UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

DIPARTIMENTO DI INGEGNERIA "ENZO FERRARI"

Laurea Triennale in Ingegneria Informatica

Planning locale di una vettura da Formula Student a guida autonoma

Relatore:

Prof: Francesco Guerra

Candidato:

Filippo Gibertini

Anno Accademico 2022/2023

Indice

Elenco delle figure

Capitolo 1

introduzione

Nell'ambito della guida autonoma il Planning è una componente software che si occupa di delineare la traiettoria che la macchina deve seguire. In particolare, nelle auto a guida autonoma da competizione, il Planning si suddivide in Locale e Globale. Il Planning Globale è specializzato nel navigare attraverso percorsi conosciuti, mentre il Planning Locale si occupa di generare la traiettoria della macchina attraverso percorsi di cui non si posseggono informazioni pregresse. Gli aspetti cruciali di cui si è dovuto tenere conto nella realizzazione dell'algoritmo di Planning Locale sono: *(1)* la molteplicità delle prove di gara che si devono affrontare, ognuna delle quali necessita di un Planning Locale dedicato, *(2)* la corretta comunicazione con gli altri elementi software (nodi) che compongono l'algoritmo di guida autonoma, *(3)* la necessità di un algoritmo che si avvicini il più possibile ad essere real-time.

Capitolo 2

Stato dell'arte

arte dello stato

Capitolo 3

Conoscenze di base

In questo capitolo verranno esplorate le conoscenze fondamentali che costituiscono il punto di partenza teorico per lo sviluppo del progetto. Queste conoscenze sono essenziali per comprendere appieno il contesto e i concetti chiave che saranno discussi nei prossimi capitoli.

3.1 ROS2 - Robotic Operating System

ROS2 è un middleware basato su un meccanismo di publisher e subscriber che per mette l'interazione, attraverso lo scambio di informazioni, tra diversi processi ROS. Il progetto è stato sviluppato all'interno di un workframe ROS2 (ROS Foxy)[**?**] introduciamo quindi alcuni componenti della sua terminologia fondamentale [**?**]:

1. **ROS graph**: Si riferisce ad una rappresentazione visuale o concettuale della rete di comunicazione e delle interconnesioni tra i nodi all'interno di ROS. Questa rappresentazione grafica mostra come i nodi comunicano tra loro, inviando e ricevendo messaggi tramite i topic e come sono collegati all'interno di ROS.

2. **ROS topic**: I topic vengono utilizzati come canali di comunicazione tra diversi processi, in modo da scambiarsi dati attraverso dei messaggi.

3. **ROS nodes**: I nodi possono essere visualizzati come unità di elaborazione, ovvero sono quelle entità che si devono scambiare dati fra di loro, questo avviene tramite i publisher o subscriber, infatti i nodi hanno la capacità di:

- **Sottoscriversi** ad un topic e quindi rimanere in ascolto su di esso e ricevere tutti i messaggi che vengono pubblicati in quel determinato topic. Un comportamento di questo tipo è quello dei subscriber.
- **Pubblicare** messaggi su un certo topic e quindi invocare una chiamata di funzione per mandare un certo messaggio sul topic selezionato che poi verrà letto da tutti quei nodi che hanno un componente che si è sottoscritto a quel topic. Questi sono i publisher.

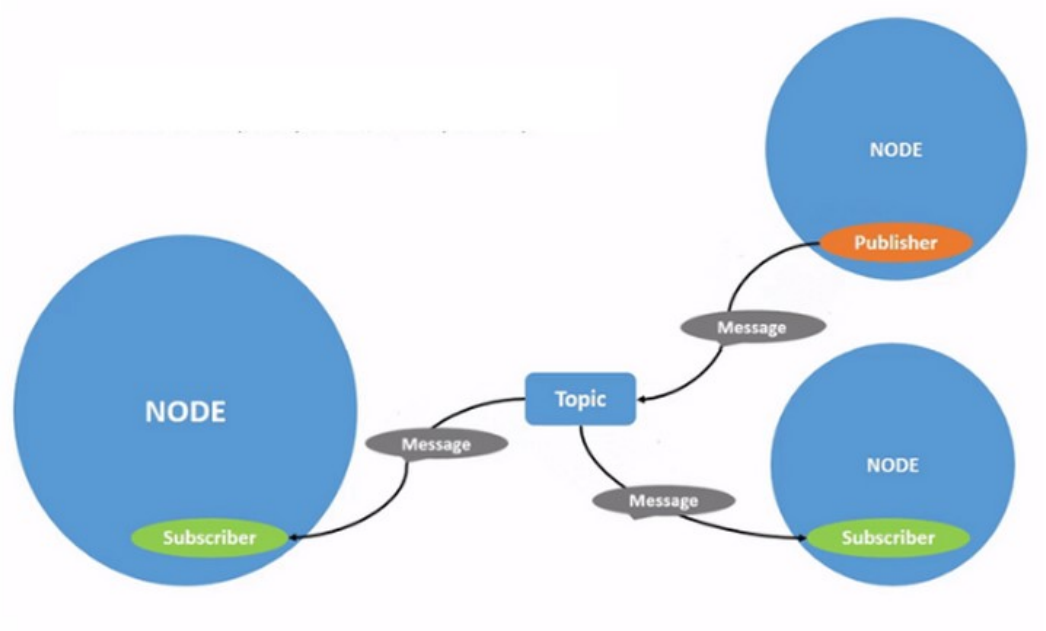

Figura 3.1: Funzionamento protocollo pub/sub su ROS

4. **ROS message**: I messaggi utilizzati in ROS sono strutture dati organizzate in campi, e per ciascun campo è specificato il tipo di dato che contiene. È possibile personalizzare i messaggi creando un file in cui si definiscono i campi necessari e salvandoli nel formato msg. ROS offre la possibilità non solo di definire nuovi tipi di messaggi, ma include anche una vasta libreria di messaggi predefiniti sviluppati dagli stessi creatori di ROS, che coprono una vasta gamma di scenari. Questi tipi di messaggi personalizzati possono essere utilizzati non solo all'interno del nodo in cui sono stati definiti, ma anche in tutti i nodi in cui sono installate le dipendenze del nodo che utilizza quel messaggio.

