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

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. Questi processi fondamentali 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 le 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 Scopo della tesi

L'obiettivo principale di questa tesi è 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. 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.

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 un'interfaccia di controllo basata sul protocollo CAN (Controller Area Network), che consente una comunicazione efficiente e affidabile tra i diversi sistemi elettronici del veicolo. 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, che rappresenta la misura dello spostamento del veicolo basata sul movimento delle ruote. L'odometria 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.

2.2 GPGPU e sensore lidar

Un elemento cruciale per la realizzazione di questa tesi è stato l'identificazione e la selezione di un calcolatore embedded idoneo a gestire l'intera logica di controllo e la pianificazione, oltre alla scelta di un sensore in grado di fornire dati essenziali per la percezione dell'ambiente circostante. Per il calcolatore embedded, è stata presa la decisione di impiegare una GPGPU (General Purpose Graphic Processing Unit), una scelta motivata dalla sua elevata capacità di elaborazione parallela, che risulta particolarmente vantaggiosa per eseguire complessi algoritmi di controllo e di pianificazione in tempo reale. La GPGPU selezionata opera con il sistema operativo Ubuntu 20.04, noto per la sua stabilità, ampia compatibilità con hardware di ultima generazione, e supporto per lo sviluppo di applicazioni avanzate, inclusi strumenti specifici per l'elaborazione grafica e la gestione di risorse computazionali. Per quanto concerne il sensore, la scelta è ricaduta su un sensore Lidar (Light Detection and Ranging). 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 accurate, fondamentali per il riconoscimento degli ostacoli, la navigazione e la pianificazione del percorso del veicolo. 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.

3 ROS

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

3.1 Nodi

Il Robotic Operating System, comunemente noto come ROS, è una piattaforma software composta da un insieme di librerie e strumenti che facilitano lo sviluppo di applicazioni dedicate al controllo e alla gestione di sistemi robotici. ROS offre un'infrastruttura flessibile e modulare che permette agli sviluppatori di creare, testare e implementare applicazioni complesse per robot in modo efficiente.
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 (Controller Area Network) 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.

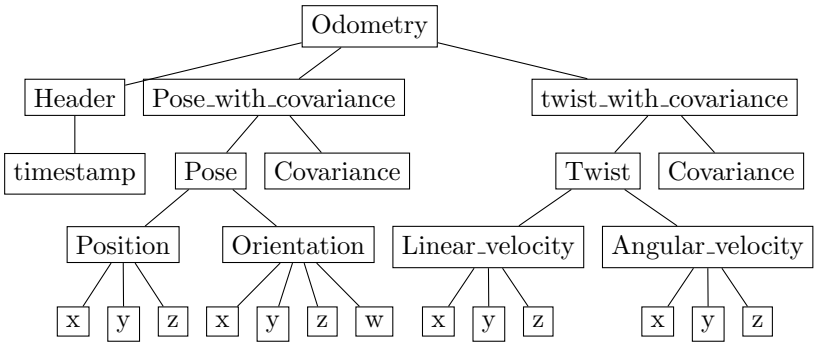
Tutti questi nodi operano in sinergia, scambiandosi informazioni critiche attraverso il sistema di messaggistica ROS, contribuendo all'esecuzione coordinata delle funzioni del robot. Questo approccio modulare e interconnesso consente di affrontare in modo efficiente le complesse esigenze operative del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

3.2 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 nodo (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 si dovrà anche decidere quali informazioni saranno ammesse su questo, 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:



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

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.

4.2 Infrastruttura

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.

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, infatti grazie a poche righe di codice è possibile ricavare i topic MQTT partendo da quelli ROS.
Conoscendo infatti la struttura dei topic ROS utilizzati e quella dei topic MQTT ci basterà eseguire il seguente codice per passare dall'uno all'altro:

```
std::string ros_topic_to_mqtt_topic(std::string ros_topic){
  std::stringstream ss;

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

  ss << loader->get_mqtt_parameters("BASIC_TOPIC");
  ss << "/" << loader->get_mqtt_parameters("VEHICLE_TOPIC");
  ss << "/" << loader->get_mqtt_parameters("VEHICLE_ID");
  ss << "/" << loader->get_mqtt_parameters("
      VEHICLE_TELEMETRY_TOPIC");
  ss << "/" << last_word;

  return ss.str();
}
```

5 Funzionamento

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

5.1 Stack di guida autonoma

La prima cosa da analizzare è il funzionamento dello stack di guida autonoma. Lo 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*.

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 manualmentequesto percorso 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 di tutto il meccanismo:

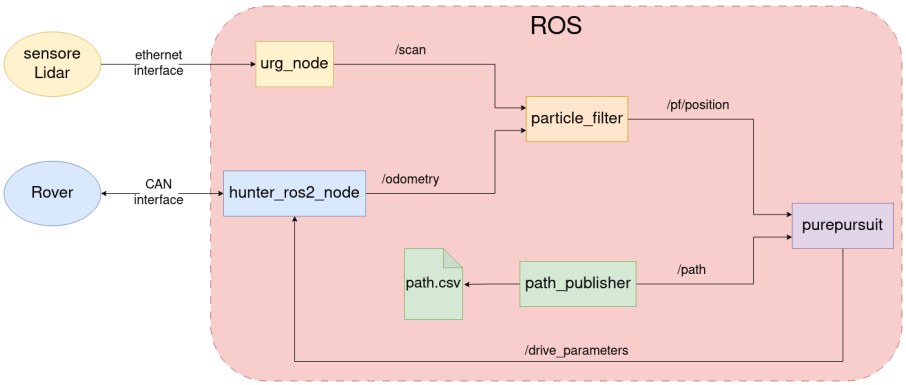


Figure 1: Schema rissuntivo dello stack di guida autonoma

5.2 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'hardare.
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.

6 Sviluppi futuri e conclusioni

6.1 Problemi

6.2 Soluzioni

6.3 Applicazioni pratiche