

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-100863-111124

**VYTVORENIE OVLÁDAČA V PROSTREDÍ ROS PRE
MOBILNÉHO ROBOTA**
BAKALÁRSKA PRÁCA

2023

Filip Loppreis

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-100863-111124

**VYTVORENIE OVLÁDAČA V PROSTREDÍ ROS PRE
MOBILNÉHO ROBOTA**
BAKALÁRSKA PRÁCA

Študijný program: Robotika a kybernetika

Názov študijného odboru: kybernetika

Školiace pracovisko: Ústav robotiky a kybernetiky

Vedúci záverečnej práce: Ing. Michal Dobiš

Konzultant: Ing. Michal Dobiš

Bratislava 2023

Filip Lobpreis



ZADANIE BAKALÁRSKEJ PRÁCE

Autor práce:	Filip Lobpreis
Študijný program:	robotika a kybernetika
Študijný odbor:	kybernetika
Evidenčné číslo:	FEI-100863-111124
ID študenta:	111124
Vedúci práce:	Ing. Michal Dobiš
Vedúci pracoviska:	prof. Ing. Jarmila Pavlovičová, PhD.
Miesto vypracovania:	Ústav robotiky a kybernetiky
Názov práce:	Vytvorenie ovládača v prostredí ROS pre mobilného robota
Jazyk, v ktorom sa práca vypracuje:	slovenský jazyk
Špecifikácia zadania:	Mobilná robotika využívaná v kombinácii s logistickými alebo servisnými úkonmi sa stáva čoraz viac populárnejšou. Úlohou študenta je naštudovať si mobilné robotické zariadenie, ktoré bude mať k dispozícii na Národnom centre robotiky a k nemu príslušné materiály. Študent bude pracovať s reálnym hardvérom a otvoreným systémom, ktorý bude potrebné preštudovať a pochopiť jeho fungovanie. Cieľom práce bude následne vytvoriť nadradený ovládač implementovaný v ROS (Robotickom operačnom systéme), ktorý bude schopný riadiť daného robota.
Úlohy:	<ol style="list-style-type: none">1. Analyzujte súčasný stav riešenia a prostredie Robotického operačného systému.2. Analyzujte možnosti a metodiku implementácia riadiaceho balíka pre daný robot3. Navrhnite spôsob implementácie a architektúru riešenia4. Implementujte riadiaci systém pre mobilného roba5. Vypracujte dokumentáciu k dosiahnutým výsledkom.6. Vyhodnote dosiahnuté výsledky.
Termín odovzdania práce:	02. 06. 2023
Dátum schválenia zadania práce:	
Zadanie práce schválil:	

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Robotika a kybernetika
Autor:	Filip Lobpreis
Bakalárska práca:	Vytvorenie Ovládača v prostredí ROS pre mobilného robota
Vedúci záverečnej práce:	Ing. Michal Dobiš
Konzultant:	Ing. Michal Dobiš
Miesto a rok predloženia práce:	Bratislava 2023

Mobilný robot vykonáva také pohyby aké mu zadáme. Preto aby tento spôsob bol čo najefektívnejší, musí byť čitateľný dobre tak ako pre robot tak aj pre užívateľa. Našou úlohou je vytvoriť také prostredie na ovládanie robota, tak aby bol kód efektívny a jednoduchý. Tým pádom bude čitateľný pre ľudí aj robota. Rozhodli sme sa pre ROS druhej verzie. To prečo a ako sme spravili jednotlive časti sa dočítate d'alej.

Kľúčové slová: ROS, BlackMetal, uzly, témy, služby, akcie

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Robotics and cybernetics
Author:	Filip Lobpreis
Bachelor's thesis:	Implementation of ROS Driver for Mobile Robot
Supervisor:	Ing. Michal Dobiš
Consultant:	Ing. Michal Dobiš
Place and year of submission:	Bratislava 2023

The mobile robot performs such movements as we give it. Therefore, for this method to be as effective as possible, it must be readable both for the robot and for the user. The major task is to create such an environment for controlling the robot, so that the code is efficient and simple. Thus, it will be readable by both humans and robots. We decided for ROS2. You can read more about why and how we made the individual parts in the next sections.

Keywords: ROS, BlackMetal, nodes, topics, services, actions

Pod'akovanie

I would like to express a gratitude to my thesis supervisor.

Obsah

Úvod	1
1 Použité technológie	2
1.1 Operačný systém	2
1.1.1 Windows	2
1.1.2 Ubuntu	3
1.2 Programovací jazyk a jeho prostredie	4
1.3 ROS	5
1.3.1 Témy	5
1.3.2 Služby	7
1.3.3 Akcie	7
1.3.4 Parametre	7
1.4 ROS1	8
1.5 ROS2	9
1.6 Rozdiely	10
1.6.1 Štandard jazyka	10
1.6.2 Inicializácia nody (uzla)	11
1.6.3 Komunikácia	11
1.6.4 Parametre	11
1.6.5 Nodelet alebo komponent	12
1.6.6 Kompilácia	12
1.6.7 Vlákna	12
2 Pôvodný stav robota	13
2.1 Robot a jeho ovládanie	13
2.2 Komunikácia s robotom	15
2.2.1 Logovanie	15
2.2.2 Ovládanie	15
2.3 Vysvetlenie kľúčov retázca	16
3 Zmeny na robote	17
3.1 Pokazený systém	17
3.2 Chyba v dokumentácii	17
3.3 Rozšírenie príkazov robota	17
3.4 Nesprávna spätná väzba	18

3.5 Zašumený výstup	20
4 Implementácia ovládača	22
4.1 Úvod do čítania grafu	23
4.2 Uzly	23
4.3 Vstup	23
4.4 Komunikácia s robotom	24
4.5 Odometria	24
4.6 Zdieľanie polohy	25
5 Filtrovanie zašumeného signálu	26
5.1 Zistovanie parametru α	27
Záver	31
Zoznam použitej literatúry	32

Zoznam obrázkov a tabuliek

Obrázok 1.1	Vizualizácia témy v ROSe [2]	6
Obrázok 1.2	Vizualizácia služby v ROSe [2]	7
Obrázok 1.3	Vizualizácia akcie v ROSe [2]	7
Obrázok 1.4	Porovnanie štruktúr ROS1 a ROS2 [3]	8
Obrázok 2.1	Zobrazenie spodnej časti mobilného robota [5]	13
Obrázok 2.2	Schéma zapojenia jednotlivých častí na robote	14
Obrázok 3.1	Ustálené hodnoty rýchlosťi ľavého a pravého motora.	20
Obrázok 3.2	Prechodová charakteristika rýchlosťi kolies [5].	21
Obrázok 4.1	Graf vykonávania programu na ovládanie robota pomocou ROS2.	22
Obrázok 5.1	Získanie prevodu z impulzov na rýchlosť v SI jednotkách.	26
Obrázok 5.2	Získanie prevodu z impulzov na rýchlosť v SI jednotkách. $\alpha = 0,9$	27
Obrázok 5.3	Získanie prevodu z impulzov na rýchlosť v SI jednotkách. $\alpha = 0,7$	28
Obrázok 5.4	Získanie prevodu z impulzov na rýchlosť v SI jednotkách. $\alpha = 0,75$	28
Obrázok 5.5	Získanie prevodu z impulzov na rýchlosť v SI jednotkách. $\alpha = 0,8$	29
Obrázok 5.6	Získanie prevodu z impulzov na rýchlosť v SI jednotkách. $\alpha = 0,8$ a frekvenciou 10Hz.	29
Obrázok 5.7	Získanie prevodu z impulzov na rýchlosť v SI jednotkách. $\alpha = 0,8$ a frekvenciou 10Hz a prvou prepočítanou hodnotou.	30

Zoznam skratiek

API	Application Programming Interface
DDS	Data Distribution Service
GCC	GNU Compiler Collection
GNU	GNU is not Unix
GUI	Grafical User Interface
IoT	Internet of Things
IPC	Inter Process Communication
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LAN	Local Area Network
NCR	Národné Centrum Robotiky
QoS	Quality of Service
ROS	Robot Operating System
RTOS	Real time operating system

Úvod

Táto bakalárska práca popisuje ako naprogramovať ovládač pre mobilného robota pomocou druhej verzie Robotického Operačného Systému (ROS2). Ovládač obsahuje funkcie ROS2 napr. uzly, parametre, služby a témy. Hlavným cieľom tohto projektu je vytvoriť rozhranie tak, aby bolo možné ovládať robota pomocou jednoduchých príkazov cez ROS.

1 Použité technológie

Našou úlohou v tejto bakalárskej práci je naprogramovať a otestovať nami vytvorený ovládač pre mobilného robota. Na vyber máme viacero technológií, ktoré môžeme použiť. Na vytvorenie ovládača budeme potrebovať operačný systém na nami používanom počítači, ktorý budeme používať na programovanie a testovanie. Na výber máme dve možnosti. Menovite to sú operačný systém Windows alebo operačný systém založený na jadre Linux. Ďalšiu vec, ktorú budeme potrebovať je programovací jazyk, na výber máme veľmi veľa možností a to hlavne C, C++, Python alebo Java. Ked'že, zo zadania vypláva, že máme použiť Robotický operačný systém, tak máme na výber dve možnosti. Sú to Robotický operačný systém prvej verzie (ROS1) a Robotický operačný systém druhej verzie (ROS2). Jazyky, ktoré sú podporované týmto systémom sme už spomenuli, jazyk ktorý je ešte podporovaný a my sme si ho nespomenuli je programovací jazyk Matlab. Samozrejme na implementáciu a otestovanie potrebujeme samotného robota. Pred tým ako začneme preberať spomenutý ovládač a robota, ktorého budeme riadiť, tak si predstavíme technológie, ktoré budeme používať pri vytváraní ovládača.

1.1 Operačný systém

Na začiatku našej práce sme sa museli rozhodnúť, či budeme písat program na náš ovládač v operačnom systéme Windows 11 alebo v operačnom systéme Ubuntu, ktorý je založený na jadre Linux. Ako programátori sme už dlhšie pracovali s operačným systémom Ubuntu a poznáme aj jeho výhody a nevýhody. To isté platí aj pre operačný systém Windows. Toto sú fakty, ktoré sme zvažovali pri výbere operačného systému, ktorý sme používali počas celej bakalárskej práce.

1.1.1 Windows

Windows je najpoužívanejším operačným systémom na počítače na svete. Je to hlavne preto, že je jednoduchý na používanie pre netechnicky založených užívateľov. Windows ale nie je bezpečným operačným systémom a taktiež aj jeho stabilita a rýchlosť nie sú na najvyššej úrovni. Tento operačný systém používajú lúdia hlavne na voľnočasové aktivity ako je hranie hier alebo v práci ako napríklad v kanceláriach na úpravu tabuľiek. Pre nás ako programátorov tento operačný systém nie je výhodný. Nastavovanie prostredia pre programovanie je v tomto operačnom systéme oveľa náročnejšie ako v operačnom systéme s Linuxovým jadrom. Podpora preklaďačov programov (z anglického compiler) je v tomto operačnom systéme oveľa horšia ako už v spomenutom Linuxe.

1.1.2 Ubuntu

Ubuntu je stabilný a bezpečný operačný systém, ktorý je založený na Linuxe. Väčšina ľudí hovorí o Linuxe ako o operačnom systéme. Pravdou to ale nie je. Linux je totižto len **kernel** (jadro) operačného systému. Kernel je hlavný program, ktorý sa stará o správu hardvéru, riadenie procesov a správu pamäte. Zároveň poskytuje možnosť operačnému systému komunikovať s hardvérom a poskytuje jednotlivým programom prístup k hardvéru a jeho časťam.

Linux je používaný skoro vo všetkých mobilných zariadeniach, serveroch, cloudoch (dátové servery, na ktoré sa pripájame cez internet) a ďalších elektronických zariadeniach, ktoré potrebujú rýchlosť, stabilitu a bezpečnosť. Je taktiež používaný vo všetkých mobilných robotoch. Nad operačným systémom Windows má niekoľko výhod.

