



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

SIMULAZIONE DI UN DRONE
CONTROLLATO E DI UN ROBOT A TRAZIONE
DIFFERENZIALE A GUIDA AUTONOMA IN
UN MAGAZZINO

SIMULATION OF A CONTROLLED DRONE
AND SELF-DRIVING DIFFERENTIAL
WHEELED ROBOT IN A WAREHOUSE

MARCO AGATENSI

Relatore: *Francesco Tiezzi*

Anno Accademico 2023-2024

Marco Agatensi: *Simulazione di un drone controllato e di un robot a trazione differenziale a guida autonoma in un magazzino*, Corso di Laurea in Informatica, © Anno Accademico 2023-2024

INDICE

1	Introduzione	1
1.1	Obiettivo della tesi e materiale di studio	1
1.2	Metodologia seguita per lo sviluppo del progetto di tesi .	2
1.3	Struttura della tesi	3
1.4	Codice della simulazione	4
2	ROS 2, Ignition e strumenti di supporto	5
2.1	Robot Operating System 2 (ROS 2)	5
2.2	Ignition Gazebo come strumento di simulazione	7
2.3	RViz2: un utile strumento di supporto alla simulazione .	7
3	Preparazione dell'ambiente di simulazione	9
3.1	Magazzino	9
3.2	Configurazione del robot a trazione differenziale: robot_scan	11
3.3	Configurazione del drone a 4 eliche: quadcopter	14
3.4	Configurazione del bridge tra Ignition e ROS 2	15
3.5	Configurazione del robot_state_publisher per RViz2	16
4	La simulazione	19
4.1	Programmazione del drone	20
4.1.1	Action Server	20
4.1.2	Action Client	23
4.2	Guida autonoma	24
4.2.1	Preparazione della mappa statica con slam_toolbox	24
4.2.2	Configurazione e avvio dei nodi nav2	26
4.3	Gestione ed avvio della simulazione	31
4.3.1	Nodo Orchestratore	31
4.3.2	Avvio della simulazione	32
5	Conclusioni e sviluppi futuri	37
5.1	Sviluppi futuri	37
5.2	Considerazioni finali	38
	Bibliografia	39

ELENCO DELLE FIGURE

Figura 1	Logo ROS [21] e ROS 2 Humble [18]	6
Figura 2	Logo Gazebo [9] e Ignition Fortress [11]	7
Figura 3	Logo Rviz [22]	8
Figura 4	Vista del magazzino dall'alto al termine della configurazione	10
Figura 5	Modello del robot Pioneer 2DX	12
Figura 6	Modello del drone X3 UAV Config 5	14
Figura 7	Traiettoria seguita dal drone (arancione) e dal robot (rossa)	19
Figura 8	Struttura del workspace ROS 2	22
Figura 9	Mappa statica del magazzino my_map.pgm	26
Figura 10	Mappa statica del magazzino con costmap visibili	29
Figura 11	Zoom sulla local_costmap	30
Figura 12	Diagramma nav2 [15]	31
Figura 13	UML Sequence Diagram dell'esecuzione della simulazione	33
Figura 14	Robots ad inizio simulazione in Ignition e RViz2	34
Figura 15	Drone a metà percorso prima di volare sopra ad un ostacolo	35
Figura 16	robot_scan a metà percorso con local_costmap ben visibile	36
Figura 17	robot_scan e quadcopter affiancati al termine della simulazione	36

*"Non sapremo mai quanto bene
può fare un semplice sorriso."
— Madre Teresa di Calcutta*

INTRODUZIONE

"La robotica è la disciplina dell'ingegneria che studia e sviluppa metodi che permettono a un robot di eseguire dei compiti specifici riproducendo in modo automatico il lavoro umano" [23].

Siamo ormai sempre più abituati ad interagire con dispositivi meccatronici nella vita di tutti i giorni a partire dai robot taglia-erba o che puliscono le case, passando per le operazioni chirurgiche con utilizzo di tecnologie robotiche fino ad arrivare alle industrie intelligenti con magazzini automatizzati.

Oltre alle discipline come la meccanica e l'elettronica, un ruolo cruciale nella robotica è svolto dall'informatica. Infatti è grazie ad essa che i dispositivi elettro-meccanici possono essere programmati affinché essi siano capaci di svolgere determinati compiti.

Poiché non è facile scrivere applicazioni robotiche, esistono dei framework che agevolano questo compito: uno tra questi è il Robot Operating System (ROS). Esso è un software gratuito ed open source che definisce i componenti, le interfacce ed i tools per costruire applicazioni robotiche avanzate. Molti robot sono composti da attuatori, ovvero componenti che svolgono azioni nel mondo reale, da sensori, che leggono i dati dell'ambiente, e dal sistema di controllo che altro non è che il cervello del modello. ROS e il relativo ambiente di simulazione, oltre a facilitare lo sviluppo e la comunicazione di tali componenti, consentono di sviluppare il software del sistema robotico e di fare testing prima di sperimentare nel mondo reale. Infatti, poiché utilizzare robot fisici può portare al sostenimento di costi dovuti sia ai loro componenti sia agli eventuali danni da essi provocati, è spesso conveniente eseguire prima delle simulazioni in ambienti virtuali.

1.1 OBIETTIVO DELLA TESI E MATERIALE DI STUDIO

L'obiettivo di questa tesi è la comprensione dei concetti alla base del framework ROS 2 per essere in grado di generare una simulazione nella

quale i robot devono eseguire compiti ben definiti. La principale fonte che è stata utilizzata per la comprensione del framework e degli strumenti che esso mette a disposizione consiste nei tutorial forniti dalla pagina ufficiale ROS 2, in particolare ROS 2 Humble Hawksbill [19]. Essi consentono di capire a fondo il funzionamento del sistema partendo dai concetti base, quali ad esempio i nodi ed i metodi di comunicazione tra essi, fino ad arrivare ai concetti più avanzati, quali la gestione delle trasformazioni dei frame e l'integrazione del framework con i simulatori. Attraverso questi tutorial si è quindi in grado di acquisire le conoscenze necessarie per costruire applicazioni robotiche avanzate interagendo con strumenti di simulazione.

Al termine di questo studio è stato poi approfondito il simulatore Ignition Gazebo, nella sua versione LTS Fortress, attraverso i tutorial forniti dalla pagina ufficiale [10].

Per poter raggiungere a pieno gli obiettivi previsti dalla tesi e per una comprensione dei concetti alla base della navigazione autonoma è poi stato studiato ed utilizzato il framework nav2. Per approfondire questo ultimo argomento sono stati analizzati i file nei repository GitHub ros-navigation/navigation2 [20] e ROBOTIS-GIT/turtlebot3 [17], così come la documentazione presente nella pagina ufficiale nav2 [16]. Poiché il linguaggio scelto per la programmazione dei nodi è il C++, è stata inoltre studiata una buona parte del libro "*Programmazione Object-Oriented in C++, Design Pattern e introduzione alle buone pratiche di programmazione*" del prof. Marco Bertini.

Dopo aver studiato il materiale sopra citato è stato quindi definito il progetto di tesi. Questo consiste nella simulazione di un drone controllato mediante nodi ROS 2 e di un robot a trazione differenziale a guida autonoma in un magazzino. La simulazione è composta di due parti: la prima consiste nella navigazione del drone che, a seguito del decollo, segue una traiettoria prefissata al termine della quale atterra sopra ad un blocco. La seconda parte della simulazione consiste nella navigazione autonoma di un robot a trazione differenziale che segue una traiettoria verso la posizione obiettivo che gli è stata comunicata da un nodo ROS 2 subito dopo l'atterraggio del drone.

1.2 METODOLOGIA SEGUITA PER LO SVILUPPO DEL PROGETTO DI TESI

La fase di sperimentazione è cominciata scegliendo i modelli dei robot che sono poi stati utilizzati nella simulazione. Essi sono stati quindi

configurati e testati uno per volta in un ambiente vuoto all'interno del simulatore. Il testing è consistito nell'invio di messaggi sui topic Ignition per controllare la correttezza della configurazione dei robot. Al termine di questa fase è stato poi costruito l'ambiente di simulazione, il magazzino. La seconda fase è relativa alla programmazione dei robot con ROS 2. A seguito di una analisi approfondita dell'esempio turtlebot 3, utile per comprendere le potenzialità ed il funzionamento di nav2, l'attenzione si è poi spostata sulla definizione della mappa statica dell'ambiente simulato e della configurazione dei nodi necessari per implementare l'autonomous driving. Infine è stato poi programmato il drone in modo che fosse in grado di seguire una traiettoria prefissata.

Oltre al simulatore, per avere una miglior comprensione dei dati letti dai sensori, è stato configurato il tool RViz2. All'avvio della simulazione si aprono quindi due schermate, una relativa al simulatore Ignition ed una relativa ad RViz2.

1.3 STRUTTURA DELLA TESI

- Nel Capitolo 2 vengono presentati il framework ROS 2, il simulatore Ignition Gazebo e il tool RViz2. In particolare in tale capitolo sono descritti: i concetti alla base di ROS 2, ovvero le tipologie di comunicazione tra i nodi e la struttura che deve avere un progetto di robotica; le modalità di definizione dei modelli nel simulatore e gli strumenti che esso mette a disposizione per configurarli (plugin); infine viene mostrata l'utilità del tool RViz2 e le funzioni da esso fornite.
- Nel Capitolo 3 viene descritta la fase di definizione dell'ambiente di simulazione, la configurazione dei modelli dei robot, la configurazione del bridge, necessaria per rendere possibile lo scambio dei messaggi tra il sistema ROS 2 e quello Ignition, ed infine viene mostrata la configurazione del nodo `robot_state_publisher` necessario per poter vedere i modelli dei robot anche all'interno di RViz2.
- Nel Capitolo 4 viene descritta la parte relativa ai nodi ROS 2 che definiscono il comportamento dei robot nella simulazione. In particolare vengono mostrati: le scelte relative alla progettazione dei nodi ROS 2 utilizzati per controllare il drone; la fase di scansione

della mappa statica del magazzino; i nodi nav2 utilizzati e le funzioni da loro svolte e la descrizione del nodo orchestratore che gestisce l'ordine degli eventi nella simulazione e come questa deve essere avviata.

- Nel Capitolo 5 viene brevemente descritta la simulazione e vengono indicati possibili sviluppi futuri del progetto di robotica che è stato implementato.

1.4 CODICE DELLA SIMULAZIONE

I file relativi al codice della simulazione possono essere visionati su GitHub nel repository `marcoaga02/ros2_ignition_thesis` al seguente link https://github.com/marcoaga02/ros2_ignition_thesis [14].

2

ROS 2, IGNITION E STRUMENTI DI SUPPORTO

"ROS 2 è un middleware basato su un meccanismo di pubblicazione/sottoscrizione anonimo e fortemente tipizzato che consente lo scambio di messaggi tra diversi processi. Al centro di ogni sistema ROS 2 c'è il ROS graph. Esso si riferisce al network di nodi in un sistema ROS e alle connessioni tra loro attraverso le quali comunicano" [1].

2.1 ROBOT OPERATING SYSTEM 2 (ROS 2)

ROS 2 (Robot Operating System, Figura 1) è un insieme di librerie software e di strumenti per costruire applicazioni robotiche [25, 27]. Consiste in un network di nodi che comunicano tra loro per raggiungere determinati obiettivi.

