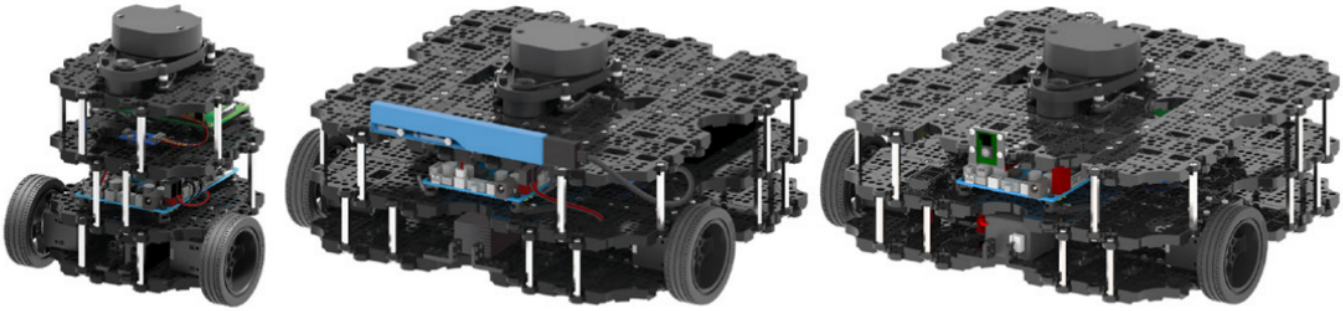
TURTLEBOT 3

Nella pagina del wiki (http://robots.ros.org/) è possibile trovare una vasta collezione di robot (terrestri, aerei, marini) che supportano ROS. Alcuni di questi includono robot personalizzati rilasciati pubblicamente dagli sviluppatori ed è un elenco notevole considerando che un singolo sistema supporta robot così diversi. Oltre 200 robot reali sviluppati con ROS sono ad oggi commercializzati sul mercato internazionale. Fra questi, uno dei più noti è sicuramente il **TurtleBot**.
Il **Turtlebot 3** è un robot compatto molto usato in ambito accademico e di ricerca grazie alla sua architettura *open-source* e alla facilità di riprogrammazione delle sue componenti. Grazie anche alle numerose integrazioni di cui può essere equipaggiato, il Turtlebot si presta anche molto bene alla prototipazione rapida (soprattuto per la validazione di nuovi algoritmi). Il Turtlebot 3 ha rilasciato 3 modelli ufficiali: Burger, Waffle e Waffle Pi.