- **Open source** Linux je open source jadro pre operačný systém, čo znamená, že kód, ktorý tvorí jeho základ, je k dispozícii pre všetkých zadarmo a každý môže prispiesť k jeho vylepšeniu. Tento otvorený prístup umožňuje programátorom prispôsobiť si Linux pre svoje potreby a vytvárať programy, ktoré sú optimálne pre ich prácu. Na Linuxe sú založené aj operačné systémy reálneho času alebo software pre smart mobilné telefóny ako je napríklad operačný systém Android.
- **Programovacie nástroje** Linux obsahuje mnoho vynikajúcich programovacích nástrojov, ktoré sú k dispozícii zadarmo a sú široko používané programátormi. Napríklad k dispozícii v Linuxe sú (GCC) GNU Compiler Collection (Kolekcia GNU prekladačov) (GNU je skratka pre GNU is not Unix) a GNU Debugger (GDB) sú vynikajúce nástroje pre C/C++ programátorov.
- **Bezpečnosť** Linux má reputáciu bezpečnejšieho operačného systému v porovnaní s operačným systémom Windows. V Linuxe je ľahšie infikovať sa škodlivými programami ako sú vírusy alebo maliari, pretože programy majú obmedzené oprávnenia a systém je navrhnutý tak, aby bol odolný voči útokom.
- **Stabilita** Linux má tendenciu byť stabilnejší než Windows, pretože nie je závislý na ovládačoch a softvéri od tretích strán. Linux poskytuje jednotný prístup k správe a aktualizácii softvéru. Pri vylepšovaní systému nie je za potreby reštartovať počítač.
- **Flexibilita** Linux je veľmi flexibilný operačný systém, ktorý umožňuje programátorom prispôsobiť si svoje pracovné prostredie a používať programy, ktoré najlepšie vyhovujú ich potrebám.

- **Podpora a komunita** Linux má veľkú a aktívnu komunitu programátorov, ktorá poskytuje podporu a rieši problémy. Vďaka tomu, že Linux je open source, mnoho ľudí prispeva k jeho vylepšeniu a vytvára nové programy, čo znamená, že je vždy niečo nové na objavovanie.
- **Komplexnosť** Napriek všetkým týmto výhodám ma Linux aj svoje nevýhody. Najväčšou nevýhodou, ktorú je treba spomenúť je komplexnosť operačného systému. Linux je komplexnejší ako operačný systém Windows, hlavne kvôli svojej flexibilnosti. Tým, že si vieme nastaviť vlastné prostredie, typ systému správy súborov, programy na stiahovanie, publikovanie a správy balíčkov.

1.2 Programovací jazyk a jeho prostredie

Medzi programovacími jazykmi, ktoré sme mohli použiť boli na vyber nasledovne: *C++, C, Python, Matlab, Java*.

- **C a C ++:** C a C ++ sú hlavné jazyky používané pri vývoji ROS2 aplikácií. Sú to výkonné a efektívne jazyky, ktoré možno použiť na nízko úrovňové programovanie, ako sú riadiace slučky, ovládače a systémové programovanie. C ++ je najčastejšie používaný jazyk v ROS2.
- **Java:** Java je ďalší jazyk, ktorý sa môže použiť na implementáciu aplikácií v ROS2. Je menej bežne používaný ako jazyk C++, ale má výhodu, a to takú, že je prenositeľný jazyk, ktorý môže byť spustený na akejkoľvek platforme, ktorá podporuje JVM (Java Virtual Machine). Java je vhodná pre vývoj vysoko úrovňových komponentov robotického systému, ako sú GUI užívateľské rozhrania (z anglického Grafcical User Interface) a sieťová komunikácia.
- **Python:** Python je populárny jazyk pre vývoj ROS2 kvôli svojej čitateľnosti a jednoduchosti použitia. Je vhodný pre vývoj vysoko úrovňových komponentov robotického systému, ako sú algoritmy na rozhodovanie, spracovanie dát a vizualizácia.
- **Matlab:** Matlab je vysoko úrovňový programovací jazyk a prostredie používané vo vedeckom prostredí (scientific computing), analýze dát a vizualizácií. Je to výkonný jazyk pre numerické výpočty a často sa používa na simulovanie a testovanie algoritmov pre robby. Avšak, nie je bežne používaný v ROS2 vývoji. Na implementáciu tohto rozhrania je spravený modul *ROS Toolbox*.

- **Výkon:** C a C++ poskytujú najlepší výkon pre ROS2 aplikácie kvôli ich nízkoúrovňovej povaze a priamemu prístupu k pamäti. Python a Java sú pomalšie ako C a C ++ kvôli ich interpretácii a nadhlaví virtuálneho stroja.
- **Čas vývoja:** Python a Java sa vyvíjajú rýchlejšie ako C a C ++ kvôli ich jednoduchosti a jednoduchosti použitia. Matlab môže byť tiež rýchlejší pri vývoji pre určité úlohy vďaka svojim zabudovaným knižniciam a nástrojom.
- **Kompatibilita:** Všetky uvedené jazyky môžu byť použité s ROS2, ale môž

Na základe vyššie spomenutých rozdielov sme sa rozhodli použiť programovací jazyk C++. Toto rozhodnutie sme spravili hlavne kvôli jeho podpore, výkonu a podpore v komunite.

1.3 ROS

Robotický operačný systém (Robot Operating System) je súbor voľne dostupných softvérových knižíc a nástrojov, ktoré vytvárajú vhodné podmienky pre programátorov na písanie aplikácií pre mnohé druhy robotov. ROS má dve verzie. Vo všeobecnosti sa stretнемe s tým, že pod názvom ROS1 alebo ROS sa myslí ROS verzie 1. Pod názvom ROS2 sa myslí ROS verzie 2. Aby nenastali nejasnosti budeme v tomto dokumente označovať ROS verzie 1 ako ROS1 a ROS verzie 2 ako ROS2. V prípade, keď budeme hovoriť o spoločných vlastnostiach a funkciałitách, ROS1 a ROS2 budeme označovať dokopy ako ROS.

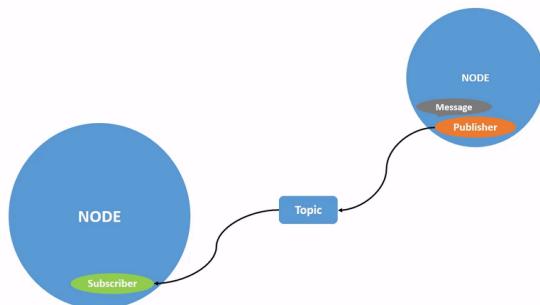
ROS je open source, čo znamená, že ide o softvér uvoľnený pod licenciou, kde má používateľ práva na použitie, štúdium, zmenu a redistribúciu. Mnohé open source licencie však určujú určité obmedzenia na tieto práva pre tento softvér, ale v podstate môžeme predpokladať tieto práva. Najbežnejšie licencie pre ROS2 softvérové balíčky sú Apache 2 a BSD, hoci vývojári majú slobodu používať aj iné. [1] Využitia Robotického operačného systému v reálnom svete sa môžu objavívať v automobiloch, mobilných robotoch, droboch, manipulátoroch.

Komunikácia v ROSe je zabezpečená cez IPC (Inter Process Communication), TCP/IP UDP/IP komunikáciou pomocou troch zakladacích metód: **témy** (Topics), **služby** (Services) a **akcie** (Actions). Všetky tieto metódy slúžia na posielanie správ na synchronizáciu vykonávania programu a komunikáciu jednotlivých častí programu. Na to aby sme ich vedeli správne využiť musíme poznávať ich základnú implementáciu a funkciałitu. Správanie týchto metód je nasledovné.

1.3.1 Témy

Témy sú sprostredkovane pomocou IPC - Medzi procesová komunikácia z anglického Inter Process Communication. Je to najjednoduchší spôsob komunikácie. Vieme si ich prirovnavať k UDP/IP protokolu, s tým že neprebiehajú cez siet'. Definujeme si jedného poskytovateľa

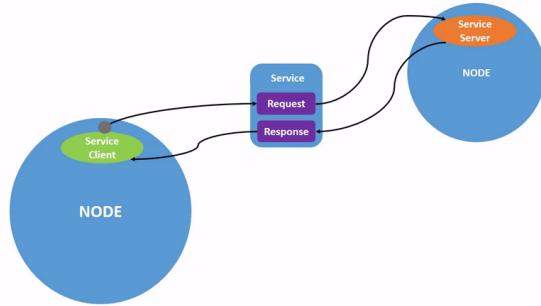
(publishers) a jedného alebo viacerých príjemcov (subscribers). Medzi týmito dvoma alebo viacerými účastníkmi sa následne posielajú správy (messages), ktoré sme si dopredu definovali.



Obr. 1.1: Vizualizácia témy v ROSe [2]

1.3.2 Služby

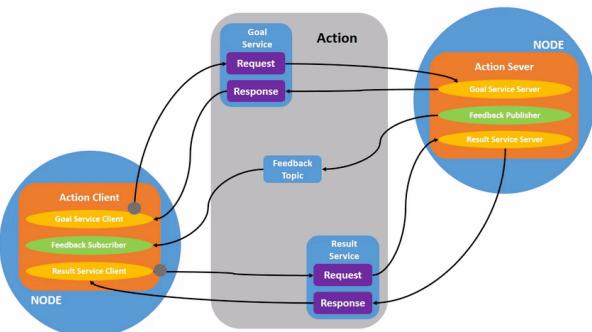
Služby sú sprostredkované pomocou TCP/IP protokolu. Poskytujú nám rovnaký spôsob komunikácie ako témy, až na to, že sa správy medzi servisom a klientom posielajú cez LAN (Local Area Network). Tieto správy sa posielajú oboma smermi. Služby sa využívajú pri komunikácii medzi viacerými zariadeniami.



Obr. 1.2: Vizualizácia služby v ROSe [2]

1.3.3 Akcie

Akcie sú takiež sprostredkované TCP/IP protokolom. Sú najzložitejším spôsobom komunikácie. Tento spôsob bol pridaný do ROS1 až neskôr. V druhej verzii ROSu je tento typ komunikácie medzi troma základnými formami komunikácie uzlov. Sú založené na službách a prebiehajú asynchrónne [1]. Máju 3 stavy vid' Obr. 1.3. Najprv pošle klient serveru, akú akciu má vykonat', server mu potvrdí, že túto požiadavku dostal. Server začne následne vykonávať danú akciu a posielat' klientovi priebežné správy o priebehu vykonávania žiadanej úlohy. Ked' server skončí, pošle klientovi výsledok akcie a klient mu obratom potvrdí obdržanie výsledku.

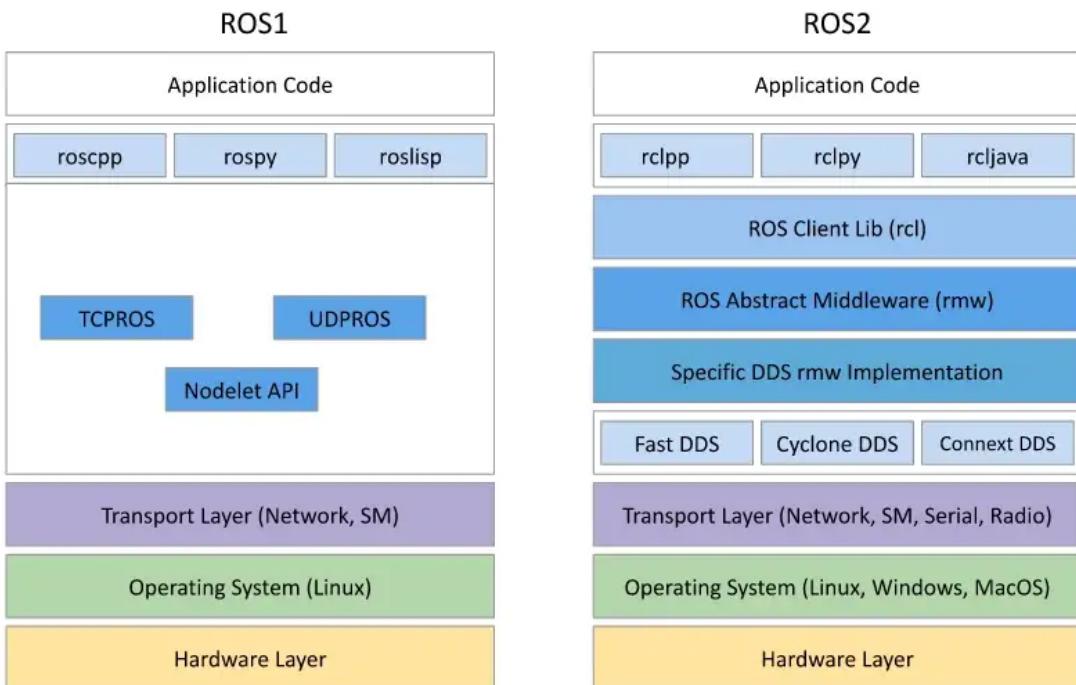


Obr. 1.3: Vizualizácia akcie v ROSe [2]

1.3.4 Parametre

Parametre sú spôsob, ako môže komunikovať užívateľ so základnými nastaveniami uzlov bez potreby zmenenia kódu a jeho následnej komplilácie, čo pri väčších projektoch môže zabrať

aj tri štvrtiny hodiny. Konfigurácie sa definujú v *yaml* konfiguračnom súbore. V ňom si môžeme zadefinovať mená jednotlivých parametrov a ich základné hodnoty. Tie si programátor vie v programe vytiahnuť pomocou API, Application Programming Interface, (Aplikačné Programovacie Rozhranie) v ROSe.