Primitive Type	Serialization	C++	Python3
bool	unsigned 8-bit int	uint8_t (2)	bool
int8	signed 8-bit int	int8_t	int
uint8	unsigned 8-bit int	uint8_t	int
int16	signed 16-bit int	int16_t	int
uint16	unsigned 16-bit int	uint16_t	int
int32	signed 32-bit int	int32_t	int
uint32	unsigned 32-bit int	uint32_t	int
int64	signed 64-bit int	int64_t	int(long)
uint64	unsigned 64-bit int	uint64_t	int(long)
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ascii string	std::string	str(bytes)
time	secs/nsecs unisgned 32-bit ints	ros::Time	rospy.Time
duration	secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration

Figura 3.2: Rappresentazione dei vari tipi su ROS

Dopo aver introdotto la terminologia necessaria per riuscire a discutere di ROS in maniera appropriata, prendiamo in considerazione un nodo e andiamo a studiarne i componenti a livello implementativo.

```
$ source /opt/ros/foxy/setup.bash
$ ros2 pkg create -build-type ament_cmake <package_name>
```

Listing 3.1: Creazione di un nodo ROS

Per ogni nodo sono presenti quattro directory principali oltre alla *CMakeLists.txt* che rende la compilazione più snella da parte dell'utente e al *package.xml* che definisce il pacchetto.

1. **config**: É presente un file di configurazione in cui possono essere definiti dei parametri statici che vengono poi recuperati dal nodo e utilizzati durante l'esecuzione. La comodità di utilizzare dei parametri di questo tipo è che non bisogna ricompilare il nodo se vengono modificati.

```
/package_name:
  ros__parameters:
  esempio_parametro: 1
  general:
  esempio_parametro: "prova"
```

Listing 3.2: Esempio di un file config

2. **include/name_node**: Si possono trovare tutti i file di intestazione che defini scono le interfacce o le dichiarazioni delle classi, delle funzioni o delle variabili utilizzate all'interno del pacchetto. Questi file possono essere inclusi in altri file sorgente all'interno del pacchetto o in pacchetti esterni che desiderano utilizzare le funzionalità fornite dal pacchetto.

3. **launch**: contengono i file che permettono al nodo di essere eseguito in maniera più snella e controllata. Per ogni file config che viene creato all'interno di un nodo è necessario aggiungere il percorso della directory in cui è presente il file di configurazione.

```
def generate_launch_description() :
  config_node = os.path.join
  (
  get_package_share_directory('package_name'),
'config',
'package_name_conf.yaml'
```

Listing 3.3: Esempio di un file launch

4. **src**: Contiene il codice sorgente effettivo del pacchetto, inclusi i file sorgente per i nodi ROS2, le librerie personalizzate le classi e le funzioni che costituiscono il comportamento del pacchetto.

L'ultima parte sulla visione iniziale di ROS2 è quella che riguarda la compilazione e l'installazione dei nodi all'interno del **ROS Environment**.

```
$ colcon build --symlink-install
--continue-on-error --package-select <package_name>

# in questo modo posso eseguire file launch come se fossero nella cartella
    corrente
```

```
$ source ./ros2_env/install/setup.bash
$ ros2 launch package_name package_name_launch.py
```

Listing 3.4: Compilazione di un pacchetto ROS e attivazione dell'Environment

Capitolo 4

Implementazione

In questo capitolo verrà descritto come è sono stati tradotti in codice gli algoritmi descritti nei capitoli precedenti. E' stato implementato un planner per ogni prova della gara: acceleration, skidpad, autocross e trackdrive, che condividendo lo stesso tracciato condividono lo stesso planner. Ogni planner viene inizializzato all'interno di un nodo ROS che ha anche le responsabilità di reperire il tipo della prova e istanziare publishers e subscribers.

4.1 Local planner node

Il nodo è stato definito nel file `local_planner_node.h` ed è stato implementato nel file `local_planner_node.cpp`. Il nodo si prende a carico le seguenti responsabilità:

- Reperire i parametri relativi ai topic e il parametro relativo al tipo di evento dal file di configurazione.
- Inizializzare i publishers che sono comuni a tutte le prove.
- Inizializzare il planner corrispondente al tipo di evento contenuto nel parametro "node/eventType".
- inizializzare i subscribers specifici facendo il bind fra le callback del planner e i topic necessari.

Capitolo 5

Conclusioni

L'applicazione, descritta all'interno di questa elaborato, ovviamente non è ultimata ma è solamente un punto di partenza per l'introduzione della programmazione modulare all'interno della libreria WLDT. Difatti il progetto può essere, sicuramente, migliorato ed ampliato sviluppando diverse tipologie di DigitalAdapter e PhysicalAdapter "complessi" (HTTPDigitalAdapter e MQTTPhysicalAdapter). Inoltre la flessibilità e la dinamicità tra le varie componenti del Digital Twin può essere, ulteriormente, migliorata riducendo la dipendenza tra il bundle rappresentante il modello del DT ed il bundle della ShadowingFunction, permettendo al Digital Twin di utilizzare più ShadowingFunction senza resettare lo stato interno. Infine potrebbe essere sviluppato anche un ulteriore bundle, che attraverso l'utilizzo di un event bus, gestisca la comunicazione tra i vari componenti del Digital Twin.