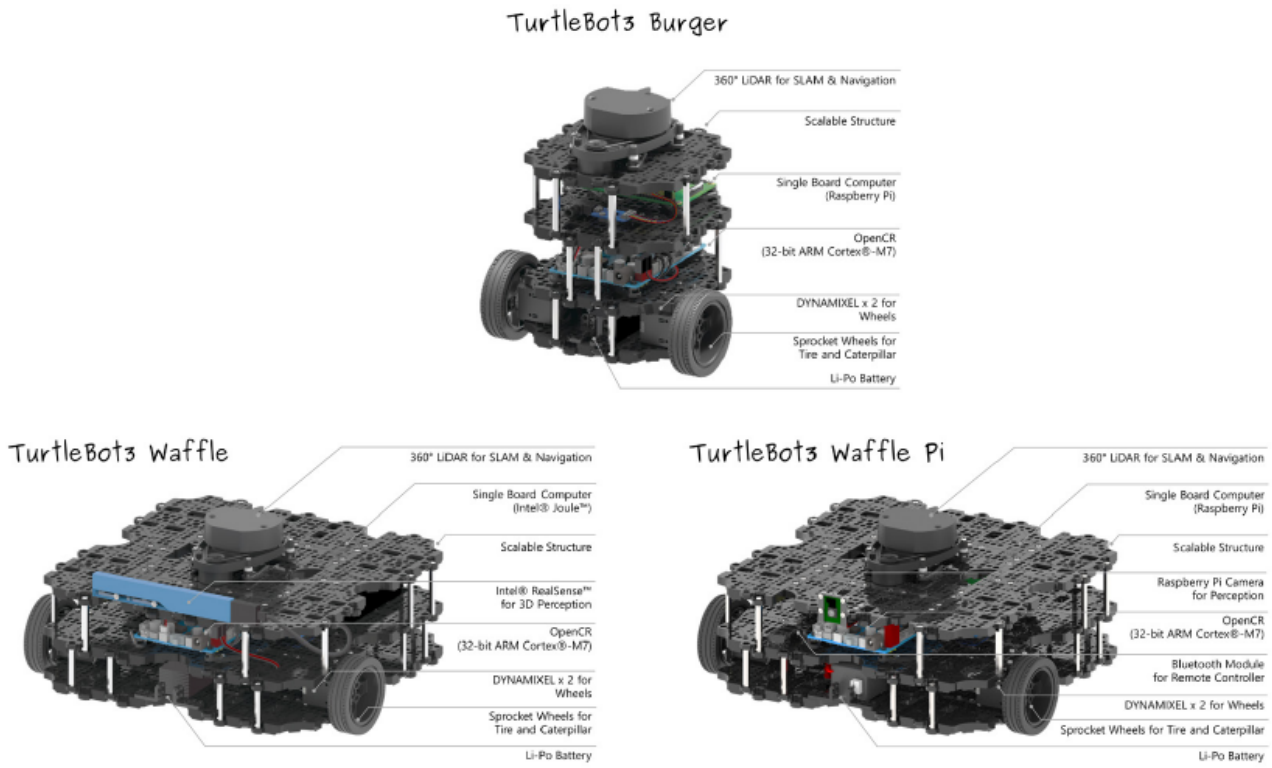


Hardware e software sono entrambi open-source (compresi i file `*.stl` del modello 3D) e distribuiti su cloud da parte della *ROBOTIS Co. Ltd*, ciò permette agli utenti di modificare e ersonalizzare ogni parte del robot secondo le proprie esigenze.

Hardware

Il Turtlebot 3 è un differential-drive basato su due motori *Dynamixel* per l'attuazione delle due ruote principali, un computer di bordo (*single-board computer*) per l'esecuzione di ROS (Raspbarry Pi o Intel Joule), una scheda a di controllo (*embedded board*) basata su microcontrollore *Cortex-M7*; è possibile inoltre equipaggiarlo di una serie di sensori, tra cui una IMU, una *depth-camera* per la ricostruzione 3D

dell'ambiente circostante, LiDaR, ecc.



ROS richiede un sistema operativo per essere eseguito, come potrebbe essere una distribuzione Linux; tuttavia è noto che i sistemi operativi convenzionali non sono in grado di garantire le prestazioni real-time necessarie per alcuni tipi di sensori e attuatori; per far fronte a i *single-board computer* sono in genere affiancati da moduli a microcontrollore per eseguire tali operazioni a basso livello.
Il TurtleBot3 fa uso di microcontrollori della famiglia ARM, serie Cortex-M7montati sulla scheda **OpenCR** (*Open-source Control Module for ROS*) a cui è demandata l'esecuzione dei task a basso livello.
La scheda OpenCR utilizza un chip *STM32F746* come MCU principale, e fornisce un'interfaccia compatibile con i prodotti *Arduino UNO*, ciò garantisce la presenza e la compatibilità di una vasta gamma di librerie, codici sorgente e schede di espansione, sviluppate per l'ambiente di sviluppo di Arduino.
La scheda embedded OpenCR supporta le principali interfacce di comunicazione: UART, I2C, SPI, CAN, TTL, RS485, e include un chip *MPU925010*, il quale integra al suo interno un giroscopio a tre assi, un accelerometro a tre assi e un magnetometro a tre assi; ciò permette di avere già a disposizione un IMU necessaria in un gran numero di applicazioni base.
OpenCR fornisce un firmware atto a comunicare con le varie periferiche della scheda, il quale può essere liberamente scaricato e caricato sulla scheda attraverso l'IDE di Arduino.

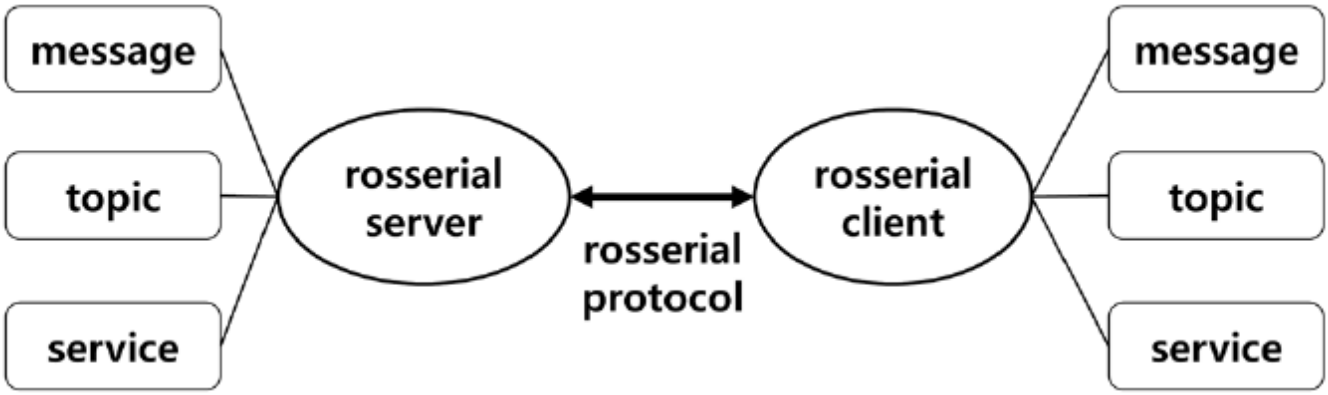
Software

Il software a bordo del Turtlebot 3 suddiviso nel seguente modo:

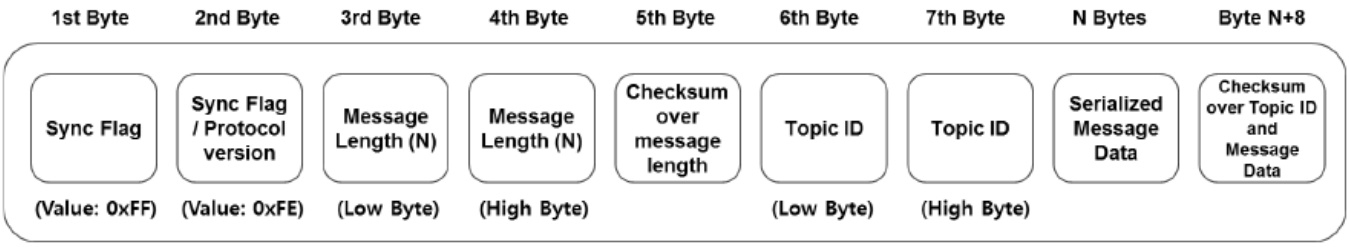
- un firmware usato come controllore fisico, in esecuzione sulla scheda embedded;
- 4 package ROS eseguibili sul computer di bordo.

Il firmware in esecuzione sulla scheda OpenCR è noto come `turtlebot3_core` e viene caricato sulla scheda embedded che si trova a bordo del Turtlebot. Tale firmware ha il compito di gestire gestire e attuare i motori, leggere i dati degli encoder e degli altri sensori a bordo (IMU) e fornire una stima odometrica della posizione del veicolo. La comunicazione tra la scheda a microcontrollore e il computer di

bordo sul quale è in esecuzione il `roscore` può avvenire sempre attraverso il meccanismo di scambio di messaggi via topic, ciò grazie a `rosserial`, un protocollo per incapsulare (*wrapping*) i messaggi standard di ROS, serializzarli e inviarli (attraverso un opportuno sistema di smistamento - *multiplexing* - verso diversi topic/servizi) tramite seriale al microcontrollore.
La libreria `rosserial` lavora secondo un meccanismo client/server per rendere possibile lo scambio di messaggi tra la scheda a microcontrollore e il computer di bordo: il PC che esegue ROS riveste il ruolo di server, mentre la scheda a microcontrollore agisce da client.



Il server è realizzato attraverso un nodo ROS che fa uso di un'apposita libreria (si faccia riferimento al package ROS relativo al linguaggio di programmazione utilizzato: `rosserial_python`, `rosserial_server` per C++, `rosserial_java`) per implementare tali funzionalità; mentre la libreria client supporta tutte le piattaforme *Arduino* o *mbed* compatibili (si segnala fra le varie librerie client anche la presenza di `rosserial_stm32`).
Il protocollo `rosserial` specifica la struttura a livello byte dei pacchetti dati che vengono scambiati dalle librerie client e server attraverso interfaccia di comunicazione seriale; ogni pacchetto contiene, oltre al dato uile, una serie di informazioni usate per la sincronizzazione e la validazione dei dati.



Un pacchetto `rosserial` è composto da un *header* che contiene informazioni generali sul dato e dei campi per la verifica dell'integrità del dato (*checksum*).

- **Sync Flag**: questo byte è posto sempre a 0xFF, e sta a indicare l'inizio del pacchetto.
- **Sync Flag / Protocol version**: questo byte indica la versione del protocollo ROS, dove per Groovy è 0xFF, mentre per Hydro, Indigo, Jade, Kinetic, Melodic e Noetic è 0xFE.
- **Message Length**: questi due bytes contengono la lunghezza del dato che rappresenta il messaggio che si sta scambiando. Il *low byte* viene inviato per primo, seguito dallo *high byte* (notazione *little endian*).
- **Checksum over message length**: il valore del checksum viene utilizzato per verificare la validità del valore di lunghezza del dato ricevuto, calcolato secondo la formula `255 - ((message_length_low_byte + message_length_high_byte) % 256)`.

- **Topic ID**: il valore dell'ID consiste di due byte ed è usato come identificatore per distinguere il tipo di messaggio ricevuto. I valori di ID compresi tra 0 e 100 sono riservati per funzioni di sistema. I principali ID utilizzati dal sistema sono i seguenti:
  ```
  uint16 ID_PUBLISHER=0
  uint16 ID_SUBSCRIBER=1
  uint16 ID_SERVICE_SERVER=2
  uint16 ID_SERVICE_CLIENT=4
  uint16 ID_PARAMETER_REQUEST=6
  uint16 ID_LOG=7
  uint16 ID_TIME=10
  uint16 ID_TX_STOP=11
  ```
- **Serialized Message Data**: questo campo contiene il messaggio vero e roprio serializzato.
- **Checksum over topic ID and Message Data**: il valore di tale checksum è utilizzato per validare Topic ID e messaggio ricevuto, questo viene calcolato secondo la seguente formula: `255 - ((topic_ID_low_byte + topic_ID_high_byte + data_byte_values) % 256)`
- **Query Packet**: quando il server `rosserial` viene eseguito, questo richiede alcune informazioni al client, quale i nomi dei topic e i relativi tipi di messaggio. Quando viene effettuato lo scambio di tali informazioni, vengono utilizzati dei pacchetti speciali definiti *query packet*, il cui Topic ID e la dimensione del pacchetto sono settati ambedue a 0. Di seguito sono riportati i dati presenti in un `query packet`.
  `0xff 0xfe 0x00 0x00 0xff 0x00 0x00 0xff` Quando un client riceve un pacchetto di questo tipo, esso invia in risposta un messaggio al server contenente le seguenti informazioni. `uint16 topic_id`
  `string topic_name`
  `string message_type`
  `string md5sum`
  `int32 buffer_size`