Obr. 1.4: Porovnanie štruktúr ROS1 a ROS2 [3]

1.4 ROS1

ROS bol prvýkrát vydaný v roku 2007. Ide o softvér, ktorý sa začal vyvíjať so zámerom zjednodušiť programovanie a ovládanie robotov. Od doby, kedy vznikol prešiel mnohými verziami a úpravami. Jeho neoddeliteľnou súčasťou sú štrukturovanie programu do uzlov (nodov), komunikácia medzi uzlami, podpora viacerých programovacích jazykov ako sú C, C++ alebo Python a vytváranie balíčkov dostupných širokej verejnosti.

Štrukturalizovanie základov ROS1 je spravené monoliticky čo najstabilnejším spôsobom. Na počiatku musí byť spustený hlavný program (roscore), ktorý zabezpečuje vytváranie jednotlivých uzlov. Komunikácia medzi uzlami je zabezpečená prostredníctvom prepojenia uzlov cez LAN/WLAN alebo IPC komunikáciu. Ak sú uzly spustené na iných zariadeniach, tak sa využíva len komunikácia cez siet. Roscore d'alej poskytuje parametre jednotlivým uzlom z parametrového servera. Jeho najdôležitejšou úlohou je zabezpečenie komunikácie uzlov v programe.

Aj napriek mnohým výhodám má ROS1 aj nedostatky, ktoré sa t'ahajú už od jeho počiatkov. Sú to napríklad:

- Nepostačujúca distribuovanosť systému. Všetky uzly sa spoliehajú na funkčnosť roscore-u,
- ROS1 je písaný v starom štandarde, to vnáša do programu technologický dlh a bezpečnostné riziká,
- Kvalita komunikácie sa nedá ovplyvniť,
- Preddefinované vláknové moduly [4],
- Možnosť užívateľa predefinovať základne prvky ROS-u.

Kvôli takýmto problémom a nedostatkom sa začala vyvíjať nová verzia ROSu, ROS2. Tá mala vyriešiť tieto problémy a zlepšiť funkcionalitu prvej verzie. V roku 2025 sa skončí podpora poslednej distribúcie ROS1 menom *Noetic*. Preto je odporúčané začínať nové projekty v ROS2.

1.5 ROS2

Ako už bolo spomenuté zámerom vývoja ROS2 bolo zlepšenie funkcionality a bezpečnosti systému. Jeden z dôsledkov tohto vývoja je, že ROS2 nie je späť kompatibilný. Podstata toho, ako sú zoskupované uzly a ako spolu komunikujú je diametrálne odlišná od ROS1. Z tohto dôvodu bol vyvinutý takzvaný rosbridge, ktorý zabezpečuje kompatibilitu medzi verziami. Nie je to ale trvalé riešenie. Odporúčané je nástroj využívať a počas toho prepisovať kód z verzie 1 do verzie 2. Komunikácia prebieha v ROS2 rovnakým spôsobom ako v ROS1. Pomocou tém 1.3.1, služieb 1.3.2 a akcií 1.3.3.

Táto podobnosť končí na najvyššej vrstve. Ako sme videli na Obr. 1.4. Štruktúra ROS2 je rozdelená do viacerých vrstiev. Najdôležitejšie je pre nás vedieť, že komunikácia je spracovávaná modelom DDS (Služba distribúcie údajov) z anglického (Data Distribution Service). Tento model zlepšuje výkon, stabilitu a bezpečnosť modelu oproti ROS1. Je založený na TCP/IP a UDP/IP protokole. Z obrázku Obr. 1.4 vyčítame aj lepšie rozloženie modulov. To zabezpečuje jednoduchšie prispôsobovanie systému pre nové funkcionality. Podpora operačných systémov sa v ROS2 rozšírila aj o Windows, Mac OS či RTOS (Operačné systémy reálneho času) z anglického Real time operating system. Operačné systémy nie sú jediné rozšírenie ohľadom kompatibility. S ROS2 je možné programovať už aj v Java či Matlabe. Tvorcovia mysleli aj na programátorov a pridali rozšírené možnosti testovania, debugovania či nasadzovania programu do reálneho využitia.

Testovanie prebieha pomocou používania Google testov. Debugovanie je možné uskutočniť pomocou debuggera gnu-gdb. Pri spustení programu cez spúšťací súbor (launch file) je potrebné pridať príkaz na spustenie spomenutého debugovacieho programu.

ROS2 má necentralizovanú štruktúru, a preto pri spúšťaní programov už nie je potrebné mať spuštený roscore. Ak teda spadne jeden uzol všetky ostatné uzly budú fungovať nadálej. V ROS1 sme vedeli ovplyvniť počet uchovaných správ komunikácie pokým nepretiekol zásobník, ktorý ich uchovával na neskoršie použitie. V ROS2 vieme implementovať túto schopnosť použitím takzvanej *QoS* triedy (kvalita komunikácie), z anglického Quality of Service. Pomožou tejto triedy vieme aj zmeniť kvalitu komunikácie. Vieme si zadefinovať, či by sme radšej stratili niektoré správy, ale dostali by sme všetky rýchlo. Alebo aby sa zabezpečilo, že dostaneme všetky správy, ktoré boli vyslané, aj keby to trvalo dlhšie. Dokonca si vieme zadefinovať maximálny čas, ktorý budeme čakať na ďalšiu správu.

Ak by bol užívateľ veľmi schopný programátor a potreboval by si zmeniť triedy, ktoré definujú základnú funkcionality ROS-u, tak aj toto je možné. Jednou z takýchto funkcionálít je, že užívateľ si vie predefinovať triedu, ktorá bude alokovovať miesto na (IPC) komunikáciu (medziprocesovú komunikáciu). K tomuto bodu je dodať, že tento prípad je špecifický a väčšina programátorov sa s takouto možnosťou do kontaktu nedostane.

Pri všetkých týchto zlepšeniach nemôžeme zabudnúť spomenútajúci nasledovný nedostatok. Keďže ROS2 je mladší ako ROS1 nájdeme k nemu menej dokumentácie. Pridaním veľkého počtu funkcionálít začal vznikať problém pre začiatočníkov s porozumením niektorých kódov. Avšak tento problém je nedostatkom, ktorý časom zanikne. V čase písania tejto práce pribudli na stránke dokumentácie minimálne 2 strany popisujúce pokročilejšie Funkcionality druhej verzie ROSu.

1.6 Rozdiely

Čo je určite dobrou správou pre všetkých programátorov, ktorí robili v prvej verzii a sú zvyknutí na jej štandardy a funkcionality. Tak tito sa nemajú čoho obávať. Prechod z ROS1 na ROS2 je dosť priamočiary. Čo sa zmenilo je spôsob písania kódu, ale koncepty ostali všetky rovnaké. V tejto sekcii nebudeme písat konkrétné kódy, budeme len opisovať čo je podobné a čo zasa rozdielne medzi verziami spomínaného systému. Keďže celý projekt bol písaný v programovačom jazyku C++ tak sa aj tieto zmeny budu týkať hlavne jazyka C++.

1.6.1 Štandard jazyka

Pokým ROS1 bola písaná v štandarde C++03 tak ROS2 je už písaná v novom štandarde. A to hlavne C++11, ale používa aj nejaké časti z C++14 a C++17. To zahŕňa inicializovanie templatov a ich používanie. Tým, že ROS2 je stále nová a stále vyvíjajúca sa platforma,

tak môžeme očakávať aj časti kódu, ktoré budú podporovať najnovší C++ štandard a to štandard z rokov 2020 a 2023.

Definície a deklarácie templatov sú na knihu samú o sebe, preto do detailov nebudeme zachádzat'. Stačí nám vedieť', ako ich inicializovať'. V prvej verzii sme definovali všeobecného publishera (publikovateľa) a definovali sme mu len cez akú tému má posielat' správy. V druhej verzii navádzame publishera na špecifický tip správy akú posielame. Nemôže sa teda stat', že takýto program by sme skompolovali a následne, keď ho spustíme, tak by spadol z dôvodu, že čítame iný typ správy ako posielame.

1.6.2 Inicializácia nody (uzla)

Tak isto ako v prvej verzii aj v druhej verzii musíme definovať uzol (node). Rozdiel je v tom, že prvá verzia obsahovala NodeHandle (Ovládač uzla) a druhá verzia obsahuje priamo Node (Uzol). V druhej verzii je zaužívaným štandardom túto nodu prededit' a použiť polymorfizmus pri objekte, ktorý bude existovať počas celej doby vykonávania programu. Pri prvej verzii tomu tak nebolo. Tam sme museli vytvoriť už spomenutý NodeHandle. Ten sa nemusel využiť ako base trieda a nemusel ani existovať počas celého behu programu.

1.6.3 Komunikácia

DDS (Služba distribúciami údajov) je protokol strednej vrstvy (middleware) implementovaný nad UDP [1]. Na implementáciu tohto protokolu je použitý protokol z IoT (internet vecí) (Internet of Things) sféry. Je to protokol MQTT. DDS je používaný v ROS2 na komunikáciu medzi uzlami. Je to systém správ publikovania (publish) / odoberania (subscribe), ktorý umožňuje uzlom komunikovať medzi sebou bez toho, aby poznali identitu ostatných uzlov. Druh komunikácie je v ROS2 rozšírený ešte o akcie vid' 1.3.3.

1.6.4 Parametre

ROS1 používa parametrový server, ktorý sa nachádza v roscore-e. Každý uzol si mohol vytiahnuť parametre, ktoré boli zapísané v konfiguračnom súbore. ROS2 žiadny roscore nemá, preto sa parametre musia distribuovať iným spôsobom. Parametre v druhej verzii ROSu patria jednotlivým uzlom. To znamená, že jednotlivé parametre sa dajú vytiahnuť len daným uzlom. Tieto parametre taktiež existujú len počas existencie daného uzlu. Parametre sú d'alej distribuované pomocou už spomínанého DDS protokolu. V prípade, že sa tieto parametre nepodarí vytiahnuť z konfiguračného súboru. Či už z dôvodu, že daný súbor neexistuje, alebo iného dôvodu, tak sa aplikujú základné hodnoty, ktoré si zvolil užívateľ pri používaní funkcie na ich zist'ovanie.

1.6.5 Nodelet alebo komponent

ROS1 ponúka možnosť definície uzlov ako uzlík (`nodelet`). Je to definovanie uzlu ako zdielanej knižnice (shared library). Je to spôsob ako ulahčiť prácu CPU. Keď sa definuje uzol ako uzlík, tak jeden proces môže spracovať programy z viacerých takýchto uzlíkov. Táto funkcia sa nachádza aj v ROS2. Volá sa komponent (`component`). Vylepšením oproti nodelet-om je zjednotenie aplikačnej implementácie (API). Pokým nodelet-y mali vlastný spôsob implementácie v ROS1 tak v ROS2 je implementácia uzla a komponentu rovnaká. Pri komponente sa musí len naviac definovať, že daný komponent existuje pomocou makra. Použitie komponentov zjednoduší prácu CPU a používa sa hlavne v zariadeniach, ktoré majú obmedzený výkon výpočtovej techniky. Sú to napríklad mikroprocesory, ktoré ovládajú roboty.