Data la sua architettura distribuita, ROS 2 mette a disposizioni tre tipologie di comunicazione tra i nodi del sistema:

- **topic**: agiscono come bus tra nodi per lo scambio dei messaggi. Sono basati sul meccanismo publish-subscribe in cui i publishers inviano i messaggi attraverso topic ed i subscribers si iscrivono ad uno o più topic per leggerli;
- **service**: si basano su un meccanismo di tipo client-server nel quale è presente un solo server che fornisce un servizio ad uno o più client che ne fanno richiesta. Il client riceve una risposta solo a seguito di una richiesta di servizio al server (call-and-response);
- **action**: come per i servizi, anch'esse sono basate su un meccanismo di tipo client-server ma le actions sono pensate per comunicazioni di lunga durata. Infatti, oltre ai messaggi di request (goal) e di result, è presente anche il messaggio di feedback che ha lo scopo di fornire lo stato attuale dell'esecuzione del "servizio" richiesto dal client.

Nel lavoro di tesi è stata utilizzata prevalentemente la comunicazione basata su topic e, solo in alcuni casi, quella basata su action.

Per vedere il grafo che rappresenta la rete di nodi ROS 2 attivi nel sistema è possibile utilizzare il tool `rqt_graph` che può essere avviato da terminale Linux eseguendo il comando `rqt_graph`.

I progetti necessitano di una struttura ben precisa composta da un workspace al cui interno sono presenti package ROS 2. Possono esserci più package in uno stesso workspace ma non possono essere annidati tra loro (pacchetti dentro pacchetti). I package possono essere creati con il comando `ros2 pkg create --build-type ament_cmake --license Apache-2.0 <package_name>` che, oltre a creare la cartella "`<package_name>`", crea anche le sottocartelle `src` e `include/<package_name>` ed i file `CMakeLists.txt` e `package.xml`.

Per effettuare la build del progetto viene utilizzato lo strumento `colcon` con il comando da terminale Linux `colcon build`. Per poter costruire il progetto sono necessari i file `CMakeLists.txt` e `package.xml` [2]: nel primo sono specificati i pacchetti necessari ai nodi ROS 2 nel comando "`find_package(nome_pacchetto REQUIRED)`", le librerie che devono essere create, gli eseguibili associati ai nodi che devono essere creati e le cartelle del package che devono essere rese condivise; nel secondo invece devono essere dichiarate tutte le dipendenze dei nodi.

I pacchetti forniti con l'installazione di ROS 2 fanno parte del così detto `underlay`. Per poter utilizzarne i file eseguibili con il comando `ros2 run <nome_pkg> <nome_eseguibile>`, deve essere fatto il "sourcing" dell'`underlay` con il comando `source /opt/ros/humble/setup.bash` in ogni nuovo terminale (è possibile aggiungerlo al file `.bashrc` per non dover eseguire il comando ogni volta) [5]. Invece, i pacchetti che fanno parte dei workspace da noi definiti costituiscono il così detto `overlay`. Esso è quindi uno spazio di lavoro secondario in cui si possono aggiungere nuovi pacchetti senza interferire con il workspace ROS 2 che stiamo estendendo. Anche l'`overlay` necessita di essere caricato: dobbiamo quindi entrare nel folder che rappresenta il workspace ed al suo interno eseguire il comando `source /install/setup.bash`.



Figura 1: Logo ROS [21] e ROS 2 Humble [18]

2.2 IGNITION GAZEBO COME STRUMENTO DI SIMULAZIONE

Poiché utilizzare robot nel mondo reale può portare al sostenimento di costi dovuti sia ai componenti dei robot sia agli eventuali danni da essi provocati, è spesso conveniente eseguire prima delle simulazioni in ambienti virtuali.

Qui entrano in gioco i simulatori, un cui esempio è Ignition Gazebo (vedi Figura 2). Esso consente di definire ambienti virtuali dotati di proprietà fisiche quanto più simili a quelle del mondo reale ed un'infinità di modelli da rappresentare al suo interno. Ignition, per descrivere i modelli dei robot ed il loro ambiente di lavoro, utilizza un particolare formato XML, il Simulation Description Format (SDF).

Per rendere possibile l'interazione tra i modelli e l'ambiente di simulazione, vengono messi a disposizione una serie di plugin che altro non sono che frammenti di codice compilato che possono essere collegati ai modelli per ottenere determinati comportamenti, quali ad esempio la sottoscrizione dei giunti di un robot ad un topic così da renderne possibili i movimenti attraverso la pubblicazione di messaggi.

Ignition è dotato di un'interfaccia grafica per mezzo della quale è possibile interagire con l'ambiente di simulazione visualizzato. Inoltre, sono presenti anche una serie di plugin da interfaccia grafica, i `plugin-gui`, che possono essere aggiunti manualmente da Ignition GUI.

Poiché definire modelli SDF da zero è spesso complicato, è possibile trovarne molti già pronti su Ignition Fuel [12].



Figura 2: Logo Gazebo [9] e Ignition Fortress [11]

2.3 RVIZ2: UN UTILE STRUMENTO DI SUPPORTO ALLA SIMULAZIONE

I modelli dei robot, siano essi virtuali o reali, sono spesso dotati di un insieme di sensori. Leggere i dati che essi forniscono è quindi essenziale

per poter conoscere lo stato attuale dei robot e poter quindi definire le azioni da compiere. Per rendere più chiara ed immediata la comprensione di queste informazioni è spesso necessaria una visualizzazione grafica di esse ed è qui che entra in gioco RViz2 (vedi Figura 3).

Questo tool, che può essere configurato tramite file con estensione .rviz, è dotato di un'interfaccia grafica intuitiva e facilmente configurabile. Attraverso essa è infatti possibile scegliere le tipologie di informazioni che vogliamo visualizzare tra cui, per citarne alcune, i dati dei lidar, le trasformazioni dei frame e anche i modelli dei robot stessi. Per far ciò, ci sono molte impostazioni predefinite che necessitano di conoscere solamente i topic su cui leggere i dati che verranno mostrati sullo schermo.

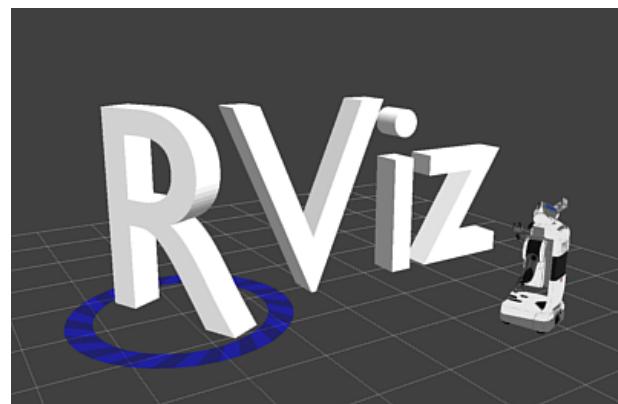


Figura 3: Logo Rviz [22]

3

PREPARAZIONE DELL'AMBIENTE DI SIMULAZIONE

Per lo sviluppo di questa tesi è stato utilizzato il simulatore Ignition Fortress (vedi Sezione 2.2) ed in esso è stato modellato un magazzino (vedi Sezione 3.1) con al suo interno un robot a trazione differenziale (vedi Sezione 3.2) ed un drone a quattro eliche (vedi Sezione 3.3).

3.1 MAGAZZINO

Come sopra anticipato, l'ambiente di lavoro dei robot che è stato model-lato è un magazzino. Come base per la definizione dell'environment è stato utilizzato l'`Empty World`, fornito con l'installazione di Gazebo [35], che descrive un mondo vuoto, con un piano terra ed una luce simile a quella del sole, dotato di proprietà fisiche fornite dal plugin `Physics` di Ignition.

Poiché non è sempre facile descrivere robot ed ambienti di simulazione, Ignition fornisce una moltitudine di file SDF in Ignition Fuel, che è quindi stato utilizzato per scaricare i modelli di base utilizzati nella simulazione. Per rappresentare il magazzino è stato incluso nel world il modello `Depot` fornito da OpenRobotics [31]. Esso consiste in un magazzino con al suo interno degli scaffali ed una serie di decorazioni proprie dell'ambiente rappresentato. Data l'impossibilità di spostare gli oggetti, è stato modificato il file SDF del modello mediante la rimozione di gran parte delle decorazioni e degli oggetti al suo interno mantenendone solo le mura, il pavimento e parte del tetto.

Le mura non erano però dotate di proprietà di collisione ed era quindi possibile attraversarle. Dopo la misurazione in Gazebo delle dimensioni di queste, sono quindi state aggiunte tali proprietà al magazzino per rendere veritiera la simulazione (codice al Listato 1). Dopo queste modifiche il magazzino appariva vuoto. Sono stati quindi aggiunti degli scaffali e dei pancali alla simulazione creando così un percorso che verrà poi attraversato dai robot. I pancali e gli scaffali sono i modelli `pallet_-`

Listato 1: Parte di codice SDF per le proprietà di collisione

```
<collision name="wall_collision_left">
  <geometry>
    <box>
      <size>30.36 0.2 9.30</size>
    </box>
  </geometry>
</collision>
```

`box_mobile`, `shelf` e `shelf_big` messi a disposizione su Ignition Fuel da MovAi [28, 29, 30].

Il model `Depot` prevedeva che il magazzino avesse tre ingressi della stessa dimensione. Poiché lo scopo della simulazione era di confinare i robot al suo interno, è stato definito un modello per la porta del garage, `garage-door`, che è stato quindi incluso nell'ambiente di simulazione (codice al Listato 2).

Alla fine della configurazione il magazzino, visto dall'alto, appare come in Figura 4.

Ignition fornisce una serie di plugin, sia da interfaccia grafica (`plugin-gui`) sia da integrare nel codice SDF, che forniscono una serie di funzionalità necessarie per poter rendere la simulazione interattiva. Per far sì che i sensori fossero in grado di leggere i dati, nel file SDF del magazzino è stato configurato il plugin `ignition::gazebo::systems::Sensors`.

