Università degli studi di Modena e Reggio Emilia
Dipartimento di ingegneria "Enzo Ferrari"

2024

Implementazione di un sistema di controllo remoto per veicoli semi-autonomi connessi

Relatore: Paolo Burgio
Candidato: Alessandro Appio

Anno accademico: 2023/2024

Alla mia famiglia

Abstract

Col passare degli anni ci si approccia sempre di più a un mondo caratterizzato da una progressiva integrazione di sistemi automatici e connessi, che incide profondamente sulle modalità di interazione con l'ambiente e sulle abitudini di vita.

La presente tesi esplora i concetti di guida autonoma e guida remota, due paradigmi emergenti nel settore dei trasporti che promettono di rivoluzionare la mobilità. La guida autonoma è caratterizzata da veicoli in grado di percepire l'ambiente circostante, pianificare traiettorie e eseguire manovre senza intervento umano diretto, la guida remota invece prevede che le decisioni relative alla guida siano prese a distanza da un operatore umano o da un sistema intelligente.

L'analisi comparativa tra le due tecnologie evidenzia le loro specificità e potenzialità. La guida autonoma offre la prospettiva di veicoli completamente autonomi, in grado di operare in modo sicuro ed efficiente in una vasta gamma di scenari. La guida remota, da parte sua, consente di estendere le capacità di guida a veicoli non tradizionali o di operare in ambienti complessi.

Il presente lavoro si concentra sullo sviluppo di un sistema di guida caratterizzato da una dualità funzionale: la capacità di operare sia in modalità autonoma che in modalità remota. Nello specifico per lo sviluppo è stato utilizzato un rover di marca AgileX come piattaforma di sviluppo, su cui sono stati montati una GPGPU Nvidia AGX Xavier come computer di bordo ed un sensore Lidar per l'analisi ambientale. Per permettere lo sviluppo dello stack di guida remota ci si è avvalsi del protocollo MQTT, ampiamente utilizzato nelle tecnologie IOT e della suite di librerie ROS utilizzato per permettere una gestione modulare delle varie funzionalità del robot.

Il sistema, in conclusione si rivela funzionale, ma presenta alcune criticità relative alla latenza riscontrata nelle comunicazioni remote, con tempistiche misurate che si rivelano eccessive per l'ambito della guida remota, richiedendo dunque ulteriori approfondimenti ed eventuali miglioramenti nello stack realizzato.

Indice

1 Introduzione **7**
1.1 Guida remota e guida autonoma 7
1.2 Tecnologie IOT 9
1.3 Scopo della tesi 10

2 Piattaforma di sviluppo **11**
2.1 Rover AgileX 11
2.1.1 Protocollo CAN 12
2.2 GPGPU 12
2.3 Lidar 13
2.4 Router 14

3 Middleware di comunicazione **17**
3.1 ROS 17
3.1.1 Nodi 18
3.1.2 Comunicazione tra nodi 19
3.2 MQTT 21
3.2.1 Infrastruttura 21
3.2.2 Formattazione messaggi 22
3.2.3 Topic 24

4 Scelte progettuali **25**
4.1 Affidabilità 25
4.2 Sicurezza 26
4.3 Struttura 27
4.4 Controllo del QoS 27

5 Funzionamento **29**
5.1 Stack di guida autonoma 29

5.1.1 Perception . . . 29
5.1.2 Planning . . . 30
5.1.3 Control . . . 31
5.2 Guida remota . . . 31
5.3 Nodi sviluppati . . . 32
5.3.1 MQTT telemery node . . . 33
5.3.2 MQTT remote control . . . 36
5.3.3 MQTT server node . . . 36
5.4 Gestione dei topic . . . 37

6 Validazione sperimentale **39**
6.1 Controllo . . . 39
6.2 Mappatura . . . 40
6.3 Localizzazione . . . 41
6.4 Guida autonoma . . . 41
6.5 Guida remota . . . 41

7 Sviluppi futuri e conclusioni **45**
7.1 Problemi ed eventuali soluzioni . . . 45
7.2 Sviluppi futuri . . . 46
7.3 Applicazioni pratiche . . . 47

Capitolo 1.0

Introduzione

1.1 Guida remota e guida autonoma

La guida autonoma e la guida remota sono due tecnologie che promettono di trasformare radicalmente il modo in cui ci muoviamo, rendendo i trasporti più sicuri, efficienti e sostenibili, e stanno al centro di una rivoluzione tecnologica: la mobilità intelligente.

La guida autonoma rappresenta un complesso sistema tecnologico che integra una serie di avanzate tecnologie, metodologie e tecniche finalizzate a consentire il movimento di un veicolo senza necessità di intervento umano diretto. Un veicolo autonomo è infatti dotato della capacità di analizzare l'ambiente circostante, elaborare un percorso ottimale in base ai dati raccolti, e seguire tale percorso in modo autonomo.

Secondo la Society of Automotive Engineers international (SAE)[7] il concetto di guida autonoma si può dividere in 6 livelli:

- Livello 0: Il veicolo non ha capacità di guida autonoma e tutta la responsabilità è conferita al guidatore
- Livello 1: Il veicolo assiste il guidatore in maniera limitata, agendo su acceleratore, freno e sterzo. La responsabilità rimane però completamente nelle mani dell'umano
- Livello 2: Il veicolo ha le capacità di controllare simultaneamente la direzione, l'accelerazione e la frenata, ma il guidatore deve sempre e comunque rimanere vigile e monitorare le scelte prese dal veicolo
- Livello 3: Il veicolo può prendere completamente il controllo della guida ma solo in determinate situazioni (e.g. in autostrada) e il conducente deve essere pronto a prendere il controllo quando richiesto dal sistema

- Livello 4: Il veicolo può gestire tutte le funzioni di guida in maniera autonoma in quasi tutte le condizioni, ma può comunque richiedere l'intervento umano in situazioni straordinarie (e.g. zone non mappate)
- Livello 5: Il veicolo è completamente autonomo e non necessita di intervento umano

I processi fondamentali della guida autonoma vengono generalmente suddivisi in tre fasi distinte: percezione (perception), pianificazione (planning) e controllo (control). La percezione riguarda la capacità del veicolo di raccogliere informazioni dall'ambiente circostante attraverso sensori avanzati, che possono includere telecamere, radar, Lidar e altre tecnologie di rilevamento. Questi dati vengono poi elaborati nella fase di pianificazione, durante la quale il sistema valuta le possibili traiettorie e sceglie il percorso più sicuro ed efficiente da seguire. Infine, la fase di controllo si occupa dell'esecuzione del movimento del veicolo lungo il percorso stabilito, garantendo che vengano seguite le decisioni prese nella fase di pianificazione.

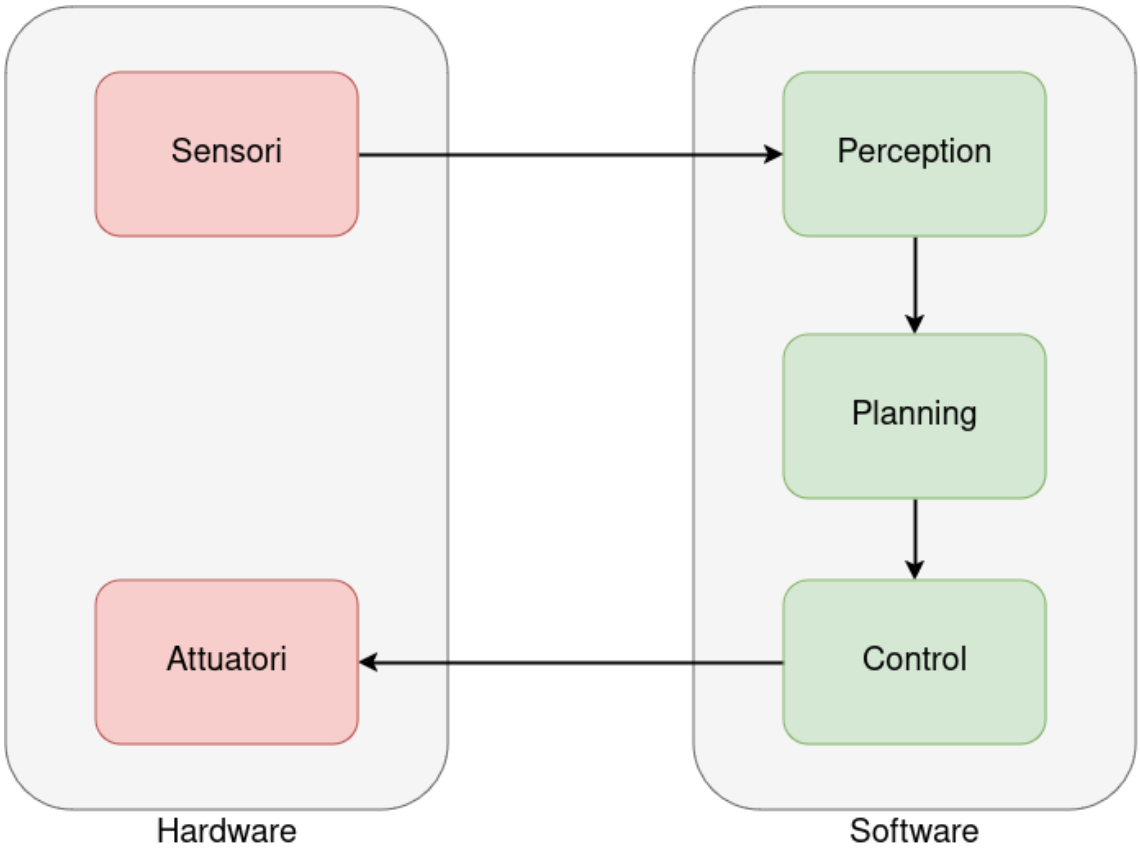

Figura 1.1: Semplice stack di guida autonoma

Se nella guida autonoma le decisioni vengono elaborate da un sistema informatico a bordo del veicolo, il concetto di guida remota si riferisce a un tipo di guida in cui le decisioni relative alla direzione e al movimento del veicolo vengono elaborate in remoto. A seconda della strategia con la quale vengono prese queste decisioni, la guida remota si suddivide in:

- Driver: le decisioni relative alla direzione e al movimento del veicolo vengono prese da un operatore umano che opera a distanza, capace di osservare i dati rilevati dai sensori presenti a bordo del mezzo
- Driverless: le decisioni vengono prese da uno stack di guida autonoma eseguito da un server remoto che prende in input i dati rilevati dai sensori presenti a bordo del mezzo e trasmette al veicolo istruzioni puntuali sul movimento da compiere