1.6.6 Kompilácia

Zmenou verzii sa zmenil aj spôsob komplikácie programu. ROS1 bol komplikovaný pomocou `catkin build` systému. Catkin je založený na programe `cmake`. Jeho nastavenie dependencií je konfigurované pomocou súboru `package.xml`. ROS2 prešiel na viac nastaviteľný systém `Colcon`. Tento systém je na rozdiel od catkin-u založený na Python-e a jeho dependencie sa nastavujú pomocou `setup.py` súboru. V prípade colcon-u si môžeme definovať spôsob komplikácie to znamená, že môžeme nastaviť, ako sa budú spracovať dependencie. Ponúkané možnosti sú `catkin_make`, `catkin_make_isolated`, `catkin_tools` a `ament_cmake`. Jednou s najviac používaných možností je `ament_cmake`. Je založený na programe `cmake` a spolupracuje so systémom `colcon`. Z tohto dôvodu mu vieme definovať dependencie pomocou xml súboru ako tomu bolo v ROS1 pričom možnosť definície pomocou Python skriptu ostáva. Je to jeden zo spôsobov, ako zmeniť rozdiel medzi ROS1 a ROS2.

1.6.7 Vlákna

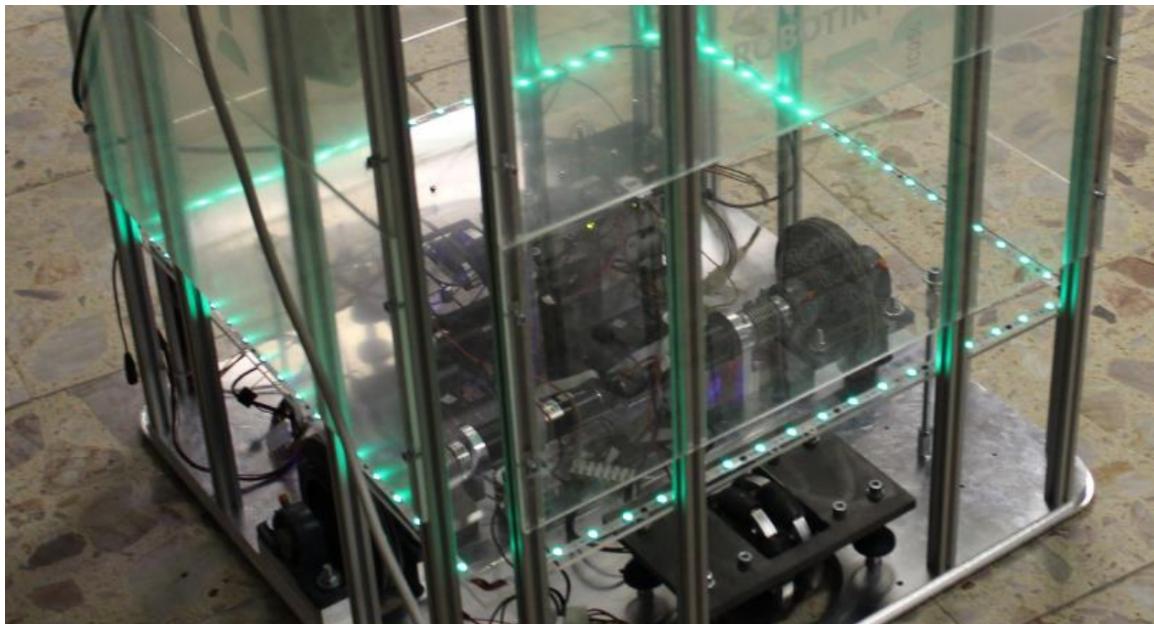
ROS1 dovoľuje programátorom vybrať si medzi jedno vláknovým a viac vláknovým vykonávaním programu. Tvorcovia ROS2 si dali zaležať na modularite aj tejto oblasti kódu. V druhej verzii ROS-u si vieme zadefinovať typ vykonávania programu separátne pre každý uzol a vieme si tento typ zadefinovať aj sami [4].

2 Pôvodný stav robota

2.1 Robot a jeho ovládanie

Robot, s ktorým sme pracovali bol výsledkom tímového projektu viacerých študentov z roku 2019. Pri vysvetľovaní a opisovaní robota sa budeme odvolávať na dokumenty, stránky a kód, ktorý napísali. Všetky tieto údaje si sprístupnené na mobilnom robote v záložke § (HOME) /Desktop/Blackmetal [5].

Robot je v tvare kvádra. Jeho šírka je 60cm a je vyzdvihnutý nad zem o 1.5cm. Nachádza sa na kolesách o polomere 8cm. Jeho kostra, až na oceľové pláty, ktoré držia robot, je spravená z hliníku. Konkrétnie z hliníkových tyčí, ktoré sú pospájané plexisklovými plátkmi. Jeho podobizeň vidíme na nasledujúcom obrázku.



Obr. 2.1: Zobrazenie spodnej časti mobilného robota [5]

Na obrázku d'alej vidíme olemovanie robota pásom s LED-kami. Tie svietia nasledovným spôsobom. Keď sa robot nehýbe všetky LED-ky svietia na zeleno. Keď sa robot pohne do nejakej strany, LED-ky znázornia jeho pohyb tým, že svietia na strane, do ktorej sa robot hýbe. Keď nastane situácia, kedy počítač ovládajúci motory prestane komunikovať s Arduinom, ktoré sa stará o detekciu stavov robota tak LED-ky začnú blikat červeno-modrými farbami.

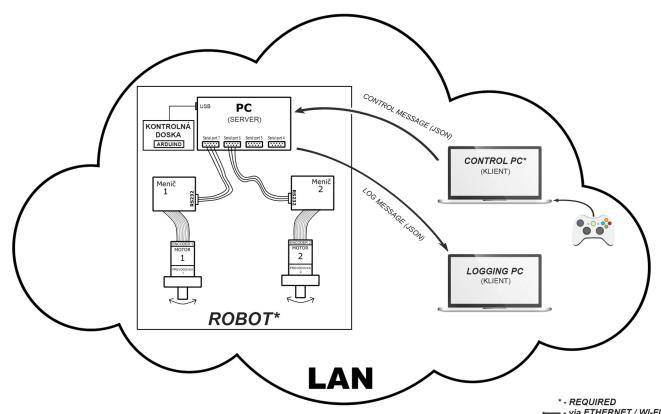
Ako bolo spomenuté LED-ky znázorňujú pohyb robota. Ten sa pohybuje za pomocí diferenciálneho podvozku s dvoma podpornými všesmerovými kolesami. Motory robota sú pripojené na meniče. Tie sú ovládané priamo príkazmi z počítača.

Hardware robota sa skladá z:

- kontrolnej dosky Arduino Uno,
- Počítača ADVANTECH MIO-5272 [6]
Počítač obsahuje operačný systém Ubuntu 16.04.
- Extension board MIOe-210 [7]
- Meniče MAXON EPOS 24/5 (s číslom 275512) [8]
Sú napájané jednosmerným napäťom 11 - 24 V a 5 A.
- Enkódery MAXON Encoder MR Type L (s číslom 225787) [9]
Rozlíšenie enkóderov je 1024 impulzov s troma kanálmi.
- Motory MAXON RE 40 (s číslom 148867) [10]
Motory s výkonom 150W. Maximálna rýchlosť je 12 000 rpm a efektivita 91%.
- Prevodovka MAXON Planetary Gearhead GP 42 C (s číslom 202120) [11]
Redukcia prevodovky je 43:1. Jej účinnosť je 72%.

Ovládanie robota je zabezpečené externými počítačmi

- Control PC (Kontrolný počítač) – Počítač posielajúci príkazy na robot cez TCP/IP protokol.
- Logging PC (Logovací počítač) – Počítač prijímajúci stav robota cez TCP/IP protokol.



Obr. 2.2: Schéma zapojenia jednotlivých častí na robote

Tieto počítače sú len reprezentácia servera. V realite to môže byť jeden a ten istý počítač.

Na obrázku Obr. 2.2 vidíme zapojenie jednotlivých častí robota. Čo sme nespomenuli a je na obrázku je XBox ovládač je to kvôli tomu, že tímový projekt bol zameraný na ovládanie robota pomocou tohto ovládača. My ho ale používat' nebudem.

2.2 Komunikácia s robotom

S robotom sa vieme spojiť pomocou dvoch portov. Jeden port je otvorený na prijímanie požiadavok (requestov) a ten druhý je na monitorovanie stavu robota. Port *664* je otvorený pre tisíc užívateľov, ktorí môžu len sledovať stav robota. Druhý port je na prijímanie requestov *665* a je otvorený len pre jedného užívateľa.

2.2.1 Logovanie

Spomínaný port *664* je otvorený jednému užívateľovi. Keď sa užívateľ pripojí začne dostávať nepretržité správy typu **JSON** (JavaScript Object Notation), ktoré hlásia stav robota. Správy, ktoré dostávame sú nasledujúceho formátu

```
{"state":1,"direction":1}
```

Hodnoty sa pri stave (state) a ani pri smere (direction) nemenia. Sú to stále jednotky. Po kóym robota nezastavíme bud' príkazom, stlačením tlačidla vypnutia alebo zablokováním jedného z kolies, tak sa tieto správy budú posielat'. Môžeme potom začať polemizovať o tom či by nebolo lepšie už tieto správy využiť na to čo reálne spomenutý **JSON** reťazec ukazuje. A to udávať smer a stav robota. Momentálne tieto správy slučia len na to, aby sme vedeli, že tento robot je aktívny a vie primat' a spracúvať informácie.

2.2.2 Ovládanie

Port *665* je sprístupnený na prijímanie a odosielanie požiadavok a ich odpovedí. Príkazy sa na počítač posielajú cez siet' z externého počítača vo formáte **JSON**. Študenti, ktorí navrhovali systém posielania požiadavok (request) a odpovedí (response) robili tieto správy ručne. Preto nastávajú situácie, kedy robot pošle správu, ktorá nespadá do štandardu písania JSON textu. Z tohto dôvodu sme nemohli použiť už existujúci kód (parser), ktorý by nám zjednodušil prehľadávanie týchto správ. Podľa dokumentácie sa robot mal ovládať správami typu [12]

```
{"UserID":1,"Command":3,"RightWheelSpeed":50,"LeftWheelSpeed":50}
```

Význam jednotlivých parametrov:

- **UserID** – Znázorňuje ID užívateľa, ktorý je pripojený na robot. Predvolená hodnota je 1.
- **Command** – Číselná hodnota znázorňujúca príkaz, ktorý ma robot vykonat'
 0. Prázdny príkaz slúžiaci na overenie spojenia
 1. Núdzové zastavenie
 2. Normálne zastavenie
 3. Príkaz nastavujúc rýchlosť kolies mobilného robota
 4. Prázdny príkaz
 5. Prázdny príkaz
 6. Príkaz pýtajúci si aktuálne rýchlosťi pravého a ľavého kolesa.
 7. Pripravenie motorov robota
 8. Príkaz pýtajúci si aktuálnu pozíciu pravého a ľavého kolesa.
- **RightWheelSpeed** – Nastavenie rýchlosťi pre pravé koleso
- **LeftWheelSpeed** – Nastavenie rýchlosťi pre ľavé koleso

2.3 Vysvetlenie kľúčov ret'azca

UserID

Táto možnosť je v momentálnom stave robota nevyužitá. Počet zariadení, ktoré sa môžu pripojiť na port, cez ktorý sa dá robot ovládať je 1. Je to ale dobrá možnosť na rozšírenie kódu. Keď sa budú môcť pripojiť viacerí užívatelia, tak sa bude musieť vyriešiť, koho príkaz bude mať akú prioritu.

RightWheelSpeed/LeftWheelSpeed

Nastavovanie rýchlosťi pravého a ľavého kolesa nie sú povinné parametre. Musíme ich zadávať len v prípade posielania rýchlosťí cez príkaz s číslom 3 alebo 4.

3 Zmeny na robote

3.1 Pokazený systém

Prvýkrát, keď sme prišli k robotu do NCR (Národné Centrum Robotiky), tak sme si určili ako prvú úlohu zálohovať systém. Ak by sa teda stala nejaká chyba a robot by prestal fungovať, tak by sme sa vedeli dostať do posledného funkčného stavu robota. Zálohovanie systému nanešťastie nebolo možné hned po zapnutí robota. Bolo to preto, lebo po tom, ako na robote robili študenti tímový projekt [5], tak na robote niekto stiahol ROS1 (Robot Operating System) verzia *Lunar Loggerhead*. Toto by problém nespôsobilo, čo ale problém spôsobilo, bolo jeho nesprávne odinštalovanie. Táto akcia mala za dôsledok vypisovanie nasledovnej chybovej hlášky pri zálohovaní.

```
E: Unable to~correct problems, you have held broken packages.
```

Tento problém sme vyriešil vymazaním všetkých knižníc, ktoré boli nainštalované spolu s ROS1. Toto vyriešilo problém. Na vymazanie týchto knižníc sme použili nasledovný príkaz.

```
sudo apt autoremove && sudo dpkg --remove $(dpkg --get-selections | grep hold)
```

Tento príkaz najprv odstráni všetky balíčky, ktoré nie sú používané žiadou aplikáciu v systéme. Následne vyhľadá všetky knižnice, ktoré taktiež nie sú používané žiadnym balíčkom a násilu odinštaluje. Po vykonaní tejto operácie sme ešte museli odstrániť všetky referencie na ROS1 v systéme. Toto sme vykonali odstránením prepojení na už neaktívne repozitáre, ktoré sa nachádzali v súbore */etc/apt/sources.list*.