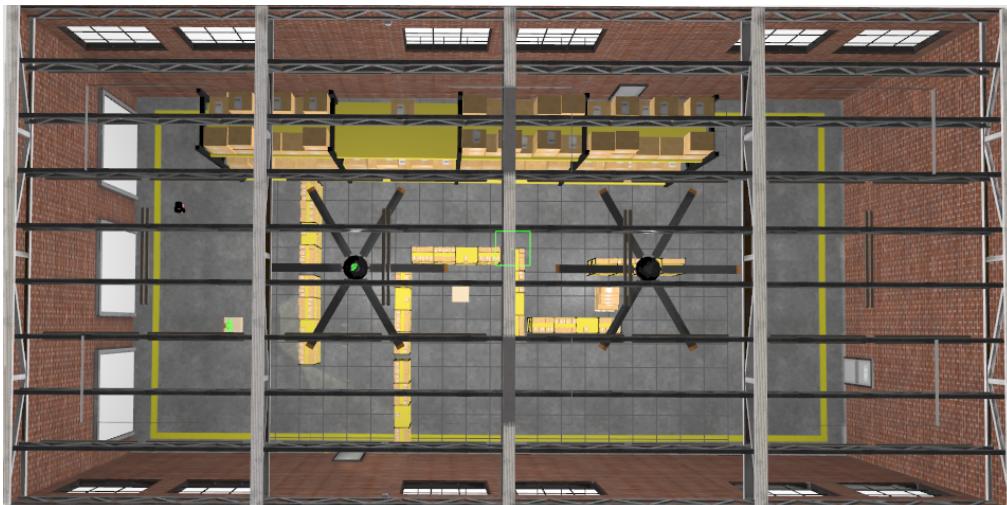


Figura 4: Vista del magazzino dall'alto al termine della configurazione

Listato 2: Codice SDF del modello garage_door

```

<?xml version="1.0"?>
<sdf version="1.8">
  <model name = "garage_door">
    <static>true</static>
    <link name = "door_link">
      <visual name = "door_visual">
        <geometry>
          <box>
            <size> 0.15 3.30 3.25 </size>
          </box>
        </geometry>
        <material>
          <ambient>0.3 0.3 0.3 1</ambient>
          <diffuse>0.6 0.6 0.6 1</diffuse>
          <specular>0.9 0.9 0.9 1</specular>
          <emissive>0 0 0 1</emissive>
        </material>
      </visual>
    </link>
  </model>
</sdf>

```

3.2 CONFIGURAZIONE DEL ROBOT A TRAZIONE DIFFERENZIALE: ROBOT_SCAN

Dopo aver preparato l’ambiente è quindi necessario configurare i robot che dovranno eseguire dei task al suo interno. Tra questi è stato scelto un modello di macchina a trazione differenziale, ovvero dotato di due ruote ognuna delle quali con trazione autonoma. Il robot per sterzare invia quindi comandi di velocità opposti sui giunti delle due ruote in modo che esso possa ruotare su se stesso.

Il modello scelto è il Pioneer 2DX di OpenRobotics [32], messo a disposizione su Ignition Fuel, che è stato incluso nella simulazione col nome `robot_scan` (vedi Figura 5). Dopo un’analisi del contenuto del file SDF sono state apportate alcune modifiche necessarie per rendere la macchina in grado di compiere task durante la simulazione.

Come già anticipato, Ignition fornisce una serie di plugin che devono essere utilizzati per poter comandare il robot. Tra questi, uno che è

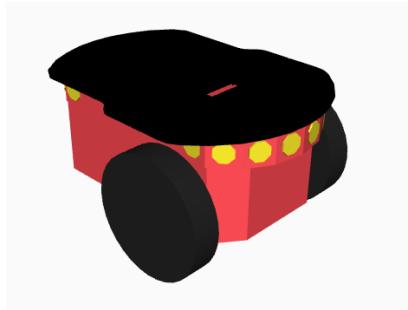


Figura 5: Modello del robot Pioneer 2DX

stato integrato e configurato sulla base delle proprietà strutturali della macchina è il plugin `ignition::gazebo::systems::DiffDrive` (codice al Listato 3).

Listato 3: Configurazione del plugin DiffDrive

```
<plugin
    filename="ignition-gazebo-diff-drive-system" name=""
    ↪ ignition::gazebo::systems::DiffDrive">
    <left_joint>left_wheel_joint</left_joint>
    <right_joint>right_wheel_joint</right_joint>
    <wheel_separation>0.34</wheel_separation>
    <wheel_radius>0.11</wheel_radius>
    <odom_publish_frequency>5</odom_publish_frequency>
    <max_linear_acceleration>1.5</max_linear_acceleration>
    <min_linear_acceleration>-1.5</min_linear_acceleration>
    <max_angular_acceleration>1.6</max_angular_acceleration>
    <min_angular_acceleration>-1.6</min_angular_acceleration>
    <max_linear_velocity>0.4</max_linear_velocity>
    <min_linear_velocity>-0.4</min_linear_velocity>
    <max_angular_velocity>1</max_angular_velocity>
    <min_angular_velocity>-1</min_angular_velocity>
</plugin>
```

Questo plugin rende il nostro modello un subscriber del topic `/model/robot_scan/cmd_vel`. Su di esso verranno inviati messaggi di tipo `ignition.msgs.Twist` con i quali è possibile far muovere i giunti indicati nei tag XML `<left_joint>` e `<right_joint>` in modo che essi facciano muovere le ruote a cui sono connessi e di conseguenza la macchina. Questo robot è stato configurato per poter essere controllato tramite

le frecce della tastiera: le motivazioni legate a questa scelta saranno esposte nella Sottosezione 4.2.1. Per ottenere il comportamento descritto è stato inoltre integrato, nel file SDF del modello, il plugin `ignition::gazebo::systems::TriggeredPublisher` che consente di inviare un messaggio su un topic a seguito di un determinato evento. Nel nostro caso l'evento consiste nella ricezione di un messaggio di tipo `ignition.msgs.Int32`, che rappresenta il numero intero associato al tasto premuto sulla tastiera, sul topic `/keyboard/keypress`.

Non è automatica la sottoscrizione del modello a questo topic, ma avviene da interfaccia grafica Ignition GUI attivando il plugin-gui Key Publisher. Per poter conoscere i numeri interi associati ai tasti di interesse, mentre Ignition GUI è in esecuzione e con attivo il plugin sopra definito, dobbiamo eseguire su terminale Linux il comando `ign topic -e -t /keyboard/keypress` che mostra sul terminale, grazie al flag `-e`, i messaggi pubblicati sul topic specificato dopo il flag `-t`, ovvero `/keyboard/keypress`. A questo punto, nella finestra di Gazebo premiamo i tasti di interesse, nel nostro caso le 4 frecce e il tasto s, usato per azzerare la velocità del robot, e vediamo sul terminale i numeri interi associati così da poter configurare il plugin `TriggeredPublisher` (esempio di configurazione del plugin, che associa alla "up arrow key" l'azione di movimento in avanti ad una velocità lineare di 0.4 m/s lungo l'asse x del robot, al Listato 4).

Listato 4: Configurazione del plugin `TriggeredPublisher`

```
<!-- Moving Forward-->
<plugin
    filename="libignition-gazebo-triggered-publisher-system.so"
    name="ignition::gazebo::systems::TriggeredPublisher">
    <input type="ignition.msgs.Int32" topic="/keyboard/keypress">
        <match field="data">16777235</match>
    </input>
    <output type="ignition.msgs.Twist" topic="/model/robot_scan/
        ↪ cmd_vel">
        linear: {x: 0.4}, angular: {z: 0.0}
    </output>
</plugin>
```

Per poter raccogliere dati dall'ambiente durante la simulazione, i robot necessitano di sensori. Al modello `robot_scan`, poiché ne era sprovvisto, è stato aggiunto un sensore Lidar di tipo `gpu_lidar`. Esso consente alla

macchina di individuare gli ostacoli permettendogli quindi di avere una panoramica dell'ambiente che la circonda.

Il lidar deve essere configurato nel file `pioneer2dx/model.sdf` in modo che durante la scansione della mappa alla Sottosezione 4.2.1 abbia gli "horizontal samples" impostati a 1200 e il "range max" impostato a 15; durante la navigazione autonoma alla Sottosezione 4.2.2, gli "horizontal samples" devono essere impostati a 450 e il "range max" a 5. Questo poiché il lidar deve essere configurato in modo che sia più potente durante la scansione della mappa per acquisire dati precisi.

Il link del sensore è poi stato fissato al telaio della macchina tramite un `fixed joint`.

3.3 CONFIGURAZIONE DEL DRONE A 4 ELICHE: QUADCOPTER

L'altro robot che è stato scelto è un modello di drone a 4 eliche.

Il modello è il X3 UAV Config 5 di OpenRobotics [33], messo a disposizione su Ignition Fuel, che è stato incluso nella simulazione col nome `quadcopter` (vedi Figura 6). Dopo un'analisi del contenuto del file SDF sono state apportate alcune modifiche necessarie per rendere il drone in grado di compiere task durante la simulazione.

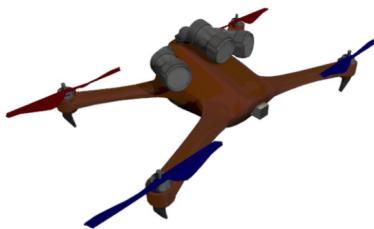


Figura 6: Modello del drone X3 UAV Config 5

Esso era dotato di una moltitudine di sensori che non erano necessari e che sono stati quindi rimossi per alleggerire il carico computazionale della simulazione. Tra questi, l'unico che è stato utilizzato è il Lidar di tipo `gpu_ray`, con nome `bottom_laser`, che raccoglie un solo campione attraverso un raggio puntato verticalmente verso il suolo. Questo consente così di ottenere l'altitudine in metri del drone (lungo l'asse z).

I 4 motori sono rappresentati nel file SDF come `joint` di tipo `revolute`, ovvero capaci di ruotare attorno ad un asse, il tal caso l'asse z. Per far sì che `quadcopter` possa ricevere comandi di velocità sono necessari due

plugin Ignition le cui configurazioni sono state selezionate e modificate da una Demo disponibile sul repository GitHub gazebosim/gz-sim [34]. Le modifiche apportate a tali configurazioni riguardano principalmente i nomi dei links, dei joints, dei topics e del namespace.

I plugin sopra citati sono:

- `MulticopterMotorModel`: ne deve essere configurato uno per ogni motore, per un totale di 4. Tra i parametri necessari ci sono il nome del link e del giunto del motore, la direzione di rotazione ed il numero del motore.
- `MulticopterVelocityControl`: ne deve essere configurato uno solo e necessita di conoscere i nomi dei giunti dei motori, il namespace del robot (`/quadcopter`) ed il topic su cui leggere i messaggi di `Twist`, nel nostro caso `command/cmd_vel`.

Con questa configurazione, è quindi possibile far muovere il drone inviando messaggi di tipo `ignition.msgs.Twist` sul topic `/quadcopter/-command/cmd_vel`.

3.4 CONFIGURAZIONE DEL BRIDGE TRA IGNITION E ROS 2

Dopo aver configurato i robot ed essere quindi a conoscenza dei topic di cui essi sono publisher o subscriber, è necessaria una struttura che consenta il transito dei messaggi tra il sistema Ignition Transport e quello ROS 2. Infatti, i topic Ignition non sono direttamente accessibili in ROS 2 e viceversa, perché i tipi dei messaggi supportati dai due sono differenti. Poiché non tutti le tipologie di messaggi ROS 2 hanno una diretta corrispondenza in Ignition, è necessario affidarsi alla tabella fornita su GitHub al seguente link https://github.com/gazebosim/ros_gz/blob/ros2/ros_gz_bridge/README.md [3] per individuare sia i messaggi supportati dal bridge sia le corrispondenze che questi hanno tra i due sistemi (è importante sottolineare che nella colonna "Gazebo Transport Type" della tabella, il prefisso "gz" dei messaggi, utilizzato in Gazebo Classic, deve essere sostituito col prefisso "ignition" usato in Ignition Transport).

Il bridge è stato quindi configurato mediante file `yml` per mappare i topic Ignition in topic ROS 2 (codice al Listato 5).

Il file eseguibile del bridge ha nome `parameter_bridge` ed è contenuto nel package `ros_gz_bridge` di ROS 2. È possibile eseguire il file direttamente da terminale Linux oppure da launch file Python, che è stata l'opzione scelta in questa tesi.