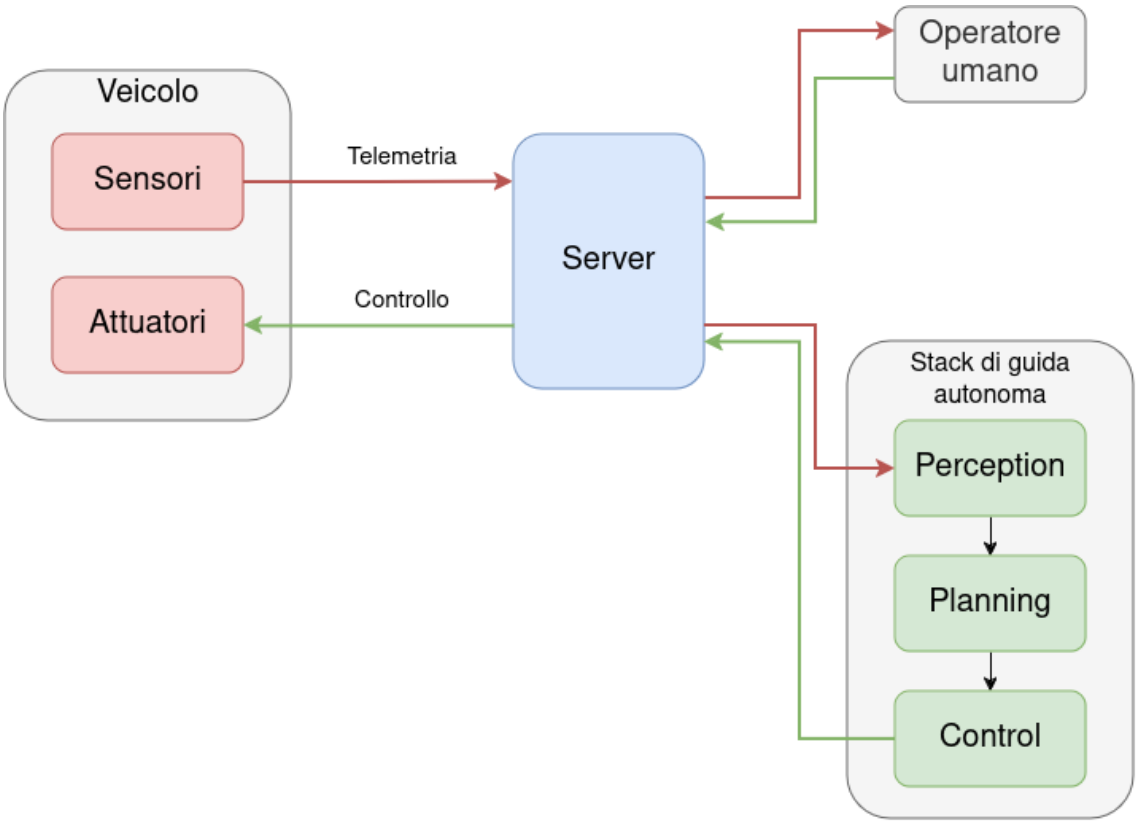

Figura 1.2: Semplice stack di guida remota

La guida remota opera utilizzando tecnologie quali sensori, attuatori, e reti di comunicazione. In questo scenario, la logica che impartisce ordini non si trova fisicamente all'interno del veicolo, ma interagisce con esso attraverso un'interfaccia remota, sfruttando la trasmissione dei dati in tempo reale per monitorarlo e controllarlo. Tale approccio rende possibile la guida di veicoli in situazioni in cui la presenza fisica del conducente potrebbe non essere necessaria o praticabile

1.2 Tecnologie IOT

L'avvento dell'Internet of Things (IoT) ha inaugurato una nuova era tecnologica, caratterizzata dalla connettività pervasiva e dall'intelligenza distribuita. Questa rivoluzione digitale sta trasformando profondamente numerosi settori, tra cui la

logistica. La crescente complessità delle catene di approvvigionamento globali, unita alla crescente domanda di efficienza e tracciabilità, rende l'IoT una tecnologia sempre più strategica per le aziende del settore.

L'IoT è un concetto che descrive una rete di dispositivi fisici connessi tra loro attraverso Internet, che sono capaci di scambiare dati e di comunicare con altri dispositivi, sistemi e/o servizi. Questi dispositivi, che possono includere sensori, elettrodomestici, veicoli (come nel nostro caso), sistemi di sicurezza, macchinari industriali e molto altro, sono dotati di sensori, software e altre tecnologie che permettono loro di raccogliere e condividere dati in tempo reale.

La guida remota, ovvero la possibilità di controllare un veicolo a distanza attraverso una connessione di rete, rappresenta una delle applicazioni più promettenti dell'IoT nel settore dei trasporti.

1.3 Scopo della tesi

L'obiettivo principale di questa tesi è quello di sviluppare un sistema di guida capace di operare sia in modo autonomo che in modalità di guida remota.

Si consideri lo scenario in cui il veicolo e l'hardware a bordo siano intatti e che il conducente sia sempre vigile, controllando il comportamento del mezzo anche in caso di errore o malfunzionamento. Su richiesta del conducente, il sistema potrà operare in questo caso in modalità autonoma.

Supponiamo ora che il guidatore non sia nelle sue condizioni ottimali e che, per esempio, a causa di un malore, non possa rimanere vigile e controllare adeguamente il veicolo. In questo caso, un ente terzo sarà in grado collegarsi al veicolo, passare alla modalità remota e mettere in sicurezza l'operatore conducendolo eventualemte all'ospedale più vicino.

Questo approccio duale consente al veicolo di navigare in modo completamente indipendente quando le circostanze lo permettono, utilizzando tecnologie di percezione, pianificazione e controllo integrate, ma offre al contempo la flessibilità di essere controllato a distanza da un operatore umano o da uno stack di guida autonoma qualora la situazione lo richieda. La possibilità di commutare tra guida autonoma e remota mira a garantire la massima sicurezza, adattabilità e versatilità del veicolo in una varietà di scenari operativi.

Capitolo 2.0

Piattaforma di sviluppo

In questa sezione si descrive come è composta e come è stata assemblata la piattaforma per lo sviluppo e il testing.

2.1 Rover AgileX

Il veicolo selezionato per lo sviluppo della presente tesi è un rover terrestre prodotto da AgileX, modello Hunter. Questo rover è dotato di 3 motori elettrici (2 per la trazione posteriore ed 1 per lo sterzo), di sensori dediti all'analisi del movimento delle ruote e di una board dedicata al controllo dei motori e all'interfacciamento del mezzo con un calcolatore esterno. L'intera scocca è realizzata in alluminio rendendolo molto resistente ma allo stesso tempo non eccessivamente pesante.

La scheda di controllo del rover è collegata alla porta usb del computer embedded tramite un'apposita periferica e consente una comunicazione efficiente e affidabile. Il modello Hunter è stato scelto per le sue avanzate caratteristiche tecniche e per la sua versatilità, che lo rendono particolarmente adatto alle esigenze del progetto.

Oltre a fornire un'interfaccia per il controllo diretto, il veicolo è in grado di raccogliere e trasmettere una serie di dati diagnostici e operativi fondamentali per il monitoraggio e l'analisi delle sue prestazioni. Tra questi dati, un ruolo cruciale è ricoperto dall'odometria.

L'odometria è una misura che permettte la stima dello spostamento di un veicolo a partire dallo spostamento delle ruote, ed è essenziale per la navigazione e la stima della posizione del rover, poiché permette di determinare il percorso seguito dal veicolo e la distanza percorsa. Questi dati, insieme ad altre informazioni sullo stato del veicolo, contribuiscono a garantire un controllo preciso e ad alimentare i sistemi di guida autonoma e remota previsti dal progetto.

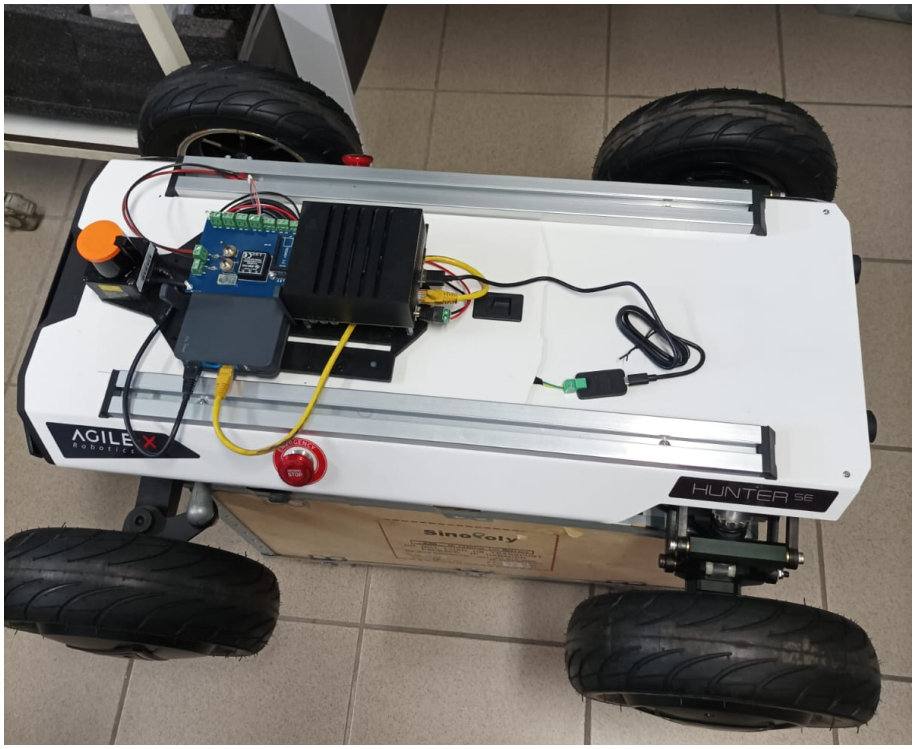

Figura 2.1: Rover assemblato

2.1.1 Protocollo CAN

La scheda di controllo del rover è basata sul protocollo Controller Area Network (CAN), un protocollo seriale molto versatile sviluppato dall'azienda Bosh nel 1993 e molto utilizzato in ambito automotive e automazione industriale. Questa interfaccia seriale è accessibile tramite 2 segnali denominati CAN-HIGH e CAN-LOW, collegati ad una periferica USB interfacciata al computer di bordo.

Il protocollo CAN è utilizzato in quanto è un protocollo molto resistente alle interferenze (grazie a una tecnica di bit dominante e recessivo), veloce e per niente costoso.

2.2 GPGPU

Un elemento cruciale per la realizzazione di questa tesi è stato l'identificazione e la selezione di un calcolatore embedded che possa comunicare con i sensori e il rover e che possa gestire il carico di tutti gli algoritmi e i processi necessari per la guida autonoma.

La scelta è ricaduta su una scheda di casa Nvidia, modello AGX Jetson Xavier, una General Purpose Graphic Processing Unit (GPGPU). Questa scelta è motivata dall'elevata capacità di elaborazione parallela che solo una GPGPU può fornire, capacità che risulta particolarmente vantaggiosa per l'esecuzione di complessi algoritmi di percezione, pianificazione e controllo, da svolgere in tempo reale. La GPGPU se-

lezionata opera con il sistema operativo Ubuntu 20.04 Focal Fossa, noto per la sua stabilità e compatibilità con l'hardware scelto.

Figura 2.2: Nvidia AGX Jetson Xavier

Di seguito una tabella riassuntiva delle principali caratteristiche della GPGPU Utilizzata:

GPU	GPU con architettura NVIDIA Volta a 512 core con 64 Tensor Core
CPU	CPU NVIDIA Carmel Arm® v8.2 8-core 64-bit 8 MB L2 + 4 MB L3
Memoria	LPDDR4x 32 GB 256-bit 136,5 GB/s
Storage	eMMC 5.1 32 GB espandibile con scheda di memoria e/o SSD NVMe
Alimentazione	10 W - 30 W

Tabella 2.1: Specifiche tecniche della GPGPU Nvidia AGX Jetson Xavier[3]

2.3 Lidar

Per permettere al sistema di orientarsi e navigare correttamente è necessario l'utilizzo di un sensore che permetta al veicolo di percepire l'ambiente circostante. La scelta è ricaduta su un sensore Light Detection and Ranging (LiDaR) 2D di marca Hokuio. Questo dispositivo sfrutta la tecnologia laser per determinare la distanza di vari punti nell'ambiente circostante, calcolando il tempo di ritorno dei raggi laser emessi. Il Lidar fornisce una mappa dettagliata della topografia dell'ambiente, consentendo al sistema di percezione di creare rappresentazioni, tridimensionali o bidimensionali,

dello stesso e fondamentali per il riconoscimento degli ostacoli, la navigazione, la pianificazione del percorso del veicolo e la mappatura dell'ambiente circostante.