> ⚠️ **LIMITAZIONI DI ROSSERIAL**
>
> Attraverso la libreria `rosserial` è possibile scambiare messaggi standard di ROS con dispositivi embedded basati su microcontrollore che dispongano di una periferica di comunicazione UART; tuttavia vi sono alcune limitazioni dovute all'hardware che potrebbero sollevare qualche problema. Per esempio, vincoli dovuti alle risorse di memoria (un numero di topic troppo elevato - dalla documentazione ufficiale: superiore ai 25 - potrebbe sollevare problemi) o tipi di dato non supportati, quali `Float64` e `String`.

Sul computer di bordo vengono invece eseguiti gli altri package ROS, i quali sono orgamizzati come segue:

- `turtlebot3`: contiene i modelli del Turtlebot, i package contenti gli algoritmi necessari allo SLAM e alla navigazione, il package per il controllo remoto e il package `bringup`;
- `turtlebot3_msgs`: contiene i file dei messaggi utilizzati negli altri package;
- `turtlebot3_simulations`: contiene i file necessari a simulare il Turtlebot su Gazebo.
- `turtlebot3_applications`: fornisce alcuni esempi di applicazioni con il Turtlebot 3.

Installazione dei package del TurtleBot3

Installazione delle dipendenze

Prima di procedere all'installazione dei package relativi al Turtlebot 3, è consigliata l'installazione di alcuni package di dipendenza nella cartella principale di ROS: `/opt/ros/melodic`.

```
$ sudo apt-get install ros-melodic-joy ros-melodic-teleop-twist-joy ros-
melodic-teleop-twist-keyboard ros-melodic-laser-proc ros-melodic-rgbd-
launch ros-melodic-depthimage-to-laserscan ros-melodic-rosserial-arduino
ros-melodic-rosserial-python ros-melodic-rosserial-server ros-melodic-
rosserial-client ros-melodic-rosserial-msgs ros-melodic-amcl ros-melodic-
map-server ros-melodic-move-base ros-melodic-urdf ros-melodic-xacro ros-
melodic-compressed-image-transport ros-melodic-rqt-image-view ros-melodic-
gmapping ros-melodic-navigation ros-melodic-interactive-markers
```

Installazione dei package del TurtleBot3

È possibile installare i package relatii al Turtlebot 3 all'interno del proprio workspace scaricandoli dal repository github ufficiale: https://github.com/ROBOTIS-GIT.

```
$ cd ~/ros_ws/src/
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3.git
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
$ cd ..
$ catkin_make
```

> **PROBLEMI DURANTE LA COMPLIAZIONE**
>
> È possibile che si incontrino dei problemi in fase di compilazione con il pacchetto Python *em*; installare il pacchetto nel seguente modo:
>
> ```
> $ python -m pip install empy
> $ python3 -m pip install empy
> ```

Simulazione

Per lavorare nell'ambiente simulativo con Gazebo e Rviz è necessario specificare con quale modello di Turtlebot 3 si vuole operare (`burger`,`waffle` e `waffle_pi`). Ciò viene fatto specificando una variabile d'ambiente come riportato di seguito:

```
$ export TURTLEBOT3_MODEL=waffle_pi
```

> ⏩ **SET ENVIRONMENT VARIABLE**
>
> Per migliorare il flusso di lavoro, potrebbe essere conveniente includere il setaggio della variabile d'ambiente nel file `.bashrc` per evitare di dover eseguire il comando in ogni nuovo terminale.
>
> ```
> # Export Robot Model
> export TURTLEBOT3_MODEL=waffle_pi
> ```

Spawn del modello in Gazebo

Per eseguire l'ambiente di simulazione Gazebo eseguire da terminale

```
$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

o

```
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

nel package `turtlebot3_description` sono presenti i file che descrivono la struttura del Turtlebot 3 attraverso file `*.urdf`, le specifiche fisiche per le simulazioni in Gazebo e i file `*.mesh` usati per modellare la struttura 3D del robot. Il package `turtlebot3_gazebo` contiene principalmente i file `*.launch` per effettuare lo *spawn* del modello del robot in differenti scenari di Gazebo.

