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

# Contents

# Capitolo 1

# Introduzione

## 1.1 Guida remota e guida autonoma

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 SAE international (Society of Automotive Engineers)[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 vigilare sulle 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.
D'altro canto, 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 prese da un essere umano, che opera a distanza utilizzando tecnologie quali sensori, attuatori, e reti di comunicazione. In questo scenario, l'essere umano 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 monitorare e controllare il veicolo. Tale approccio combina l'intelligenza umana con l'automazione tecnologica, rendendo 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 delle Cose (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'Internet of Things (IoT) è un concetto che descrive una rete di dispositivi fisici connessi tra loro attraverso Internet, capaci di scambiare dati e comunicare con altri dispositivi, sistemi e 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 veicolo capace di operare in modo autonomo in condizioni di guida normali, sfruttando un avanzato sistema di guida autonoma, ma che possa anche, su richiesta, passare alla modalità di guida remota.
Con condizioni normali si intende condizioni in cui il veicolo e l'hardware a bordo siano intatti e in cui il guidatore ,se presente a bordo, sia in grado di

condurre il veicolo senza problemi. Per fare un esempio possiamo immaginarci un semplice scenario che comprende un veicolo a guida autonoma con un conducente a bordo, in questi casi ci si aspetta che il conducente sia sempre vigile e che controlli il comportamento del mezzo, così che il mezzo in caso di errore o malfunzionamento, possa essere condotto dal guidatore al bisogno. Supponiamo ora invece che il guidatore non sia nelle sue condizione ottimali condizione causata da un malore o altro, in questo particolare caso il guidatore non può rimanere vigile e controllare il veicolo autonomo. In questo esempio, dunque, un operatore potrà collegarsi da remoto al veicolo per poterlo condurre all'ospedale più vicino o comunque in una zona sicura.
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 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

# 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 utile per l'interfacciamento del mezzo con un calcolatore esterno. L'intera scocca è realizzata in alluminio rendendolo molto resistente ma allo stesso tempo non eccessivamente pesante.

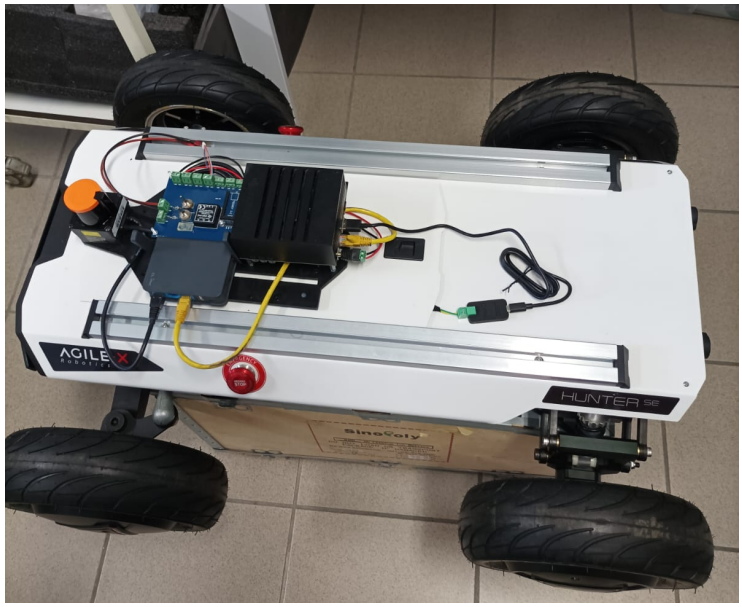


Figura 1: Rover assemblato

L'interfaccia di controllo del rover è collegata alla porta usb del computer embedded e consente una comunicazione efficiente e affidabile con esso. 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 su ruote a partire dallo spostamento di esse, questa 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.

### 2.1.1 Protocollo CAN

L'interfaccia di controllo del rover è basata sul protocollo CAN (Controller Area Network), un protocollo seriale molto versatile sviluppato dall'azienda Bosh nel 1993 ed è molto utilizzato in ambito automotive e automazione industriale. Questa interfaccia seriale è accessibile tramite 2 cavi denominati HIGH e LOW, collegati ad un interfaccia USB a cui poi il computer di bordo si va a interfacciare.
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 prima di tutto comunicare con i sensori e il rover ed inoltre che possa anche gestire il carico di tutti gli algoritmi e i processi necessari per la guida autonoma.
La scelta è dunque ricaduta su una scheda di casa Nvidia, nello specifico sul modello AGX Jetson Xavier, ovvero una GPGPU (General Purpose Graphic Processing Unit). Questa scelta è motivata dall'elevata capacità di elaborazione parallela che solo una GPGPU può fornire, questa capacità risulta particolarmente vantaggiosa per l'esecuzione di complessi algoritmi di percezione e pianificazione, oltre che di controllo, da svolgere in tempo reale. La GPGPU selezionata opera con il sistema operativo Ubuntu 20.04 Focal Fossa, noto per la sua stabilità e ampia compatibilità con l'hardware scelto.