La combinazione di una GPGPU performante e un sensore Lidar avanzato rappresenta una solida base tecnologica per lo sviluppo di un sistema di guida autonoma e remota altamente efficiente.

Il sensore Lidar genera una nuvola di punti dell'ambiente circostante attraverso l'emissione di impulsi laser in un cono di 270 gradi. Ciascun punto della nuvola corrisponde alla distanza misurata tra il sensore e un elemento dell'ambiente. La distanza è determinata con accuratezza cronometrando il tempo impiegato dall'impulso laser a percorrere il tragitto andata e ritorno.

Figura 2.3 riporta un'immagine del sensore scelto.

Figura 2.3: Sensore Lidar

2.4 Router

In considerazione delle elevate esigenze di comunicazione proprie di un veicolo connesso, si è optato per l'integrazione a bordo di un router di rete. Tale dispositivo ha la duplice funzione di instaurare una connessione stabile e ad alta banda passante con l'infrastruttura di rete esterna, garantendo così la trasmissione fluida dei dati, e di fungere da nodo centrale per la comunicazione interna al veicolo. In particolare, il router è preposto a interconnettere il calcolatore di bordo, deputato all'elaborazione dei dati provenienti dai vari sensori, con il sensore Lidar, il quale, mediante interfaccia Ethernet, trasmette ingenti volumi di dati destinati al calcolatore. Il router

scelto per il progetto è il modello GL-AR750S di marca gl.inet. Alimentato tramite porta USB direttamente sul rover, questo ruter è noto per le sue piccole dimensioni e il suo basso consumo energetico

Figura 2.4: Router scelto per la piattaforma di sviluppo

Capitolo 3.0

Middleware di comunicazione

In questa sezione si discute dei principali componenti utilizzati per l'implementazione del middleware di comunicazione

3.1 ROS

Il Robotic Operating System (ROS) rappresenta un framework software di riferimento nel panorama della robotica. Si tratta di una suite di librerie e strumenti open-source progettata per semplificare e accelerare lo sviluppo di applicazioni complesse destinate a sistemi robotici di varia natura. ROS offre un'infrastruttura flessibile e modulare che consente agli sviluppatori di creare, testare e distribuire software in modo efficiente.

La scelta della versione di ROS è strettamente correlata alla distribuzione Linux utilizzata. In ambito robotico, Ubuntu è il sistema operativo più diffuso grazie alla sua stabilità e al supporto a lungo termine. Nel contesto del presente lavoro è stata selezionata la distribuzione Ubuntu 20.04 Focal Fossa, in quanto rappresenta l'ultima versione LTS (Long Term Support) compatibile con la scheda grafica GPGPU adottata come computer di bordo.

In linea con questa scelta, è stata adottata la versione ROS2 Foxy Fitzroy che, rispetto alla precedente generazione ROS1 (Noetic Ninjemys), introduce una serie di miglioramenti significativi, tra cui:

- Supporto multi-linguaggio: ROS2 offre un supporto più ampio per diversi linguaggi di programmazione, tra cui C++ e Python. Tuttavia, nel caso specifico di questo progetto, si è optato per l'utilizzo esclusivo del C++ al fine di ottenere un controllo più granulare sul codice e sfruttare le elevate prestazioni offerte da questo linguaggio

- Modernizzazione delle librerie: le librerie di ROS2 sono state aggiornate e ampliate, offrendo funzionalità più avanzate e una maggiore integrazione con altri strumenti software
- Architettura distribuita e scalabile: ROS2 è progettato per gestire sistemi robotici complessi e distribuiti, con una maggiore attenzione alla scalabilità e alla resilienza

Nel corso di questa tesi l'implementazione del codice sarà interamente svolta in linguaggio C++. Questa decisione è stata guidata da diversi fattori:

- Controllo a basso livello: il C++ consente un controllo più preciso sulle risorse hardware e software del sistema, permettendo di ottimizzare le prestazioni e di implementare algoritmi complessi in modo efficiente
- Prestazioni elevate: le elevate prestazioni che il C++ può fornire sono fondamentali in applicazioni robotiche real-time dove la latenza e la capacità di risposta del sistema sono critiche
- Ampia diffusione e supporto: data la sua vicinanza all'hardware, il C++ è il linguaggio di programmazione più utilizzato nell'ambito della robotica, in quanto permette di avere molteplici librerie e strumenti a disposizione, oltre che ad una maggiore facilità nella ricerca di soluzioni online a eventuali problemi

La suite ROS, oltre a fornire le componenti software di base, offre un ampio ventaglio di strumenti per lo sviluppo e il debugging. Tra questi, ROS-Visualization (RViz) riveste un ruolo cruciale, fornendo una potente interfaccia di visualizzazione tridimensionale. Progettato specificatamente per operare all'interno dell'ecosistema ROS, RViz consente agli sviluppatori di visualizzare in modo intuitivo e dettagliato i dati provenienti dai sensori e dagli attuatori del robot. Durante la fase di sviluppo, RViz è stato utilizzato in modo estensivo per la visualizzazione dei dati di telemetria, facilitando la comprensione del funzionamento del sistema e l'individuazione di eventuali anomalie. La sua capacità di rappresentare in modo grafico le informazioni provenienti da diversi nodi del sistema ha reso RViz uno strumento indispensabile per la messa a punto e la validazione degli algoritmi di localizzazione.

3.1.1 Nodi

Uno degli aspetti fondamentali di ROS è la sua architettura basata su nodi, che rappresentano la singola unità di esecuzione all'interno del sistema. Un nodo può essere responsabile di una vasta gamma di funzioni, tra cui eseguire calcoli, interfacciarsi con dispositivi hardware, raccogliere dati dai sensori, e molto altro. Tuttavia, la caratteristica più distintiva di un nodo ROS è la sua capacità di comunicare in maniera integrata con altri nodi attraverso un sistema di messaggistica distribuita. Questo sistema consente ai nodi di scambiare informazioni in tempo reale, permettendo una coordinazione precisa e affidabile tra i diversi componenti di un robot.

Questa struttura modulare e comunicativa rende possibile la rappresentazione di ciascuna funzione operativa del robot come un nodo distinto, favorendo una chiara

separazione delle responsabilità e una maggiore facilità di sviluppo e manutenzione. Ad esempio, lo stack software utilizzato per il controllo autonomo del rover all'interno di questo progetto è costituito da una serie di nodi ROS, ognuno dei quali svolge un ruolo specifico e critico nel funzionamento complessivo del sistema.

Si può dunque concepire lo stack come un sistema in cui ciascun nodo è deputato a una specifica funzione. Ad esempio, un nodo gestisce esclusivamente la localizzazione, un altro è interamente dedicato alla trasmissione dei dati da ROS all'interfaccia di controllo del veicolo, e un terzo calcola la traiettoria. Questa suddivisione modulare consente una chiara separazione delle responsabilità, favorendo una maggiore manutenibilità e flessibilità del sistema e mentre ciascun nodo è responsabile di un sottoinsieme ben definito delle funzionalità, essi sono interconnessi e si scambiano attivamente le informazioni necessarie per coordinare le loro azioni.

3.1.2 Comunicazione tra nodi

La comunicazione tra nodi ROS è basata su 3 elementi fondamentali:

- Topic: il protocollo utilizzato da ROS è di tipo publish/subscribe, il quale prevede che durante l'esecuzione dei nodi si vadano a creare dei topic, ovvero stringhe che utilizzano come separatore il carattere '/' e che permettono di smistare i diversi dati da inviare e/o ricevere. Un esempio sono le scan lidar che vengono pubblicate sul topic "/scan". Ogni nodo può decidere sottoscriversi (subscribe) ad un certo topic, ovvero ricevere tutti i dati inviati attraverso esso, pubblicare dati (publish), ovvero pubblicare dati su di esso, o se semplicemente ignorarlo
- Message type: una volta scelto il topic è necessario stabilire la tipologia di informazioni che verranno trasmesse attraverso lo stesso. ROS fornisce diversi tipi di dato pubblicabile su un singolo topic. Un esempio è il tipo di dato utilizzato dal particle filter e dall'odometria del mezzo, ovvero "Odometry messages", che descrivono la posizione e il movimento (o meglio l'odometria) di un oggetto nello spazio. Il messaggio specifico per l'odometria fornito da ROS è strutturanto nel seguente modo:

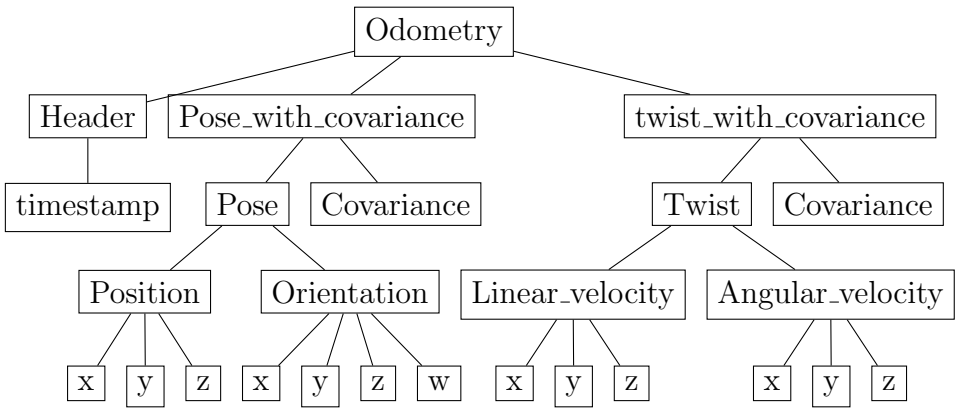

Figura 3.1: Struttura del messaggio ROS "Odometry"

In questo particolare messaggio, come si può notare, sono contenuti: la posa, ovvero la posizione e l'orientamento del veicolo rispetto alla posa di partenza ed il Twist, ovvero la velocità lineare e quella angolare dell'oggetto al momento della misura. Un altro esempio di messaggio ROS è l'ackermann message, che fornisce come dati principali una velocità ed un angolo di sterzo e viene utilizzato per comunicare al robot il movimento da compiere. Tutti i tipi di messaggio sono consultabili online sulla documentazione di ROS

- Content: è il dato che dobbiamo inviare e che deve essere incapsulato nel tipo di dato fornitoci da ROS

3.2 MQTT

Il protocollo Message Queuing Telemetry Transport (MQTT) è un protocollo di rete di tipo publish/subscribe, progettato per la trasmissione di messaggi tra dispositivi in ambienti caratterizzati da connessioni di rete con larghezza di banda limitata, latenza elevata, o affidabilità intermittente.

Le principali caratteristiche del protocollo MQTT includono:

- **Efficienza nella larghezza di banda**: MQTT è progettato per minimizzare l'overhead di rete, il che lo rende particolarmente adatto per applicazioni in cui la larghezza di banda è limitata o costosa
- **Affidabilità e livelli di qualità del servizio (QoS)**: MQTT offre tre livelli di QoS, che consentono di bilanciare la necessità di affidabilità con le risorse disponibili. I livelli QoS vanno da "almeno una volta" a "esattamente una volta", garantendo diversi gradi di consegna del messaggio in base ai requisiti dell'applicazione
- **Supporto per la persistenza delle sessioni**: I client MQTT possono disconnettersi e riconnettersi senza perdere i messaggi inviati durante la disconnessione, grazie alla capacità del broker di mantenere lo stato delle sessioni e gestire i messaggi pendenti
- **Sicurezza**: MQTT può essere configurato per utilizzare connessioni cifrate (SSL/TLS) e supporta l'autenticazione tramite username e password, garantendo la protezione dei dati scambiati e l'accesso controllato alle risorse
- **Scalabilità**: La natura leggera e la flessibilità del modello publish-subscribe rendono MQTT altamente scalabile, consentendo di supportare un gran numero di dispositivi e applicazioni con un impatto minimo sulle risorse di rete

Grazie a queste caratteristiche, MQTT è ampiamente utilizzato in una vasta gamma di applicazioni, tra cui la telemetria industriale, il monitoraggio ambientale, le smart cities, l'automazione domestica e i sistemi di gestione energetica, rappresentando una soluzione robusta ed efficiente per la comunicazione tra dispositivi eterogenei in contesti IoT.

