupné z: https://www.nxp.com/docs/en/data-sheet/PCA9685.pdf.

[7] Perera, S., Barnes, N. a Zelinsky, A. Exploration: Simultaneous Localization and Mapping (SLAM). In: Ikeuchi, K., ed. *Computer Vision: A Reference Guide*. Cham: Springer International Publishing, 2021, s. 412–420. ISBN 978-3-030-63416-2.

[8] Raspberry Pi Foundation. *Raspberry Pi Camera Documentation* [online]. [cit. 2023-31-12]. Dostupné z: https://www.raspberrypi.com/documentation/accessories/camera.html.

[9] Rico, F. M. *A concice introduction to robot programming with ROS2*. First edition. Boca Raton: CRC Press, 2023. Computer science, č. 1. ISBN 978-1-032-26465-3.

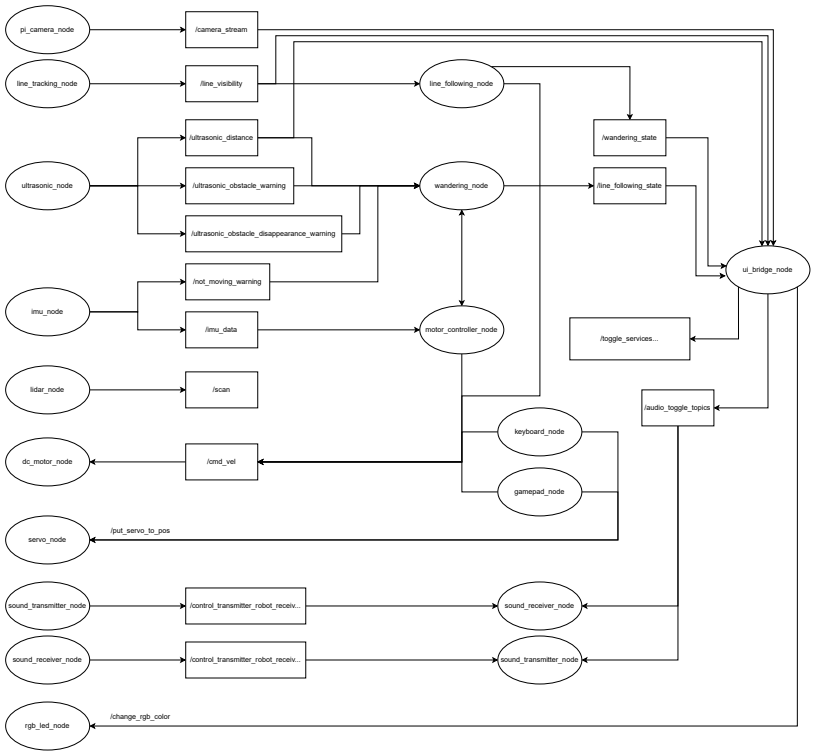
[10] Salhuana, M. *Sensor I2C Setup and FAQ* [online]. an4481. Freescale Semiconductor, Inc, červenec 2012 [cit. 2023-11-12]. Dostupné z: https://www.nxp.com/docs/en/application-note/AN4481.pdf.

[11] Sojka, P. et al. *ROS 2 Documentation* [online].

[12] STMicroelectronics. *L298: Dual Full-Bridge Driver*. 2000 [cit. 2023-11-12].

[13] Wikipedia contributors. *Differential wheeled robot* [online]. Wikipedia, The Free Encyclopedia, únor 2024. Dostupné z: https://en.wikipedia.org/w/index.php?title=Differential_wheeled_robot&oldid=1204597545.

[14] WORLDSEMI CO., LIMITED. *WS2812 Intelligent control LED integrated light source* [online]. [cit. 2023-31-12]. Dostupné z: https://html.alldatasheet.com/html-pdf/553088/ETC2/WS2812/95/1/WS2812.html.



Obrázek 1: Výsledný ROS2 systém bez slam navigace a gazeba