3.2 Chyba v dokumentácii

V sekcii 2.2.2 sme uviedli pôvodný stav ovládania robota pomocou správ typu JSON. Z kódu 2.2.2 je jasné, že sa majú posielat celé čísla a na základe tohto vstupu sa bude robot hýbať. Čo sme zistili až po skompilovaní a spustení tímového projektu je, že sa majú posielat desatinné čísla z intervalu 0 až 1. Toto nebolo písané v dokumentácii, ktorá nám bola dodaná na začiatku programu. Môžeme preto príklad prepísať na reťazec, ktorý by fungoval:

```
{"UserID":1,"Command":3,"RightWheelSpeed":0.50,"LeftWheelSpeed":0.50}
```

3.3 Rozšírenie príkazov robota

Prvotne sme mali v plane získavať pozíciu robota pomocou príkazov, ktoré sa nachádzajú v knižnici enkóderov. Tieto príkazy mali vracať počet impulzov IRC snímačov prejdených od spustenia robota. Pre zaobchádzanie s touto funkcionálou sme definovali dva príkazy:

Command: 4

Tento príkaz je prázdny. My sme ho ale neskôr prepísali na príkaz, cez ktorý sa dá nastaviť žiadana pozícia kolies robota (ich natočenie) pomocou funkcie z knížnice EPOS [13]. Táto funkciu nie je v takom stave ako sme si priali. Je to spôsobené hlavne nepostačujúcou dokumentáciou enkóderov na robote. Síce sme našli v dokumentácii funkciu, ktorá by mala túto možnosť povolovať. Čo sa ale stane pri poslaní príkazu je to, že kolesá sa začnú točiť rýchlosťou 0,5 metra za sekundu.

Command: 8

Príkaz na zisťovanie polohy kolies neboli originálne naprogramované na robote. Pridali sme ho za cieľom presného dostania sa robota na preddefinované miesto. Táto funkciu nefunguje správne rovnako ako v predchádzajúcom príklade, keď si vypýtame polohu kolies od robota, dátu ktoré obdržme sú, že jedno koleso je priamo nastavené na hodnotu, ktorú sme si vyžiadali a to druhé koleso vráti náhodnú hodnotu. Počas toho sa ale kolesá robota stále točia.

Tento postup sa ukázal ako neuskutočiteľný, lebo príkazom 8 sme z jedného enkódery získavali informácie priamo nastavenej polohy a z druhého enkódera sme dodržiavali čisto náhodné dátu. Z tohto dôvodu sme sa rozhodli využívať spätnú väzbu, ktorá obsahuje rýchlosťi robota.

3.4 Nesprávna spätná väzba

Ako bolo spomenuté vyššie, pri poslaní príkazu s číslom 6 nám robot vráti aktuálne rýchlosťi kolies. Počas skúšania tejto funkciu sme narazili na problém. Keď sme sa robota spýtali na jeho rýchlosťi. Dostali sme reťazec, ktorý obsahoval náhodne veľké čísla. Tieto čísla sa menili, keď sme zadávali nejaké hodnoty pre rýchlosťi kolies aby sa robot hýbal. Ich magnitúda ale ostávala nezmenená. V nasledujúcom príklade môžeme vidieť ako tento reťazec vyzerá:

```
{"LeftWheelSpeed":236223201280 "RightWheelSpeed":4294967296}
```

Ako si môžeme všimnúť. Pri tomto type správ nie je dodržaná správna forma reťazca typu **JSON**. Namiesto ‘:’ máme ‘=’ a medzi argumentmi sa nenachádza čiarka. Hned’ ako prvú vec sme chceli tento štandard napraviť. Bohužiaľ na tomto robote už bolo spravených niekoľko projektov a museli by sme prejsť každý z nich a zistíť či používajú túto spätnú väzbu. Ak by ju používali museli by sme tieto kódy upraviť.

V dokumentácii robota bohužiaľ nebolo písané v akom formáte sa tieto rýchlosťi kolies majú nachádzať. Preto jeden z nápadov ako zistíť presne v akom formáte sa posielali tieto čísla

bolo vyskúšať' pár možností:

- *long* - celé číslo s malým endianom
- *long* - celé číslo s veľkým endianom
- *float* - desatinné číslo s malým endianom
- *float* - desatinné číslo s veľkým endianom

Ked'že robot má počítač so 64 bytovým procesorom [6], tak *long* aj *float* budú mať 64 bitovú dĺžku. Po skúsení všetkých štyroch možností sa ukázalo, že ani jedna nebola správna a problém je niekde inde.

Aby sme pochopili, ako sa máme správať k prijatým dátam musíme vedieť ako funguje program na robote. Skopírovali sme si preto kód z robota a pustili sme sa do jeho analýzy. Pri poslaní požiadavky na nastavenie rýchlosť kolies si ich robot premení na celé čísla v rozsahu 0 až 1000. To je hodnota, na ktorú nastaví rýchlosť otáčania pravého a ľavého kolesa respektíve rýchlosť otáčania ich motorov. Na druhú stranu, keď si vypýtame od robota rýchlosť kolies. On zoberie informáciu z enkóderov a pošle nám ju bez spracovania. Aj napriek týmto poznatkom sa nám nepodarilo získať z týchto dát žiadane rýchlosťi.

Po dôkladnom preštudovaní kódu sme zistili, že hodnoty ktoré nám posielala robot nie sú ani vytahované z enkóderov správnu funkciou. Preto sme ju zmenili a začali sme dostávať hodnoty, s ktorými by sa mohlo dať pracovať'.

Funkcie z knižnice zabezpečujúce komunikáciu z enkóderov motorov pochádzajú z firmy Maxon [13]. Táto dokumentácia nebola moc nápomocná. Opisy jednotlivých funkcií boli len ich rozložené názvy na osobitné slová. Aj napriek tomu sa nám podarilo nájsť funkcie, ktoré sme potrebovali. Funkcie obsahujúce slovo ‘Target’, majú návratné hodnoty reprezentujúce žiadane hodnoty. Funkcie s príponou ‘-Is’ vracajú aktuálne hodnoty. Z tohto dôvodu sme museli prepísať funkciu na robote, ktorá sa vykonávala. Pre získanie aktuálnych hodnôt rýchlosťi motora poslaním príkazu 6 sme museli zmeniť nasledujúcu funkciu:

```
BOOL VCS_GetTargetVelocity(
    HANDLE KeyHandle,
    WORD NodeId,
    long* pTargetVelocity,
    DWORD* pErrorCode);
```



```
BOOL VCS_GetVelocityIs(
```

```

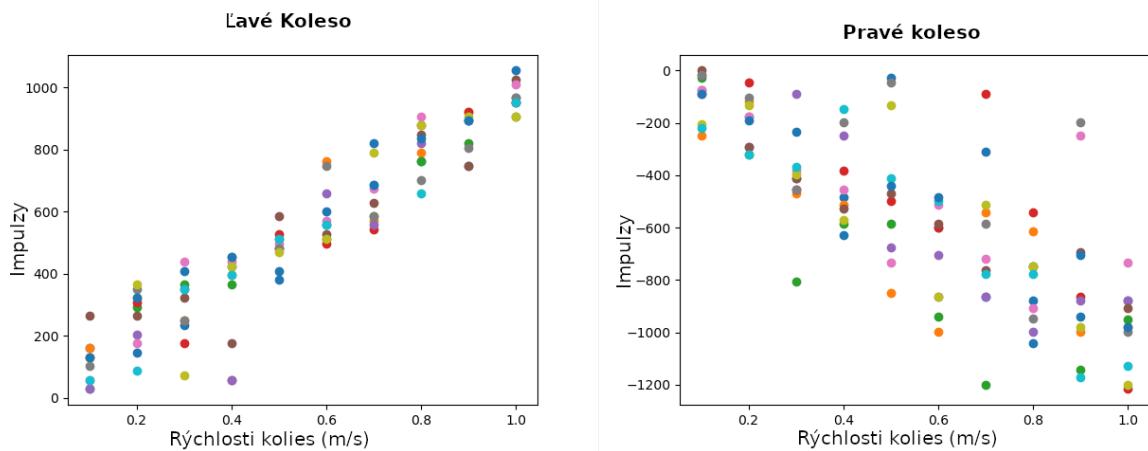
HANDLE KeyHandle,
WORD NodeId,
long* pVelocityIs,
DWORD* pErrorCode);

```

Ako môžeme vidieť v týchto predpisoch funkcií, bolo treba zmeniť názov funkcie a ostatné parametre ostali rovnaké. Nebolo treba meniť implementáciu kódu.

3.5 Zašumený výstup

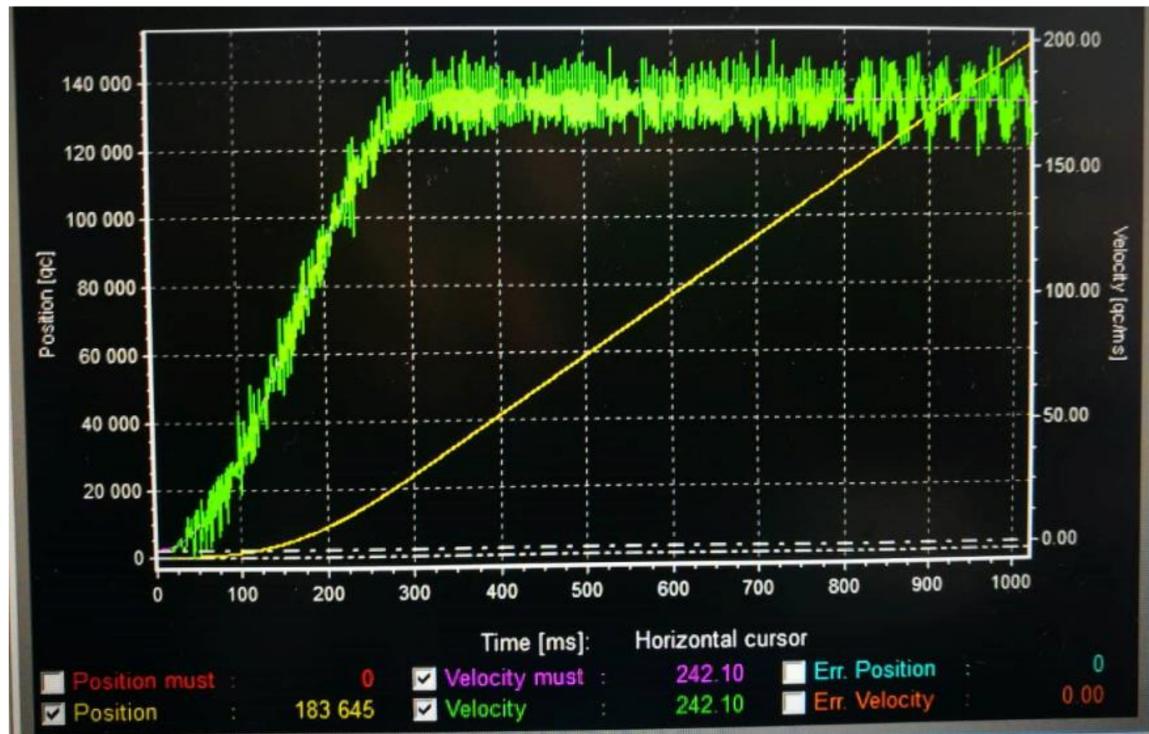
Po prepísaní funkcie na získavanie rýchlosťí robota sme spravili pári meraní, aby sme zistili, aké presné informácie o rýchlosťach motorov dostávame. Aby nám robot neodbiehal postavili sme ho na vyvýšené miesto, tak aby sa kolesá nedotýkali zeme. V takomto postavení sa robot nepohne z miesta a my môžeme bez problémov odmerať prechodové a prevodové charakteristiky rýchlosťí pravého a ľavého motora.



Obr. 3.1: Ustálené hodnoty rýchlosťi ľavého a pravého motora.