Per l'implementazione dello stack MQTT ci si avvale, all'interno del progetto, della libreria paho.mqtt.cpp[5], una libreria open source fornita dalla eclipse foundation.

3.2.1 Infrastruttura

Il protocollo MQTT opera secondo un'architettura client-server, dove i client (dispositivi o applicazioni) si connettono a un server centrale (broker) che gestisce la distribuzione dei messaggi. I client che desiderano inviare dati pubblicano messaggi su specifici argomenti (topics), mentre i client interessati a ricevere quei dati si iscrivono (subscribe) agli stessi argomenti. Il broker, che agisce come intermediario,

si occupa di ricevere i messaggi pubblicati e di inoltrarli a tutti i client iscritti agli argomenti corrispondenti.

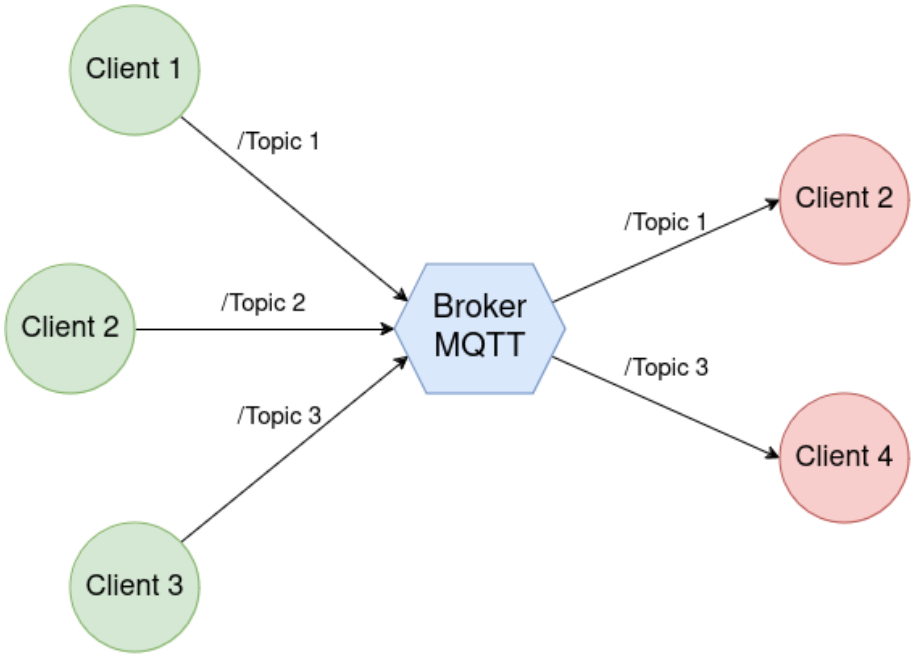

Figura 3.2: Struttura generica di un sistema di omunicazione basato su MQTT

Nel corso della presente tesi, lo stack MQTT viene implementato per mettere in comunicazione l'intero stack di guida remota, che si compone di 3 diversi agenti al quale ci si riferirà con la seguente nomenclatura:

- "veicolo" per indicare il rover AgileX
- "server" per indicare il calcolatore incaricato di ricevere i dati dal mezzo e a cui inviare i risultati (ovvero i messaggi di controllo)
- "broker" per indicare il broker MQTT

Siccome i dati provenienti dal veicolo sono formattati secondo i tipi di dato ROS, per implementare la comunicazione tramite lo stack MQTT sarà necesasrio convertire questi dati in un formato compatibile con MQTT.

3.2.2 Formattazione messaggi

I messaggi scambiati tramite protocollo MQTT si compongono fondamentlamente di stringhe di testo. È quindi necessario utilizzare una formattazione per il testo che renda possibile distinguere i vari campi di un messaggio ROS (la cui struttura è illustrata nella sezione precedente) che vogliamo inoltrare. Per questa motivazione si è deciso di avvalersi del formato JSON.

Per fare un esempio, si illustra di seguito come un messaggio di odometria (illustrato nella sezione precedente) si presenterà in formato JSON:

```
{
  "header":{
    "timestamp":{
      "sec": 0,
      "nanosec": 0,
    }
  }
  "pose_with_covariance":{
    "pose":{
      "position":{
        "x": 0,
        "y": 0,
        "z": 0
      },
      "orientation":{
        "x": 0,
        "y": 0,
        "z": 0,
        "w": 0
      }
    },
    "covariance":{
      [...]
    }
  }
  "twist_with_covariance":{
    "twist":{
      "linear_velocity":{
        "x": 0,
        "y": 0,
        "z": 0
      },
      "angular_velocity":{
        "x": 0,
        "y": 0,
        "z": 0
      }
    },
    "covariance":{
      [...]
    }
  }
}
```

Come si può vedere, grazie a questa struttura, è possibile rappresentare fedelmente i dati riportati dal messaggio ROS.

3.2.3 Topic

In MQTT, come in ROS, per gestire la comunicazione si utilizzano i topic, motivo per cui si è deciso di utilizzare MQTT per il controllo remoto. Questo fa si che per mettere in comunicazione i due sia sufficiente, una volta ricevuto il dato in ROS, generare l'analogo nel formato MQTT e viceversa.

È inoltre utile rendere nota la diversa struttura delle due tipologie di topic. Infatti, se per i topic ROS è utile utilizzare solo pochi identificatori, alle volte di una sola parola (eg. */drive_parameters*, */scan*, */odometry*) dato che tutto il traffico ROS è presente solo all'interno del computer di bordo, per i topic MQTT è invece necessario utilizzare topic più lunghi, comprendenti diversi campi.

Per fare un esempio, la stringa "/hipert/vehicle/rover_1234/telemetry/odometry" illustra il topic utilizzato nell'odometria, i cui campi indicano:

- **/hipert**: è il campo che identifica il laboratorio che sta svolgendo la comunicazione, ed è utile in quanto se lo stesso broker MQTT è utilizzato da più laboratori abbiamo un filtro che ci permette di avere solo i dati rilevanti al nostro campo di interesse
- **/vehicle**: identifica il tipo di device osservato
- **/rover_1234**: è l'effettiva stringa ID del topic, ed è utile per dare un identificatore univoco per il veicolo osservato. Nel caso in cui siavesse più di un veicolo osservato, è necessario sapere quale esattamente di questi si sta prendendo in considerazione
- **/telemetry**: identifica il tipo di dato preso in considerazione, che in questo caso specifico possono essere del tipo **telemetry** o **control**
- **/odometry**: indica il dato osservato che, in questo caso, è l'odometria del veicolo

Capitolo 4.0

Scelte progettuali

Una delle peculiarità di ROS è che non limita la comunicazione tra i nodi al solo device che esegue l'istanza, ma può distribuire i pacchetti pubblicati dai nodi anche tramite rete, avvalendosi del protocollo UDP.

Questo significa che per implementare la comunicazione necessaria al funzionamento della guida remota sarebbe stato suficiente sfruttare questa peculiarità di ROS. Si riporta nelle seguenti sezioni le motivazioni per le quali si è deciso di utilizzare invece lo stack MQTT.

4.1 Affidabilità

Come descritto precedentemente, il framework ROS prevede l'esclusivo utilizzo del protocollo UDP per l'invio di messaggi in rete. Il protocollo UDP è un protocollo a livello trasporto molto veloce e versatile, e viene soprattutto utilizzato in applicazioni orientate a minimizzare l'overhead e la latenza. Tuttavia, questa scelta comporta alcuni compromessi.

Essendo un protocollo connectionless, il protocollo UDP non garantisce la consegna dei pacchetti né il loro ordine di arrivo. Inoltre, non fornisce meccanismi di controllo dell'errore come la ritrasmissione automatica dei pacchetti persi. Di conseguenza, in ambienti con elevata interferenza o congestione di rete, si possono verificare perdite di dati e una degradazione della qualità del servizio.

Questa caratteristica del protocollo non ne permette un'utilizzo affidabile in casi critici come quello di studio, in quanto una perdita di pacchetti potrebbe molto facilmente scalare in potenziali danni al veicolo, persone od oggetti terzi.

Al contrario, il protocollo MQTT si avvale del protocollo TCP a livello di trasporto. Questa scelta progettuale offre una serie di vantaggi che lo rendono particolarmente adatto per applicazioni IoT e di messaggistica in generale.

Il protocollo TCP garantisce infatti la consegna ordinata e affidabile dei messaggi, riducendo al minimo il rischio di perdite di dati. Questa caratteristica è fondamentale in scenari come quello studiato, dove l'integrità dei dati è cruciale. Questa garanzia ci viene fornita da diverse tecniche che TCP implementa come i meccanismi di acknowledge e timeout dei pacchetti.

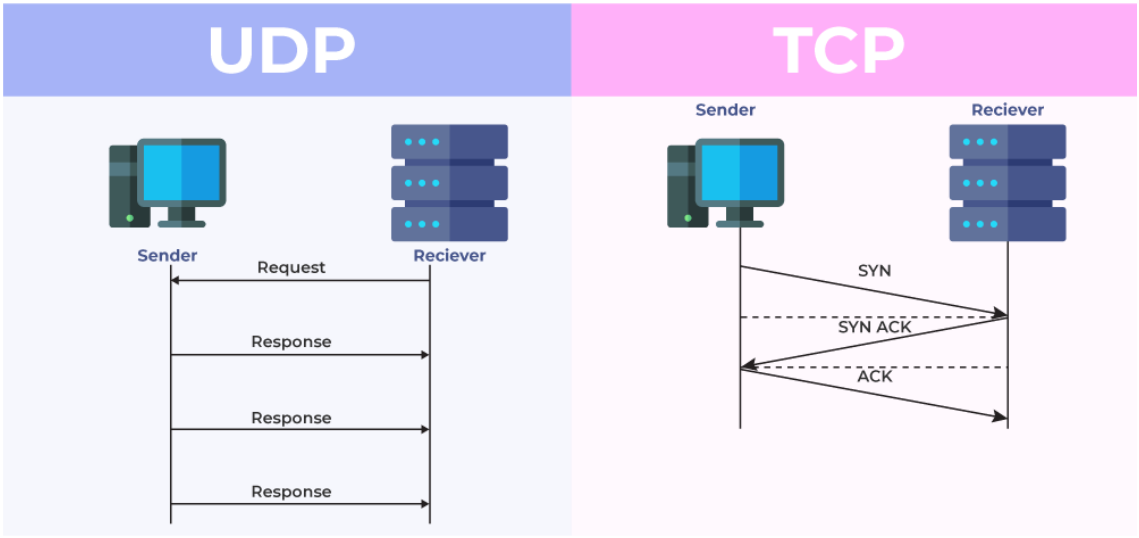

Figura 4.1: Comparazione graficata tra i 2 protocolli. Immagine tratta da [1]

Come mostrato in figura 4.1, il protocollo TCP implementa infatti meccanismi di controllo del flusso che evitano la congestione della rete, garantendo una comunicazione efficiente anche in condizioni di carico elevato.

Tuttavia, il protocollo TCP è un protocollo sensibilmente più "pesante" e lento rispetto al protocollo UDP. Nonostante questo, la latenza introdotta dall'uso di questo protocollo risulta accettabile per il funzionamento del sistema e si è deciso di sfruttare il protocollo MQTT per l'affidabilità nella trasmissione. Questa scelta è stata dettata dalla necessità di garantire la consegna dei messaggi in modo sicuro e ordinato, anche in condizioni di rete instabili.