Listato 5: Parte del file yaml di configurazione di ros_gz_bridge

```

- ros_topic_name: "/robot_scan/cmd_vel"
  gz_topic_name: "/model/robot_scan/cmd_vel"
  ros_type_name: "geometry_msgs/msg/Twist"
  gz_type_name: "ignition.msgs.Twist"
  direction: ROS_TO_GZ

- ros_topic_name: "/robot_scan/odometry"
  gz_topic_name: "/model/robot_scan/odometry"
  ros_type_name: "nav_msgs/msg/Odometry"
  gz_type_name: "ignition.msgs.Odometry"
  direction: GZ_TO_ROS

- ros_topic_name: "/robot_scan/scan"
  gz_topic_name: "/model/robot_scan/scan"
  ros_type_name: "sensor_msgs/msg/LaserScan"
  gz_type_name: "ignition.msgs.LaserScan"
  direction: GZ_TO_ROS

```

3.5 CONFIGURAZIONE DEL ROBOT_STATE_PUBLISHER PER RVIZ2

Molte volte diventa necessario visualizzare graficamente le trasformazioni dei frame dei robot, la loro odometria, i progressi nella scansione di una mappa oppure i dati provenienti dai vari sensori. Qui entra in gioco il tool RViz2 (vedi Sezione 2.3) che consente di visualizzare queste informazioni.

A differenza delle informazioni sopra citate, per poter osservare il modello del robot in RViz2 non basta agire da GUI ma è necessario configurare il nodo `robot_state_publisher` [6]. Quest'ultimo per essere eseguito necessita di una descrizione del modello del robot. Nativamente ROS 2 supporta la descrizione URDF dei modelli ma, poiché in questa tesi è stato usato il formato SDF supportato da Ignition, è necessario includere nel file `package.xml`, tra le dipendenze del pacchetto, anche il package `sdformat_urdf`. Infatti, il parametro `robot_description` del nodo riceve una stringa contenente la descrizione completa del robot che è stata letta dal file SDF. Questa è quindi utilizzata dal `robot_state_publisher` per calcolare e pubblicare le trasformazioni 3D dei giunti del robot. Quando ROS 2 cerca di interpretare il contenuto di `robot_description` avvia il plugin `sdformat_urdf` che converte SDF in un formato URDF compatibile.

le.

Come per il bridge, anche in questo caso il `robot_state_publisher` viene eseguito da launch file Python. Nel file di lancio finale, quello di avvio dell'intera simulazione (vedi Sottosezione 4.3.2), sono stati configurati due nodi `robot_state_publisher`, uno per `robot_scan` ed uno per `quadcopter`, in modo che i robot siano visibili entrambi su Rviz2 e su Gazebo.

4

LA SIMULAZIONE

La simulazione oggetto di questa tesi è suddivisibile in due parti: la prima utilizza il drone quadcopter opportunamente programmato per mezzo di nodi ROS 2 per seguire un determinato percorso all'interno del magazzino per poi posizionarsi sopra un parallelepipedo; la seconda, che comincia dopo l'atterraggio di quadcopter, utilizza invece il robot `robot_scan` e i nodi `nav2` di ROS 2 per implementare un sistema di autonomous driving. Le traiettorie seguite dai due robot all'interno del magazzino sono mostrate nella mappa in Figura 7.

Per la tesi è stato utilizzato il sistema operativo Ubuntu 22.04 jammy, ROS 2 nella sua versione LTS Humble [13] ed Ignition nella versione LTS Fortress [8].

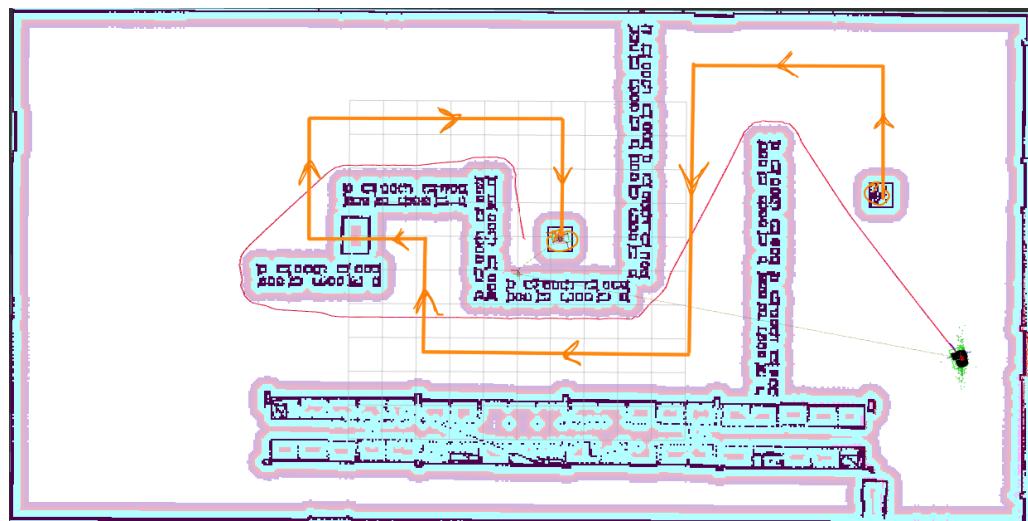


Figura 7: Traiettoria seguita dal drone (arancione) e dal robot (rossa)

4.1 PROGRAMMAZIONE DEL DRONE

Il drone quadcopter, per come è stato configurato (vedi Sezione 3.3), è subscriber del topic Ignition /quadcopter/command/cmd_vel che è mappato tramite il bridge sul topic ROS 2 /quadcopter/cmd_vel. Su quest'ultimo vengono inviati messaggi di tipo geometry_msgs/msg/Twist.

Per le comunicazioni di lunga durata, ROS 2 mette a disposizione le action (vedi Sezione 2.1). Poiché i movimenti del drone tra i waypoints necessitano di abbastanza tempo, questo meccanismo è stato adottato per controllare il robot quadcopter.

Il linguaggio di programmazione usato per la scrittura dei nodi è il C++ e sono state utilizzate le librerie ROS 2 necessarie. Sono stati scritti due file C++ e la dichiarazione delle classi e dei metodi sono in file header.

I nodi ROS 2 si trovano nel package `my_sim_tesi_ros2_nodes` all'interno della sua sottocartella `src`. Gli header sono nello stesso package ma nella sottocartella `include` (struttura del workspace in Figura 8).

4.1.1 Action Server

Il file header nel quale è dichiarata la classe server `PoseControlActionServer` è `pose_control_action_server.hpp`; la sua implementazione è fornita dal file `pose_control_action_server.cpp`.

Il costruttore crea il server della action `drone_pose_control` con il metodo `rclcpp_action::create_server<PoseControl>`. La action `DronePoseControl` è stata definita nel package ROS 2 `my_sim_tesi_ros2_interfaces` (codice al Listato 6). Il costruttore inizializza anche i subscribers dei topic relativi all'odometria ed all'altimetro così come il publisher del topic `/quadcopter/cmd_vel`. Quando un nuovo dato di odometria viene pubblicato sul topic, viene effettuata la callback al metodo `odometry_topic_callback` che aggiorna gli attributi della classe in accordo con i nuovi valori odometrici ricevuti. Similmente a quanto sopra descritto, alla ricezione di un messaggio sul topic `/quadcopter/altimeter`, viene eseguita una callback che aggiorna il valore della variabile `current_altitude_` con il nuovo valore ricevuto.

Alla creazione del server vengono effettuati tre `std::bind`. Questi sono necessari affinché esso esegua i task alla ricezione di un nuovo goal. Quando il client invia un goal alla action `drone_pose_control`, il server invoca una callback a `handle_goal`. In caso di accettazione del goal lo rende noto al client per poi invocare la callback al metodo `handle_accepted`. Quest'ultimo avvia un nuovo thread per eseguire i task (funzione

Listato 6: File DronePoseControl.action che definisce la action DronePoseControl

```

# Request
float64 target_x
float64 target_y
float64 target_yaw
float64 target_altitude
---

# Result
float64 res_x
float64 res_y
float64 res_yaw
float64 res_altitude
---

# Feedback
float64 current_x
float64 current_y
float64 current_yaw
float64 current_altitude

```

execute) evitando così di bloccare il thread principale durante l'esecuzione.

Ad ogni nuovo goal ricevuto il server corregge sempre l'altitudine del drone e poi esegue una sola tra le azioni di rotazione, movimento lineare lungo l'asse x o movimento lineare lungo l'asse y (assi del sistema di riferimento solidale con l'ambiente di simulazione e rispetto al quale il quadcopter si muove). Quindi, per come è stato programmato il server, esso deve ricevere dal client solo goal che differiscono dal precedente in altitudine ed al più uno tra i parametri target_x, target_y, target_yaw. Esso esegue quindi il metodo `moveToGoal` che pubblica sul topic `/quadcopter/cmd_vel` i messaggi di `Twist` necessari per raggiungere la goal pose. Per rendere fluido il movimento del drone, quando esso si avvicina alla posizione obiettivo la velocità viene ridotta chiamando la funzione `reduceVelocity`.

Infine, in accordo con quanto previsto dalle action ROS 2, durante l'esecuzione della funzione `execute` vengono pubblicati i feedback al client: `goal_handle->publish_feedback(feedback)`.

```

~/tesi_ws
├── build/
├── install/
├── log/
└── src/
    ├── my_sim_tesi_bringup/
    │   ├── config/
    │   │   ├── final_simulation_config.rviz
    │   │   ├── map_scan_config.rviz
    │   │   ├── nav2_param_config.yaml
    │   │   ├── ros_gz_bridge_final_config.yaml
    │   │   ├── ros_gz_bridge_scan_config.yaml
    │   │   └── slam_toolbox_config.yaml
    │   ├── files/
    │   │   └── coordinates_goal.txt
    │   ├── launch/
    │   │   ├── my_ros2_nodes.launch.py
    │   │   ├── my_sim_final.launch.py
    │   │   ├── my_sim_map_scan.launch.py
    │   │   └── nav2.launch.py
    │   ├── map/
    │   │   ├── my_map.pgm
    │   │   └── my_map.yaml
    │   ├── CMakeLists.txt
    │   └── package.xml
    ├── my_sim_tesi_gazebo/
    │   ├── models/
    │   │   └── all folders containing meshes and file model.sdf / model.config
    │   ├── worlds/
    │   │   └── my_world.sdf
    │   ├── CMakeLists.txt
    │   └── package.xml
    ├── my_sim_tesi_ros2_interfaces/
    │   ├── action/
    │   │   └── DronePoseControl.action
    │   ├── CMakeLists.txt
    │   └── package.xml
    └── my_sim_tesi_ros2_nodes/
        ├── include/
        │   └── my_sim_tesi_ros2_nodes/
        │       ├── orchestrator_node.hpp
        │       ├── pose_control_action_client.hpp
        │       └── pose_control_action_server.hpp
        └── src/
            ├── orchestrator_node.cpp
            ├── pose_control_action_client.cpp
            ├── pose_control_action_server.cpp
            ├── CMakeLists.txt
            └── package.xml

```

Figura 8: Struttura del workspace ROS 2

4.1.2 Action Client

Poiché il meccanismo delle action è di tipo `client-server`, è necessario definire anche il client.

Il file header nel quale è dichiarata la classe client `PoseControlActionClient` è `pose_control_action_client.hpp`; la sua implementazione è fornita dal file `pose_control_action_client.cpp`.

Il nodo ROS 2 `pose_control_action_client` dichiara il parametro `file_path`: se non viene passato alcun percorso a file viene sollevato un `RCLCPP_ERROR(this->get_logger(), "File path not provided.")` che blocca la creazione del nodo.

