

Sviluppo di sistemi robotici con ROS

La tecnologia ROS applicata alla guida
autonoma

12/12/2023



UNIVERSITÀ DI PISA

SMART INDUSTRY®
EXCELLENT FUTURE CONDITION



SMART INDUSTRY è una società di servizi e **consulenza industriale**

LE SEDI

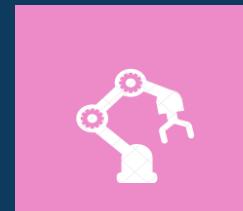
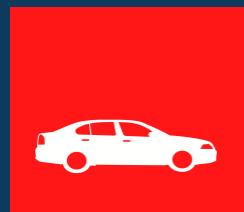
SMART INDUSTRY ha sede legale a Ozzano dell'Emilia (BO) e sedi operative in Italia e all'estero:



- Trento (TN)
- Torino (TO)
- Amaro (UD)
- Schio (VI)
- Mestre (VE)
- Cernusco sul Naviglio (MI)
- Modena (MO)
- Ozzano dell'Emilia (BO)
- Ancona (AN)
- Roma (RM)
- Napoli (NA)
- Palermo (PA)

- Albania
- Bulgaria
- Romania
- Serbia

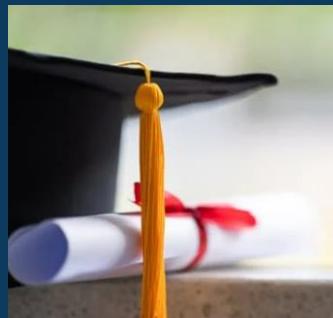
IN QUALI SETTORI OPERIAMO



PERCORSI PER GLI STUDENTI



TIROCINI CURRICULARI



TESI DI LAUREA



SEMINARI



Introduzione

Chi siamo

Alberto Troiani

Control and Software Engineer- System Integrator

alberto.troiani@oncode.it

Alessandro Rossignoli

Founder and Technical Manager

alessandro.rossignoli@oncode.it



Introduzione

Di cosa parleremo oggi



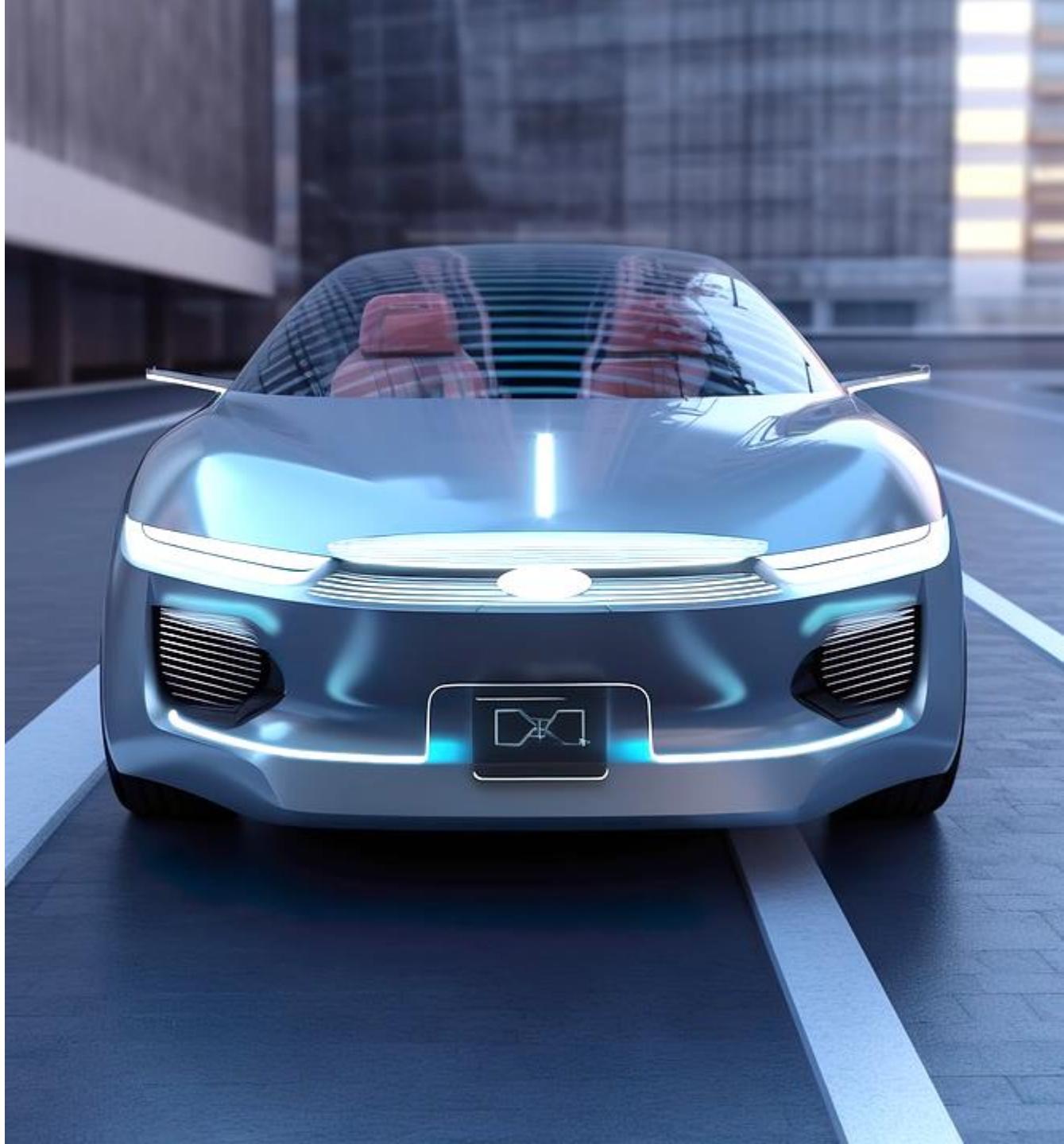
Capire il funzionamento di **ROS**, dei suoi **componenti** e i **tool** che possono aiutarci durante il suo utilizzo



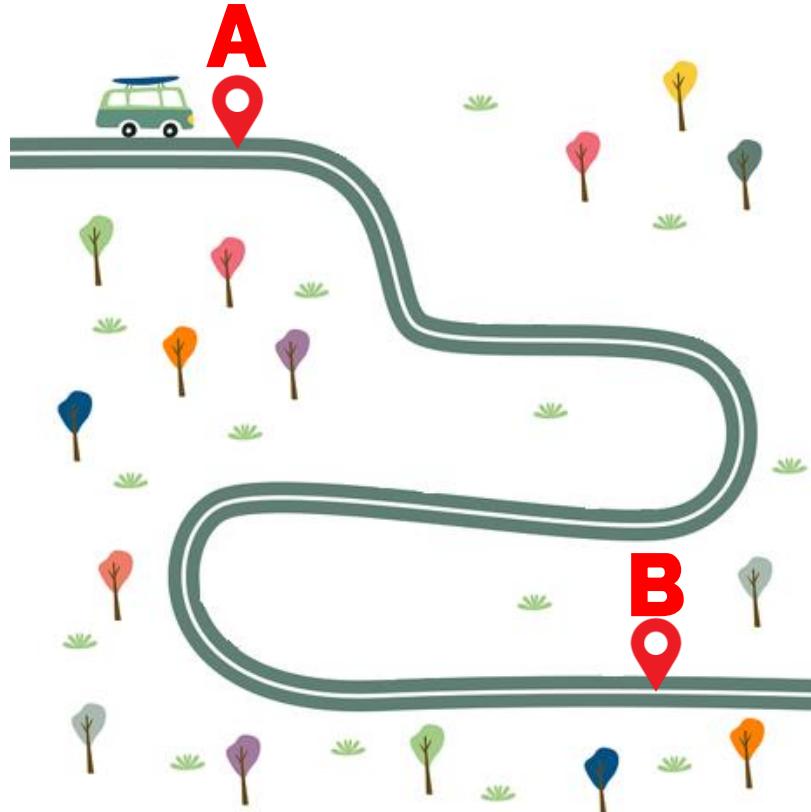
Il tutto applicato ad un **caso pratico** di sviluppo di sistema di controllo di un **veicolo a guida autonoma**

Veicoli a guida autonoma

**Cosa è e come funziona un veicolo a
guida autonoma?**



Cos'è e come funziona un veicolo a guida autonoma?

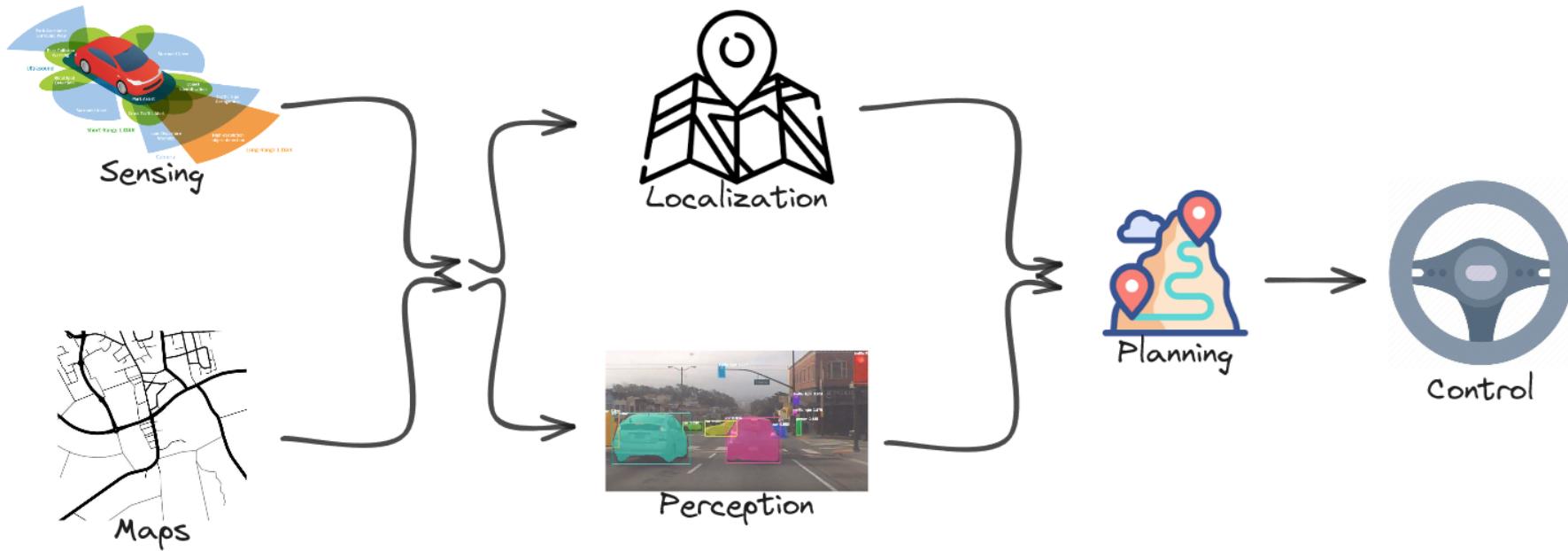


Un **veicolo a guida autonoma** è un veicolo in grado di **rilevare** le caratteristiche dell'ambiente circostante tramite l'utilizzo di sensori di diverso tipo per poter svolgere il compito di **muoversi** da un punto A ad un punto B senza l'intervento e la supervisione umana.



Veicoli a guida autonoma

Cos'è e come funziona un veicolo a guida autonoma?



Cos'è e come funziona un veicolo a guida autonoma?

L'argomento dei veicoli a guida autonoma si distingue per la sua **straordinaria complessità**, emergendo come un caso applicativo affascinante all'interno del vasto panorama della robotica.