Guida del Turtlebot 3

Per teleguidare il Turtlebot è disponibile il nodo `turtlebot3_teleop` nell'omonimo package, che permette la lettura dei dati da tastiera e l'invio di comandi di velocità sul topic `cmd_vel`. In un terminale digitare:

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Visualizzazione su Rviz

Per visualizzare le percezioni del robot - come la sua posizione, il suo assetto, o i dati raccolti dai sensori - è possibile usare Rviz per raccogliere le informazioni pubblicate sui topic e visualizzarle graficamente:

```
$ roslaunch turtlebot3_gazebo turtlebort3_gazebo_rviz.launch
```

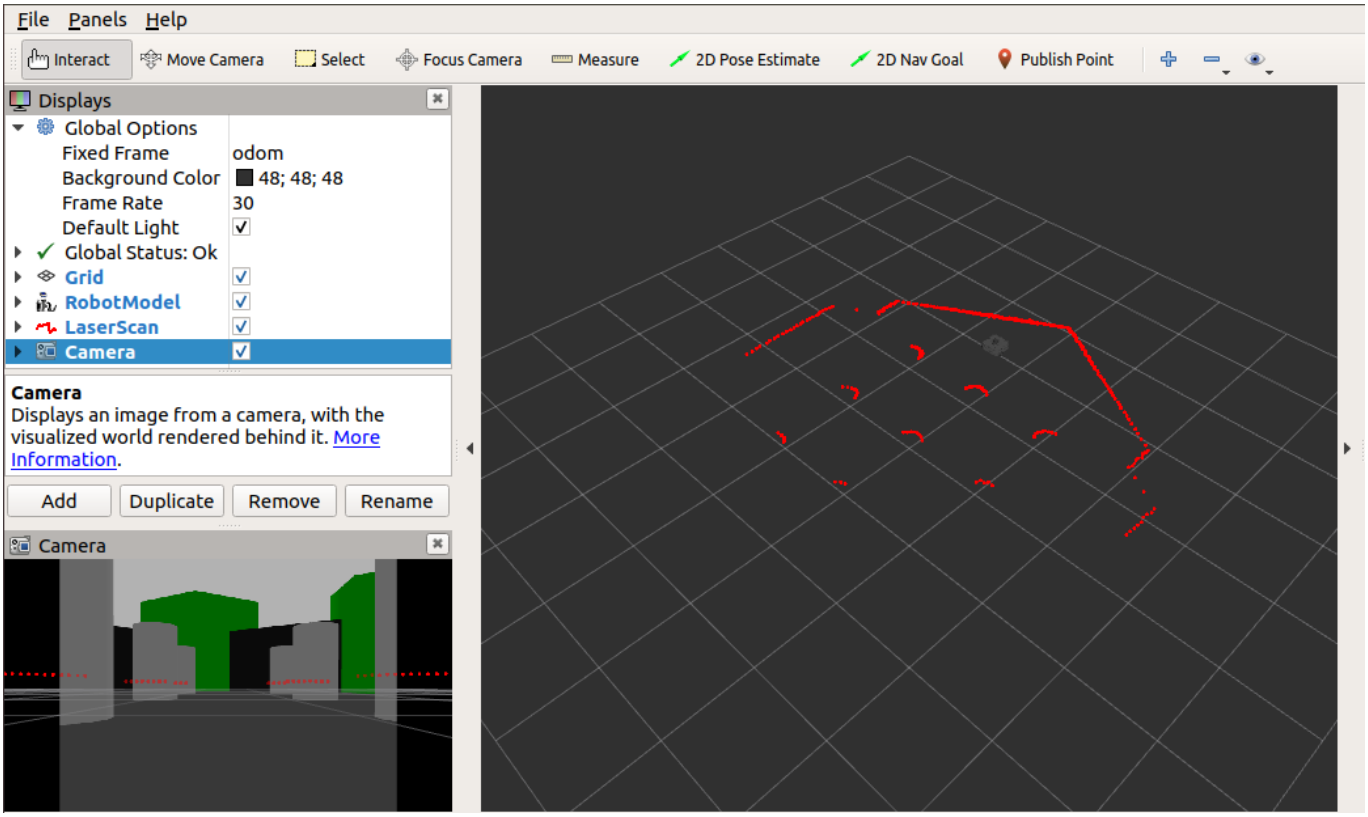
oppure far partire manualmente Rviz selezionando quali informazioni si vogliono visualizzare aggiungendo gli appositi oggetti.

```
$ rviz
```

Se viene seguita quast'ultima strada è necessario, oltre all'esecuzione *standalone* di Rviz, l'esecuzione del nodo `robot_state_publisher` (dell'omonimo package)