Figura 2: Nvidia AGX Jetson Xavier

Di seguito una tabella riassuntiva delle principali caratteristiche della GPGPU Utilizzata: 1 1.5

| | |
|---|---|
| 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 |

Table 1: Specifiche tecniche della GPGPU Nvidia AGX Jetson Xavier[3]

## 2.3 Lidar

Per quanto concerne il sensore, la scelta è ricaduta su un sensore Lidar (Light Detection and Ranging) a 2 dimensioni 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

a seconda della tipologia, accurate 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 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 stabilire 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, noto per le sue piccole dimensioni e il suo scarso utilizzo energetico, esso è infatto alimentato direttamente da una delle porte USB del computer di bordo.


Figura 4: Router scelto per la piattaforma di sviluppo

# Capitolo 3

# ROS

In questa sezione si passa alla descrizione di ROS, al suo funzionamento e al suo utilizzo.

## 3.1 Descrizione

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 tutta l'implementazione sarà 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: Il C++ è noto per le sue elevate prestazioni, che sono fondamentali in applicazioni robotiche real-time dove la latenza e la capacità di risposta del sistema sono critiche
- Ampia diffusione e supporto: Il C++ è un linguaggio di programmazione molto diffuso nel settore della robotica, con una vasta comunità di sviluppatori e una ricca gamma di librerie e strumenti a disposizione

Il pacchetto ROS, oltre a fornire le componenti software di base, offre un ampio ventaglio di strumenti per lo sviluppo e il debugging. Tra questi, RViz (ROS-Visualization) 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.

## 3.2 Nodi

Uno degli aspetti fondamentali di ROS è la sua architettura basata su nodi, che rappresentano unità di esecuzione autonome 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.
I principali nodi che compongono questo stack sono i seguenti:

- **hunter_ros2_node**: Questo nodo è responsabile della gestione della comunicazione tra i vari nodi ROS e l'interfaccia CAN del veicolo. Attraverso questo nodo, i comandi e le informazioni vengono trasmessi efficacemente tra il sistema di controllo e il rover, assicurando un'interazione fluida e coerente con l'hardware del veicolo

- **urg_node**: Il compito di questo nodo è quello di raccogliere e trasmettere le informazioni provenienti dal sensore Lidar agli altri nodi del sistema. La scansione dell'ambiente effettuata dal Lidar viene elaborata e distribuita, fornendo dati essenziali per la navigazione autonoma e l'evitamento degli ostacoli

- **particle_filter**: Questo nodo implementa un algoritmo di localizzazione basato su filtri particellari, che consente di determinare con precisione la posizione del rover rispetto alla mappa dell'ambiente circostante. Il nodo utilizza le informazioni della mappa e le scansioni del sensore Lidar per aggiornare continuamente la stima della posizione del veicolo

- **telemetry_node** e **control_node**: Questi nodi gestiscono la comunicazione tra ROS e il protocollo MQTT (Message Queuing Telemetry Transport), garantendo la trasmissione di dati di telemetria e comandi di controllo in modo efficiente e affidabile

Tutti questi nodi operano in sinergia, scambiandosi informazioni critiche attraverso il sistema di messaggistica ROS, contribuendo all'esecuzione coordinata delle funzioni del robot. Questo approccio modulare e interconnesso consente di affrontare in modo efficiente le complesse esigenze operative del rover, garantendo una gestione robusta e scalabile delle diverse attività richieste durante la sua operazione autonoma e remota.

## 3.3 Comunicazione tra nodi

Sorge però spontaneo domandarsi come questi nodi comunichino tra loro e come facciamo soprattutto a riconoscere di che tipo di informazione si tratti. Una comunicazione ROS è formata da 3 elementi:

- Topic: Il protocollo utilizzato da ROS è di tipo publish/subscribe, ciò vuol dire che durante l'esecuzione dei nodi si vanno a creare dei topic, ovvero stringhe che utilizzano come separatore il carattere '/' e che ci permettono di suddividere tutti i diversi dati da inviare. Un esempio sono le scan lidar che vengono pubblicate sul topic "/scan". Ogni nodo può decidere se fare la subscribe a quel topic (ovvero ricevere tutti i dati inviati attraverso esso), fare delle publish (ovvero pubblicare dati su di esso) o se semplicemente ignorarlo

- Message type: Una volta scelto un topic però è necessario stabilire quali informazioni saranno ammesse su questo. ROS fornisce diversi tipi di dato