Il file che viene fornito come parametro è `coordinates_goal.txt` che contiene le posizioni obiettivo del drone, una per ogni riga, nel seguente formato: `target_x target_y target_yaw target_altitude`.

Il file viene quindi aperto, letto ed i dati al suo interno vengono memorizzati nel vettore di quartuple `coordinates_vector_` che viene scandito con l'iterator `current_goal_`. Vengono poi inizializzati il client della action `drone_pose_control`, con la funzione `rclcpp_action::create_client<PoseControl>`, il subscriber del topic `/quadcopter/start_navigation` ed anche il publisher del topic `/quadcopter/end_navigation`.

Quando viene ricevuto un messaggio sul topic cui il nodo è sottoscritto (vedi Sottosezione 4.3.1 per informazioni legate al publisher), viene invocata la callback al metodo `drone_start_callback` che controlla se la stringa ricevuta è `"start_drone_navigation"`. Se questa è la stringa ricevuta, viene avviata la navigazione del quadcopter inizializzando un timer che, ogni 500 millisecondi, effettua una callback al metodo `send_goal`. Per evitare di inviare richieste al server prima di aver terminato il goal precedente viene utilizzata la variabile booleana `is_goal_active_`. Se questa ha valore `true`, viene effettuato immediatamente un `return`; altrimenti, il server è libero: viene quindi inviata la richiesta relativa alla nuova posizione da raggiungere e la variabile booleana viene impostata con valore `true`. Quando l'iteratore ha raggiunto la fine del vettore, il timer viene eliminato per evitare inutili callback ed il nodo pubblica sul topic `/quadcopter/end_navigation` un messaggio contenente la stringa `"end_drone_navigation"` (vedi Sottosezione 4.3.1 per maggiori informazioni).

Durante l'esecuzione di un goal il server invia messaggi di feedback che il client provvede a stampare su console per mostrare all'utente la `"current position"` del drone, così come ne stampa la posizione finale alla ricezione del messaggio di risposta inviato dal server al raggiungimento

dell’obiettivo.

4.2 GUIDA AUTONOMA

Come anticipato all’inizio di questo capitolo, la seconda parte della simulazione coinvolge un robot a trazione differenziale a guida autonoma.

4.2.1 *Preparazione della mappa statica con slam_toolbox*

Per poter configurare un robot a guida autonoma, usando i package ROS 2 forniti dal framework nav2, è necessaria una mappa statica che riproduca l’ambiente in cui il robot dovrà muoversi. Questa mappa consiste di due file: `map.yaml` e `map.pgm`. Il primo è composto principalmente da 4 parametri:

- `image`: path assoluto del file `map.pgm`;
- `mode`: che è stato impostato col valore `trinary` ad indicare che la mappa è composta di tre possibili stati (occupato, libero, sconosciuto);
- `resolution`: indica l’occupazione in metri di ogni pixel della mappa;
- `origin`: indica le coordinate dell’origine della mappa espresso in metri. È una terna `[x, y, yaw]`, dove il terzo valore rappresenta la rotazione della mappa in radianti.

Se il file `map.pgm` viene aperto con un editor grafico è possibile vedere la mappa scansionata; se invece viene aperto con un editor di testo, possiamo allora vedere il numero di pixel che la mappa possiede sia in larghezza che in altezza. Queste informazioni, in aggiunta a quella sulla risoluzione specificata nel file `map.yaml`, ci permettono di calcolare, mediante una semplice operazione di prodotto, la larghezza e l’altezza in metri della mappa (operazione necessaria per configurare correttamente le costmap nav2, vedi Sottosezione 4.2.2).

Per poter ottenere questi file sono necessarie delle operazioni preliminari, prima tra queste la configurazione e l’avvio del nodo `slam_toolbox`, eseguibile `async_slam_toolbox_node` del ROS 2 package `slam_toolbox` [24, 37]. Esso consente di generare una mappa dell’ambiente sulla base dei dati letti dal sensore Lidar del robot. Per acquisire questi dati è stato utilizzato `robot_scan` che era stato configurato in modo da poter

essere guidato tramite le frecce della testiera (vedi Sezione 3.2). È infatti importante poter direzionare manualmente il robot in ogni punto del magazzino affinché i sensori possano raccogliere tutti i dati necessari. Il nodo ROS 2 è stato avviato da launch file Python ed è stato configurato con il file `slam_toolbox_config.yaml` (il cui codice è Listato 7). Per avviare il simulatore, RViz2 e tutti i nodi necessari per la scansione della mappa, è stato utilizzato il file di lancio `my_sim_map_scan.launch.py`. Prima di lanciarlo, con il comando `ros2 launch my_sim_tesi Bringup my_sim_map_scan.launch.py`, devono essere eseguite le operazioni preliminari indicate nella Sottosezione 4.3.2.

Listato 7: File yaml di configurazione di `slam_toolbox`

```
slam_toolbox:
  ros__parameters:
    scan_topic: '/robot_scan/scan'
    odom_topic: '/robot_scan/odometry'
    map_frame: 'map'
    odom_frame: 'robot_scan/odom'
    base_frame: 'robot_scan/base_footprint'
```

Per poter visualizzare la parte di mappa di volta in volta generata è stato utilizzato il tool RViz2 (vedi Sezione 2.3). Al termine della scansione, per poter salvare la mappa è stato eseguito da terminale Linux il comando `ros2 run nav2_map_server map_saver_cli` che avvia l'eseguibile `map_saver_cli` del pacchetto `nav2_map_server` di ROS 2 e salva i file `.pgm` e `.yaml` nella "current directory" del terminale. I due file sono stati poi rinominati come `my_map.yaml` e `my_map.pgm`.

Come è possibile osservare dalla Figura 9, relativa al file `my_map.pgm`, nella mappa ottenuta sono presenti 3 colori diversi (parametro `mode: trinary` di `my_map.yaml`):

- bianco -> indica un'area navigabile priva di ostacoli;
- nero -> indica la presenza di ostacoli;
- grigio -> corrisponde a zone sconosciute di cui `slam_toolbox` non è stato in grado di raccogliere informazioni (spesso corrisponde all'interno degli ostacoli).

A tali zone sconosciute potranno essere applicati alcuni criteri durante la configurazione dei nodi `nav2` affinché esse siano considerate occupate o navigabili; quest'ultima è la scelta che è stata adottata.

4.2.2 Configurazione e avvio dei nodi nav2

"Nav2 è un framework di navigazione di alta qualità che consente ai robot mobili di navigare attraverso ambienti complessi. Fornisce percezione, pianificazione, controllo, localizzazione, visualizzazione e molto altro per costruire sistemi autonomi altamente affidabili" [16, 26, 36].

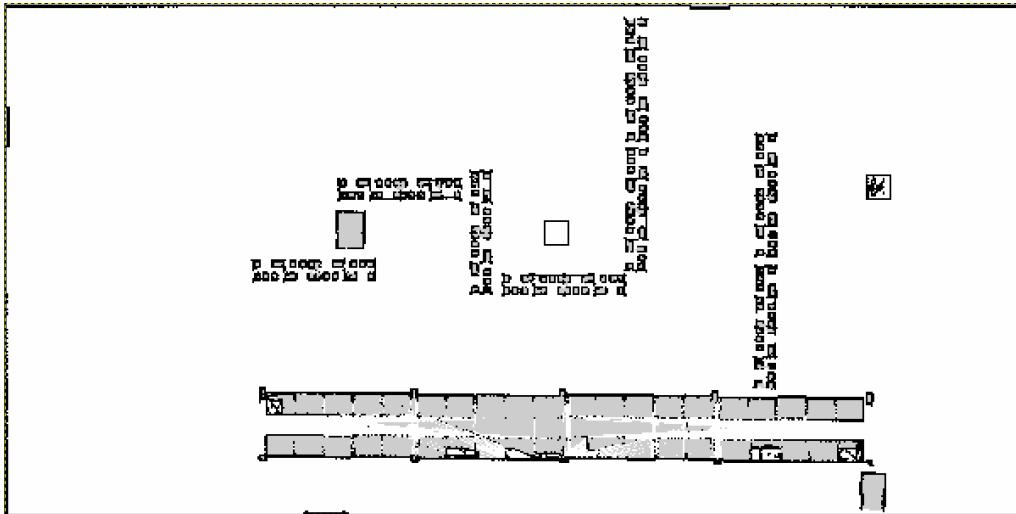


Figura 9: Mappa statica del magazzino my_map.pgm

Questo framework fornisce un insieme di package ROS 2 contenenti file eseguibili che possono essere integrati in progetti di robotica per implementare un sistema di guida autonoma. Durante la scrittura della tesi sono stati consultati alcuni file sui repository GitHub ros-navigation/navigation2 [20] e ROBOTIS-GIT/turtlebot3 [17].

Poiché i nodi devono essere eseguiti in un determinato ordine, nav2 fornisce un nodo "orchestratore" il quale gestisce il ciclo di vita dei componenti che riceve per mezzo del parametro `node_names`. Esso è il `lifecycle_manager` del package `nav2_lifecycle_manager`. Nel launch file Python gli sono stati quindi passati tutti i nodi nav2 utilizzati nella simulazione.

La visualizzazione si svolge anche in questo caso sia su Ignition che in RViz2. Sul primo è possibile osservare i movimenti del robot all'interno del magazzino, mentre nel secondo possiamo osservare i movimenti della macchina all'interno della mappa statica scansionata (vedi Sezione 4.2.1). Quest'ultima, per poter essere visualizzata su RViz2, necessita del nodo `map_server` il cui eseguibile è nel pacchetto `nav2_map_server`. Esso prende come argomento il path assoluto al file `my_map.yaml` e poi pubblica la

mappa sul topic `/map`, il quale viene letto da RViz2 che potrà così renderla visibile.

Parte essenziale della navigazione autonoma è il planner, il cui compito è quello di definire la traiettoria che la macchina deve seguire per raggiungere la "goal pose". Sulla base della posizione corrente e di quella obiettivo calcola quindi la funzione di navigazione che in output restituisce la traiettoria. Il nodo planner viene avviato con l'eseguibile `planner_server` del package `nav2_planner`.

Il percorso calcolato viene quindi utilizzato dal nodo controller (eseguibile `controller_server` del package `nav2_controller`). Esso è incaricato di inviare messaggi di Twist sul topic `/cmd_vel`. Per far sì che la velocità della macchina fosse modulabile, è stato configurato il nodo `velocity_smoother` del package `nav2_velocity_smoother`. Esso prende come parametri di configurazione i dati sulle velocità ed accelerazioni lineari ed angolari, massime e minime, di `robot_scan`. Sulla base dei parametri di configurazione e dei messaggi pubblicati dal controller, esso modifica, se necessario, i messaggi Twist per rendere la navigazione più fluida. I nuovi messaggi vengono poi pubblicati sul topic `/cmd_vel_smoothed` che è stato rimappato sul topic `/robot_scan/cmd_vel`. Su di esso transitano messaggi di tipo `geometry_msgs/msg/Twist` che, tramite il bridge, vengono inoltrati sul topic `/model/robot_scan/cmd_vel` di Ignition con tipo `ignition.msgs.Twist`. Grazie al bridge i messaggi vengono quindi trasmessi al sistema Ignition comandando i movimenti del robot affinché esso raggiunga la destinazione fissata.