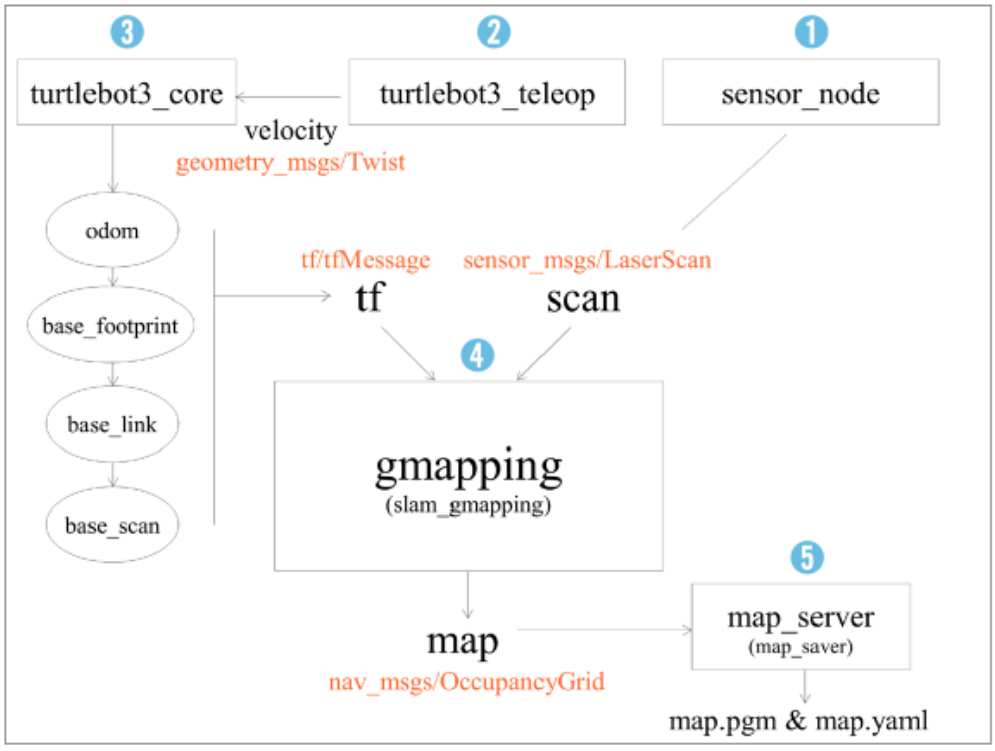
```
$ rusrun robot_state_publisher robot_state_publisher
```

In funzione del modello di Turtlebot 3 scelto, sarà possibile aggiungere la visualizzazione di diversi sensori; per conoscere l'insieme di sensori con cui è equipaggiato il modello scelto, una possibile strada è quela di andare a visionare il contenuto del relativo file `*.gazebo.xacro` presente nel package `turtlebot3_description`.
Per esempio, volendo utilizzare il Turtlebot 3 Waffle Pi, è possibile ottenere informzaioni sull'ambiente circostante attraverso i dati raccolti dal LiDaR e dalla Camera RGB. Dopo aver aggiunto il modello del robot, è possibile provare ad aggiungere oggetti di tipo `LaserScan` e `Camera`, dopodiché selezionare i topic `/scan` per il sensore laser e il topic `/camera/rgb/image_raw` per la camera.



Costruzione di una Mappa

Un'aspetto molto importante al fine della navigazione autonoma di un robot è la capacità di costruire una mappa dell'ambiente in cui si muove. La capacià di costruzione autonoma permette di non dover effettuare in precedenza un'operazione di costruzione, in alcuni casi non possibile.
In ROS le mappe di navigazione possono essere rappresentate con una griglia bidimensionale (nel caso di movimenti su un piano), dove ogni casella della matrice contiene un valore numerico che corrisponde al *"grado di occupazione"* della porzione di spazio corrispondente: il bianco identifica uno spazio libero, il nero spazio occupato, il grigio è associato a spazio ignoto (non ancora splorato). I file rappresentanti una mappa possono essere salvati come file immagine in uno dei classici formati: `*.png`, `*.jpg`, `*.pmg`. Associato ad ogni immagine vi è un file `*.yaml` contenente informazioni aggiuntive su come interpretare la mappa: la risoluzione (la dimensione del lato di una cella espressa in metri), l'origine della mappa, e i valori di soglia (*threshold*) necessari a stabilre se una cela può essere considerata libero, occupata o ignota.
ROS mette a disposizione più di un package per la costruzione delle mappe. Tra i vari package, uno dei più noti è `gmapping`, che registra i valori acquisiti dai sensori laser mentre simultaneamente localizza il robot facendo uso dei dati odometrici pubblicati. Affinché la costruzione della mappa avvenga in maniera efficace, è necessario ripercorrere più volte l'intera mappa, prestando attenzione ad eseguire movimenti lenti (soprattuto nelle operazioni di rotazione) affinché l'algoritmo riesca a raccogliere quanti più dati possibile dell'ambiente circostante.
Utilizzando il nodo `slam_gmapping` del package `gmapping` è possibile cosruire una mappa dell'ambiente di simulazione entro cui ci si sta muovendo.