4.2 Sicurezza

Nell'ambito dell'analisi di scenari di rete pubblica come il nostro, l'aspetto della sicurezza riveste un'importanza cruciale. In questo contesto, l'adozione di ROS per la guida remota potrebbe rivelarsi una scelta non ottimale.

ROS, nella sua configurazione standard, non prevede l'implementazione di meccanismi di sicurezza a livello di pacchetto. Di conseguenza, i dati scambiati tra i nodi della rete vengono trasmessi in chiaro, rendendoli potenzialmente accessibili

a qualsiasi entità connessa alla rete. Questa vulnerabilità espone i sistemi a rischi di intercettazione, manipolazione o alterazione dei dati, con potenziali conseguenze negative sulla privacy e sulla sicurezza operativa.

Al contrario, MQTT supporta nativamente lo strato di trasporto sicuro SSL/TLS, che fornisce una crittografia end-to-end dei dati scambiati tra i dispositivi. Questo significa che anche in caso di intercettazione delle comunicazioni, i dati rimarranno incomprensibili agli esterni.

4.3 Struttura

Nell'ambito della guida remota, la resilienza del sistema è un fattore critico. Un'infrastruttura deve essere in grado di operare in modo affidabile anche in condizioni avverse, garantendo la continuità del servizio e la sicurezza dei dati. In questo contesto, MQTT dimostra una maggiore resilienza rispetto a ROS.

MQTT è progettato per operare in ambienti distribuiti, con molti dispositivi connessi a un broker centrale. Questa architettura rende il sistema più resistente a guasti locali, poiché la perdita di un singolo dispositivo o di una connessione non compromette necessariamente l'intero sistema.

Al contrario, ROS spesso si basa su un unico organo, chiamato *ROS Master*, che coordina tutte le comunicazioni. La perdita o l'irraggiungibilità di questo organo può causare il blocco dell'intero sistema.

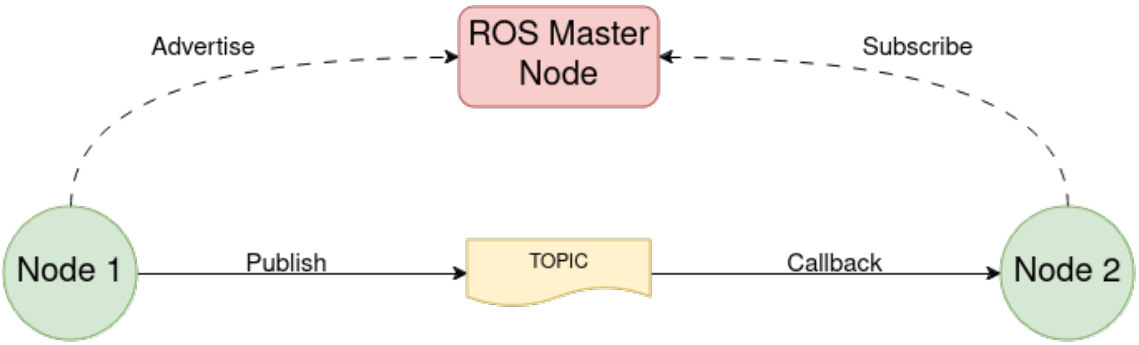

Figura 4.2: Struttura del sistema ROS

4.4 Controllo del QoS

Un ulteriore punto da considerare è quella della gestione della Quality Of Service (QoS). Il concetto di QoS indica il livello di garanzia che una rete offre per quanto riguarda la consegna dei dati. Una bassa QoS indica che i pacchetti non vengono consegnati correttamente o che addirittura non è certa la ricezione di questi messaggi.

Entrambi i protocolli offrono meccanismi di gestione della QoS, nelle seguenti modalità.

MQTT offre tre livelli di gestione (0, 1, 2) che consentono di adattare il livello di affidabilità della rete. Tutti e tre i livelli si basano sul come e quante volte il messaggio deve essere riinviato perchè questo possa essere considerato ricevuto.

- QoS 0 (At most once): il messaggio viene consegnato al massimo una volta. Non c'è alcuna garanzia di consegna, e il messaggio potrebbe perdersi. È il livello più veloce ma meno affidabile
- QoS 1 (At least once): il messaggio viene consegnato almeno una volta. Il broker invia un messaggio di conferma al publisher, e se non riceve conferma, reinvia il messaggio. Questo livello garantisce che il messaggio arrivi, ma potrebbe arrivare più di una volta
- QoS 2 (Exactly once): il messaggio viene consegnato esattamente una volta. Il broker invia un messaggio di conferma al publisher, e solo dopo aver ricevuto questa conferma, considera il messaggio consegnato. Questo è il livello più affidabile ma anche il più lento

Diversamente, ROS non offre un meccanismo di QoS così definito e flessibile come MQTT. La gestione della qualità del servizio in ROS è più legata alla configurazione dei nodi e dei topic che, spesso, richiede un'implementazione personalizzata per garantire un livello di affidabilità specifico.

Le possibili configurazioni che è possibile fare sui nodi ROS sono:

- History: definisce la quantità di dati che possono essere memorizzati in un buffer prima che vengano scartati.
- Depth: indica la dimensione massima del buffer.
- Reliability: determina il livello di affidabilità della comunicazione (best-effort, reliable, etc.).
- Durability: specifica se i dati devono essere persistenti anche se i nodi non sono connessi.
- Liveliness: definisce come spesso un nodo deve dimostrare di essere attivo.

Capitolo 5.0

Funzionamento

Nella seguente sezione si descrive il funzionamento del veicolo, degli algoritmi utilizzati e dei nodi realizzati.

5.1 Stack di guida autonoma

La prima cosa che si analizza è il funzionamento dello stack di guida autonoma. Come descritto nel capitolo 1, questo stack funziona grazie ai processi di perception, planning e control.

5.1.1 Perception

La parte di perception viene implementata avvalendosi delle informazioni del sensore Lidar e mediante la realizzazione dell'algoritmo di localizzazione, concretizzato nei seguenti nodi:

- **urg_node**: è il nodo che permette di pubblicare sul topic ROS */scan* le point-cloud rilevate dal sensore Lidar. Il tipo di dato utilizzato è chiamato *LaserScan* e fornisce la serie di distanze rilevate da Lidar.
- **hunter_ros2_node**: fornisce un'interfaccia con il veicolo, restituendo l'odometria calcolata a partire dal movimento delle ruote e, come vedremo successivamente, permettendo il comando dello stesso. Dal punto di vista della perception, quello che interessa è l'odometria, pubblicata sul topic */odometry* sottoforma di dato *Odomentry*.
- **particle_filter**: implementa un algoritmo di localizzazione basato sull'utilizzo di un particle filter e un algoritmo di ray casting. Il suo funzionamento sfrutta la mappa dell'ambiente in cui il robot si sta muovendo, l'odometria del veicolo

e la pointcloud del sensore Lidar per il calcolo della posizione. Questo dato viene poi pubblicato sul topic */pf/position* sottoforma di dato *Odometry*.

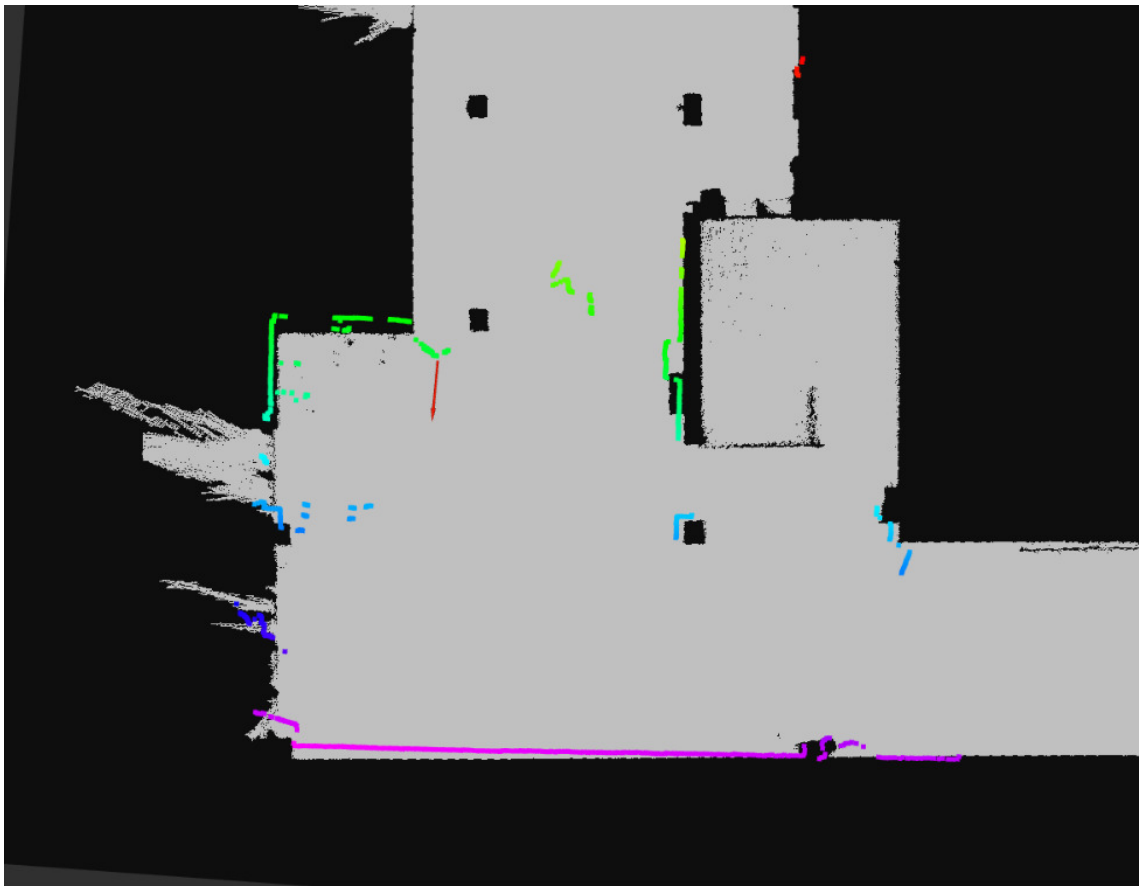

Figura 5.1: Dimostrazione funzionamento del sensore Lidar

Figura 5.1 llustra una breve dimostrazione del funzionamento del sensore Lidar. Nello specifico si riporta una porzione di mappa (grigia su sfondo nero) e la pointcloud generata dal sensore Lidar (punti colorati). Il colore dei punti non è casuale, ma è bensì un metodo intuitivo per mostrare graficamente la distanza di quel particolare punto dalla posizione calcolata del robot (freccia rossa).

5.1.2 Planning

La parte di planning si avvale di due nodi:

- **path_logger**: permette la registrazione di un percorso quando il veicolo viene guidato manualmente. Il percorso registrato viene poi salvato in un file apposito.
- **path_publisher**: questo nodo si occupa di pubblicare un percorso preregistrato o precalcolato da seguire, pubblicato sul topic */path*.