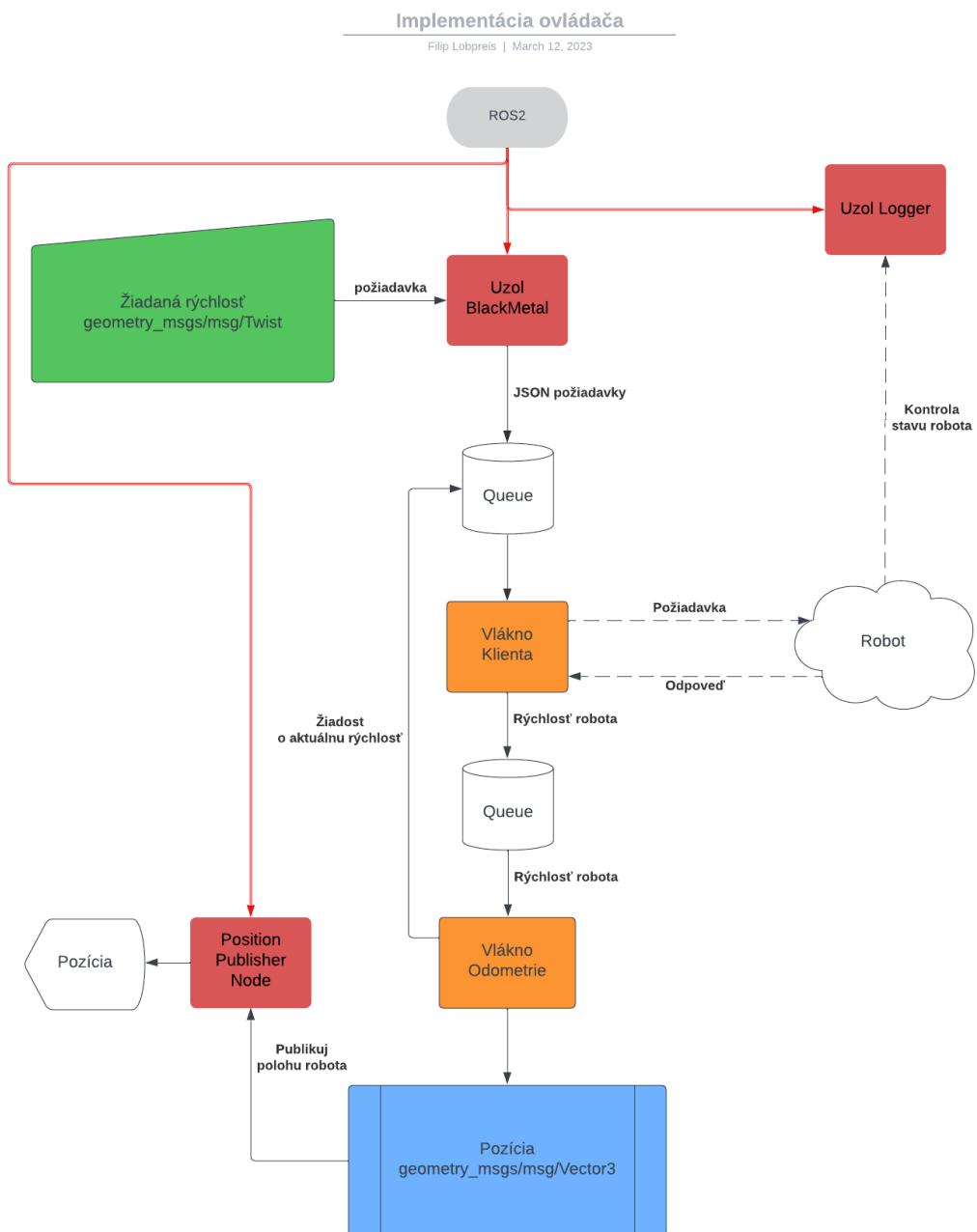
Na obrázku Obr. 3.2 vidíme zašmený signál rýchlosťi, ktorú sme obdržali z enkóderov. Prvá otázka, ktorá nám napadla bola, či aj iné projekty na tomto robote mali rovnaký problém. Kontaktovali sme preto jedného z autorov tímového projektu [5], pána Adriána Kasperkeviča. On nám odpísal s tým, že aj oni mali problémy s enkódermi. Ich problémy boli naviazané na staré enkódery, ktoré neskôr vymenili. Na nových enkodéroch avšak netestovali ich spätnú väzbu. Na túto tému sme sa rozprávali aj s pánom Ing. Martin Dekan PhD. Pán doktor Dekan nám spomínal, že on mal so študentmi raz podobný problém. Tento problém bol spôsobený vzájomným rušením neizolovaných častí káblu. Keď sa pozrieme na obrázok Obr. 2.1, tak môžeme vidieť nesprávny manažment káblu. Toto neusporiadane vedenie káblu môže spôsobovať rušenie signálu spätnej väzby. Túto teóriu potvrdzuje aj fakt, že spätná väzba rýchlosťi ked' robot stoji

neobsahuje rušenie a je zaručene 0.



Obr. 3.2: Prechodová charakteristika rýchlosť kolies [5].

4 Implementácia ovládača



Obr. 4.1: Graf vykonávania programu na ovládanie robota pomocou ROS2.

4.1 Úvod do čítania grafu

Na obrázku Obr. 4.1 môžeme vidieť viacero objektov rôznych farieb. Objekty zobrazené červenou farbou sú uzly spracovávané a vytvárané v rámci ROS2. Každý tento objekt sa vykonáva v osobitnom procese. Objekty zobrazené oranžovou farbou sú objekty, ktoré majú svoje vlastné vlákno. Tieto objekty boli vytvorené uzlom BlackMetal. Dátové štruktúry Queue, zobrazené bielou farbou sú vytvorené tak, aby zabezpečovali bezchybnú komunikáciu a synchronizáciu medzi viacerými vláknenami. Objekt so zelenou farbou je vstup do programu. Je zadávaný užívateľom a reprezentuje žiadanú rýchlosť robota. Modrý objekt je výstupom programu. Je to téma, na ktorú sa publikuje aktuálna pozícia robota. Robot samotný je zobrazený bielou farbou vo forme malého obláčika. Prerušované čiary na diagrame znázorňujú siet'ovú komunikáciu ovládača a robota. Červené dvojité čiary udávajú, ktoré objekty patria ROS-u. Nakoniec čierne plné čiary reprezentujú tok dát medzi objektmi.

4.2 Uzly

Na obrázku Obr. 4.1 môžeme vidieť postup vykonávania programu na ovládanie robota BlackMetal pomocou ROS2. Na začiatku programu sa vytvoria 3 uzly. Prvý uzol **Position Publisher** je uzol, na ktorý sa publikuje vypočítaná pozícia robota. Tento uzol existuje v programe len na overenie publikovaných informácií a ich následné uchovávanie. Pozícia sa počíta sa na základe obdržaných dát z enkóderov robota. Uzol **Logger** slúži na zaznamenávanie stavu robota 2.2.1. Posledný uzol **BlackMetal** ovláda robota podľa zadaných dát užívateľom.

4.3 Vstup

Uzol **BlackMetal** vytvorí príjemcu, ktorý počúva na téme `/cmd_vel` a príma správy typu `geometry_msgs/msg/Twist`. Tento vstup je vo forme príkazu zadaného v príkazovom riadku. Vyzerá nasledovne:

```
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist
    "linear:
        x: 0.0,
        y: 0.0,
        z: 0.0,
    angular:
        x: 0.0,
        y: 0.0,
        z: 0.0" -1
```

Tento príkaz publikuje jednu správu (-1) na tému `/cmd_vel`. Obsahuje lineárne a uhlové rýchlosťi. Z tejto správy sa využijú dva údaje lineárna rýchlosť po osi x a uhlová rýchlosť po osi z . Ostatné lineárne rýchlosťi neovplyvňujú chod robota rovnako ako ďalšie uhlové rýchlosťi. Táto správa typu `geometry_msgs/mgs/Twist` je následne spracovaná a uložená do rady `Queue`. Táto rada je prioritne založená. To znamená, že požiadavka s nižším kódom ma vyššiu prioritu. Požiadavky a ich kódy môžeme vidieť v sekciách 2.2.2 a 3.3.

4.4 Komunikácia s robotom

Ako je naznačené na Obr. 4.1, klient si vo svojom vlastnom vlákne vytiahne prvú správu z rady a pretransformuje ju do formy JSON. Tento typ správy môžeme vidieť v sekcii 2.2.2. Príkaz je poslaný robotu a ten obratom dá vedieť, či danú požiadavku obdržal. Ak obsah tejto správy žiadal o vrátenie rýchlosťí kolies, tak robot ďalšou správou odpovie na danú požiadavku. Typ tejto odpovede môžeme vidieť v 3.4. V tomto prípade sa správa spracuje a uloží sa do ďalšej rady.

4.5 Odometria

Odometria, počítanie polohy na základe rýchlosťi kolies, sa vykonáva rovnako ako komunikácia s robotom, v separátnom vlákne. Tu je potreba si uvedomiť jednu skutočnosť. To je tá, že keď posielame žiadosť na nastavenie rýchlosťí kolies robota, tak robot si hodnoty v žiadosti prepočíta. Prepočítané dátá následne poskytne enkóderom. Keď si ale tieto rýchlosťi vyžiadame z enkóderov, tak sa robot týchto dát nechytá a my si ich musíme prepočíta na metre za sekundu. Aby sme dostali rýchlosťi kolies od robota, tak si ich musíme od neho vypýtať. Ako sme už spomenuli v 4.4, tieto správy majú nižšiu prioritu ako nastavenie rýchlosťí kolies alebo bezpečnostné zastavenie robota. Preto sa môže stať, že správy posielané robotu nebudú dodržiavať presne stanovenú frekvenciu v čase keď mu bude užívateľ posielat príkazy. Toto nám až tak neprekáža. My sa sice pýtame od robota jeho rýchlosť každých 100 milisekúnd, ale v odometrii si zistíme aktuálny čas a zmenu času od poslednej odpovede.

Po obdržaní rýchlosťi robota v impulzoch za sekundu si tieto dátá kvôli zašumeniu preženieme cez filter a následne ich spracujeme. Spracovanie prebieha nasledovne:

- Prepočítanie rýchlosťí kolies robota z impulzov za sekundu na metre za sekundu,
- Prepočítanie lineárnej a uhlovej rýchlosťi robota na jeho ďažisko,
- Zistenie aktuálnej zmeny času oproti predchádzajúcemu meraniu,
- Zapísanie aktuálneho času, lineárnej a uhlovej rýchlosťi do spravy,

- Prepočítanie aktuálnej polohy z nameraných rýchlosťí.