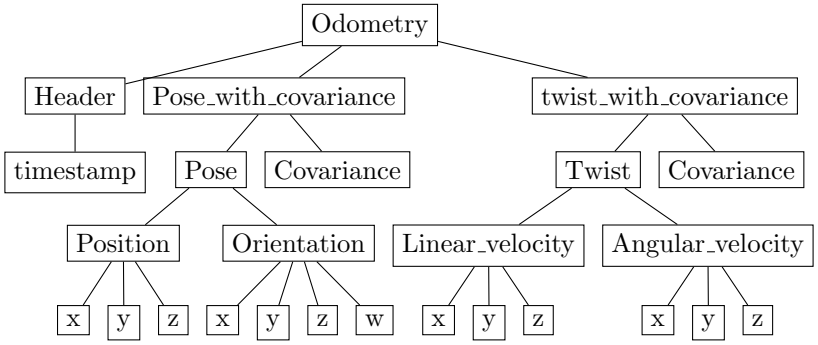


Figura 5: Struttura del messaggio ROS "Odometry"

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:

In questo particolare messaggio come si può notare sono contenuti: la posa, ovvero la posizione e l'orientamento del veicolo rispetto al punto di partenza ed il Twist ovvero la velocità lineare e quella angolare dell'oggetto al momento della misura. Esistono molti tipi di dato forniti da ROS alcuni altri esempi sono 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

# Capitolo 4

# MQTT

In questa sezione si passa alla descrizione del protocollo di rete MQTT ed al perchè si è scelto di utilizzare questa tecnologia.

## 4.1 Descrizione

Il protocollo MQTT (Message Queuing Telemetry Transport) 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.
All'interno del progetto per implementare lo stack MQTT si è avvalsi della libreria paho.mqtt.cpp[5], una libreria open source fornita dalla eclipse foundation.

## 4.2 Infrastruttura

Il protocollo MQTT opera secondo un'architettura client-server, dove i client (dispositivi o applicazioni) si connettono a un server (broker) centrale 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.

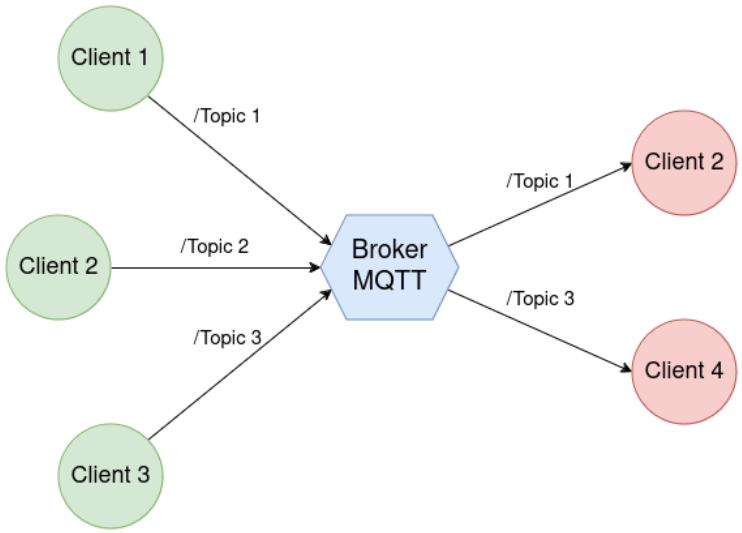


Figura 6: Struttura generica di un sistema di omunicazione basato su MQTT

Nonostante questa numeclatura però, nel corso della presente tesi ci si riferirà al broker come "broker", mentre con il termine server ci si riferirà al calcolatore incaricato di ricevere i dati dal mezzo e a cui inviare i risultati (ovvero i messaggi di controllo).

## 4.3 Formattazione messaggi

I messaggi scambiati tramite protocollo MQTT non sono altro che stringhe di testo. È quindi necessario utilizzare una formattazione per il testo che ci 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 che si presta bene a questo impiego. Per fare un esempio di seguito si illustra come un messaggio di odometria (illustrato nella sezione precedente)si presenterà in forma di testo 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.