La posizione obiettivo può essere indicata in due modi:

- da GUI Rviz2, selezionando la modalità "Nav2 Goal" e cliccando la destinazione sulla mappa;
- col topic `/goal_pose`, inviando un messaggio `geometry_msgs/PoseStamped`. Questo è il meccanismo che è stato utilizzato così da non richiedere un'interazione con l'utente.

Per poter migliorare la localizzazione della macchina all'interno della mappa, nav2 fornisce il nodo `amcl` (Adaptive Monte-Carlo Localization, avviabile mediante l'eseguibile `amcl` del package `nav2_amcl`). Ad inizio simulazione questo nodo richiede la pubblicazione della posizione iniziale del robot: come per la "goal pose", anche la posizione iniziale può essere specificata da GUI Rviz2, selezionando la modalità "2D Pose Estimate" ed indicandola sulla mappa, oppure tramite un messaggio di tipo `geometry_msgs/msg/PoseWithCovarianceStamped` pubblicato sul

topic `/initialpose` (che anche in questo caso è stata la modalità scelta). Dopo che essa è stata indicata, sulla base dei parametri nel file yaml di configurazione di `amcl`, la localizzazione del robot viene mantenuta aggiornata durante la navigazione per consentire al planner di ricalcolare dinamicamente la traiettoria del percorso sulla base della posizione relativa del robot all'interno della mappa. La localizzazione adattiva è resa possibile grazie ai dati forniti dal sensore Lidar che è integrato con `robot_scan` e i cui dati sono pubblicati, mediante il bridge, sul topic `/robot_scan/scan` di ROS 2.

Il file di configurazione del nodo `amcl` prevede una moltitudine di parametri i cui valori di default possono essere modificati per aumentare la precisione della localizzazione. Per la simulazione oggetto di questa tesi i valori di default si sono rivelati in molti casi sufficientemente precisi; principalmente sono stati infatti modificati soltanto i valori dei parametri relativi ai topic, ai frame ed alle caratteristiche del Lidar.

Arrivati a questo punto sorge naturale porsi la seguente domanda: "Sulla base di quali informazioni la funzione di navigazione calcola la traiettoria da seguire?"

Essa utilizza le informazioni della mappa statica e quelle delle costmap. Come indicato nella Sottosezione 4.2.1, ogni pixel della mappa si trova in uno tra gli stati libero, occupato, sconosciuto. Ovviamente, la traiettoria definita dal planner non passerà mai su pixel che si trovano nello stato occupato, mentre su quelli liberi può passare. Per quelli che si trovano nello stato "sconosciuto" possiamo scegliere di considerarli "esplorabili" impostando il parametro `allow_unknown` a `true`. Questo parametro, che è stato impostato a `true`, non assume molta rilevanza in questa simulazione poiché le uniche parti della mappa sconosciute sono quelle interne agli ostacoli, delimitate quindi da pixel che si trovano nello stato occupato e pertanto di fatto non percorribili.

Il planner predilige i percorsi che "costano meno". Questi costi sono definiti dalle costmap le quali utilizzano dei layer per modificare i costi in presenza di ostacoli, così da rendere il passaggio in tali zone più oneroso e di conseguenza sconsigliato. In nav2 sono utilizzate principalmente due tipi di costmap: la `global_costmap` e la `local_costmap`. Come è facile intuire, la prima definisce una mappa dei costi a livello globale, per tutta la mappa. La seconda, invece, ne definisce una per una porzione ristretta di mappa attorno al robot. Entrambe sono configurate con un file yaml. Come indicato nella Sottosezione 4.2.1, combinando le informazioni sulla risoluzione e sul numero dei pixel della mappa contenute nei file `my_map.yaml` e `my_map.pgm`, possiamo ottenere la larghezza e l'altezza in

metri della mappa. Queste informazioni devono essere impostate nel file di configurazione della `global_costmap` nei parametri `width` e `height`. Per la `local_costmap` è stata invece impostata un'area di 9 metri quadrati. È importante sottolineare come nel parametro `global_frame` della mappa dei costi globale sia stato impostato il frame `map`, mentre per quella locale sia stato impostato il frame di odometria `robot_scan/odom`.

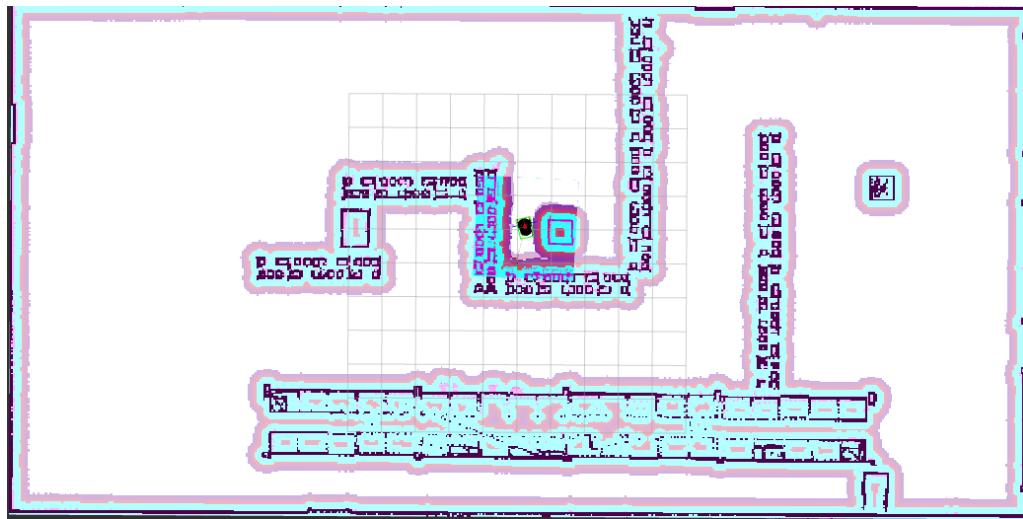


Figura 10: Mappa statica del magazzino con costmap visibili

Entrambe le costmap sono configurate con i plugin `obstacle_layer`, `static_layer` e `inflation_layer`. Il primo definisce alcune proprietà degli ostacoli ed indica le "observation_sources", nel nostro caso solo il sensore Lidar, i cui dati devono essere esaminati per il rilevamento dinamico degli oggetti; il secondo utilizza invece i dati della mappa per individuare preventivamente gli ostacoli statici che devono essere evitati; infine, l'`inflation_layer` si occupa di creare una inflazione, ovvero una crescita dei costi, in prossimità degli ostacoli. Questo ultimo layer è molto utile in quanto rende le aree attorno agli oggetti molto onerose da attraversare e quindi di fatto sconsigliate. Per configurare il grado di inflazione e il raggio di azione dell'`inflation_layer`, devono essere opportunamente impostati i parametri `cost_scaling_factor` e `inflation_radius`.

Come è possibile osservare dalla Figura 10, le costmap sono visualizzabili in RViz2. Infatti, è stata utilizzata una configurazione [17] di questo tool che permetta la visualizzazione colorata delle costmap: nelle aree prive di

ostacoli non è presente alcun colore poiché sono aree a basso costo; invece, gli oggetti sono circondati da aree colorate la cui larghezza dipende dal parametro `inflation_radius`. Come è possibile notare nella Figura 11, la mappa dei costi locale copre un'area più piccola e prevede, nella configurazione scelta di RViz2, colori più accesi per indicare gli ostacoli che circondano il robot ed a cui quest'ultimo dovrà fare attenzione durante la navigazione.

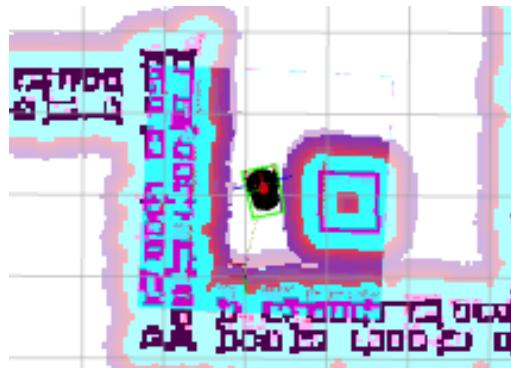


Figura 11: Zoom sulla local_costmap

Un altro nodo importante che è stato aggiunto al sistema è il `behavior_server`. Esso definisce, tramite i `behavior_plugins`, alcune azioni che il robot può compiere in caso di difficoltà durante la navigazione autonoma, ad esempio a seguito di una collisione o in caso di blocco in un punto stretto. I plugin scelti sono `spin`, `backup` e `wait` che, in base a quanto definito nei behavior trees configurati nel nodo `bt_navigator`, consentono al robot di effettuare azioni di recupero quali la rotazione, la retromarcia e l'attesa di un lasso di tempo opportunamente configurato per ambienti dinamici in cui sono presenti ostacoli in movimento.

Infine, l'ultimo nodo configurato è il `bt_navigator`. Esso, come è possibile vedere dalla Figura 12, è un componente fondamentale poiché, sulla base dei behavior trees che riceve attraverso i parametri `default_nav_to_pose_bt_xml` e `default_nav_through_poses_bt_xml`, si occupa del coordinamento degli altri componenti del sistema `nav2`. Consente quindi di orchestrare le operazioni necessarie per la navigazione autonoma del robot verso la destinazione finale che gli viene inviata attraverso il topic `/goal_pose`.

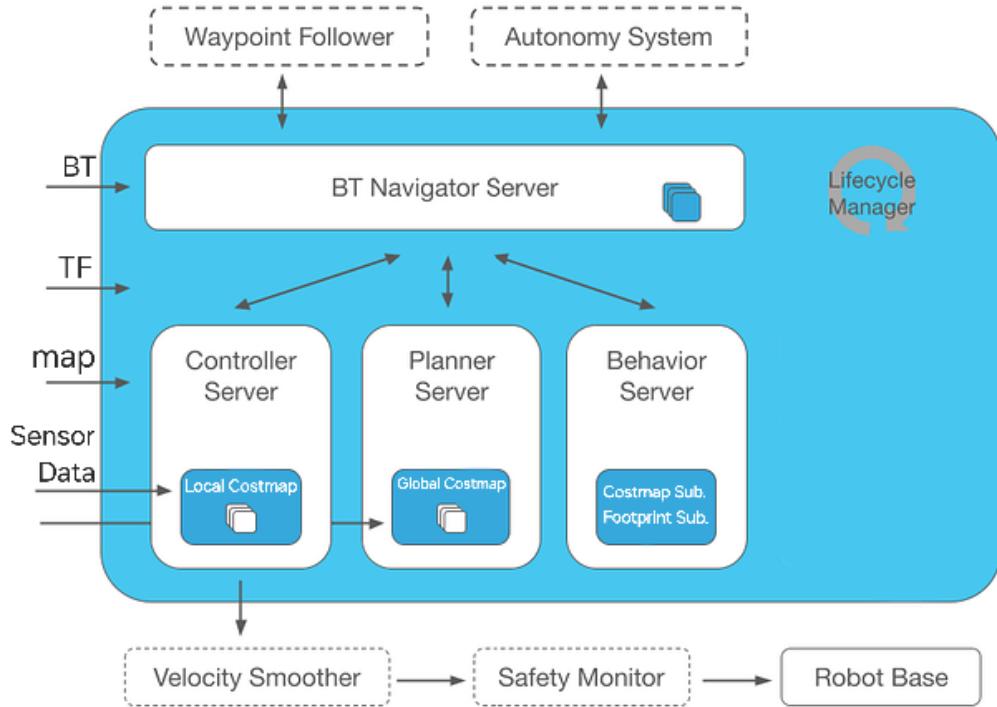


Figura 12: Diagramma nav2 [15]

4.3 GESTIONE ED AVVIO DELLA SIMULAZIONE

La simulazione viene gestita mediante un nodo orchestratore che controlla l'ordine di esecuzione dei nodi (vedi Sottosezione 4.3.1) e viene avviata da file di lancio Python (vedi Sottosezione 4.3.2).