Denso di sfide e caratteristiche distintive:

- Alte velocità
- Dinamiche complesse e rapide dei veicoli
- Sistema ad altissimo rischio
- Ambiente variabile e sempre diverso.
- Ambiente multi-agente
- Presenza variabile e densa di ostacoli
- Regolamentazione non completa al 100%

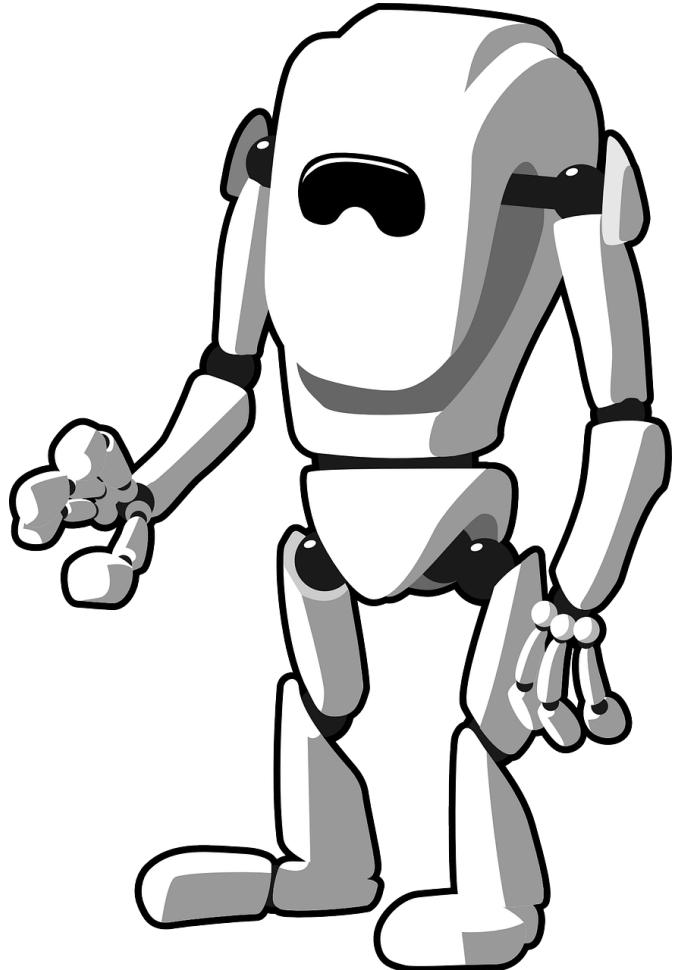


è un **meta-sistema operativo** open source per Robot. Fornisce i servizi che ci si aspetta da un sistema operativo:

- astrazione dell'hardware
- controllo dei dispositivi a basso livello
- implementazione di funzionalità comuni
- passaggio di messaggi tra processi
- gestione dei pacchetti

Fornisce inoltre strumenti e librerie per ottenere, costruire, scrivere ed eseguire codice su più computer.

<https://wiki.ros.org/>



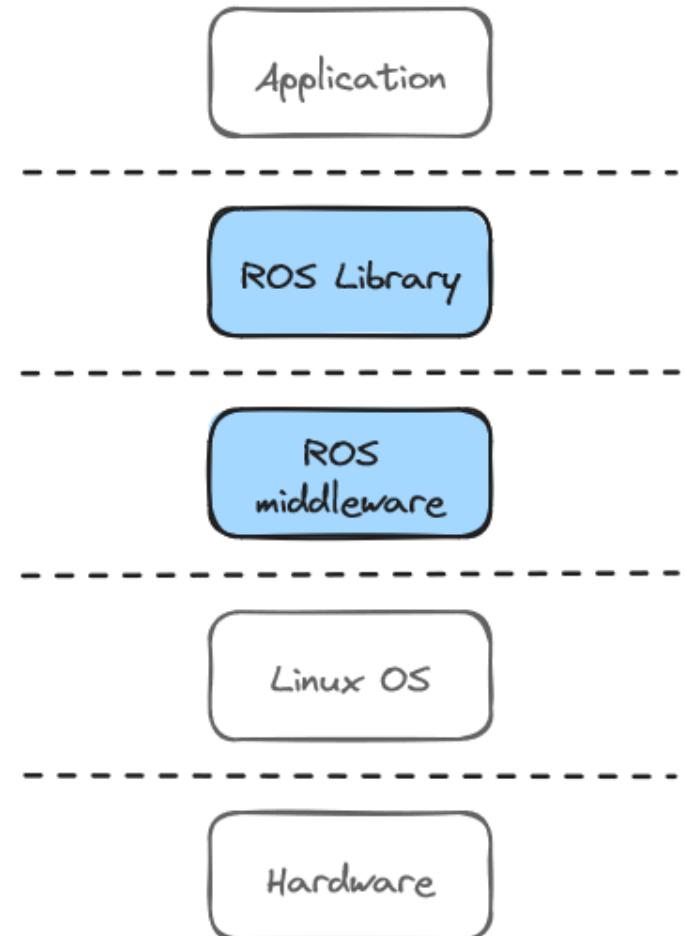


ROS

Robot Operating System

È un **framework middleware** progettato per lo sviluppo di software per la robotica

Fornisce un insieme di strumenti, librerie e convenzioni per **facilitare lo sviluppo**



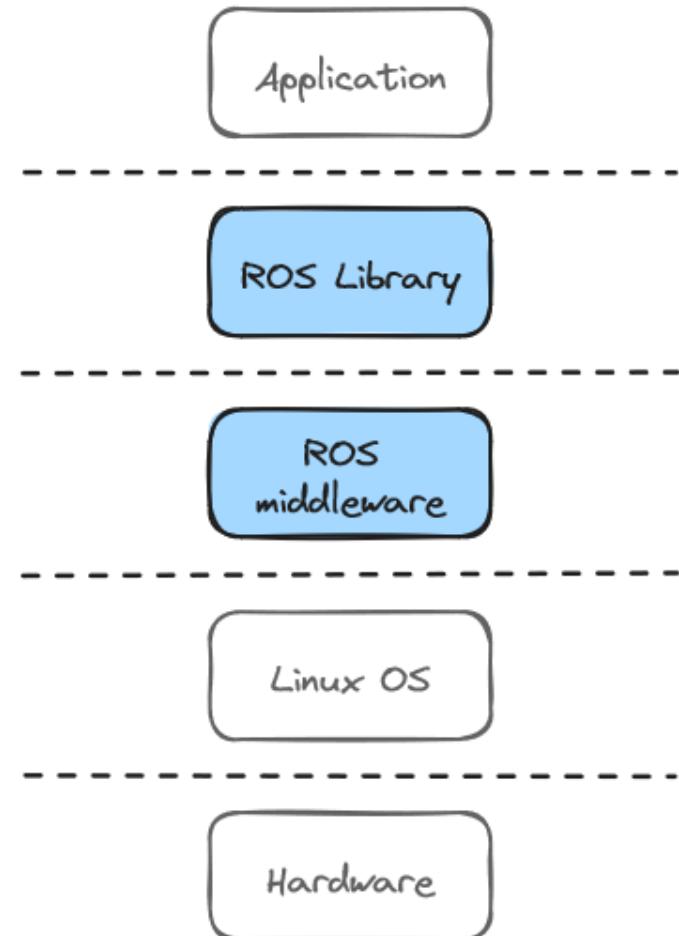


ROS

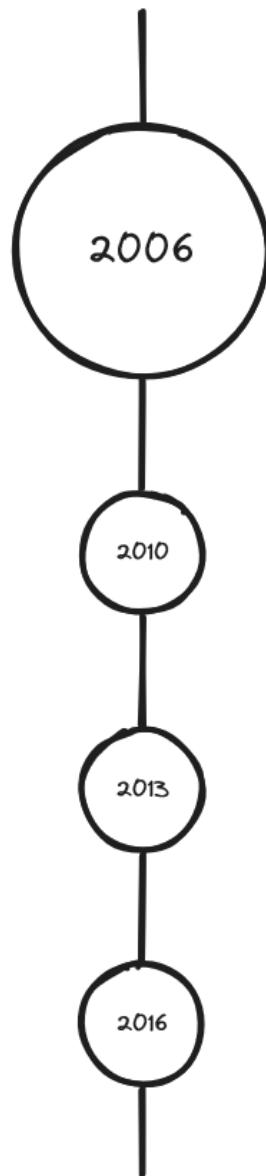
Robot Operating System

Il termine **"meta"** viene usato in riferimento al fatto che **ROS** offre un livello di astrazione superiore rispetto al sistema operativo sottostante

ROS facilita la comunicazione tra i diversi componenti software di un robot, consentendo loro di **operare in modo coordinato**



Nascita di ROS



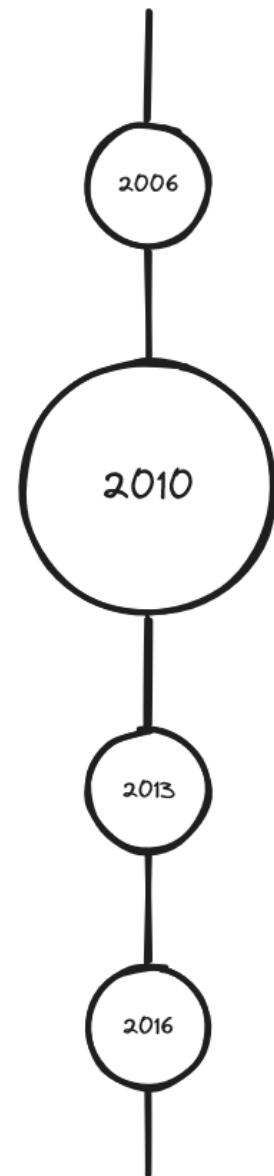
ROS è stato pensato e creato da Keenan Wyrobek ed Eric Berger durante il loro periodo universitario a Stanford

Volevano risolvere uno dei problemi più comuni del tempo nel mondo della robotica:

- Reimplementazione continua delle funzionalità base di un sistema robotico.

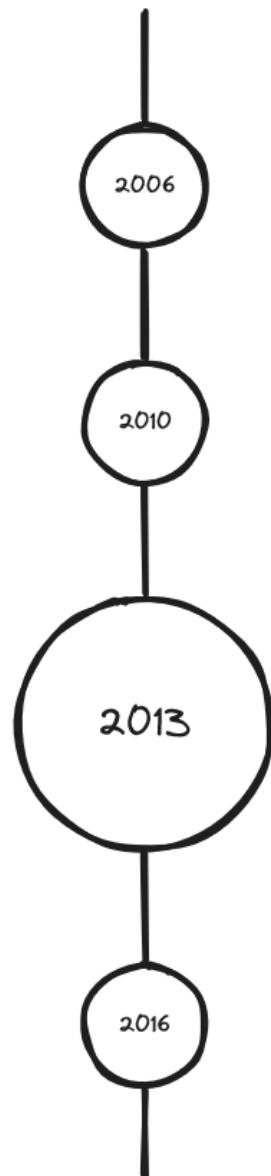
E ROS consente di evitare la riscrittura di driver e sistemi di comunicazione così da avere più tempo da dedicare allo sviluppo di funzionalità.