5.1.3 Control

Si passa infine a descrivere il funzionamento della parte di controllo, composta anch'essa da due nodi:

- **purepursuit**: si occupa di ricevere il percorso pubblicato sul topic */path* e, a partire dalla posizione pubblicata dal nodo **particle_filter**, di calcolare i comandi da impartire al veicolo. I comandi vengono pubblicati sul topic */drive_parameters* e sono di tipo *Ackermann Stamped*, un tipo di messaggio ROS che incapsula il timestamp, l'angolo di sterzo e la velocità desiderata
- **hunter_ros2_node**: come descritto in precedenza, questo nodo, oltre a fornire l'odometria del mezzo, è anche capace di ricevere i comandi da impartire al veicolo. Il nodo è infatti in perenne ascolto sul topic */drive_parameters* e ad ogni messaggio comunicherà con l'interfaccia CAN del veicolo impartendogli la velocità e l'angolo di sterzo desiderati

Di seguito uno schema riassuntivo del funzionamento della guida autonoma:

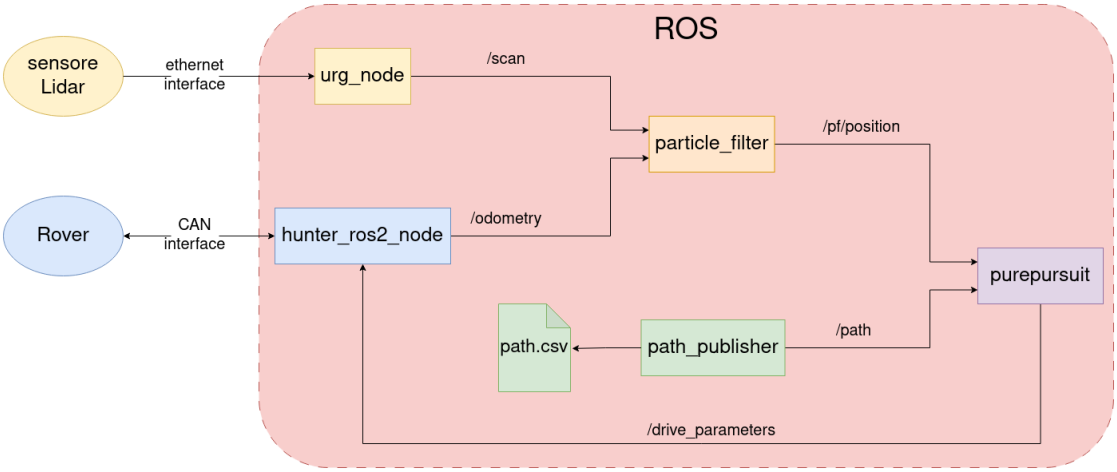

Figura 5.2: Schema rissuntivo dello stack di guida autonoma

5.2 Guida remota

Una volta illustrato e compreso il funzionamento dello stack di guida autonoma, si passa alla descrizione di quella remota.

La prima scelta tecnica è stata quella di decidere quale parte dello stack spostare in remoto. Per esempio, si potrebbe decidere di svolgere solo la parte di planning da remoto e lasciare in resto in locale, o diversamente, portare sin remoto solo la perception. Si potrebbe anche decidere di far eseguire solo specifici nodi da remoto e lasciare in resto in locale.

Nella presente tesi si è scelto di portare in remoto quasi tutto lo stack, lasciando in locale solo i nodi che hanno strettamente bisogno dell'interfacciamento con l'hardware.

Nello specifico, gli unici nodi che rimarranno in locale saranno:

- **urg_node**: che sarà necessario per ricavare i dati dal sensore Lidar
- **hunter_ros2_node**: necessario per ricavare l'odometria del mezzo e per inviare i comandi all'interfaccia CAN

Tutto il resto sarà gestito da remoto. Questo permette di poter scegliere con più flessibilità in quale modo pilotare il rover. Si potrà infatti decidere sia di eseguire l'intero stack, senza modifiche, sulla macchina in remoto e di conseguenza inviare i comandi calcolati al mezzo, sia di poter guidare il veicolo completamente in manuale da un apposito operatore e di inviare solo i comandi scelti da quest'ultimo al veicolo.

Di seguito uno schema riassuntivo del funzionamento della guida remota:

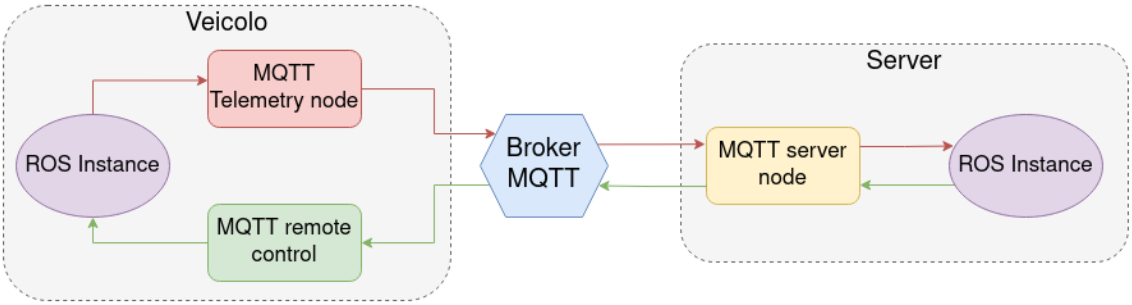

Figura 5.3: Schema rissuntivo dello stack di guida remota

5.3 Nodi sviluppati

Oltre ai nodi già compresi nello stack di guida autonoma, è stato necessario sviluppare i nodi che permettessero lo scambio di informazioni tra il veicolo e il server. Nello specifico, i 3 nodi sono:

- **mqtt_telemetry_node**: ha il compito di inviare i dati dei sensori dal veicolo al server
- **mqtt_control_node**: ha il compito di ricevere i dati di controllo dal server per poi pubblicarli su ROS
- **mqtt_server_node**: eseguito sul server, avrà il compito di ricevere sia i dati di telemetria da MQTT per poi ripubblicarli su ROS che ricevere i dati di controllo da ROS per poi ripubblicarli su MQTT

Ogni nodo è diviso in più classi ed un unico main file, che gestirà l'intero flusso di esecuzione del nodo. All'interno del progetto che contiene il nodo sono anche compresi dei file di configurazione in formato yaml, dai quali il nodo andrà a leggere informazioni necessarie per l'esecuzione del processo. Di seguito un esempio di file

di configurazione utilizzato per definire alcuni parametri relativi alla comunicazione MQTT.

```
BROKER_ADDRESS: "127.0.0.1"
BROKER_PORT: 1883
USERNAME: ""
PASSWORD: ""
VEHICLE_ID: "rover"

BASIC_TOPIC: "/hipert"
VEHICLE_TOPIC: "vehicle"
VEHICLE_TELEMETRY_TOPIC: "telemetry"
VEHICLE_INFO_TOPIC: "info"
VEHICLE_POSITION_TOPIC: "position"
VEHICLE_DRIVE_PARAMETERS_TOPIC: "drive_parameters"
VEHICLE_ODOMETRY_TOPIC: "odometry"
VEHICLE_SCAN_TOPIC: "scan"


CLEAN_SESSION: true
QOS: 0
```

È inoltre utile specificare che l'intera codebase è scritta in linguaggio C++, che risulta utile qualora sia necessario avere bassi tempi di esecuzione.

5.3.1 MQTT telemery node

La prima fase del progetto ha riguardato lo sviluppo di un componente software, ovvero un nodo ROS, in grado di interfacciarsi con i sensori del veicolo. Questo nodo, una volta configurato per sottoscriversi ai topic di interesse, è in grado di acquisire i dati provenienti dai sensori e di trasformarli in un formato adatto alla trasmissione. La scelta del formato JSON, ampiamente utilizzato per lo scambio di dati tra sistemi eterogenei, è stata dettata dalla sua leggibilità e dalla sua facilità di parsing. I dati, una volta formattati, vengono inviati al server MQTT.

I topic a cui il nodo effettua una subscribe sono:

- */scan*: per la ricezione della pointcloud rilevata dal sensore Lidar
- */odometry*: per la ricezione dei dati di odometria del mezzo

Una volta acquisito il dato grezzo, esso viene convertito in una rappresentazione strutturata e leggibile, che nello specifico si compone di una stringa in formato JSON. Tale stringa, contenente l'insieme completo dei dati costituenti il messaggio, viene quindi immessa all'interno di una cache dedicata.

La cache funge da deposito temporaneo, accumulando le stringhe JSON generate fino al raggiungimento di una determinata soglia (o intervallo) di tempo predefinito e impostabile tramite file di configurazione. Al verificarsi di tali condizioni, il contenuto

completo della cache viene trasmesso in un'unica operazione al server MQTT. Questa modalità di trasmissione, basata su un meccanismo di invio periodico, consente di ottimizzare le comunicazioni e ridurre il carico sulla rete.

Per quanto concerne la gestione concorrente di queste operazioni, si introduce il concetto di thread. Un thread può essere definito come un flusso di esecuzione autonomo all'interno di un processo. In altre parole, rappresenta una singola sequenza di istruzioni che può essere eseguita in parallelo rispetto ad altre sequenze, all'interno dello stesso programma.

Nel contesto descritto, è possibile impiegare un thread dedicato per gestire l'operazione di trasmissione dei dati. Tale thread opererà in modo concorrente rispetto al thread principale (main thread) consentendo di eseguire contemporaneamente altre attività e di migliorare la reattività dell'applicazione. Per fare questo è però necessario gestire la cache in modo che i 2 thread che ne fanno utilizzo non vadano in conflitto per accedere alla risorsa in quanto, se ciò non fosse gestito, si andrebbe ad incappare in problemi come, ad esempio, la lettura di un dato incorretto da parte del thread di invio dei messaggi o una scrittura parziale da parte del main thread.

Per la gestione di questi eventi si è dunque deciso di utilizzare una struttura chiamata mutex, che permette ad un thread di bloccare una risorsa per utilizzarla ed ad un altro di aspettare che la risorsa si liberi per poterla utilizzare. Di seguito l'implementazione della classe **msgs_cache**:

```
1  MsgsCache::MsgsCache(int history_length){
2      this->history_length = history_length;
3  }
4
5  void MsgsCache::addTopic(std::string topic){
6      topicList.push_back(topic);
7  }
8
9  void MsgsCache::setTopicList(std::vector<std::string>
       topic_list){
10     this->topicList = topic_list;
11 }
12
13 std::vector<std::string> MsgsCache::getTopics(){
14     return topicList;
15 }
16
17 void MsgsCache::addMsg(std::string topic, std::string
       message){
18     this->mutex.lock();
19     if(std::count(this->topicList.begin(), this->
           topicList.end(), topic)<1)
20         this->topicList.push_back(topic);
21
```

```
22      if(this->cache[topic].size() >= this->history_length)
            {
23        this->cache[topic].pop_front();
24      }
25      this->cache[topic].push_back(message);
26      this->mutex.unlock();
27  }
28
29  std::string MsgsCache::getLastMsg(std::string topic){
30      std::string return_value("");
31      if(this->cache[topic].size() > 0){
32        return_value = this->cache[topic].front();
33      }
34      return return_value;
35  }
36
37  std::string MsgsCache::popLastMsg(std::string topic){
38      this->mutex.lock();
39
40      std::string msg = this->getLastMsg(topic);
41      if(this->cache[topic].size() > 1){
42          this->cache[topic].pop_front();
43      }
44
45      this->mutex.unlock();
46
47      return msg;
48  }
49  int MsgsCache::getCacheSize(std::string topic){
50    return this->cache[topic].size();
51  }
```

Il nodo è suddiviso in 6 classi:

- **conf_loader**: si occupa di caricare i dati dai file di configurazione prima citati. Nello specifico, questa classe è stata implementata per rendere il main thread e il resto del processo indipendente dal formato dei file di configurazione
- **mqtt_publisher**: è una classe che contiene metodi utili all'invio di stringhe tramite protocollo MQTT, avvalendosi della libreria paho.mqtt.cpp fornita dalla eclipse foundation
- **mqtt_telemetry_node**: questa classe è quella che implementa il nodo ROS che si incaricherà di ricevere i dati utili alla telemetria
- **msgs_cache**: questa classe implementa una semplice struttura dati che accoppia ad ogni topic ROS (rappresentata come stringa), una stringa JSON contente i dati di telemetria da inviare

- **msgs_to_string**: è un insieme di funzioni statiche che permette la traduzione da messaggi ROS a stringhe JSON
- **topic_manager**: è la classe designata ad accoppiare i topic ROS con i rispettivi topic MQTT, il cui codice è incluso nella sezione 5.4

5.3.2 MQTT remote control

Successivamente, è fondamentale predisporre un meccanismo che consenta al veicolo di ricevere i comandi di controllo. A tal fine, è stato implementato un nodo dedicato che si sottoscrive al topic MQTT specificamente designato per la trasmissione di tali dati. Successivamente, il nodo ripubblica le informazioni ricevute sul topic ROS */drive_parameters*.