4.3.1 Nodo Orchestratore

Poiché l'ordine di esecuzione dei task è importante per poter ottenere comportamenti desiderabili, in questa tesi è stato utilizzato un nodo ROS 2 che assume il ruolo di orchestratore. Come per i nodi action client e server (vedi Sezione 4.1), anche in questo caso la dichiarazione della classe `OrchestratorNode` è stata effettuata in un file header, `orchestrator_node.hpp`, la cui implementazione è stata effettuata nel file `orchestrator_node.cpp`.

Come anticipato all'inizio di questo capitolo, la prima parte di simulazione è relativa al quadcopter mentre la seconda riguarda la guida autonoma di `robot_scan`.

Per gestire l'ordine di esecuzione delle operazioni nella simulazione, l'orchestratore utilizza i seguenti topic:

- `/quadcopter/start_navigation`: pubblica su questo topic la stringa `"start_drone_navigation"` così che l'action client possa cominciare ad inviare goal al server per far muovere il drone;
- `/quadcopter/end_navigation`: è sottoscritto a questo topic in attesa della ricezione della stringa `"end_drone_navigation"` che indica l'atterraggio del drone. A seguito della ricezione del messaggio avvia la navigazione autonoma del robot a trazione differenziale pubblicando la destinazione sul topic `/goal_pose`;
- `/initialpose`: pubblica su questo topic un messaggio di tipo `geometry_msgs/msg/pose_with_covariance_stamped` contenente la posizione iniziale di `robot_scan` in modo che il nodo `amcl` di `nav2` possa avviare la localizzazione adattiva della macchina all'interno della mappa;
- `/goal_pose`: pubblica su questo topic un messaggio di tipo `geometry_msgs/msg/pose_stamped` in modo che i nodi `nav2` possano avviare la navigazione autonoma verso la posizione indicata.

L'ordine degli eventi nella simulazione, ad un livello di astrazione tale da tralasciare i dettagli del sistema `nav2`, può essere rappresentato dal sequence diagram UML in Figura 13.

4.3.2 Avvio della simulazione

Per poter avviare la simulazione sono necessari due terminali Linux (per la tesi è stato utilizzato Ubuntu 22.04 `jammy`) ed è necessario anche aver installato ROS 2 Humble [13] ed Ignition Fortress [8]. Prima di poter eseguire i comandi di avvio, devono essere effettuate alcune operazioni preliminari (per praticità i comandi che verranno mostrati sono stati aggiunti nel file `.bashrc` in modo che siano eseguiti automaticamente all'apertura di ogni nuovo terminale):

- `export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp`: per poter utilizzare correttamente i pacchetti `nav2` senza errori è necessario cambiare il DDS di default `FastDDS` con `CycloneDDS` [4]. Il DDS fornisce l'architettura di comunicazione in ROS 2;

- . ~/ros2_humble/install/local_setup.bash, source /opt/ros/-humble/setup.bash, export ROS_DOMAIN_ID=0: questi comandi sono necessari per impostare le variabili d'ambiente ROS 2 e per effettuare il "sourcing" dell'underlay ROS 2 (vedi Sezione 2.1 per maggiori informazioni) [5];
- export IGN_GAZEBO_RESOURCE_PATH=\$IGN_GAZEBO_RESOURCE_PATH:\$HOME/tesi_ws/install/my_sim_tesi_gazebo/share:\$HOME/tesi_ws/install/my_sim_tesi_gazebo/share/my_sim_tesi-gazebo/models:\$HOME/tesi_ws/install/my_sim_tesi_gazebo/share/my_sim_tesi-gazebo/worlds: questa variabile di ambiente Ignition deve essere configurata per far sì che i modelli caricati nei file SDF Ignition (con i tag <uri> </uri>) siano trovati all'interno del filesystem così come le altre risorse, file meshes e plugin [7].

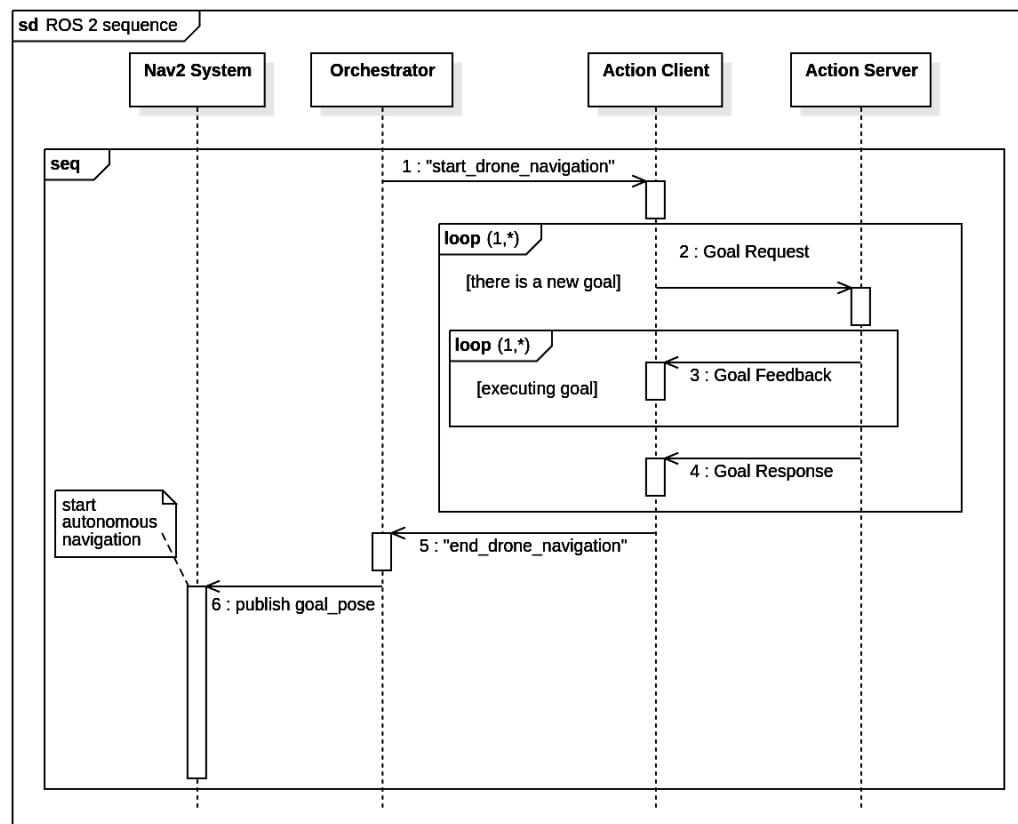


Figura 13: UML Sequence Diagram dell'esecuzione della simulazione

Se viene utilizzata una struttura di progetto diversa da quella mostrata in questa tesi (vedi Figura 8), così come una distribuzione ROS 2 che non sia Humble, le variabili sopra definite devono essere reimpostate correttamente.

A questo punto, è necessario aprire due terminali: in entrambi dobbiamo entrare all'interno della directory che rappresenta il workspace ROS 2 (nella tesi è la cartella `~/tesi_ws`). A questo punto nel primo terminale si esegue il comando `colcon build` che utilizza lo strumento `colcon` per effettuare la build del workspace ROS 2. Questo comando crea le cartelle `build`, `install` e `log` che verranno utilizzate da ROS 2 per poter eseguire il progetto. Nel secondo terminale dobbiamo effettuare il caricamento dell'overlay con il comando `source install/setup.bash`. Esso è necessario per far sì che i nuovi package ROS 2, creati nel proprio workspace e costruiti con `colcon`, possano essere individuati ed eseguiti da ROS 2. Nello stesso terminale del sourcing, possiamo ora eseguire il file di lancio con il comando `ros2 launch my_sim_tesi Bringup my_sim_final.launch.py`, dove `my_sim_tesi Bringup` è il package contenente il launch file `my_sim_final.launch.py`.

Data la pesantezza dell'ambiente simulato, l'avvio di Ignition Fortress e dei nodi ROS 2 richiede un totale di circa 20 secondi, nella macchina utilizzata per la sperimentazione in questa tesi¹, al termine dei quali possiamo osservare il drone che comincia a muoversi.

I robot ad inizio simulazione sono posizionati come in Figura 14.



Figura 14: Robots ad inizio simulazione in Ignition e RViz2

¹ Modello: HP Pavilion Laptop 15-eg0028nl; CPU: 11th Gen Intel(R) Core(TM) i7-1164.70 GHz; GPU: TigerLake-LP GT2 [Iris Xe Graphic] Integrata da 1 GB; RAM: 16 GB.

Sia quadcopter che robot_scan sono visibili in Rviz2 e in Ignition. Per quanto riguarda la prima parte della simulazione è più interessante osservare il movimento del drone all'interno di Ignition, poiché in RViz2 è presente una visualizzazione 2D della mappa che quindi non riflette l'altitudine a cui quadcopter sta viaggiando.

È possibile osservare il drone a metà percorso che sta per volare sopra ad un ostacolo in Figura 15.

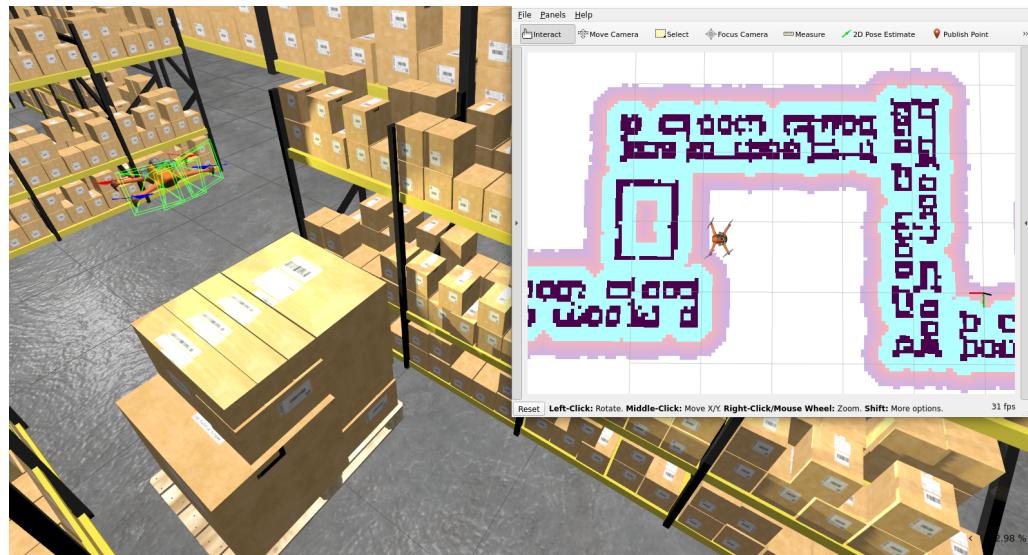


Figura 15: Drone a metà percorso prima di volare sopra ad un ostacolo

Invece, nella seconda parte, quella in cui è coinvolto robot_scan, la situazione cambia: in RViz2 possiamo in tal caso osservare informazioni utili del framework nav2 durante la navigazione autonoma del robot. Infatti è possibile visualizzare la traiettoria globale, la traiettoria locale (piccola linea blu davanti alla macchina), le costmap ed infine anche uno "sciame" di frecce verdi che indicano la possibilità nell'immediato futuro che la traiettoria locale del robot passi in un determinato punto della mappa. La local costmap circonda il robot e cambia durante la simulazione. Inoltre, in RViz2 si possono osservare i dati letti dal gpu_lidar che appaiono come punti rossi attorno agli ostacoli "colpiti" dai raggi del Lidar. Possiamo notare come varia l'allineamento tra questi punti ed i contorni degli ostacoli: durante la navigazione, specialmente a seguito di rotazioni, appare un lieve disallineamento tra di essi ma, dopo pochi centimetri, grazie alla localizzazione adattiva fornita dal nodo amcl (vedi Sottosezione 4.2.2), tornano allineati. Per la seconda parte è quindi interessante osservare sia Ignition, per vedere il robot muoversi

nel magazzino virtuale, sia RViz2, per osservare i dati letti dai sensori e i cambiamenti delle traiettorie e delle costmap nav2.