Prima versione
ufficiale

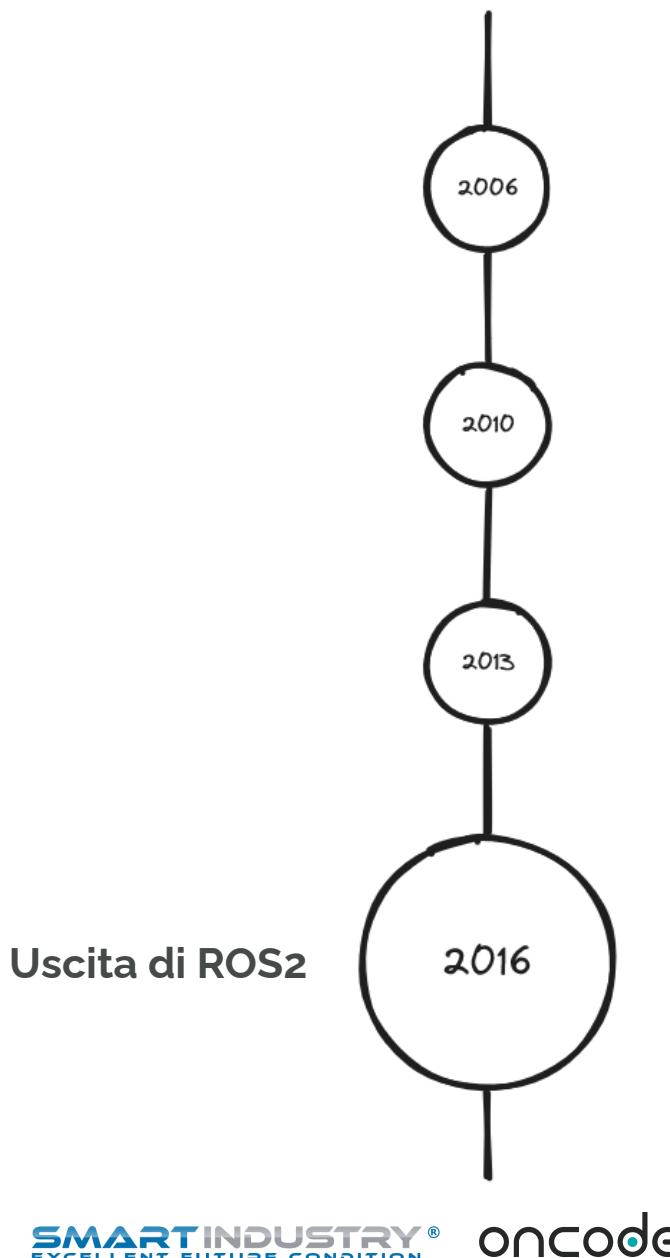


Viene rilasciata la prima versione ufficiale di ROS, «Box Turtle»

ROS diventa
open source



Gestione trasferita alla Open Source Robotics Foundation (OSRF),
successivamente ribattezzata Open Robotics



ROS2 è stato annunciato come un importante aggiornamento di ROS, con l'obiettivo di **migliorare la modularità, la portabilità e la sicurezza del sistema**.

È progettato per soddisfare le esigenze di una gamma più ampia di applicazioni, inclusi sistemi embedded e robot industriali.



ROS ha **un'adozione ampia e crescente** in numerosi settori, dalla ricerca accademica all'industria.

ROS è una risorsa fondamentale per la comunità della robotica, **offrendo uno standard** aperto per la progettazione e lo sviluppo di software robotico.

Perché ROS è basato su Linux?

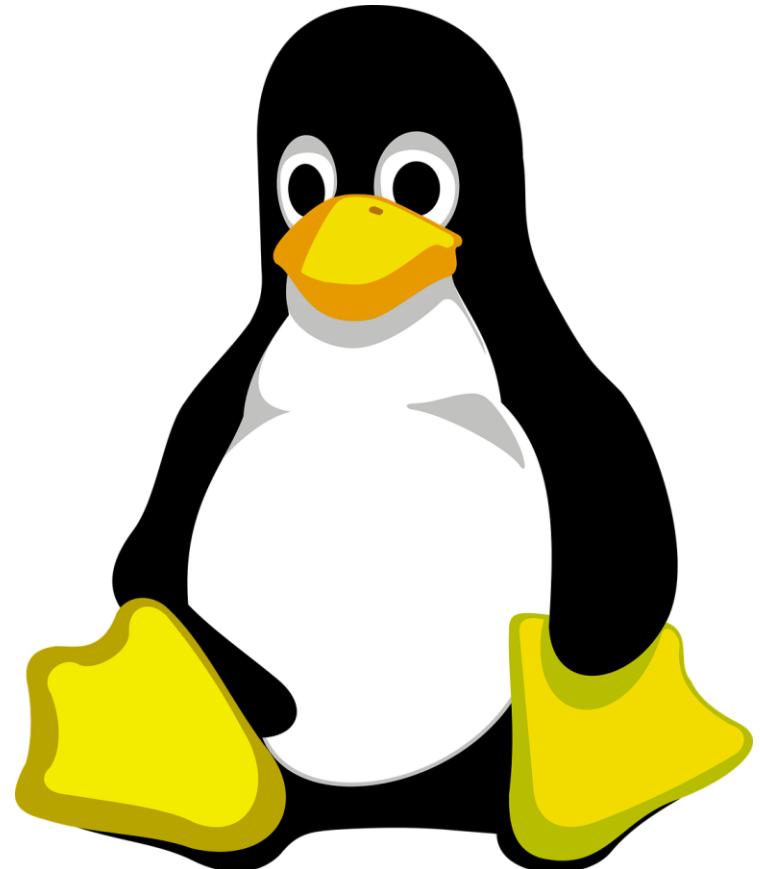
Filosofia Open Source

ROS è un framework open-source, e Linux offre un ambiente ideale per lo sviluppo e la distribuzione di software open source.

Linux favorisce la condivisione e la collaborazione

Architettura e Strumenti di Sviluppo

L'architettura di Linux è modulare e offre una vasta gamma di strumenti di sviluppo che semplificano la creazione, il debug e la gestione di software.



Perché ROS è basato su Linux?

Sistema di Controllo e Scheduling

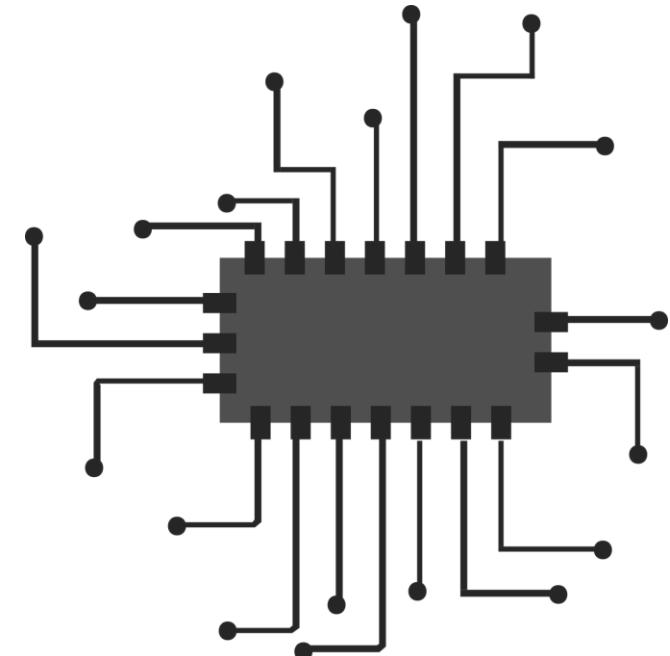
Il sistema di controllo e scheduling di ROS si basa sulle funzionalità di scheduling di Linux, garantendo una **gestione efficiente e real-time**.

Cruciale per garantire che i processi critici, come la lettura dei sensori e il controllo degli attuatori, siano eseguiti in modo tempestivo e predeterminato.

Kernel Personalizzabile

Linux offre un kernel altamente personalizzabile, consentendo agli sviluppatori di ottimizzare il sistema operativo per le specifiche esigenze della robotica.

Necessario per rispettare requisiti stringenti come operare in condizioni di real-time e gestire le risorse in maniera efficiente.