Poiché i dati trasmessi tramite MQTT sono di tipo stringa, il veicolo dovrà elaborare queste stringhe al fine di estrarre le informazioni pertinenti e popolarne i campi di un messaggio ROS conforme al tipo di dato previsto, esattamente come descritto nella sezione 4.3.

il nodo in questo caso è suddiviso in 4 classi:

- **conf_loader**: il cui funzionamento è riportato nella sezione 5.3.1
- **control_node**: è il nodo incaricato di ripubblicare sul topic giusto i dati relativi al controllo
- **json_to_ros2_msgs**: è il nodo incaricato alla conversione dei messaggi
- **mqtt_subscriber**: è il nodo che si occupa alla sottoscrizione al topic mqtt designato alla ricezione dei dati

In questo caso non è stato necessario l'utilizzo della classe **topic_manager** in quanto gli unici due topic in gioco (quello ROS e quello MQTT) sono entrambe impostabili da file di configurazione yaml.

5.3.3 MQTT server node

È stato infine implementato un nodo centrale, eseguito sul server, che funge da ponte di comunicazione tra il veicolo e l'istanza ROS del server stesso. Questo nodo è responsabile della ricezione di tutti i dati trasmessi dal veicolo tramite il protocollo MQTT e della loro successiva pubblicazione sul bus ROS. Contestualmente, il nodo si occupa di raccogliere i comandi di controllo generati all'interno dell'ambiente ROS, di incapsularli in un messaggio MQTT e di inoltrarlo al veicolo.

Il nodo in questo caso è suddiviso in 8 classi, alcune delle quali sono prese dai due nodi prima implementati:

- **conf_loader**: il cui compito è riportato nella sezione 5.3.1
- **mqtt_publisher**: il cui compito è riportato nella sezione 5.3.1
- **mqtt_subscriber**: descritto nella sezione 5.3.2
- **msgs_cache**: anch'esso descritto alla sezione 5.3.1

- **msgs_to_string**: riportato alla sezione 5.3.1
- **ros_server_node**: implementa il client ROS che andrà poi ad iscriversi e a pubblicare sui topic necessari
- **string_to_msgs**: al contrario di **msgs_to_string**, questa classe è incaricata di convertire le stringhe in formato JSON in messaggi ROS
- **topic_manager**: già riportato alla sezione 5.3.1

5.4 Gestione dei topic

Come descritto nella sezione 3.2.3, i topic ROS utilizzati hanno una struttura sostanzialmente diversa da quelli MQTT. Ai fini del progetto è stata dunque sviluppata una classe chiamata *topic_manager* che semplifica la gestione di questi due tipi di dati. La classe comprende 4 semplici metodi, 2 che riguardano il get ed il set di topic ROS e 2 che riguardano il get e il set dei topic MQTT, ad ogni topic MQTT è associato uno ROS e viceversa, di seguito i 4 metodi:

```
1  void TopicManager::add_ros_topic(std::string ros_topic,
       bool telemetry_topic){
2      // add ros topic and generate the corresponding mqtt
         topic
3      std::stringstream ss;
4
5      std::stringstream ros_topic_ss(ros_topic);
6      std::string last_word;
7      while(std::getline(ros_topic_ss, last_word, '/')){}
8
9      ss << this->loader->get_mqtt_parameters("BASIC_TOPIC"
         );
10     ss << "/" << this->loader->get_mqtt_parameters("
         VEHICLE_TOPIC");
11     ss << "/" << this->loader->get_mqtt_parameters("
         VEHICLE_ID");
12     if(telemetry_topic){
13         ss << "/" << this->loader->get_mqtt_parameters("
             VEHICLE_TELEMETRY_TOPIC");
14     }
15     ss << "/" << last_word;
16
17     this->topic_list[ss.str()] = ros_topic;
18 }
19
20 void TopicManager::add_mqtt_topic(std::string mqtt_topic)
       {
```

```
21 // add mqtt topic and generate the corresponding ros
   topic
22
23     std::stringstream ss;
24
25     std::stringstream ros_topic_ss(mqtt_topic);
26     std::string last_word;
27     while(std::getline(ros_topic_ss, last_word, '/')){}
28
29     this->topic_list[mqtt_topic] = ss.str();
30 }
31
32 std::string TopicManager::get_ros_topic_from_mqtt(std::
   string mqtt_topic){
33 //get ros topic corresponding to the passed mqtt topic
34     return this->topic_list[mqtt_topic];
35 }
36
37 std::string TopicManager::get_mqtt_topic_from_ros(std::
   string ros_topic){
38 //get mqtt topic corresponding to the passed ros topic
39     std::string mqtt_topic;
40
41     for (auto& it : this->topic_list) {
42         if (it.second == ros_topic) {
43             mqtt_topic = it.first;
44         }
45     }
46
47     return mqtt_topic;
48 }
```

Capitolo 6.0

Validazione sperimentale

In questa sezione si elencano i vari test che sono stati svolti, i problemi riscontrati e le soluzioni trovate.

6.1 Controllo

La prima fase sperimentale è stata dedicata alla verifica funzionale dei driver e del nodo ROS forniti da AgileX per il rover modello Hunter. L'obiettivo primario era determinare se tali componenti, specificamente progettati per il controllo e l'analisi del veicolo, potessero essere integrati nel sistema senza richiedere modifiche sostanziali o se, al contrario, fosse necessario apportare adattamenti o addirittura una completa riscrittura.

Per condurre questa valutazione, è stato sviluppato un nodo ROS dedicato alla ricezione dei dati da un joystick. Questi dati, dopo un'elaborazione preliminare, venivano convertiti in un angolo di sterzo e una velocità che, successivamente, venivano pubblicati sul topic ROS */drive_parameters* sotto forma di messaggi di tipo Ackermann. Tale configurazione consentiva di verificare direttamente:

- Compatibilità del nodo ROS AgileX: se il nodo ROS fornito dal produttore supportasse il formato dei messaggi Ackermann, comunemente utilizzato per il controllo di veicoli mobili
- Efficacia del driver: se il driver fosse in grado di tradurre correttamente i messaggi ROS in messaggi CAN, permettendo così al rover di eseguire i comandi impartiti

I primi test non sono andati a buon fine in quanto, dopo un'accurata analisi, si è riscontrato che il nodo ROS di casa AgileX utilizza un diverso formato di messaggi per il controllo del mezzo, interrompendo così lo stack.

La soluzione che si è deciso di adottare è stata di introdurre una modifica puntuale al nodo ROS AgileX. Tale modifica ha consentito al nodo di interpretare correttamente il tipo di messaggi previsto, ripristinando così la compatibilità con il resto del sistema

6.2 Mappatura

Il secondo test è stato quello di eseguire una mappatura bidimensionale di un intero ambiente.

Con mappatura bidimensionale si intende ricreare una vista dall'alto di un ambiente grazie all'utilizzo del sensore Lidar e dell'odometria del mezzo. Conoscendo infatti lo spostamento e la pointcloud rilevata dal sensore, tramite un apposito algoritmo è possibile ricreare questa mappa.

Per realizzare questa operazione, è stato impiegato un algoritmo di Simultaneous Localization and Mapping (SLAM). Come suggerisce il nome, lo SLAM è una tecnica che consente di localizzare un robot all'interno di un ambiente sconosciuto e, contemporaneamente, di costruire una mappa di tale ambiente. In questo caso specifico, l'algoritmo SLAM è stato implementato nel nodo ROS **slam_toolbox**, un pacchetto software open-source ampiamente utilizzato nella comunità robotica.

Dopo una fase di configurazione iniziale e alcune prove preliminari, è stato possibile generare una mappa bidimensionale accurata del primo piano dell'edificio di matematica del Dipartimento di Scienze Fisiche, Matematiche e Informatiche dell'Università di Modena e Reggio Emilia (UNIMORE). La mappa ottenuta rappresenta una fedele rappresentazione planimetrica dell'ambiente, evidenziando con precisione gli ostacoli presenti e le caratteristiche geometriche delle pareti

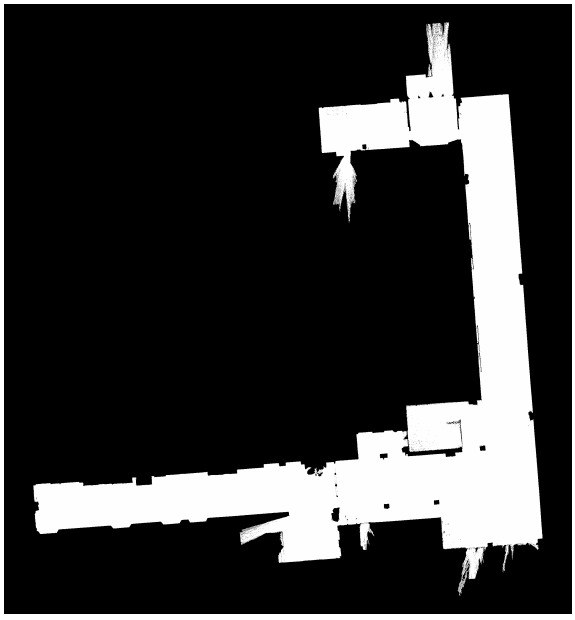

Figura 6.1: Mappa ottenuta tramite Lidar del primo piano dell'edificio Matematica

La mappa ottenuta grazie a questi test è riportata in figura 6.1.

6.3 Localizzazione

Il terzo test è stato uno dei più importanti, e riguarda la localizzazione.

Avendo una mappa dell'ambiente grazie ai test precedenti è stato infatti possibile svolgere test volti al calcolo preciso della propria posizione all'interno di un ambiente conosciuto. Questi test sono stati svolti grazie al nodo ROS **particle_filter** che implementa l'algoritmo di localizzazione già discusso alla sezione 5.1.1.

I test di localizzazione condotti hanno evidenziato alcune criticità legate alla natura dell'ambiente di prova. In particolare, sono state riscontrate difficoltà nel raggiungimento di una stima accurata della posizione in determinate aree dell'ambiente, caratterizzate da una scarsità di elementi distintivi (featureless). Tale condizione ha limitato la capacità del filtro a particelle di discriminare tra posizioni potenzialmente simili, compromettendo così la precisione della localizzazione.

Una soluzione a questo problema può essere sicuramente quella di utilizzare un Lidar più avanzato a tre dimensioni, in modo da apprezzare feature dell'ambiente che non sarebbero altrimenti rilevabili in due dimensioni, soluzione che sta venendo sperimentata al momento della stesura della presente tesi.

6.4 Guida autonoma

La penultima sperimentazione ha avuto come obiettivo la verifica delle prestazioni complessive del sistema di navigazione autonoma. A partire dalla mappa dell'ambiente generata precedentemente e dai parametri di controllo ottimizzati, è stato pianificato un percorso di riferimento. Successivamente, il robot è stato incaricato di seguire il percorso predefinito, utilizzando l'algoritmo di localizzazione per stimare la propria posizione in tempo reale e adattare la traiettoria in base alle informazioni sensoriali acquisite.