In Figura 16 è possibile osservare in RViz2 il robot a trazione differenziale che si trova a metà percorso.

La simulazione termina quindi con il drone e la macchina affiancati l'uno all'altra (Figura 17).



Figura 16: `robot_scan` a metà percorso con `local_costmap` ben visibile

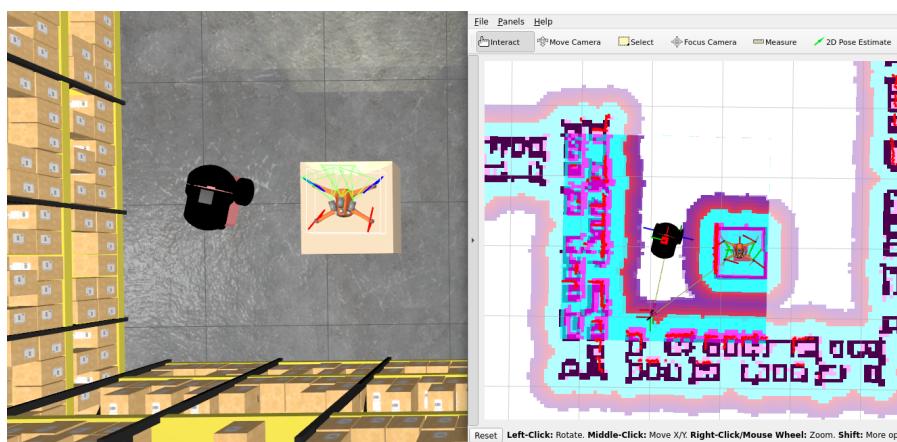


Figura 17: `robot_scan` e quadcopter affiancati al termine della simulazione

5

CONCLUSIONI E SVILUPPI FUTURI

L'obiettivo di questa tesi è lo sviluppo di una simulazione relativa ad uno scenario con robot eterogenei tramite ROS 2 e Ignition. In particolare, la simulazione realizzata si svolge all'interno di un magazzino e ha come "attori" un drone ed una macchina a trazione differenziale. La prima parte consiste nel drone quadcopter che, dopo essere decollato, segue una traiettoria nel mezzo agli scaffali per poi terminare la navigazione atterrando sopra un piano. In seguito, viene quindi avviata la navigazione autonoma del robot a trazione differenziale che, dopo aver seguito la traiettoria calcolata dai nodi del framework nav2, si posiziona di fianco a quadcopter.

5.1 SVILUPPI FUTURI

Il drone, come spiegato nella Sezione 4.1, è programmato per seguire una traiettoria mediante la definizione di una serie di obiettivi che di volta in volta deve raggiungere. Nel futuro, con maggior tempo a disposizione, sarà possibile estendere il progetto di robotica dotando quadcopter di ulteriori sensori in modo da renderlo in grado di effettuare scelte in autonomia sulla base dell'ambiente di navigazione in cui si trova. Ad esempio, se venisse dotato di un sensore Lidar per la scansione dell'area circostante, potremmo utilizzare i dati raccolti per renderlo in grado di scegliere la prossima posizione verso cui navigare per raggiungere un punto prestabilito.

Poiché la robotica non si limita a simulazioni virtuali, un altro aspetto interessante da approfondire è quello relativo ai robot fisici. Viene spesso creato un digital twin di un modello fisico in modo da effettuare sperimentazioni nel mondo virtuale per poi passare a quello reale mantenendo invariato il codice che era stato testato sul modello digitale. Quindi, il progetto di questa tesi potrebbe essere esteso passando al mondo reale attraverso l'utilizzo di robot fisici quanto più simili a quelli utilizzati

nella simulazione. È un aspetto molto interessante che comunque, oltre a richiedere la comprensione delle problematiche legate al mondo reale, necessita anche di risorse economiche legate all'acquisto dei componenti. Per il futuro, con più tempo a disposizione, è sicuramente un argomento che merita ulteriori approfondimenti e sperimentazioni.

5.2 CONSIDERAZIONI FINALI

In conclusione, per poter fornire vantaggi alla società mediante l'utilizzo di tecnologie robotiche è necessario passare dal virtuale al reale, effettuando sperimentazioni con robot fisici e raffinando le tecniche di programmazione per rendere i modelli efficienti e soprattutto sicuri per le persone con cui si interfacciano. Questo ultimo aspetto è senza ombra di dubbio quello più importante; infatti, non dobbiamo utilizzare tecnologie moderne se queste possono essere dannose per l'ambiente e per le persone con cui entrano in contatto.

Oltre alla comprensione dei concetti relativi allo sviluppo di applicazioni robotiche, è quindi necessaria una piena comprensione delle problematiche legate al mondo reale nel quale, a differenza del mondo virtuale, spesso non sono ammessi errori implicando quindi una maggior sperimentazione e testing delle tecnologie e dei modelli utilizzati.

BIBLIOGRAFIA

- [1] Basic concepts ros 2. <https://docs.ros.org/en/humble/Concepts/Basic.html>.
- [2] Beginner tutorial: Client libraries. <https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries.html>.
- [3] Bridge communication between ros and gazebo. https://github.com/gazebosim/ros_gz/blob/ros2/ros_gz_bridge/README.md. Data ultimo accesso: 2024-08-15.
- [4] Bugs with fast-dds, switch to cycloneddds. <https://github.com/ros-navigation/navigation2/issues/3298>. Data ultimo accesso: 2024-08-10.
- [5] Configuring environment ros 2. <https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Configuring-ROS2-Environment.html>. Data ultimo accesso: 2024-07-08.
- [6] Github robot_state_publisher. https://github.com/ros/robot_state_publisher?tab=BSD-3-Clause-1-ov-file.
- [7] Ignition: finding resources. <https://gazebosim.org/api/gazebo/6/resources.html>. Data ultimo accesso: 2024-07-25.
- [8] Ignition fortress binary installation on ubuntu. https://gazebosim.org/docs/fortress/install_ubuntu/. Data ultimo accesso: 2024-06-20.
- [9] Ignition fortress getting started and logo. <https://gazebosim.org/docs/fortress/getstarted/>.
- [10] Ignition fortress official tutorials. <https://gazebosim.org/docs/fortress/tutorials/>.
- [11] Ignition fortress release. <https://community.gazebosim.org/t/ignition-fortress-release/1127>.
- [12] Ignition fuel. <https://app.gazebosim.org/fuel/models>.

- [13] Installation ros 2 humble ubuntu (deb). <https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debs.html>. Data ultimo accesso: 2024-06-20.
- [14] My github repository: Ros 2 humble + ignition fortress simulation. https://github.com/marcoaga02/ros2_ignition_thesis.
- [15] [nav2] path smoother server, plugins, tutorial, new architectural diagram, oh my! <https://discourse.ros.org/t/nav2-path-smoother-server-plugins-tutorial-new-architectural-diagram-oh-my/24940>.
- [16] Overview nav2 framework. <https://docs.nav2.org/>.
- [17] Robotis-git/turtlebot3 github. <https://github.com/ROBOTIS-GIT/turtlebot3>.
- [18] Ros 2 documentation, logo. <https://docs.ros.org/en/humble/index.html#>.
- [19] Ros 2 humble hawksbill official tutorials. <https://docs.ros.org/en/humble/Tutorials.html>.
- [20] ros-navigation/navigation2 github. <https://github.com/ros-navigation/navigation2>.
- [21] Ros wikipedia, logo. https://it.wikipedia.org/wiki/Robot_Operating_System.
- [22] Rviz2 github and logo. <https://github.com/ros-visualization/rviz?tab=readme-ov-file>.
- [23] Wikipedia: Robotica. <https://it.wikipedia.org/wiki/Robotica#Voci correlate>.
- [24] S. Macenski. On use of slam toolbox, a fresh(er) look at mapping and localization for the dynamic world. *ROSCon*, 2019.
- [25] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.
- [26] Steven Macenski, Francisco Martin, Ruffin White, and Jonatan Génés Clavero. The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

- [27] Steven Macenski, Alberto Soragna, Michael Carroll, and Zhenpeng Ge. Impact of ros 2 node composition in robotic systems. *IEEE Robotics and Autonomous Letters (RA-L)*, 2023.
- [28] MovAi. *pallet_box_mobile*, March 2022. https://app.gazebosim.org/MovAi/fuel/models/pallet_box_mobile, Data ultimo accesso: 2024-08-01.
- [29] MovAi. *shelf*, March 2022. <https://app.gazebosim.org/MovAi/fuel/models/shelf>, Data ultimo accesso: 2024-08-01.
- [30] MovAi. *shelf_big*, March 2022. https://app.gazebosim.org/MovAi/fuel/models/shelf_big, Data ultimo accesso: 2024-08-01.
- [31] OpenRobotics. *Depot*, September 2023. <https://app.gazebosim.org/OpenRobotics/fuel/models/Depot>, Data ultimo accesso: 2024-08-05.
- [32] OpenRobotics. *Pioneer 2dx*, September 2023. <https://app.gazebosim.org/OpenRobotics/fuel/models/Pioneer%20DX>, Data ultimo accesso: 2024-08-10.
- [33] OpenRobotics. *X3 uav config 5*, September 2023. <https://app.gazebosim.org/OpenRobotics/fuel/models/X3%20UAV%20Config%205>, Data ultimo accesso: 2024-08-15.
- [34] OpenRobotics. *multicopter_velocity_control.sdf*, 2024. https://github.com/gazebosim/gz-sim/blob/main/examples/worlds/multicopter_velocity_control.sdf. Data ultimo accesso: 2024-08-07.
- [35] openroboticstest. *Empty world*, Aprile 2024. <https://app.gazebosim.org/openroboticstest/fuel/worlds/Empty%20world>, Data ultimo accesso: 2024-08-02.
- [36] DV Lu A. Merzlyakov M. Ferguson S. Macenski, T. Moore. From the desks of ros maintainers: A survey of modern capable mobile robotics algorithms in the robot operating system 2. *Robotics and Autonomous Systems*, 2023.
- [37] I. Jambrecic S. Macenski. Slam toolbox: Slam for the dynamic world. *Journal of Open Source Software*, 6(61), 2783, 2021.