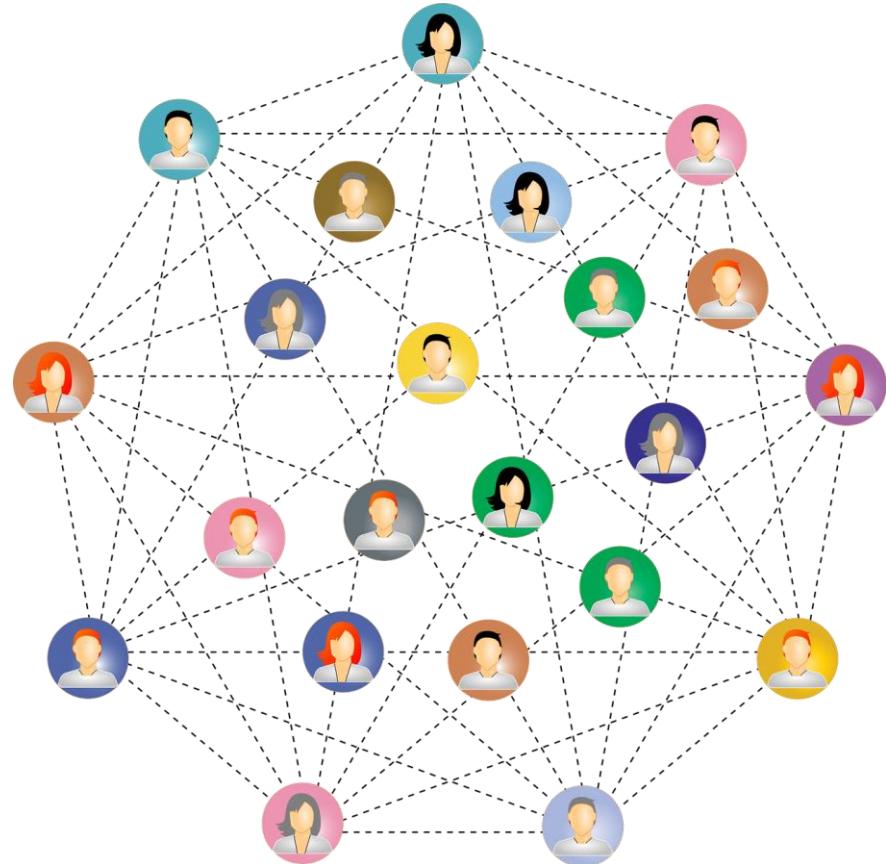


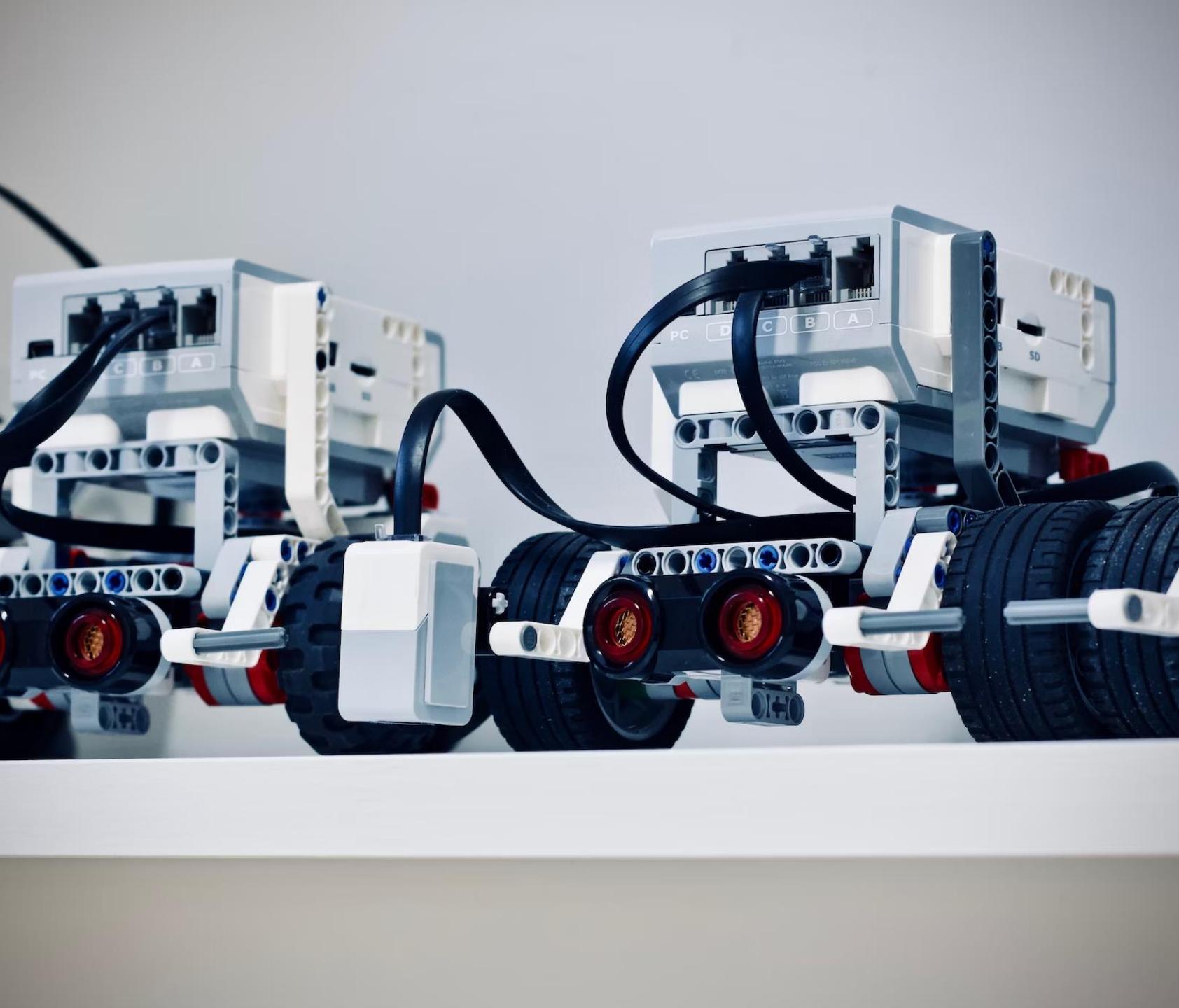
Perché ROS è basato su Linux?

Supporto e Community

Stabilità e affidabilità, essenziali nello sviluppo di sistemi robotici che operano in ambienti impegnativi, e fondamentali per garantire la **sicurezza e la robustezza dei robot** in varie situazioni.

La comunità di sviluppatori Linux è vasta e attiva e **contribuisce al supporto** di una ampia gamma di hardware. Ci sono costantemente risorse, supporto e aggiornamenti disponibili per affrontare le sfide emergenti nel campo della robotica.





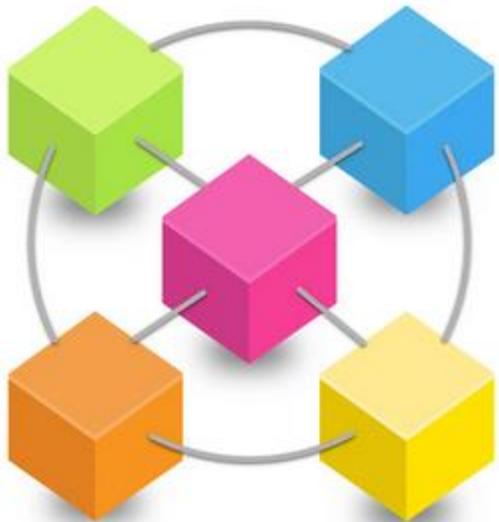
ROS

Perché usare ROS?



ROS

Architettura modulare



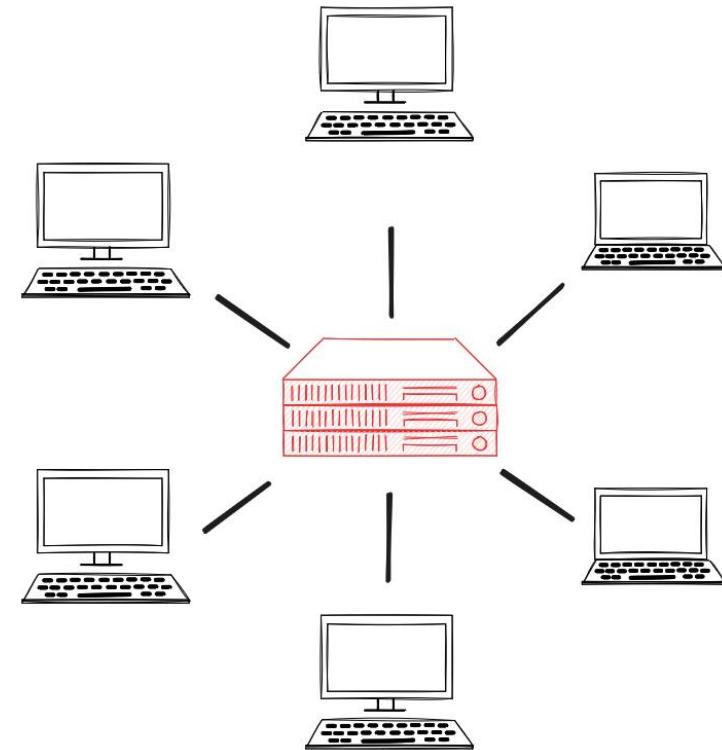
L'architettura modulare di ROS offre una piattaforma potente e flessibile per lo sviluppo di sistemi robotici avanzati.

Consente una progettazione efficiente, promuove la riutilizzabilità del codice e semplifica l'integrazione di nuovi componenti.

Gestione delle risorse

ROS fornisce strumenti per l'allocazione equa delle risorse, evitando che un nodo o un processo monopolizzi l'elaborazione della CPU.

Mantiene un ambiente di esecuzione bilanciato, riducendo il rischio di degradazione delle prestazioni dovuto al sovrautilizzo delle risorse da parte di un singolo componente.

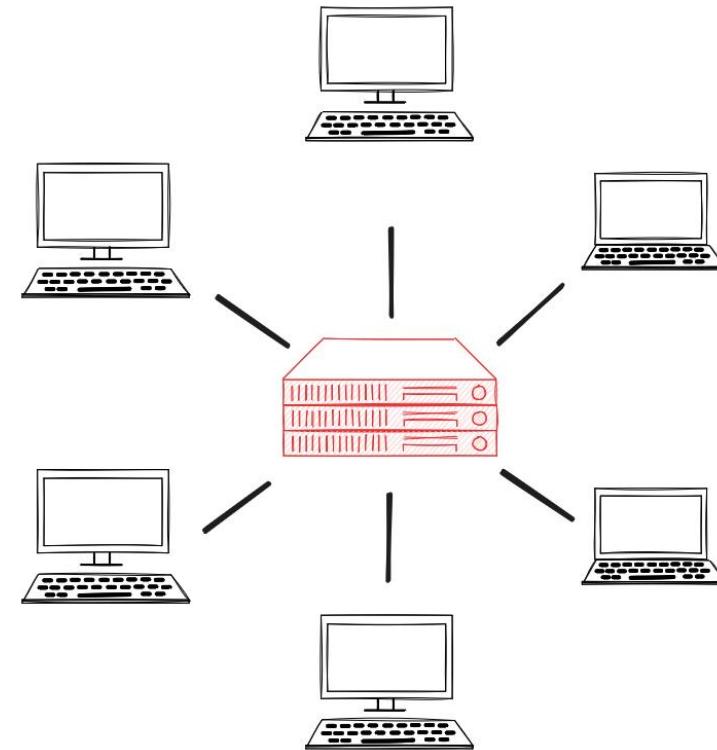


Gestione delle risorse

Organizzazione del software in «**nodi**» e in «**thread**».

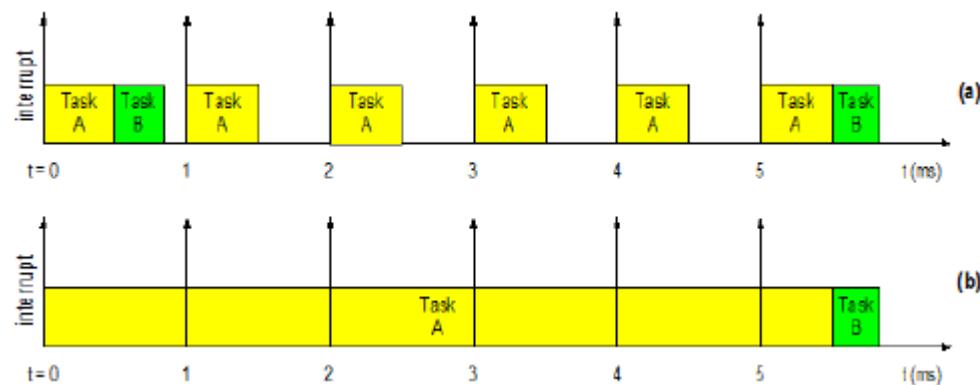
La **gestione efficace** dei nodi e dei thread è cruciale per bilanciare le priorità delle diverse attività.

Il **nodo responsabile del controllo della traiettoria di un veicolo** deve essere progettato per avere una **priorità più alta rispetto** a un **nodo** che gestisce la visualizzazione grafica dei dati.





ROS Scheduler

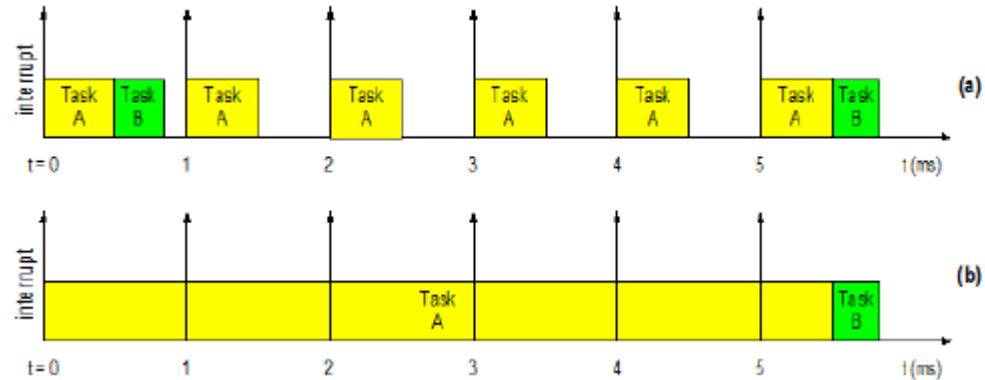


ROS fornisce un sistema di scheduling che gestisce l'esecuzione dei componenti in modo distribuito.

I componenti comunicano tra loro, e il sistema di scheduling si occupa di gestire i tempi di esecuzione.



ROS Scheduler



Il sistema di **scheduling** è cruciale per garantire la gestione tempestiva dei **processi critici**: raccolta di dati dai sensori, esecuzione di controlli, etc.

Assicurare che queste attività siano eseguite senza ritardi è fondamentale per mantenere il controllo e prevenire comportamenti indesiderati, come lo sbandamento di un veicolo.

 ROS

Simulazioni e Test in Ambiente Virtuale



ROS consente la simulazione di ambienti robotici senza la necessità di hardware fisico.

Rende possibile testare e validare il software in un ambiente virtuale prima di implementarlo su un robot reale.