## 4.4 Topic

In MQTT, come in ROS, per dividere le varie tipologie di messaggi inoltrati si utilizzano i topic, questo in realtà è esattamente il motivo per cui si è deciso di utilizzare MQTT per il controllo remoto. Grazie infatti ad una semplice stuttura dati è possibile intercambiare tra le due tipologie di topic.
Ai fini del progetto è stata quindi 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 meotodi:

```
void TopicManager::add_ros_topic(std::string ros_topic
    , bool telemetry_topic){
    // add ros topic and generate the corresponding
       mqtt topic
    std::stringstream ss;

    std::stringstream ros_topic_ss(ros_topic);
    std::string last_word;
    while(std::getline(ros_topic_ss, last_word, '/'))
       {}

    ss << this->loader->get_mqtt_parameters("
       BASIC_TOPIC");
    ss << "/" << this->loader->get_mqtt_parameters("
       VEHICLE_TOPIC");
    ss << "/" << this->loader->get_mqtt_parameters("
       VEHICLE_ID");
    if(telemetry_topic){
        ss << "/" << this->loader->get_mqtt_parameters
           ("VEHICLE_TELEMETRY_TOPIC");
    }
    ss << "/" << last_word;

    this->topic_list[ss.str()] = ros_topic;
```

```
}

void TopicManager::add_mqtt_topic(std::string mqtt_topic){
// add mqtt topic and generate the corresponding ros topic

    std::stringstream ss;

    std::stringstream ros_topic_ss(mqtt_topic);
    std::string last_word;
    while(std::getline(ros_topic_ss, last_word, '/')) {}

    this->topic_list[mqtt_topic] = ss.str();
}

std::string TopicManager::get_ros_topic_from_mqtt(std::string mqtt_topic){
//get ros topic corresponding to the passed mqtt topic
    return this->topic_list[mqtt_topic];
}

std::string TopicManager::get_mqtt_topic_from_ros(std::string ros_topic){
//get mqtt topic corresponding to the passed ros topic
    std::string mqtt_topic;

    for (auto& it : this->topic_list) {
        if (it.second == ros_topic) {
            mqtt_topic = it.first;
        }
    }

    return mqtt_topic;
}
```

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 si può illustrare il topic utilizzato nell'odometria

/hipert/vehicle/rover_1234/telemetry/odometry

Si elencano ora i diversi campi che compongono il topic

- **/hipert**: È il campo che identifica il laboratorio che sta svolgendo la comunicazione, 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, ci è utile per dare un identificatore univoco per il veicolo osservato, potremmo infatti anche avere più di un veicolo osservato e dobbiamo quindi sapere quale esattamente di questi si sta prendendo in considerazione.
- **/telemetry**: Identifica il tipo di dato preso in considerazione, infatti i dati potrebbero essere di tipo **telemetry**, **control** o eventualmente altro
- **/odometry**: Abbiamo infine l'esatto dato osservato, in questo caso, l'odometria del mezzo.

# Capitolo 5

# Scelte progettuali

Una delle peculiarità di ROS è che non limita la comunicazione tra i nodi al solo device che esegue l'istanza, ROS può infatti distribuire i pacchetti pubblicati dai nodi anche tramite rete, avvalendosi del protocollo **UDP**.
È giusto dunque chiedersi se, per soddisfare le richieste e le necessità della guida remota, non basti avvalersi di questa peculiarità di ROS, evitando di aggiungere complessità, integrando altri protocolli di rete.
Vengono di seguito riportate le motivazioni per cui non viene sfruttata la capacità di ROS di pubblicare pacchetti in rete nel progetto, andando invece a preferire un protocollo IOT come MQTT.

## 5.1 Affidabilità

Come descritto nella prefazione il framework ROS prevede l'esclusivo utilizzo del protocollo **UDP** per quanto concerne l'invio di messaggi in rete. Il protocollo UDP, un protocollo a livello trasporto molto veloce e versatile, soprattutto utilizzato in applicazioni che preferiscono minimizzare l'overhead e la latenza. Tuttavia, questa scelta comporta alcuni compromessi.
Essendo un protocollo connectionless, **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.
TCP garantisce 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 con meccanismi di acknowledge e timeout dei pacchetti.

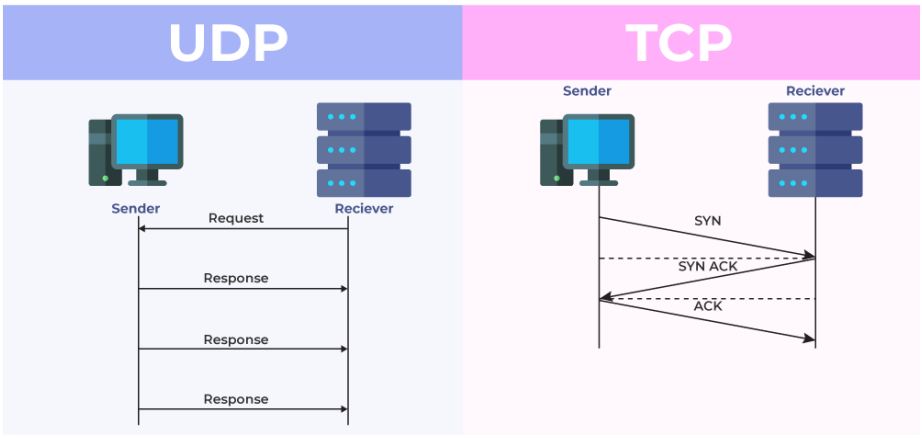


Figura 7: Comparazione graficata tra i 2 protocolli. Immagine tratta da [1]