Pre cely počítania odometrie sme si odmerali polomer kolies a vzdialenosť stredov týchto kolies. *Polomer kolies (R)* sme určili na **0,08m**. *Polomer stredov kolies L* robota sme určili na **0,56m**. Z dokumentácie robota [9] sme si zistili *rozsah enkóderov (E)* (**1024** impulzov na otočku) Tieto údaje sme použili na prepočítanie impulzov za sekundu na metre za sekundu.

$$V_{mps} = \frac{2\pi R}{E} v_{imp} \quad (4.1)$$

Týmto spôsobom si vieme prepočítať rýchlosť pre pravé a ľavé koleso robota. V druhom bode sme si prepočítali lineárnu a uhlovú rýchlosť pomocou rýchlosťi pravého a ľavého kolesa.

$$V_{mps} = \frac{2\pi R}{E} v_{imp} \quad (4.2)$$

$$\omega = \frac{v_r + v_l}{L} \quad (4.3)$$

Po prepočítaní rýchlosťí kolies sme si zistili aktuálne natočenie robota

$$\gamma = \gamma + \omega dt \quad (4.4)$$

Následne sme dopočítali polohu robota v Karteziánskej súradnicovej sústave pomocou vztahov:

$$X = v \cos(\gamma) dt \quad (4.5)$$

$$Y = v \sin(\gamma) dt \quad (4.6)$$

4.6 Zdieľanie polohy

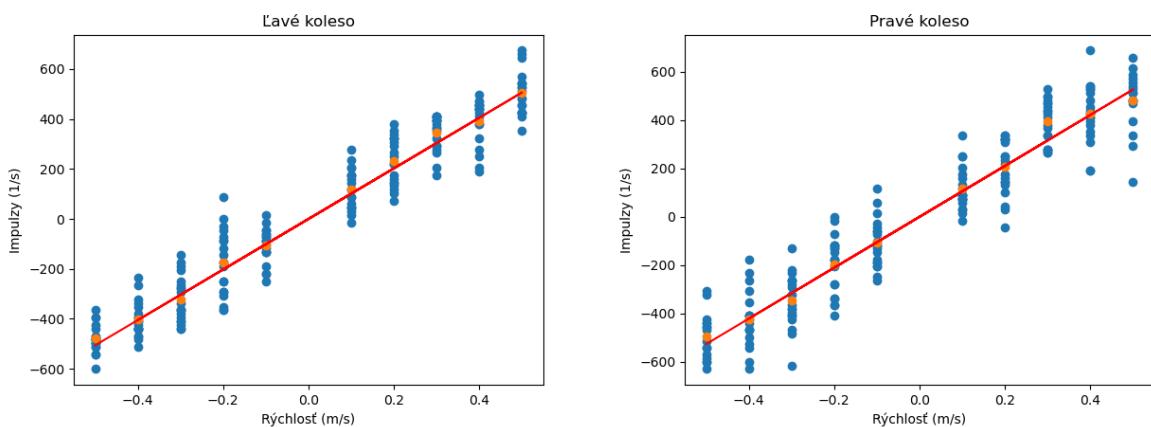
Ďalšiu vec, ktorú je treba vysvetliť ku grafu Obr. 4.1 je zdieľanie polohy. Odometria po každom prepočítaní polohy robota publikuje túto informáciu na tému */odom* táto správa je typu *geometry_msgs/msg/Odometry*. Tento typ správy obsahuje štyri hlavné časti:

- **hlavička** Hlavička obsahuje čas a identifikačný reťazec odosielajúceho rámca
- **ID dcérskeho rámca** Identifikačný reťazec prímačujúceho rámca
- **rýchlosť** Lineárne a uhlové rýchlosťi
- **poloha** Polohu v Karteziánskej súradnicovom systéme so súradnicami *x*, *y* a otočením vo forme *kavaterniónu*.

5 Filtrovanie zašumeného signálu

Ako bolo spomenuté v predchádzajúcej kapitole, rýchlosť kolies sa dajú získať z enkódarov. Tieto dátu sa posielajú v správe, ktorá pripomína JSON formát. Z týchto vzoriek poslaných robotom nevieme priamo vypočítať polohu. Musíme si tieto dátu premeniť z impulzov za sekundu $\frac{1}{s}$ na metre za sekundu $\frac{m}{s}$. Tento prevod nebude jednoznačný, pretože každý enkódery posielá dátu inak zašumené. Preto je potrebné zistiť, ako sa zmení rýchlosť pri zmene impulzov za sekundu. Tento prevod je možné získať z merania, kde po nastavení rýchlosťí zoberieme veľa dát z enkóderov a zistíme, ako sa zmení rýchlosť pri zmene impulzov za sekundu.

Prvý nápad na získanie čo najlepšej prevodovej charakteristiky bolo cez všetky dátu položiť lineárnu regresiu. To sa ukázalo ako zlé riešenie, lebo dátu, ktoré dostávame majú veľmi veľký rozptyl. Jednou z nasledujúcich úvah bolo spraviť kľzavý priemer. Toto riešenie malo tiež svoje chyby a to v tom, že zmeny zaznamenaných impulzov za sekundu sa zmenili v závislosti od rýchlosťi a smeru Obr. 5.1. V tomto bode sme vyskúšali počítať odometriu z obdržaných dát. Táto implementácia bola veľmi nepresná. Zároveň nám tento pokus potvrdil, že potrebujeme filtrovať dátu, ktoré dostávame od robota, a ktoré reprezentujú jeho rýchlosť v impulzoch. Výsledky pokusu, kde sme zistovali prevodovú charakteristiku z impulzov za sekundu na rýchlosť v SI jednotkách nájdeme na nasledovných grafoch.



Obr. 5.1: Získanie prevodu z impulzov na rýchlosť v SI jednotkách.

Ako prvú vec sme si vykreslili všetky **nazbierané dátu**. Tie sú zobrazené **modrou farbou**. Cez ne sme spravili **lineárnu regresiu**. Je zobrazená ako **červená** úsečka. Z nazbieraných dát sme si nakoniec spravili priemer, aby sme videli, ako presne aproximuje nami vypočítaná lineárna regresia priemer nazbieraných dát. **Priemery** jednotlivých rýchlosťí sú zobrazené ako **oranžové** body.

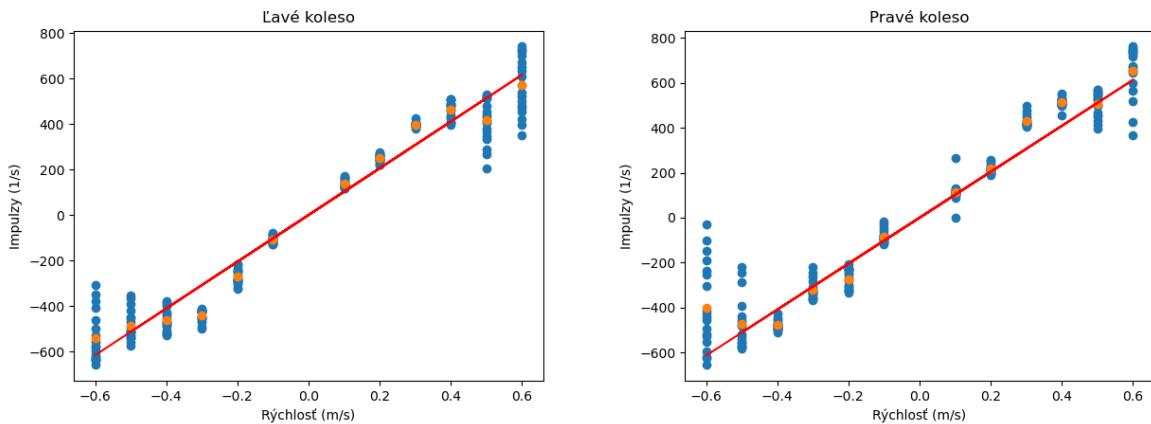
Môžeme si všimnúť, že vypočítané priemery takmer presne ležia na vypočítanej lineárnej regresii. Problémom je, ako už bolo spomenuté, že dátá, ktoré dostávame od robota sú veľmi zašumené. Preto z nich nevieme priamo počítať polohu robota. Na zobrazenie veľkosti odchýlky sme spravili meranie. Nechali sme robot aby prešiel dráhu štvorca so stranou dlhou 1 meter a rýchlosťou $0,5 \frac{m}{s}$. Výsledok bol veľmi nepresný. Metrom sme si odmerali jeho x -ovú a y -ovú súradnicu s počiatkom v bode, kde sme na robote spustili náš ovládač. Jeho skutočná poloha bola v bode (-0,3m, 0m). Čo nám ale vypočítala odometriu je, že sa robot nachádzal 4 metre od počiatku súradnicového systému.

5.1 Zistovanie parametrov α

Implementovali sme si preto dolnopriepustný kvadratický filter. Fungovanie tohto filtra spočíva v skombinovaní nového vstupného parametra a starého parametra uloženého vo filtrovom pamäti. Tento pomer je daný parametrom **alpha** α .

$$stavFiltr = \alpha * stavFiltr + (1 - \alpha) * nováVzorka$$

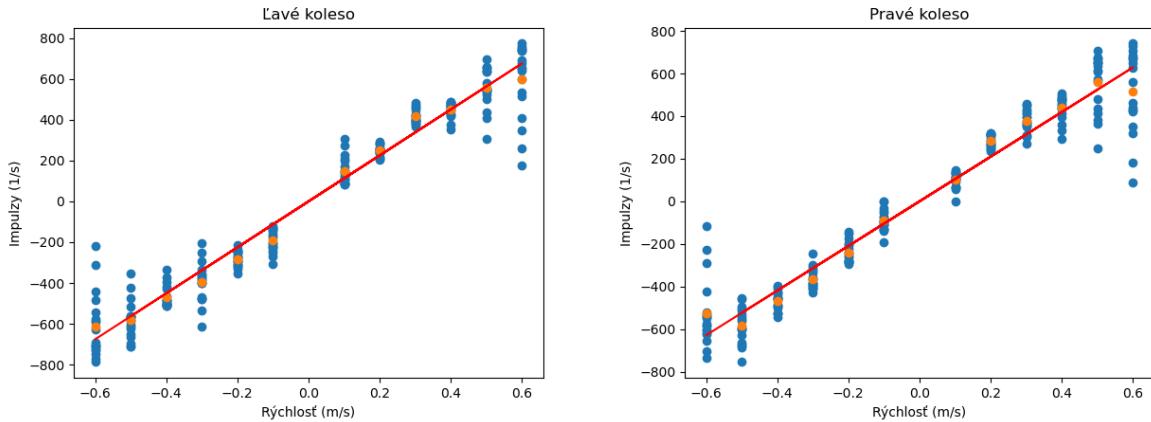
Jeho parameter α sme získali viacerými meraniami. Začali sme najsilnejším filtrom s hodnotou α rovnou 0,9.



Obr. 5.2: Získanie prevodu z impulzov na rýchlosť v SI jednotkách. $\alpha = 0,9$.

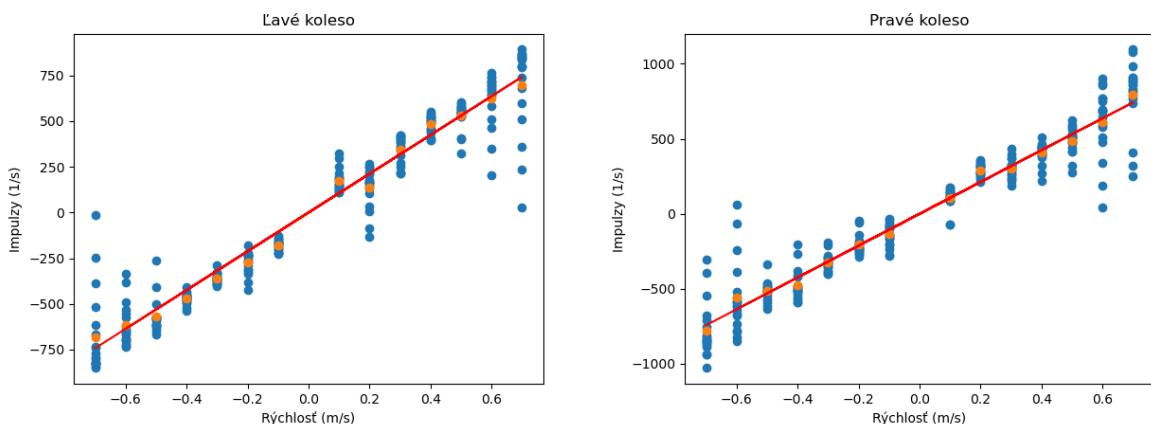
Ako môžeme vidieť na obrázkoch Obr. 5.2 a Obr. 5.1 aplikácia filtrovania výrazne pomohla proti šumu signálu. Problémom pri silnom filtrovании je to, že ak na začiatku merania dostaneme zlú hodnotu, tak sa táto hodnota t'ažko mení na správnu. Tento efekt si môžeme všimnúť skoro pri každej meranej rýchlosťi. Najviditeľnejší dopad môžeme vidieť pri pravom kolese na Obr. 5.2 pri rýchlosti $-0,6 \frac{m}{s}$. Tento problém sme riešili postupným menením parametrov filtrovania. Aby sme predišli veľkému množstvu meraní, tak sme použili metódy binárneho vyhľadáva-

nia. V tomto prípade sme začali s veľkou hodnotou a postupne sme skákali do stredu nášho intervalu. Preto sme si ako ďalšiu hodnotu zvolili alphu α rovnú 0,7.



Obr. 5.3: Získanie prevodu z impulzov na rýchlosť v SI jednotkách. $\alpha = 0,7$.