Fondamentale per accelerare lo sviluppo e ridurre i rischi associati alla sperimentazione sul campo

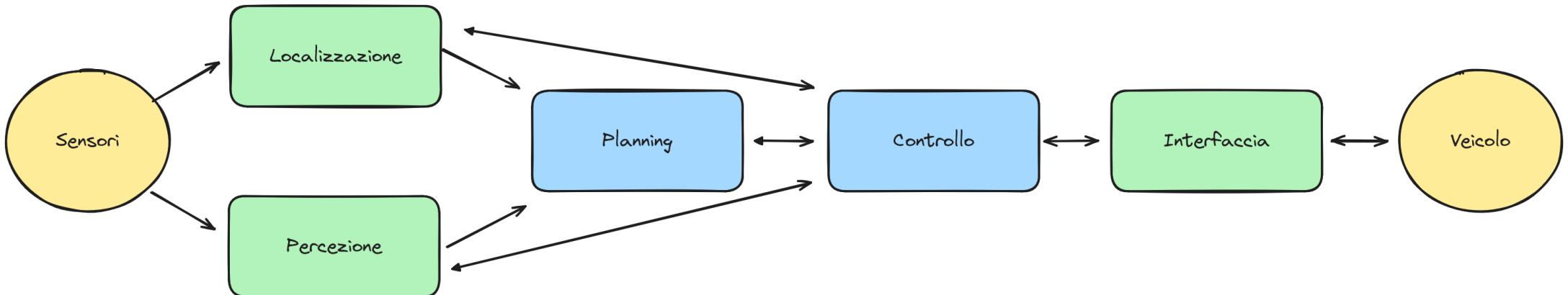
ROS nella guida
autonoma

**Uso di ROS nei veicoli a
guida autonoma**



Modularità e comunicazione

I sistemi a guida autonoma richiedono un'architettura modulare, in cui tanti componenti comunicano continuamente tra loro. È necessario coordinare diverse funzionalità e sensori.



Grazie all'architettura modulare basata su nodi, ROS facilita lo sviluppo e l'integrazione di diversi componenti software.

Facilita la comunicazione tra i componenti attraverso un sistema di messaggistica, consentendo una gestione efficiente dei dati tra i diversi componenti del sistema.

ROS nella guida autonoma

Ecosistema e supporto



Esiste già un vasto ecosistema di pacchetti che coprono molte delle funzionalità comuni allo sviluppo di sistemi a guida autonoma.

Accelerà il processo di sviluppo consentendo di utilizzare blocchi e driver esistenti.



ROS nella guida autonoma

Simulazione

La simulazione consente di:

- testare e validare i loro algoritmi e sistemi in un ambiente virtuale prima di passare alla fase di test sul campo
- ridurre il rischio di danni al veicolo in un ambiente controllato



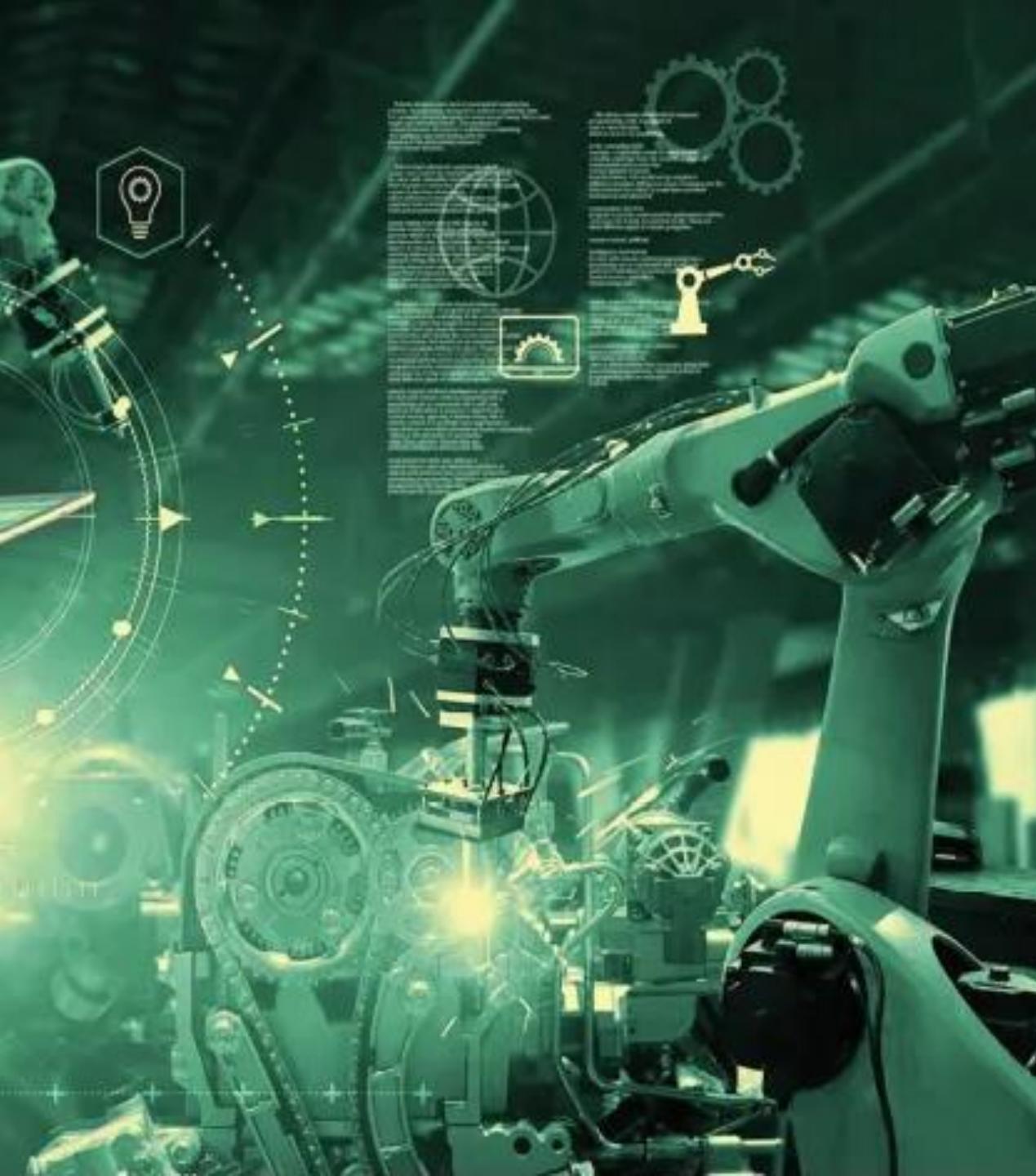
ROS nella guida autonoma

Open Source e community attiva



L'Open source è fondamentale per la ricerca e lo sviluppo di nuove tecnologie.

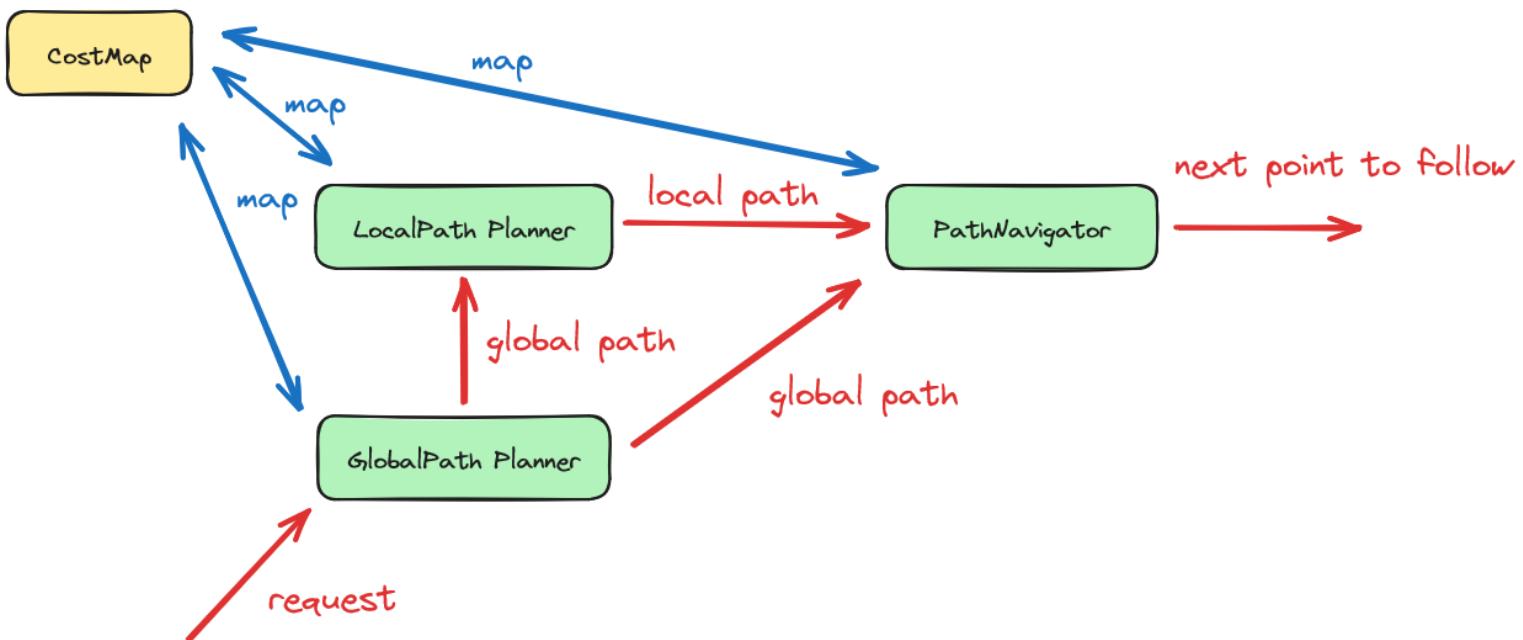
- Offre trasparenza
- Abbassa i costi di ingresso
- Facilita le università ad essere attivi nella ricerca



Componenti di ROS

Configuriamo un Path Planner

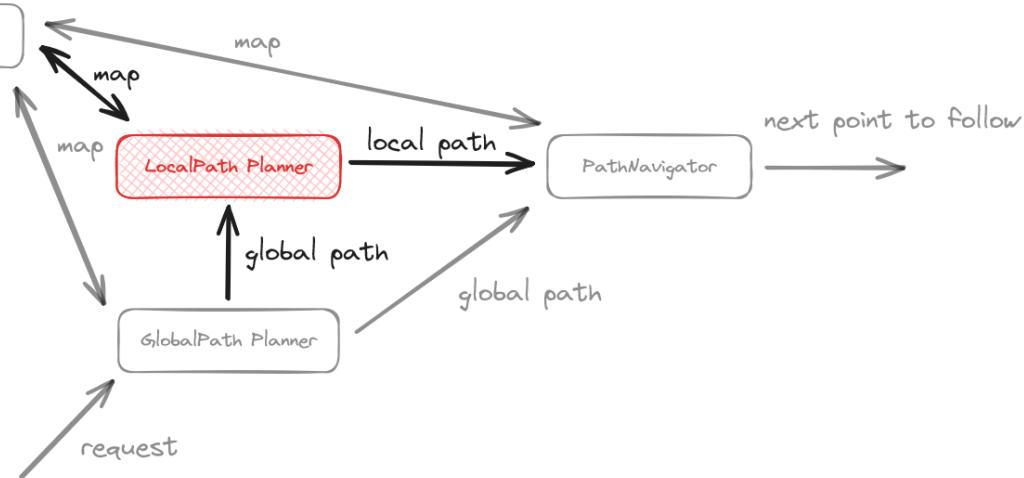
Cosa è un path planner?



Il compito principale di un path planner è quello di calcolare il percorso ottimale per muoverci da un punto A ad un punto B.