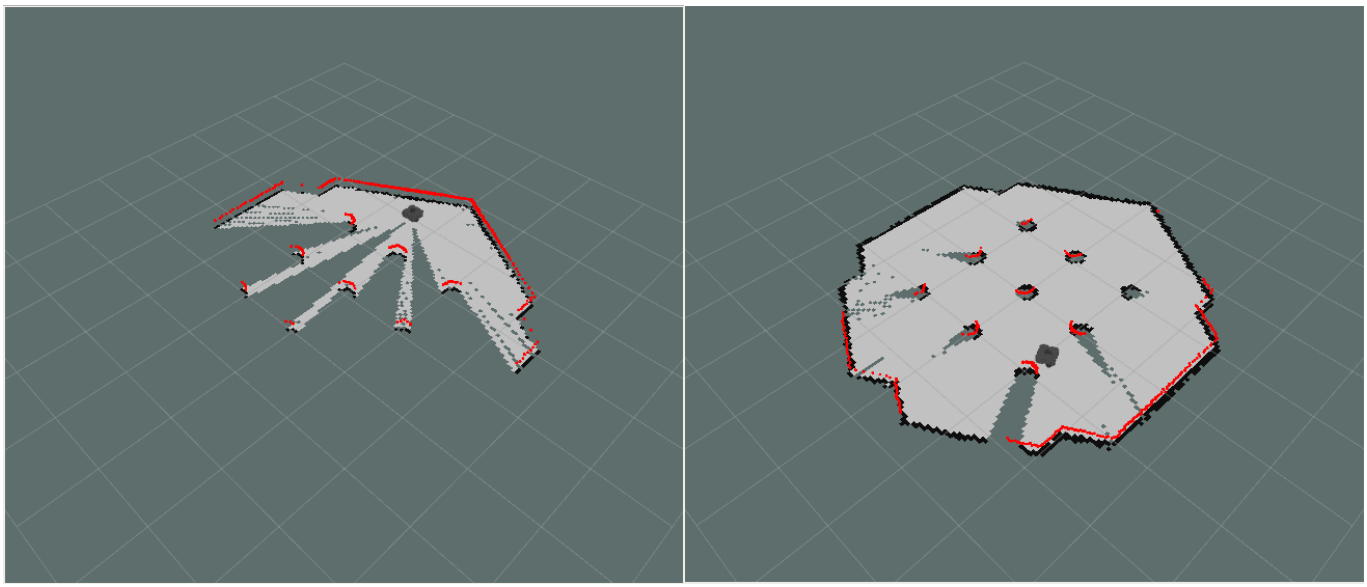
Prima di lanciare il nodo `slam_gmapping` è necessario impostare le dimensioni della mappa che si vuole costruire, in modo da poter dimensionare opportunamente la matrice che la dovrà alloggiare. Ciò viene fatto andando a configurare nel parameter server i seguenti parametri:

```
$ rosparam set /slam_gmapping/xmax 10
$ rosparam set /slam_gmapping/xmin -10
$ rosparam set /slam_gmapping/ymax 10
$ rosparam set /slam_gmapping/ymin -10
```

Giunti a tale punto è possibile eseguire

```
$ rosrun gmapping slam_gmapping
```

Su Rviz è possibile provare ad aggiungere l'oggetto *Map* e settare il topic `/map`, per visualizzare la mappa via via che la si costruisce. Assicurarsi che il *fixed frame* di Rviz sia impostato sul frame `map` per la corretta visualizzazione del movimento.
È possibile provare a muovere il Turtlebot all'interno della mappa utilizzando il nodo di teleoperazione, mentre si osserva il processo di creazione della mappa in real-time su Rviz.