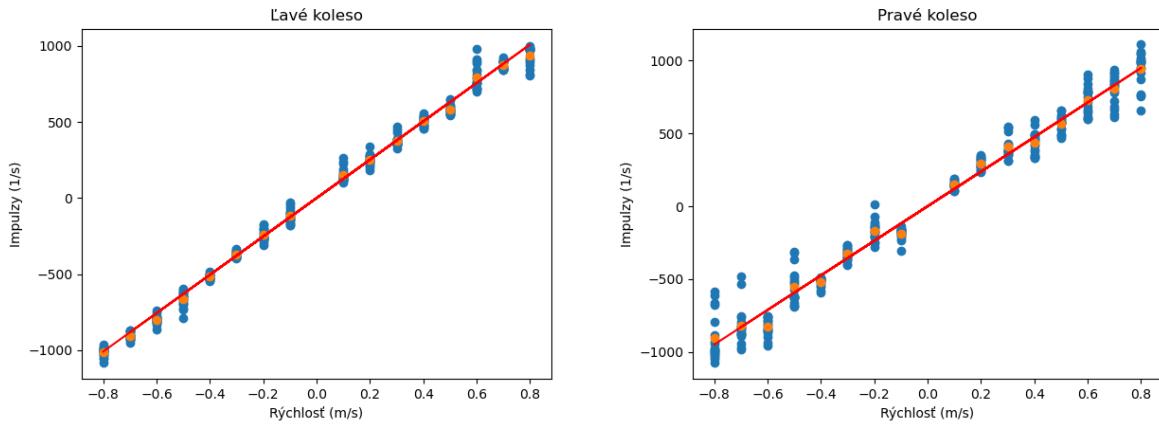
Pri použití hodnoty α rovnou 0,7 (Obr. 5.3) sme zistili, že sa hodnoty rýchlosť aj pri aplikácii filtra výrazne menili. Ustálené hodnoty zobrazené oranžovou farbou sú podobne ako pri meraní s filtrom s alphou α rovnou 0,9 mimo lineárnej regresie. Je to zapríčinené iným dôvodom ako pri silnejšom filtro. Pokým pri silnejšom filtro sme dostali zlú začiatočnú hodnotu, tak už bolo zložité ju zmeniť. Pri slabšom filtro, ak dostávame rozdielne vstupné hodnoty tak sa výstupná hodnota filtra ľahko mení. To má za dôsledok posun priemeru vstupných hodnôt. Tento efekt sa dá odstrániť zosilnením filtra, čiže zväčšením koeficientu alpha α . Spravili sme preto ďalšie meranie, kde sme použili hodnotu α rovnou 0,75.



Obr. 5.4: Získanie prevodu z impulzov na rýchlosť v SI jednotkách. $\alpha = 0,75$.

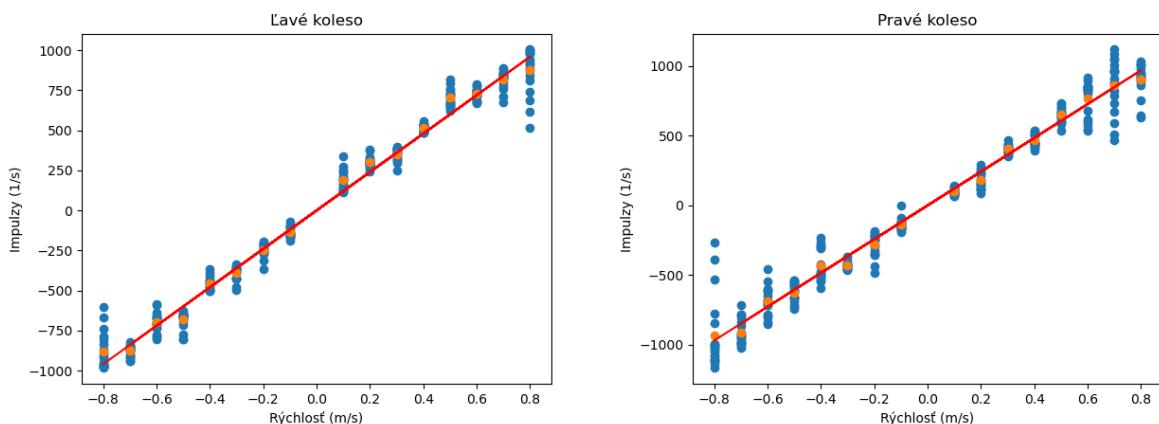
Obr. 5.4 zobrazuje graf výsledkov nameraných pri aplikácii filtra s koeficientom alpha o veľkosti 0,75. V tomto prípade sa priemerné hodnoty na rozdiel od filtrov s koeficientami

alpha 0,9 a 0,7 dostali takmer priamo na úsečku lineárnej regresie prevodu z impulzov za sekundu na metre za sekundu. Použitie silnejšieho filtra sice pomohlo rýchlejšiemu ustáleniu hodnoty, ale skúsili sme ešte silnejší filter s hodnotou alpha α rovnou 0,8.



Obr. 5.5: Získanie prevodu z impulzov na rýchlosť v SI jednotkách. $\alpha = 0,8$.

Ako môžeme vidieť na Obr. 5.5 ustálenie hodnôt je veľmi jednoznačné. Vybrali sme si preto filter s hodnotou koeficientu alpha α rovnou 0,8. Zatiaľ všetky dátá čo sme merali boli s frekvenciou 4Hz (1 vzorka za 250 milisekúnd). Pre presnejší výsledok sme túto frekvenciu ešte zvýšili. Z testov robota sme vyzozorovali, že najfrekventovaniejsia frekvencia, ktorú môže robot sprostredkovovať je 10Hz (1 vzorka za 100 milisekúnd). Spravili sme si preto test na prevod rýchlosťi ešte raz s rovnakou hodnotou koeficientu alpha α rovnou 0,8, ale s frekvenciou 10Hz namiesto už spomenutých 4Hz.

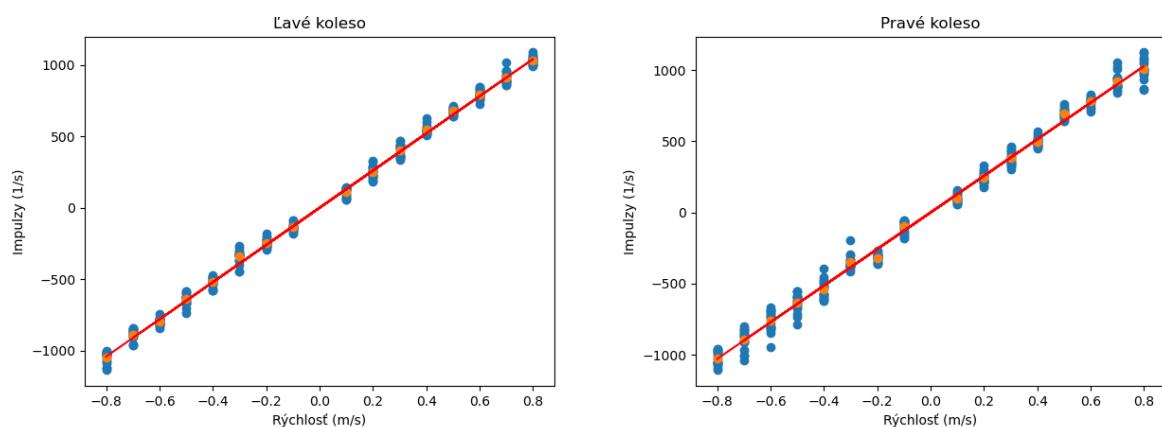


Obr. 5.6: Získanie prevodu z impulzov na rýchlosť v SI jednotkách. $\alpha = 0,8$ a frekvenciou 10Hz.

Najlepšia možná hodnota tohto koeficientu nám vyšla α rovná 0,8. Pri implementácii tohto filtra sme museli myslieť na dôležitú vec. Ked' sa zmenia jednorazovo impulzy na hodnotu 0

a hned' spat', tak nám táto vzorka pokazí výsledok. Musíme preto túto vzorku ignorovať. Ďalšia prekážka, ktorú sme mali pred sebou bola zmena rýchlosťi. Obyčajná implementácia filtra by nám spomalila zmenu vypočítanej rýchlosťi a teda aj veľkú odchýlku v polohe. Tento problém sme opravili prestavením počiatočnej hodnoty filtra na prvú hodnotu po zmene rýchlosťi. Toto riešenie sa ukázalo ako najlepšie so skúšaných riešení.

Problém so zlou počiatočnou hodnotou môžeme vidieť aj na posledom grafe Obr. 5.7. Tento problém sme vyriešili predpočítavaním prvej hodnoty filtra po zmene rýchlosťi. Keďže sme už mali koeficienty lineárnej regresie, tak sme ich využili na predpočítanie počiatočnej hodnoty. Výsledok tohto postupu vidíme na nasledujúcom grafe.



Obr. 5.7: Získanie prevodu z impulzov na rýchlosť v SI jednotkách. $\alpha = 0,8$ a frekvenciou 10Hz a prvou prepočítanou hodnotou.

Záver

V prvej časti bakalárskej prace bolo našou úlohou zoznámiť sa s robotom a neimplementovať ovládač na neho pomocou ROSu. V stave v akom sme ho dostali sme museli opraviť niektoré softvérové chyby na robote. Tie sa týkali hlavne komunikácie s robotom. V tejto práci sme sa dostali do stavu, kedy vieme robotu poslať, akou rýchlosťou sa majú hýbať jednotlive kolesa a on nám dáva spätnú väzbu, že akou rýchlosťou naozaj ide. To ale ešte nefunguje, tak ako má kvôli zašumenému signálu. V druhej časti bakalárskej prace by sme chceli opraviť alebo aspoň zredukovať toto zašumenie. Poprípade nájsť iný spôsob na kontrolovanie pozície robota v priestore.

Zoznam použitej literatúry

1. RICO, Francisco Martín. *A Concise Introduction to Robot Programming with ROS2*. 1. vyd. Chapman a Hall/CRC Press, 2023. ISBN 978-1-003-28962-3.
2. *ROS2 documentation* [online] [cit. 2022-12-23]. Dostupné z : <https://docs.ros.org/en/humble/index.html>.
3. *ROS2 from the Ground Up* [online] [cit. 2022-12-23]. Dostupné z : <https://medium.com/@nullbyte.in/ros2-from-the-ground-up-part-1-an-introduction-to-the-robot-operating-system-4c2065c5e032>.
4. *Changes between ROS 1 and ROS 2* [online] [cit. 2023-01-08]. Dostupné z : <http://design.ros2.org/articles/changes.html>.
5. BC. MAREK PACALAJ, BC. TOMÁŠ KÚTIK, BC. DOMINIK GULA, BC. DÁVID PAVLIČ, BC. DANIEL ĎURKOVIČ. *Mobilný podstavec pre robota*. 2019.
6. *MIO-5272* [online] [cit. 2022-12-26]. Dostupné z : [https://www.mouser.sk/datasheet/2/638/MIO-5272_DS\(01.17.18\)20180118153722-1570123.pdf](https://www.mouser.sk/datasheet/2/638/MIO-5272_DS(01.17.18)20180118153722-1570123.pdf).
7. *MIO-210* [online] [cit. 2022-12-26]. Dostupné z : [https://advownload.advantech.com/productfile/PIS/MIOe-210/Product%20-%20Datasheet/MIOe-210_220_230_110_120_PWR1_DS\(03.26.14\)20140327095019.pdf](https://advownload.advantech.com/productfile/PIS/MIOe-210/Product%20-%20Datasheet/MIOe-210_220_230_110_120_PWR1_DS(03.26.14)20140327095019.pdf).
8. *EPOS2 Positioning Controllers* [online] [cit. 2022-12-26]. Dostupné z : https://www.maxongroup.com/medias/sys_master/root/8831294472222/2018EN-457-458-459-461.pdf.
9. *Encoder MR Type L, 256–1024 CPT, 3 channels, with line driver* [online] [cit. 2022-12-26]. Dostupné z : <https://innodrive.ru/downloads.php?file=/wp-content/uploads/files/maxon/sensor/15032-EN-21-479.pdf>.
10. *Details RE 40 Ø40 mm, Graphite Brushes, 150 Watt* [online] [cit. 2022-12-26]. Dostupné z : <https://www.maxongroup.com/maxon/view/product/motor/dcmotor/re/re40/148867>.
11. *Details Planetary Gearhead GP 42 C Ø42 mm, 3 - 15 Nm, Ceramic Version* [online] [cit. 2022-12-26]. Dostupné z : <https://www.maxongroup.com/maxon/view/product/gear/planetary/gp42/203120>.
12. BC. MAREK PACALAJ, BC. TOMÁŠ KÚTIK, BC. DOMINIK GULA, BC. DÁVID PAVLIČ, BC. DANIEL ĎURKOVIČ. *Dokumentacia k softwaru robota BlackMetal*. 2019.

13. MAXON. *EPOS Command Library: Document ID: rel6806*. 2019.