Oltre a ciò TCP implementa meccanismi di controllo del flusso che evitano la congestione della rete, garantendo una comunicazione efficiente anche in condizioni di carico elevato.
Ovviamente per garantire questi aspetti, ci sono compromessi, **TCP** infatti è un protocollo sensibilmente più "pesante" rispetto ad **UDP**. Nonostante la latenza di TCP, 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.

## 5.2 Sicurezza

Nell'ambito dell'analisi di scenari di rete pubblica come il nostro, l'aspetto della sicurezza riveste un'importanza cruciale. In tale 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 intrusi.

## 5.3 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.

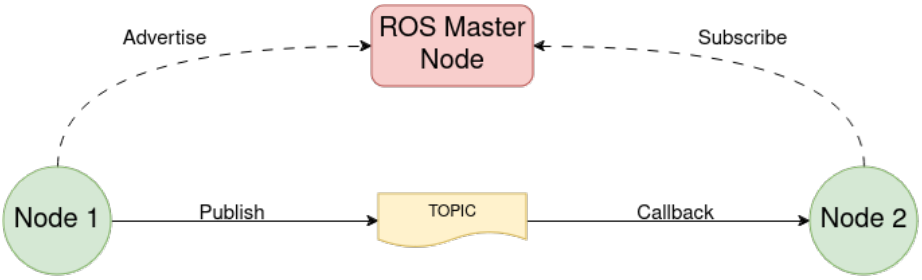


Figura 8: Struttura del sistema ROS

## 5.4 Controllo del QOS

Un'ulteriore punto da considerare è quella della gestione della **QOS** (Quality Of Service). Il concetto di **QOS**indica il livello di garanzia che una rete offre per quanto riguarda la consegna dei dati. Una bassa Quality Of Service 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**, nello specifico. **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

**ROS** invece, 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, e spesso richiede una 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 6

# Funzionamento

Nella seguente sezione si descrive il funzionamento del veicolo, degli algoritmi utilizzati e delle scelte progettuali.

## 6.1 Stack di guida autonoma

La prima cosa da analizzare è il funzionamento dello stack di guida autonoma. Questo stack funziona grazie a diversi processi, divisi (come descritto nell'introduzione) in perception, planning e control, di seguito una breve censita dei nodi ROS che permettono tale funzionamento: Per quanto riguarda la parte di perception, vediamo due aspetti, i sensori e l'algoritmo di localizzazione:

- **urg_node**: È il nodo che permette di pubblicare sul topic ROS */scan* le pointcloud rilevate dal sensore Lidar, il tipo di dato utilizzato è chiamato *LaserScan* e fornisce una serie di distanze che vanno insieme a formare ciò che il sensore Lidar rileva.
- **hunter_ros2_node**: Fornisce un interfaccia con il robot stesso, da questo nodo possiamo ricevere l'odometria calcolata a partire dal movimento delle ruote e, come vedremo successivamente, potremo pubblicare i comandi che il robot dovrà svolgere. Quello che interessa a noi al momento della perception è l'odometria del mezzo, che viene pubblicata sul tpoic */odometry* sottoforma di dato *Odomentry*.
- **particle_filter**: Implementa l'algoritmo di localizzazione chiamato particle filter, questo algoritmo sfrutta per il suo funzionamento la mappa dell'ambiente in cui il robot si sta muovendo, l'odometria del veicolo e la pointcloud del sensore Lidar. Questo nodo pubblica la posizione calcolata sul topic ROS */pf/position* sottoforma di dato *Odometry*.

Di seguito un'immagine che dimostra intuitavemente il funzionamento del sensore Lidar durante la fase di localizzazione.