Tenendo conto di fattori come:

- la mappa dell'ambiente circostante
- la posizione corrente del veicolo
- la presenza di ostacoli
- le regole del traffico



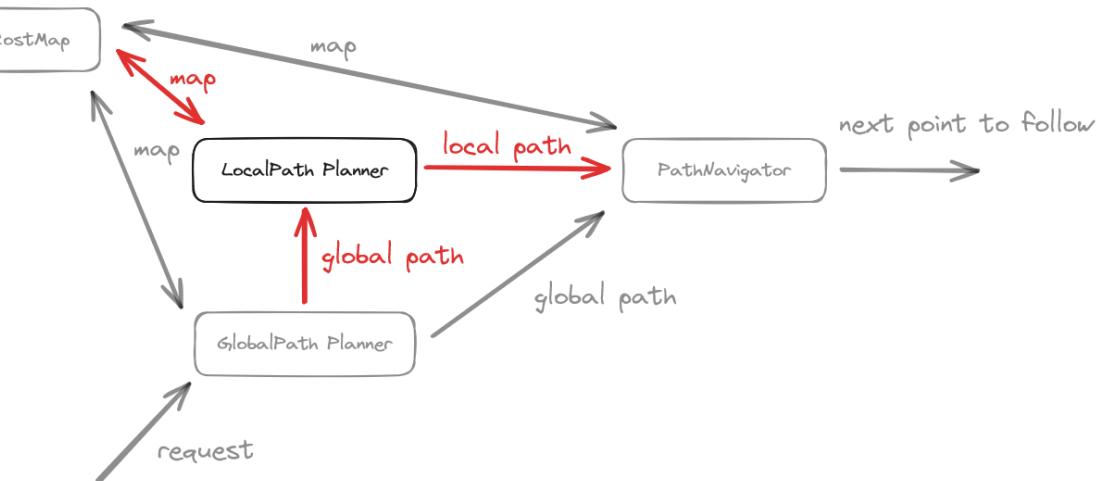
```
> rosrun path_planner_package localpath_planner.py
```

Un «nodo» è un processo autonomo che esegue un compito specifico.

I nodi comunicano tra loro attraverso il sistema di messaggistica di ROS, scambiando informazioni.

Il nodo «*local_planner*» è responsabile di generare un percorso locale per il robot, sulla base di: mappa dell'ambiente, percorso globale e informazioni dai sensori.

Componenti di ROS Messaggi

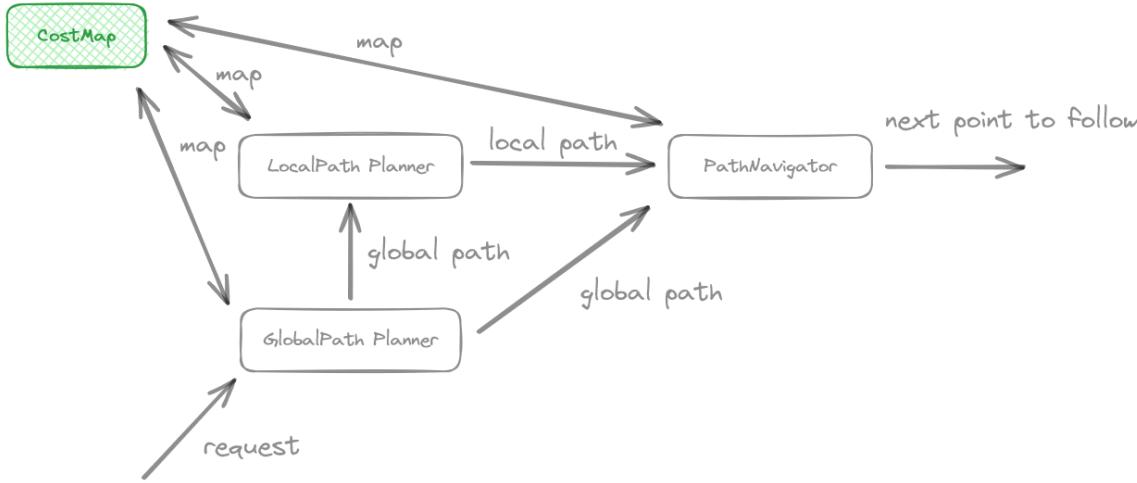


```
# Trajectory message
Header header
geometry_msgs/PoseStamped[] poses
```

```
# Position Message
Header header
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

Tra di loro i nodi si scambiano **Messaggi**.

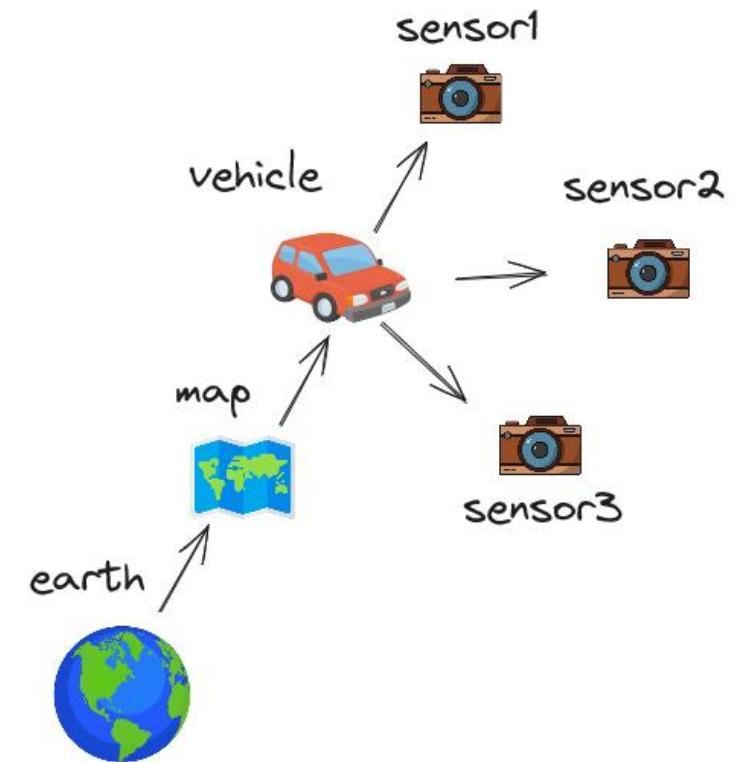
I messaggi ROS sono definiti utilizzando un linguaggio di specifica chiamato "**ROS message description language**". Questi messaggi possono rappresentare dati di sensori, comandi per i motori, informazioni sullo stato del robot o qualsiasi altra informazione che deve essere scambiata tra i nodi del sistema.

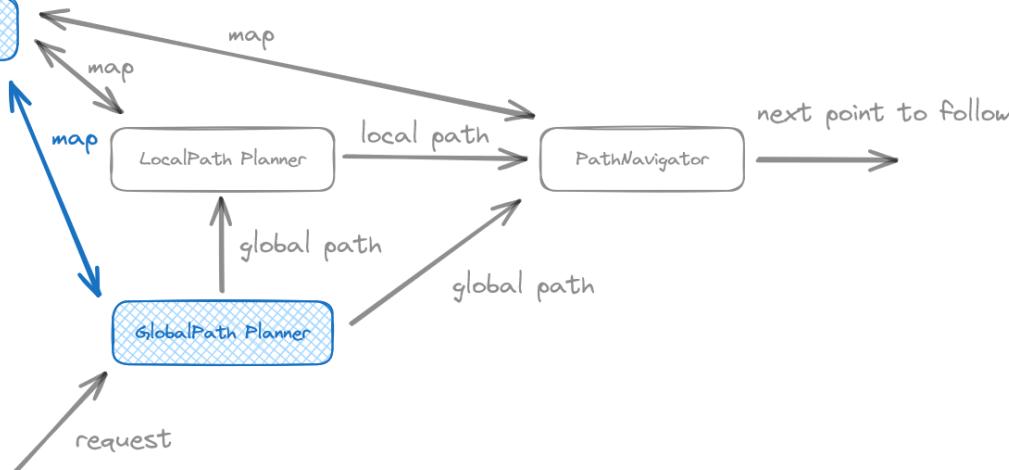


La costmap è una rappresentazione spaziale della probabilità che un determinato punto dello spazio sia occupato da un ostacolo.

Il sistema «Transform» TF tiene traccia delle trasformazioni tra i diversi frame di riferimento, e fornisce le funzioni per muoversi attraverso essi.

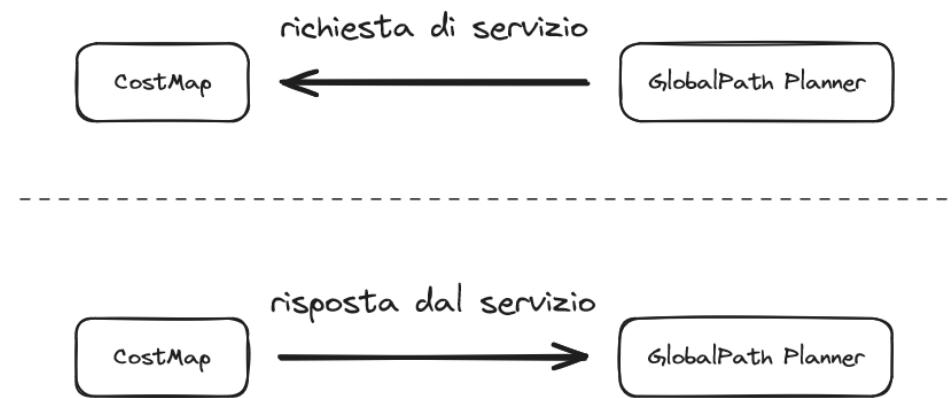
È fondamentale per gestire le relazioni spaziali tra diversi elementi di un sistema robotico.



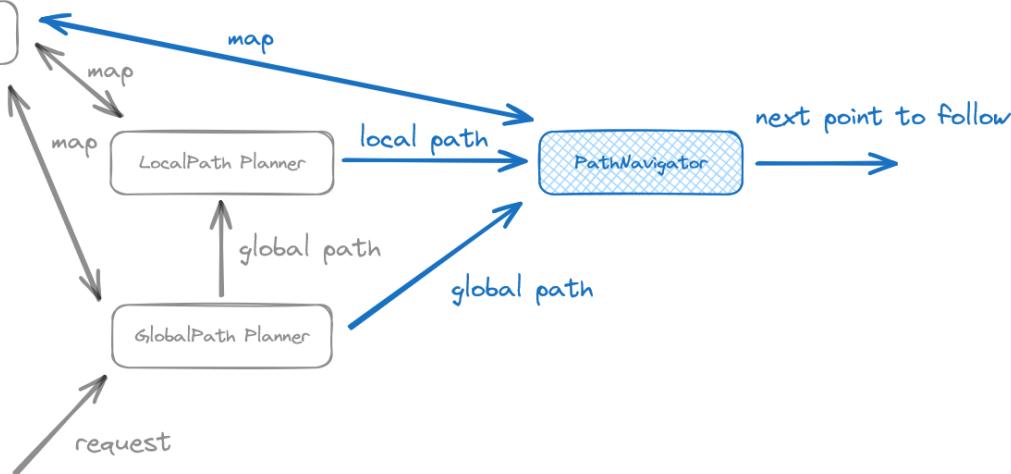


Un servizio è un meccanismo di comunicazione sincrona tra i nodi, permettono ad un nodo di richiedere un'azione specifica e ricevere un risposta.

Il nodo di pianificazione del percorso utilizza il servizio offerto dal nodo costmap per ottenere la mappa.



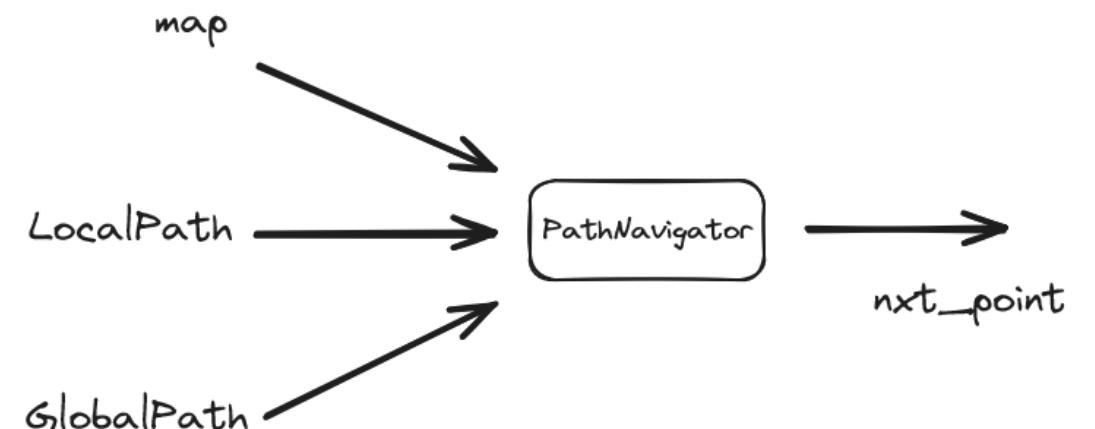
Publisher subscriber



Il modello **Publish-Subscribe** è un paradigma di comunicazione **asincrona** tra i nodi del sistema.

I nodi possono pubblicare un messaggio su un **Topic** ed un altro nodo può sottoscriversi per leggere il messaggio ed eseguire una callback.

I Topic sono un canale di comunicazione specifico per il tipo di messaggio.



Utilities di ROS

Tool molto utili per lo sviluppo



Launch file

Un file .launch è un XML che specifica come avviare un insieme di nodi e configurazioni del sistema.