Sebbene i risultati ottenuti siano stati generalmente positivi, si sono manifestati alcuni problemi di deviazione dalla traiettoria pianificata, attribuibili alle incertezze nella stima di posizione e particolarmente evidenti nelle aree dell'ambiente prive di elementi distintivi.

6.5 Guida remota

In conclusione, sono stati svolti test per quanto concerne la guida remota. Nello specifico si è utilizzato un gamepad connesso ad un computer portatile che andasse a comunicare con il nodo ROS *mqtt_server_node* per l'impartizione e per l'invio dei comandi tramite rete. Contemporaneamente, sul veicolo venivano eseguiti i nodi **mqtt_control_node** per la ricezione dei comandi e **hunter_ros2_node** per inviare i comandi all'interfaccia CAN del veicolo.

Per la comunicazione MQTT è stato utilizzato un broker accessibile da rete pubblica con cui sia il veicolo che il portatile (che nell'esperimento rappresentava il server) andavano a comunicare.

Le verifiche effettuate hanno confermato il corretto funzionamento degli applicativi, sebbene abbiano messo in luce alcune inefficienze. Tra queste, la latenza nella comunicazione tra veicolo e stazione di controllo è risultata particolarmente critica. Il tempo di risposta, misurato tra l'invio di un comando dal gamepad e la corrispondente azione del veicolo, ha mostrato valori elevati, rendendo l'applicazione non idonea a scenari che richiedono tempi di reazione rapidi come la guida remota.

Gli stessi tempi di latenza di questo test sono stati anche riscontrati nelle prove che riguardavano la telemetria: l'invio di dati da parte del mezzo che riguardano i sensori a bordo al server.

Per facilitare la ripetibilità e l'analisi dei dati, durante il test è stato generato un rosbag, ovvero un file eseguibile contenente una registrazione completa dei messaggi scambiati tra i nodi del sistema ROS. Grazie a questa risorsa, è stato possibile condurre multiple iterazioni di test sulla latenza senza la necessità di rieseguire fisicamente le manovre del rover.

Le misurazioni di latenza sono state effettuate eseguendo il rosbag e applicando un timestamp ai messaggi prima della trasmissione MQTT, registrando poi un secondo timestamp al momento della ricezione. La differenza temporale tra i due istanti, rappresentativa della latenza del sistema, è stata quindi memorizzata in un file di log.

Il file di log è stato poi analizzato tramite un semplice script python per estrarre i dati statistici riportati in tabella 6.1

tipo	Media (ms)	Varianza (ms^2)	Numero di sample
Odometry	661.7	87424.7	1235
Scan	656.6	86391.9	1237
Ackermann	722.1	86821.3	1379

Tabella 6.1: Latenze riscontrate durante i test

Come è possibile vedere dai dati mediamente si riscontra una latenza tra i 600 e i 700 millisecondi per quanto riguarda i messaggi dei sensori (odometria e scan), mentre per i messaggi di controllo (ackermann) si riscontrano valori più elevati, tra i 700 e gli 800 millisecondi. Questi alti valori di latenza riscontrati sono dovuti a vari fattori.

In primo luogo fra tutti la complessità del protocollo MQTT. Come discusso nella sezione 4.1, le procedure di conferma di ricezione e di ritrasmissione dei messaggi, pur essenziali per garantire la robustezza delle comunicazioni, introducono inevitabilmente ritardi nella trasmissione dei dati.

In secondo luogo, il mezzo di trasmissione utilizzato, ovvero una rete Wi-Fi, ha limitato le prestazioni del sistema. La tecnologia Wi-Fi, sebbene ampiamente diffusa, presenta caratteristiche intrinseche che possono influenzare negativamente la latenza, come la variabilità del segnale, l'interferenza con altre reti wireless e la saturazione del canale. A differenza di tecnologie più recenti come il 5G, che offrono una banda più elevata e una latenza significativamente inferiore, la rete Wi-Fi può costituire un collo di bottiglia nelle applicazioni che richiedono tempi di risposta estremamente rapidi.

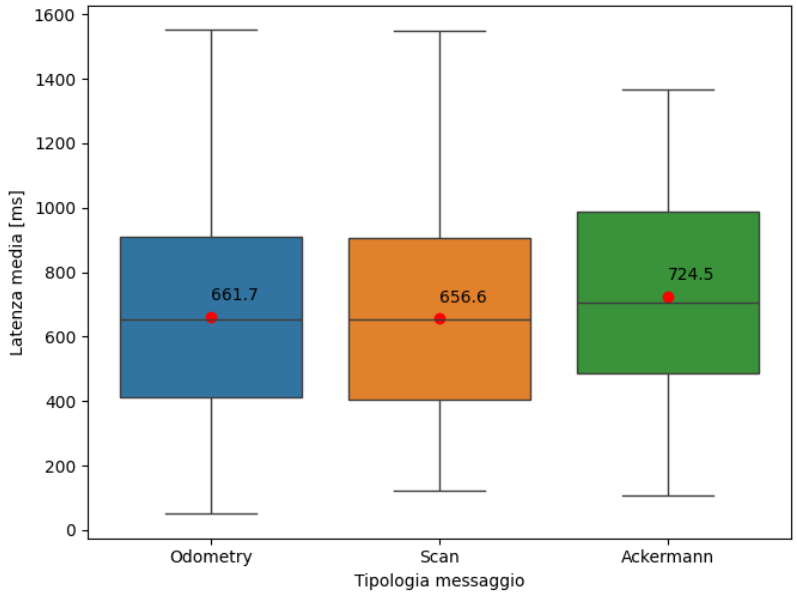

Figura 6.2: Latenze medie dei pacchetti messe a confronto

Il grafico a candela di Figura 6.2 evidenzia la notevole variabilità dei tempi di trasmissione. Questa alta instabilità, insieme ai dati in tabella 6.1 avvalorano l'ipotesi che l'utilizzo di una rete Wi-Fi non sia ottimale per questo tipo di applicazione.

Capitolo 7.0

Sviluppi futuri e conclusioni

In questo capitolo si vanno ad analizzare quali sono state le principali complicazioni che si sono riscontrate durante lo sviluppo, quali sono i punti da migliorare, eventuali soluzioni, sviluppi futuri e applicazioni pratiche della tesi.

7.1 Problemi ed eventuali soluzioni

Il primo dubbio riguardo il progetto descritto in questa tesi è sicuramente quello della latenza di rete: applicazioni come quella della guida autonoma richiedono l'esecuzione real time, ovvero che l'esecuzione di ogni singolo pezzo dello stack segue tempi predefiniti (chiamati deadline) anche nel caso peggiore.

Sorge spontanea una riflessione circa l'impatto dell'utilizzo della rete sulla puntualità dell'esecuzione delle attività. L'integrazione di reti di comunicazione in applicazioni real-time, infatti, introduce un elemento di incertezza non trascurabile, in quanto la qualità della connessione può variare significativamente, compromettendo il rispetto delle tempistiche previste. Anche in condizioni di rete ottimali, l'introduzione di protocolli di comunicazione, seppur efficienti come MQTT, comporta un overhead di sistema che può influire negativamente sulla latenza e, di conseguenza, sulla tempestività delle operazioni.

È possibile mitigare, seppur parzialmente, le problematiche legate alla variabilità della latenza di rete attraverso l'implementazione di meccanismi di Quality of Service (QoS). Riducendo il livello di QoS, è possibile diminuire l'overhead associato ai controlli di protocollo e conseguentemente ottimizzare le prestazioni in termini di latenza. Tuttavia, questa scelta comporta un compromesso significativo in termini di affidabilità, poiché la riduzione dei controlli aumenta la probabilità di perdita di pacchetti. Analogamente, la selezione del protocollo di trasporto incide in modo

determinante sulla latenza. Mentre il protocollo UDP, caratterizzato da un overhead ridotto, offre latenze inferiori, il protocollo TCP, garantendo la consegna affidabile dei dati, introduce un overhead maggiore e può comportare latenze più elevate. La scelta ottimale tra UDP e TCP dipende dal bilanciamento tra i requisiti di performance e affidabilità dell'applicazione.

7.2 Sviluppi futuri

Sebbene il progetto abbia raggiunto gli obiettivi prefissati, è evidente la necessità di integrare ulteriori funzionalità per consentirne l'utilizzo in contesti applicativi reali.

In particolare, l'implementazione di un sistema di sicurezza informatica robusto rappresenta una priorità assoluta. Tale sistema dovrà garantire la protezione dei dati del mezzo mediante l'adozione di tecniche di crittografia e offuscamento, al fine di mitigare i rischi connessi alla trasmissione di informazioni sensibili su reti pubbliche.

Un'ulteriore sviluppo possibile è l'aggiunta di una videocamera a bordo del mezzo. Tale sensore può risultare molto vantaggioso sia ai fini della guida autonoma che di quella remota. L'integrazione di questo sensore infatti si rivelerebbe strategica per diverse ragioni. In primo luogo, essa fornisce all'operatore remoto una rappresentazione visiva dettagliata dell'ambiente circostante, complementando le informazioni provenienti dal Lidar e consentendo un controllo più preciso e intuitivo del mezzo. In secondo luogo, i dati acquisiti dalla telecamera possono essere utilizzati per sviluppare e affinare algoritmi di visione artificiale, fondamentali per l'implementazione di funzionalità avanzate di guida autonoma, quali il riconoscimento di oggetti e il tracciamento di percorsi.

Un'ulteriore significativo miglioramento potrebbe essere ottenuto sostituendo l'attuale Lidar 2D con un sensore 3D. Questa evoluzione consentirebbe al sistema di guida autonoma di operare con una precisione notevolmente maggiore, riducendo al minimo gli errori derivanti da variazioni dinamiche dell'ambiente circostante. Inoltre, la rappresentazione tridimensionale dello spazio offrirebbe un quadro più completo e dettagliato per un eventuale operatore umano che dovesse intervenire in remoto.

Altro aspetto da considerare sarà l'implementazione di un sistema di platooning, per permettere al server di poter guidare oltre che un unico mezzo anche una flotta, questo può rivelarsi molto vantaggioso se si prevede l'utilizzo di questa tecnologia, ad esempio, per il trasporto di merci.

Sarà infine cruciale per una valutazione completa, è necessario effettuare test in situazioni reali, simulando le potenziali limitazioni della rete e le complessità tipiche degli ambienti di utilizzo.

7.3 Applicazioni pratiche

Per concludere, si esamineranno le potenziali applicazioni concrete di questa innovazione.

In linea con le precedenti considerazioni, si prospetta l'opportunità di implementare flotte di veicoli semiautonomi in rete, appositamente progettate per il trasporto. L'adozione di rover autonomi in grado di operare in ambienti industriali di grandi dimensioni, movimentando ingenti quantitativi di materiali, costituirebbe un volano per l'innovazione tecnologica all'interno dell'azienda.

Altro utilizzo pratico si può avere nel caso di veicoli ad utilizzo personale. Si può infatti ipotizzare uno scenario in cui il guidatore non sia in grado di controllare il veicolo in caso di emergenza medica e che quindi si avvalga ad un servizio che preveda un operatore pronto a connettersi e che possa pilotare il mezzo a distanza.

Bibliografia

[1] Differences between tcp and udp. https://www.geeksforgeeks.org/differences-between-tcp-and-udp/.

[2] draw.io: Flowchart maker & online diagram software. https://app.diagrams.net/.

[3] Nvidia jetson xavier. https://www.nvidia.com/it-it/autonomous-machines/embedded-systems/jetson-xavier-series/.

[4] Paho mqtt c library. https://github.com/eclipse/paho.mqtt.c.

[5] Paho mqtt cpp library. https://github.com/eclipse/paho.mqtt.cpp.

[6] Ros2 documentation. https://docs.ros.org/en/foxy/index.html.

[7] Sae international: Sae levels of driving automation. https://www.sae.org/blog/sae-j3016-update.