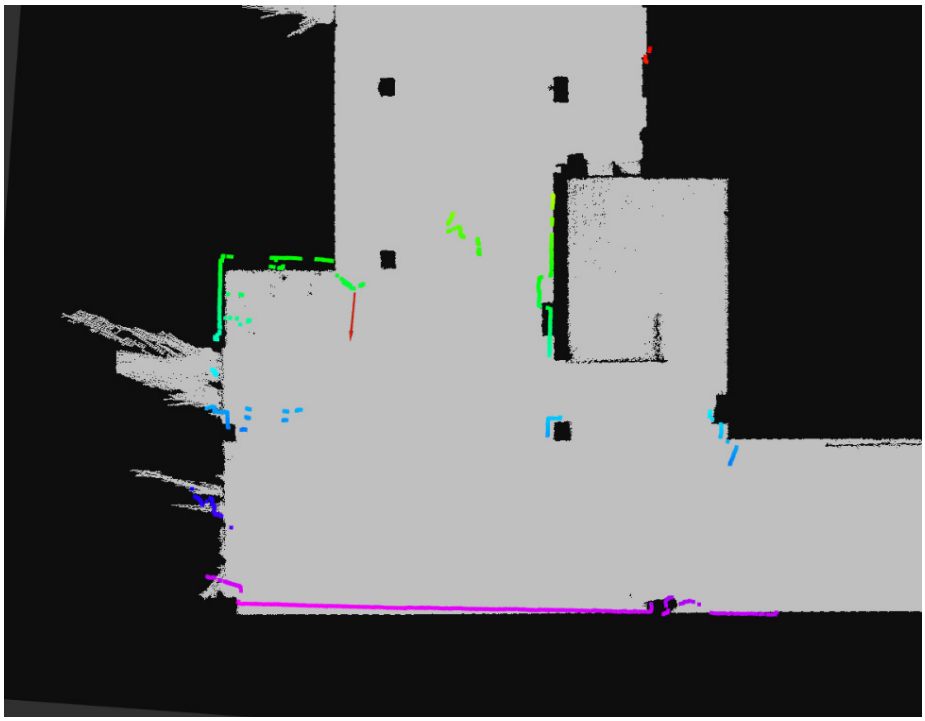

Figura 9: Dimostrazione funzionamento del sensore Lidar

In questa immagine abbiamo una porzione di una mappa(lo sfondo nero e grigio) e la pointcloud generata dal sensore Lidar (i punti colorati). Il colore dei punti non è casuale, ma è bensì un metodo intuitivo per stimare la distanza di quel particolare punto dalla posizione calcolata del robot (la freccia rossa).
Passando invece al momento del planning ci 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_logger**: Questo nodo si occupa di pubblicare un percorso preregistrato o precalcolato da seguire, il dato è pubblicato sul topic */path*.

Andiamo infine a descrivere il funzionamento della parte di controllo, questa è infatti composta da due nodi:

- **purepursuit**: Si occupa di ricevere il percorso pubblicato sul topic */path* e a partire dalla posizione pubblicata dal nodo **particle_filter** calcola i comandi da impartire al robot. I comandi vengono pubblicati sul topic*/drive_parameters* e sono di tipo *Ackermann Stamped*, questo non è altro che un semplice tipo di messaggio ROS che incapsula il timestamp, un angolo di sterzo ed una velocità

- **hunter_ros2_node**: Come descritto prima, questo nodo oltre a fornire l'odometria del mezzo, è anche capace di ricevere i comandi da impartire al robot. Il nodo è infatti in perenne ascolto sul topic */drive_parameters* e ad ogni messaggio non farà altro che comunicare con l'interfaccia CAN del veicolo comunicandogli la velocità e l'angolo di sterzo da impostare

Di seguito uno schema riassuntivo del funzionamento della guida autonoma:

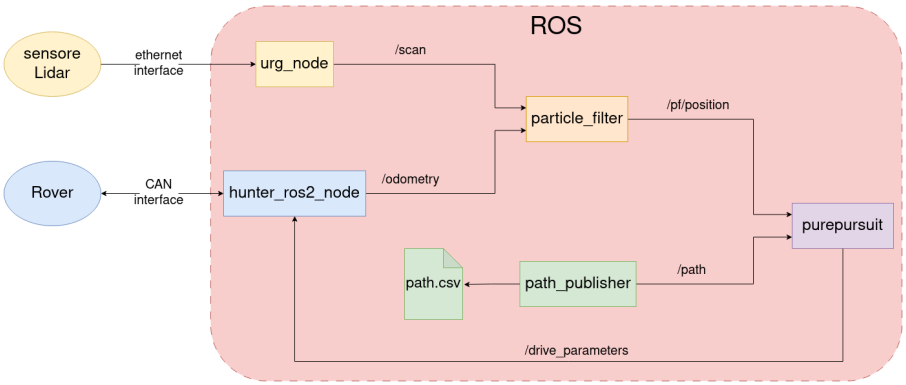


Figura 10: Schema rissuntivo dello stack di guida autonoma

## 6.2 Guida remota

Una volta illustrato e compreso il funzionamento dello stack di guida autonoma, è ora di passare alla pianificazione di quella remota.
La primissima domanda da porci è quale parte dello stack diventerà remoto, ad esempio si potrebbe decidere di svolgere solo la parte di planning da remoto e lasciare in resto in locale, o ad esempio di portare 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 ci 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:

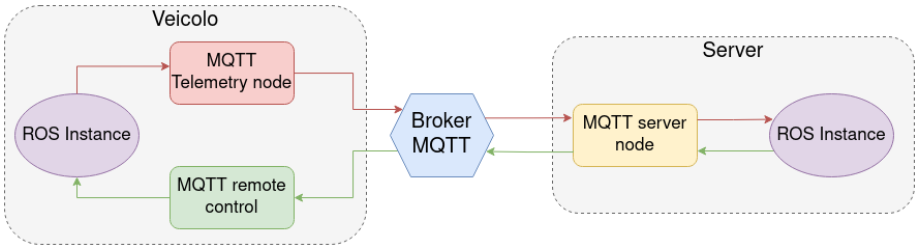


Figura 11: Schema rissuntivo dello stack di guida remota

## 6.3 Nodi sviluppati

Oltre ai nodi che già erano compresi nello stack di guida autonoma, è stato necessario sviluppare altri nodi che permettessero lo scambio di informazioni tra il veicolo e il server. Nello specifico i 3 nodi si iscriveranno a specifici topic ROS e pubblicheranno le stesse informazioni su topic 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"


```

```
17 CLEAN_SESSION: true
18 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.

### 6.3.1 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, così 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, nello specifico 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 permettere ciò 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 alcuni 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 ciò 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**:

```
MsgsCache::MsgsCache(int history_length){
    this->history_length = history_length;
}

void MsgsCache::addTopic(std::string topic){
    topicList.push_back(topic);
}

void MsgsCache::setTopicList(std::vector<std::string>
   topic_list){
    this->topicList = topic_list;
}

std::vector<std::string> MsgsCache::getTopics(){
    return topicList;
}

void MsgsCache::addMsg(std::string topic, std::string
   message){
    this->mutex.lock();
    if(std::count(this->topicList.begin(), this->
       topicList.end(), topic)<1)
        this->topicList.push_back(topic);

    if(this->cache[topic].size() >= this->
       history_length){
      this->cache[topic].pop_front();
    }
    this->cache[topic].push_back(message);
    this->mutex.unlock();
}

std::string MsgsCache::getLastMsg(std::string topic){
    std::string return_value("");
    if(this->cache[topic].size() > 0){
      return_value = this->cache[topic].front();
    }
    return return_value;
}

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 e che si avvale 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 codice è incluso nella sezione 4.4

### 6.3.2 MQTT remote control

In secondo luogo, 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**
- **control_node**: È il nodo ROS incaricato di ripubblicare sul topic giusto i dati relativi al controllo
- **json_to_ros2_msgs**: Per la conversione dei messaggi
- **mqtt_subscriber**: Per iscriversi 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.

### 6.3.3 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**
- **mqtt_publisher**
- **mqtt_subscriber**
- **msgs_cache**
- **msgs_to_string**
- **ros_server_node**: Implementa il client ROS che andrà poi ad iscriversi e a pubblicare sui topic necessari
- **string_to_msgs**
- **topic_manager**

# Capitolo 7

# Test Svolti

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

## 7.1 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, e successivamente 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

## 7.2 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, sapendo infatti lo spostamento e la pointcloud rilevata dal sensore ed un algoritmo apposito è possibile ricreare questa mappa.
Per realizzare questa operazione, è stato impiegato un algoritmo di Simultaneous Localization and Mapping (SLAM). Come suggerisce il nome, il 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
Di seguito l'immagine rappresentante la mappa ottenuta durante questi test.

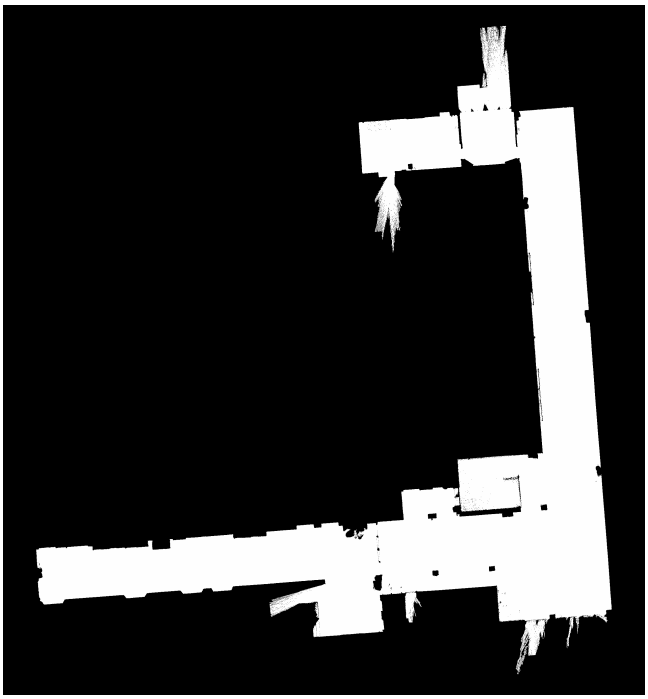

Figura 12: Mappa ottenuta tramite Lidar del primo piano dell'edificio Matematica

## 7.3 Localizzazione

Il terzo test è stato uno dei più importanti, questo 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 grazie al nodo ROS **particle_filter**, che implementa l'algoritmo di localizzazione già discusso precedentemente.
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.

## 7.4 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, particolarmente evidenti nelle aree dell'ambiente prive di elementi distintivi.

## 7.5 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, allo stesso tempo 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 mezzo.
Per la comunicazione MQTT è stato utilizzato un broker hostato su 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, fino a un secondo o due, 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, vvero l'invio di dati da parte del mezzo che riguardano i sensori a bordo al server.