```
<launch>
    <!-- Nodo della mappa -->
    <node pkg="costmap_2d" type="costmap_node" name="costmap_node"/>

    <!-- Nodo di pianificazione globale -->
    <node pkg="global_planner_pkg" type="global_planner_node" name="global_planner_node"/>

    <!-- Nodo di pianificazione locale -->
    <node pkg="local_planner_pkg" type="local_planner_node" name="local_planner_node">
        <param name="collision_threshold" type="double" value="0.2"/>
    </node>

    <!-- Nodo di movimento -->
    <node pkg="move_base" type="move_base" name="path_navigator">
        <remap from="original_topic_name" to="nxt_point_to_follow" />
        <param name="controller_frequency" type="double" value="10.0"/>
    </node>
</launch>
```

Semplifica l'esecuzione dell'applicazione, non dobbiamo avviare un singolo nodo alla volta



```
> roslaunch my_custom_pkg path_planning.launch
```

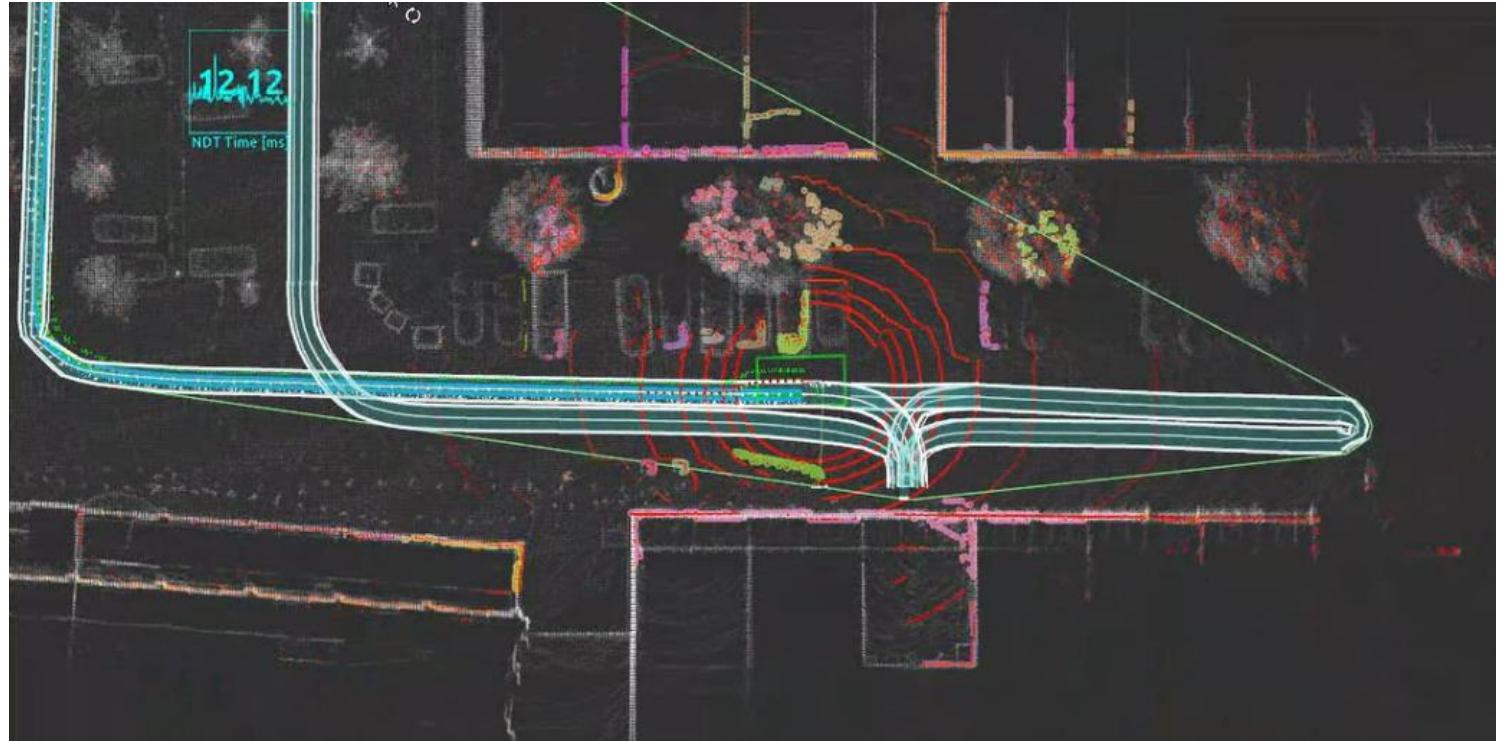
Debug e visualizzazione

ROS fornisce numerosi tool di debug utili nel processo di sviluppo, test e commissioning di un applicazione.

```
> rostopic list  
> rosnode list  
  
> rosnode info /local_planner_node  
> rostopic echo /nxt_point  
  
> rostopic pub /request position_msg "dati_del_messaggio"  
> rosparam set /local_planner_node/collision_threshold "10.0"
```

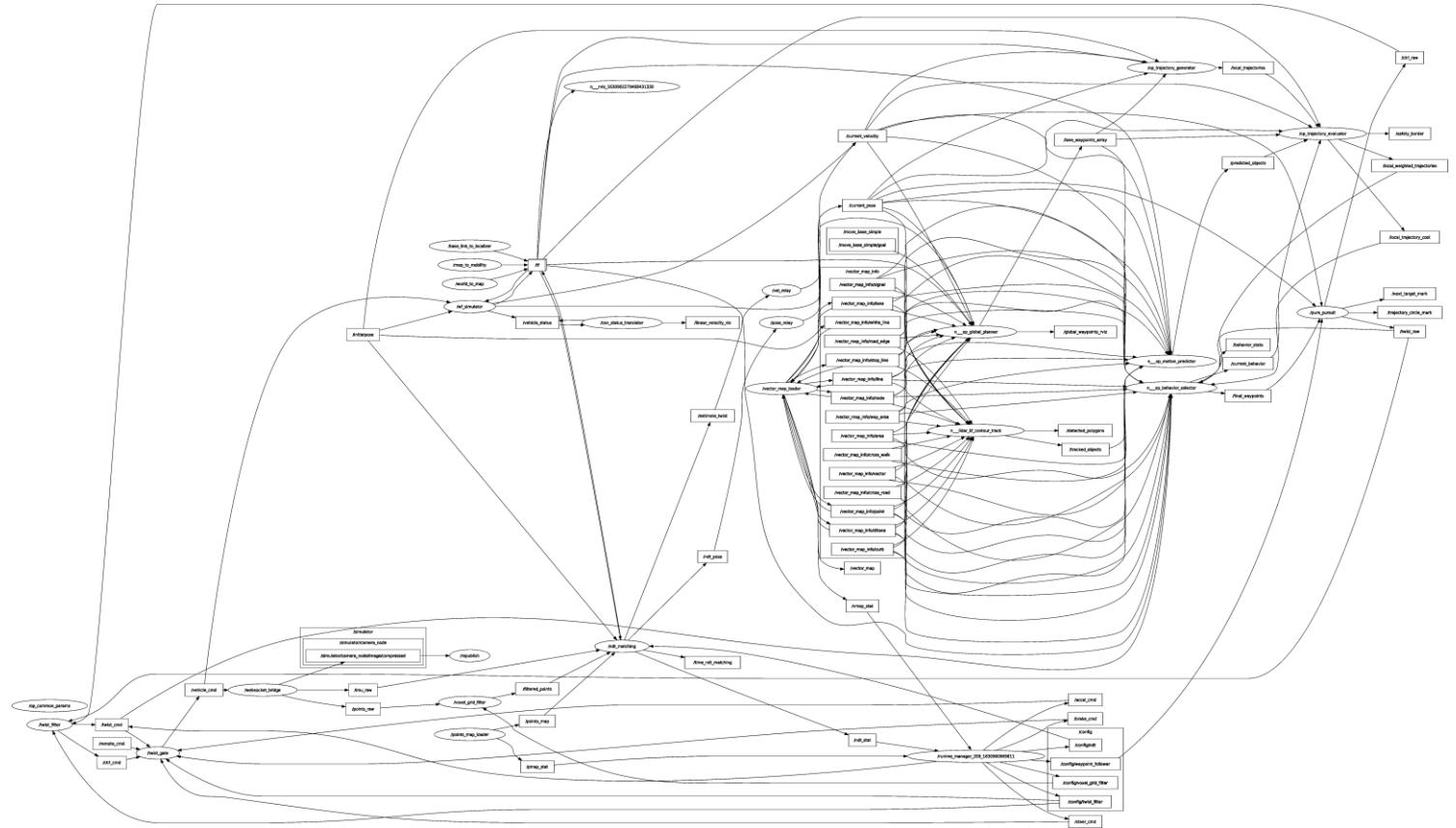
Debug e visualizzazione

Rviz è uno strumento di visualizzazione 3D che consente di visualizzare dati sensoriali, modelli robotici e altre informazioni cruciali.



Debug e visualizzazione

Rqt è un framework di sviluppo di interfaccia utente per ROS che fornisce diversi plugin utili per il debug come la visualizzazione dei messaggi di debug e il tracciamento dei dati in tempo reale.





Utilities di ROS

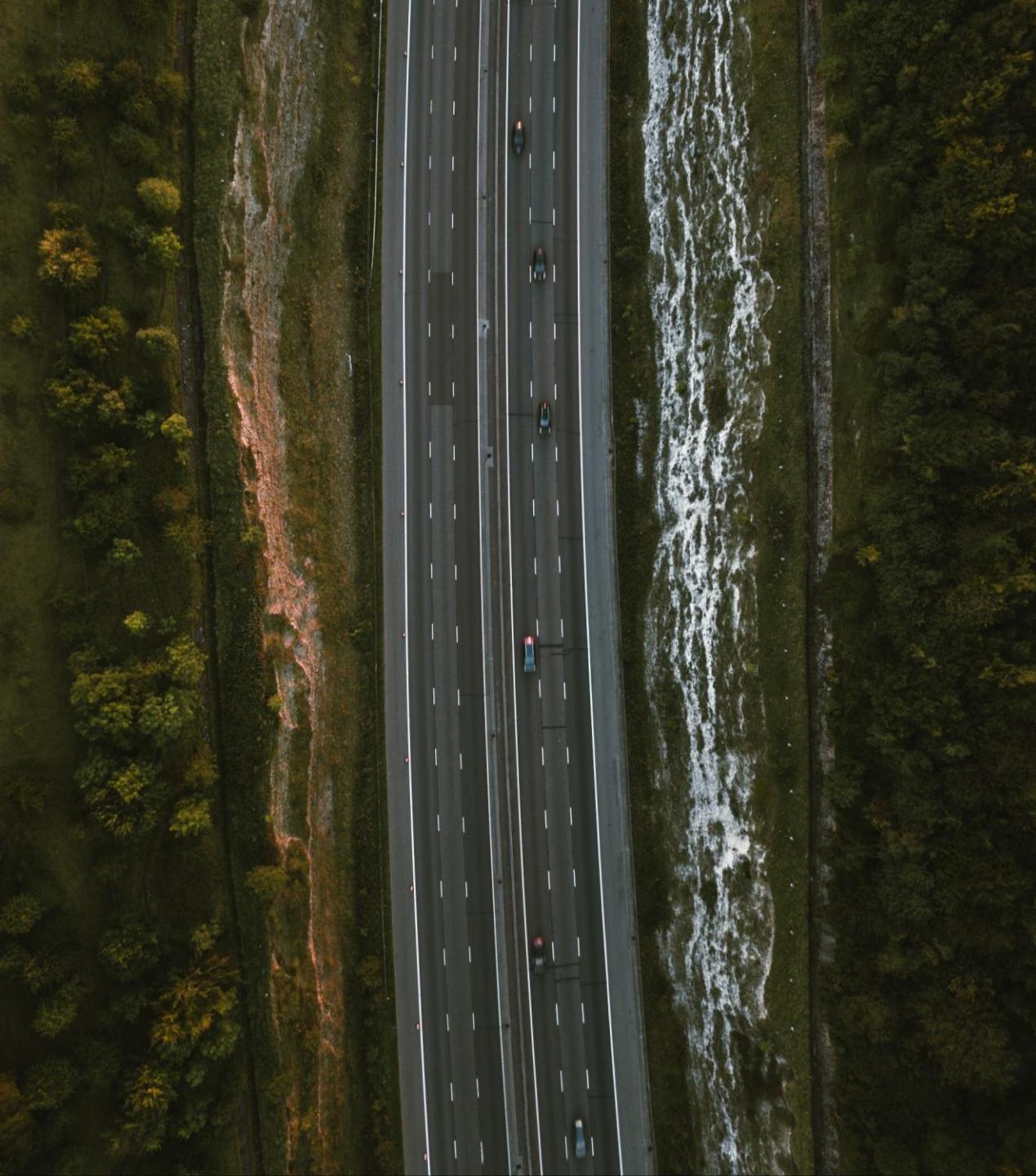
ROSBag

ROSbag è uno strumento fondamentale che consente di registrare e riprodurre dati dei topic ROS. È estremamente utile durante lo sviluppo, il testing e il debugging delle applicazioni.

L'utilizzo combinato dei .bag e degli strumenti Rviz e Rqt permette di poter testare l'applicazione un componente alla volta.



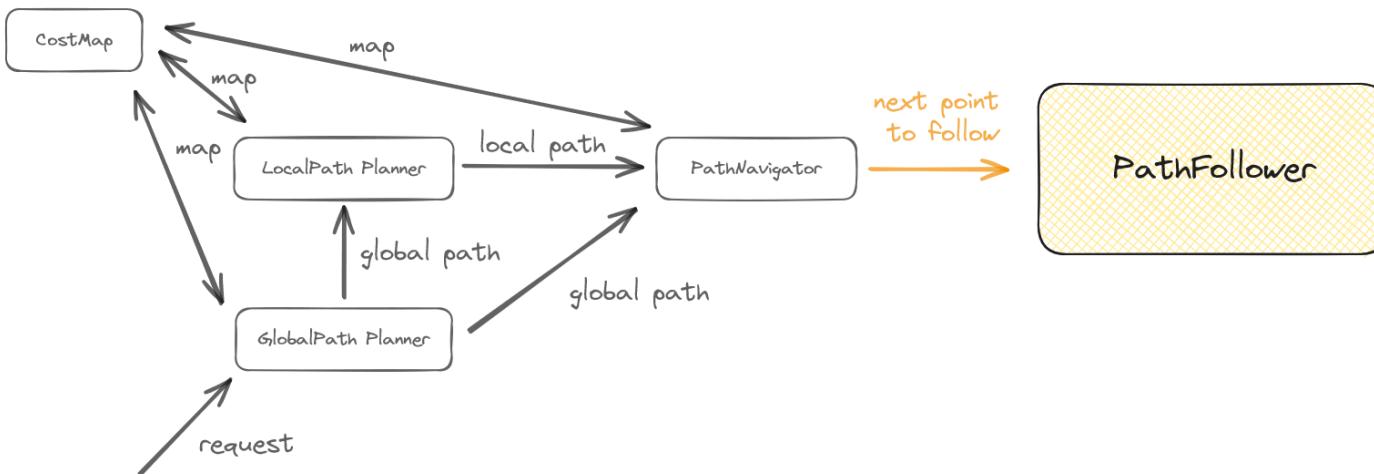
```
> rosbag play registrazione.bag
```



Path Follower

Creiamo il nostro pacchetto

Come rilasciare un pacchetto



Come è possibile usare pacchetti creati da altre persone già presenti nelle varie librerie, è anche possibile e facile creare i propri pacchetti.

Proviamo ad implementare il nostro Path Follower.

Come rilasciare un pacchetto

Per poter creare un pacchetto da poter riutilizzare bisogna prima di tutto creare un workspace.