# Capitolo 8

# Sviluppi futuri e conclusioni

Per concludere, si va ad analizzare quali complicazioni sono state riscontrate durante lo sviluppo, quali falle rimangono ed eventuali soluzioni e sviluppi futuri e applicazioni pratiche della tesi.

## 8.1 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 vengono chiamate real time, ciò vuol dire che l'esecuzione di ogni singolo pezzo dello stack deve eseguire in tempi stretti e che anche nel caso peggiore l'esecuzione non può superare una certa quantità di tempo, questa quantità viene chiamata deadline.
È quindi spontaneo porsi un dubbio, ovvero quanto l'utilizzo della rete complichi il dover far rispettare le tempistiche. Includere la rete in applicazioni real time infatti diventa rischioso, si pensa subito al caso in cui la connessione sia molto scarsa o addirittur assente, quali complicazioni questo può portare. Anche nel caso di una situazione ottimale però bisogna sempre considerare quanto l'inclusione di un protocollo di rete (sepur leggero come nel caso di MQTT) porti dell'overhead e di conseguenza delle latenze nell'esecuzione.
Possiamo ovviare a questi problemi (seppur limitatamente) grazie al meccanismo dei livelli di Quality Of Service che il protcollo ci fornisce, un livello basso di Quality Of Service infatti farà fare meno controlli al protocollo e di conseguenza rimuoverà un certo overhead. Anche la scelta del protocollo a livello transport influenza le latenze, se usare quindi UDP o TCP dato che notoriamente il protocollo UDP riduce di molto la complessità, a scapito però sempre della qualità del servizio.

## 8.2 Sviluppi futuri

Il progetto ai fini della tesi può considerarsi concluso, rimangono però eventuali implementazioni e test che potranno essere integrati in futuro.
La prima cosa che viene in mente è l'implementazione di uno stack di sicurezza informatica che permetta l'invio dei dati del mezzo su rete pubblica completa-

mente criptati ed oscurati ad un possibile attaccante, implementazione necessaria se si prevede di utilizzare questa tecnologia in casi reali.
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. Una telecamera infatti risulterebbe utile sia ad un operatore che si avvale della guida remota per poter osservare l'ambiente circostante con più chiarezza, rispetto a quanto il singolo sensore Lidar permetta, potrebbe inoltre risultare interessante per lo sviluppo di algoritmi di computer vision di cui la guida autonoma si avvarrebbe.
Rimanenendo sull'analisi dei sensori e come già specificato nella sezione dedicata ai test di localizzazione, uno sviluppo interessante potrebbe essere quello di sostituire il Lidar attuale a due dimensioni, con uno a 3, che permetterebbe al sistema di guida autonoma una maggiore precisione negli spostamenti, un minore errore dato da eventuali cambiamenti nell'ambiente mappato e una visione più chiara per un operatore dedicato alla guida remota.
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.
Infine sarà necessario svolgere test in condizioni reali, condizioni in cui l'affidabilità della rete sia limitata o con ambienti molto complessi.

## 8.3 Applicazioni pratiche

Per concludere si procede ad elencare quali possono essere delle eventuali applicazioni pratiche di questa tecnologia.
Come descritto prima, un utilizzo potrebbe essere quello della creazione di una flotta di veicoli semi-autonomi connessi a scopi di trasporto, avere una flotta di rover capaci di trasportare all'interno di ambienti lavorativi grandi quantità di materiale o semi-lavorati infatti, aiuterebbe con lo sviluppo tecnologico di un'impresa.
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 che possa pilotare il mezzo a distanza, o addirittura ad un servizio di guida autonoma che possa guidare il veicolo fino all'ospedale più vicino.

# References


[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.