```
> mkdir -p ~/catkin_ws/src  
> cd ~/catkin_ws/src  
> catkin_init_workspace  
> cd ..  
> catkin_make  
  
> cd ~/catkin_ws/src  
> catkin_create_pkg path_follower_pkg rospy geometry_msgs
```

Nota: in ROS2 il package manager si chiama «colon»

Come rilasciare un pacchetto

Il nodo sottoscrive la posizione attuale del veicolo da '/current_pose' e il prossimo punto da seguire da '/nxt_pt_follow'.

Invece il comando di sterzo viene pubblicato sul topic '/steering_cmd'.

```
import rospy
from geometry_msgs.msg import PoseStamped, Twist
from nav_msgs.msg import Path

class PathFollowerNode:
    def __init__(self):
        # Inizializzazione del nodo
        rospy.init_node('path_follower_node')

        # Sottoscrizione alla posizione attuale del veicolo
        rospy.Subscriber('/current_pose', PoseStamped, self.current_pose_callback)

        # Sottoscrizione al prossimo punto da seguire
        rospy.Subscriber('/nxt_pt_follow', PoseStamped, self.target_pose_callback)

        # Pubblicazione del comando di sterzo
        self.steering_cmd_pub = rospy.Publisher('/steering_cmd', Twist, queue_size=10)

        # Variabili di stato
        self.current_pose = None
        self.target_pose = None

    def current_pose_callback(self, current_pose):
        # Callback chiamata quando viene ricevuta la posizione attuale del veicolo
        self.current_pose = current_pose

    def target_pose_callback(self, target_pose):
        # Callback chiamata quando viene ricevuto il prossimo punto da seguire
        self.target_pose = target_pose
```

Come rilasciare un pacchetto

Abbiamo bisogno di una funzione per calcolare l'angolo di sterzo, che venga chiamata ciclicamente e che sulla base dei valori attuali della posizione e della posizione desiderata valuti il nostro angolo.

In figura è rappresentata una logica semplificata di quella che potrebbe essere una funzione reale del calcolo dell'angolo di sterzo.

```
def calculate_steering_cmd(self):
    # Logica di calcolo del comando di sterzo
    if self.current_pose is not None and self.target_pose is not None:
        # Calcolo l'angolo dall'orientamento
        current_yaw = euler_to_yaw(self.current_pose.pose.orientation)
        target_yaw = euler_to_yaw(self.target_pose.pose.orientation)

        # Calcolo il comando di sterzo
        steering_angle = calculate_steering_angle(current_yaw, target_yaw)

        # Pubblico il comando di sterzo
        steering_cmd = Twist()
        steering_cmd.angular.z = steering_angle
        self.steering_cmd_pub.publish(steering_cmd)

def calculate_steering_angle(current_yaw, desired_yaw):
    # Calcolo l'angolo di sterzo
    steering_angle = desired_yaw - current_yaw
    # Normalizzo l'angolo tra -pi e pi
    steering_angle = math.atan2(math.sin(steering_angle), math.cos(steering_angle))

    return steering_angle
```



Path Follower

Come rilasciare un pacchetto

Ci rimane da creare il main del nostro programma ed eseguirlo all'avviamento del nodo.

```
def run(self):
    # Ciclo principale
    rate = rospy.Rate(10) # Esecuzione a 10 Hz
    while not rospy.is_shutdown():
        self.calculate_steering_cmd()
        rate.sleep()

if __name__ == '__main__':
    # Istanzia il nodo PathFollower
    path_follower_node = PathFollowerNode()

    # Avvio il nodo
    path_follower_node.run()
```

Come rilasciare un pacchetto

Il file appena creato deve essere aggiunto al CMakeLists.txt del pacchetto.

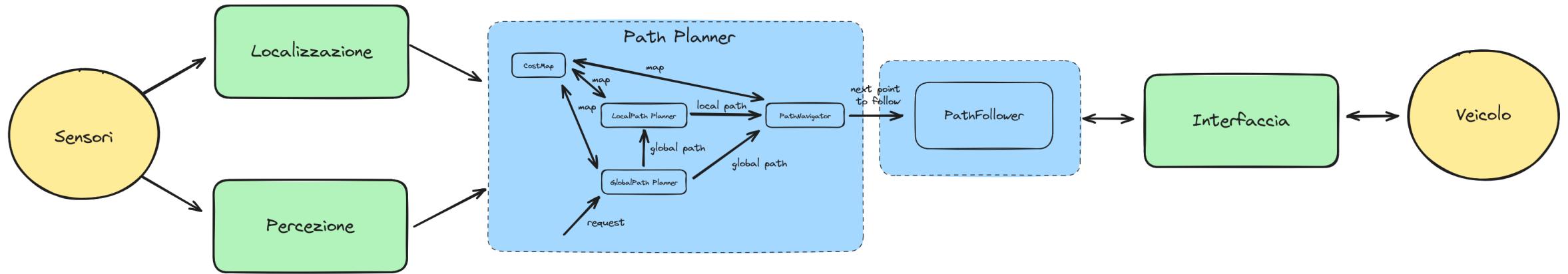
```
«~/catkin_ws/src/path_follower_pkg/CMakeList  
s.txt»
```

```
> add_executable(path_follower_node src/path_follower_node.py)  
> target_link_libraries(path_follower_node ${catkin_LIBRARIES})
```

Prima di poter eseguire il nostro pacchetto deve essere compilato e il sorgente deve essere reso eseguibile.

```
> catkin_make  
> source devel/setup.bash  
> rosrun path_follower_pkg path_follower_node.py
```

ROS per la guida autonoma





Reference

- [Documentazione ROS ufficiale](#)
- [Documentazione Autoware \(ROS per autonomous driving\)](#)
- [Video youtube: Programming for Robotics \(ROS\)](#)
- [Video youtube: Self-driving cars with ROS2 and Autoware](#)
- [Setup ROS environment](#)



Contatti

Website

<https://www.oncode.it/>

LinkedIn

<https://www.linkedin.com/company/oncodeit>

Smartindustry

<https://www.sindustry.it>

Alessandro Rossignoli

alessandro.rossignoli@oncode.it

Alberto Troiani

alberto.troiani@oncode.it

Fabio Ricci Curbastro

fabio.riccicurbastro@sengineering.it



Grazie per l'attenzione!

«Da soli si va più veloci,
insieme si va più lontano...»

[Proverbio africano]

Fabio Ricci Curbastro

Direttore Didattico & Talent Scout

fabio.riccicurbastro@sengineering.it