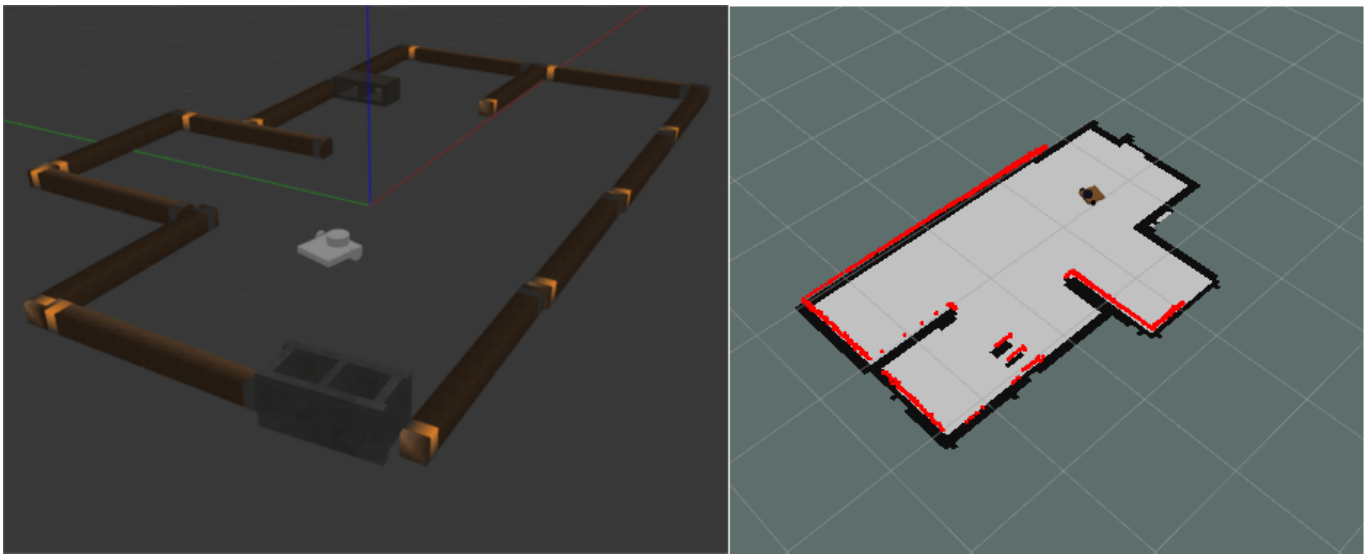
Quando si è raggiunto un livello di costruzione della mappa considerato accettabile (tutti gli spazi sono stati visitati e non vi sono più celle ignote all'nterno dello spazio desiderato), è possibile usare il nodo `map_saver` dell'omonimo package per salvare sul disco sia la mappa che il suo file di configurazione. In un nuovo terminale eseguire:

```
$ rosrun map_server map_saver
```

e, a operazione terminata, verificare la presenza dei file nella cartella `/home` dell'utente. Il nodo `map_saver` genera un'immagine che contiene la griglia di occupazione e il file `*.yaml` contenente le informazioni aggiuntive per interpretare la stessa.

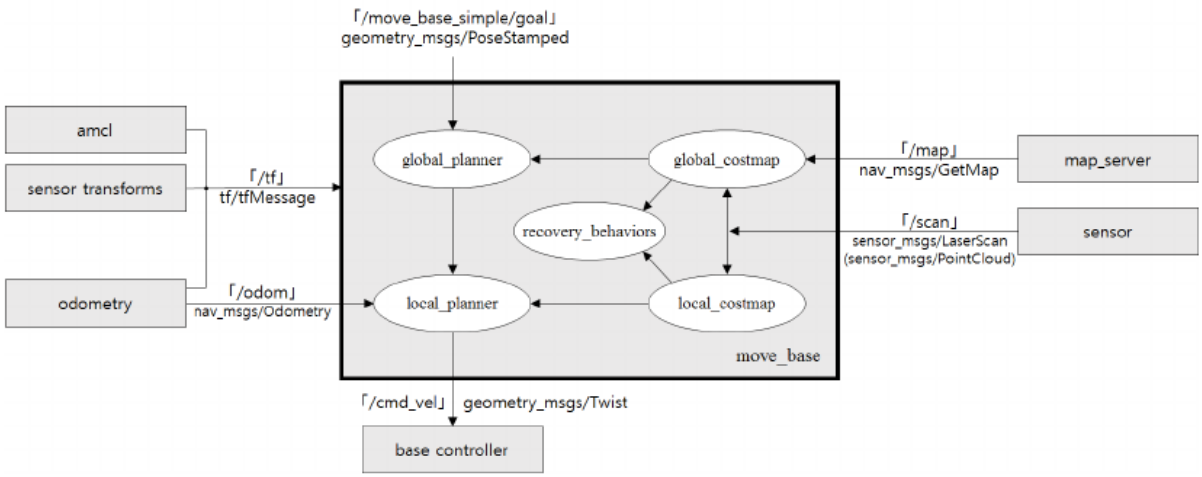
```
image: map.pgm
resolution: 0.050000
origin: [-10.000000, -10.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

È possibile spostare ambedue i file nella cartella `map/` nel package per usarla durante le applicazioni che richiedono la navigazione autonoma nell'ambiente mappato.
È possibile usare il package `gmapping` anche con il robot personalizzato costruito nelle lezioni precedenti, per ricostruire la mappa dell'ambiente personalizzato che si era creato.



Navigazione Autonoma

Lo scopo di un sistema di navigazione autonomo, chiamato anche **navigation stack** (o *nav stack*) nell'ecosistema ROS, è quello di integrare le informazioni provenienti dai vari sensori, dalla mappa dell'ambiente e dagli altri sistemi di localizzazione (ad esempio l'odometria), per pianificare un percorso che porti il robot dalla posizione iniziale ad un traguardo prefissato; inoltre è responsabile della gestione dei segnali di riferimento da passare al robot, in modo che questo possa seguire il percorso pianificato al meglio delle proprie abilità, il tutto evitando anche eventuali ostacoli imprevisti presenti lungo il tragitto.
La figura seguente mostra le principali relazioni che intercorrono tra le entità cardine del *nav stack* di ROS.



Ad alto livello, il processo di lavoro del *nav stack* può essere riassunto dalle seguenti fasi:

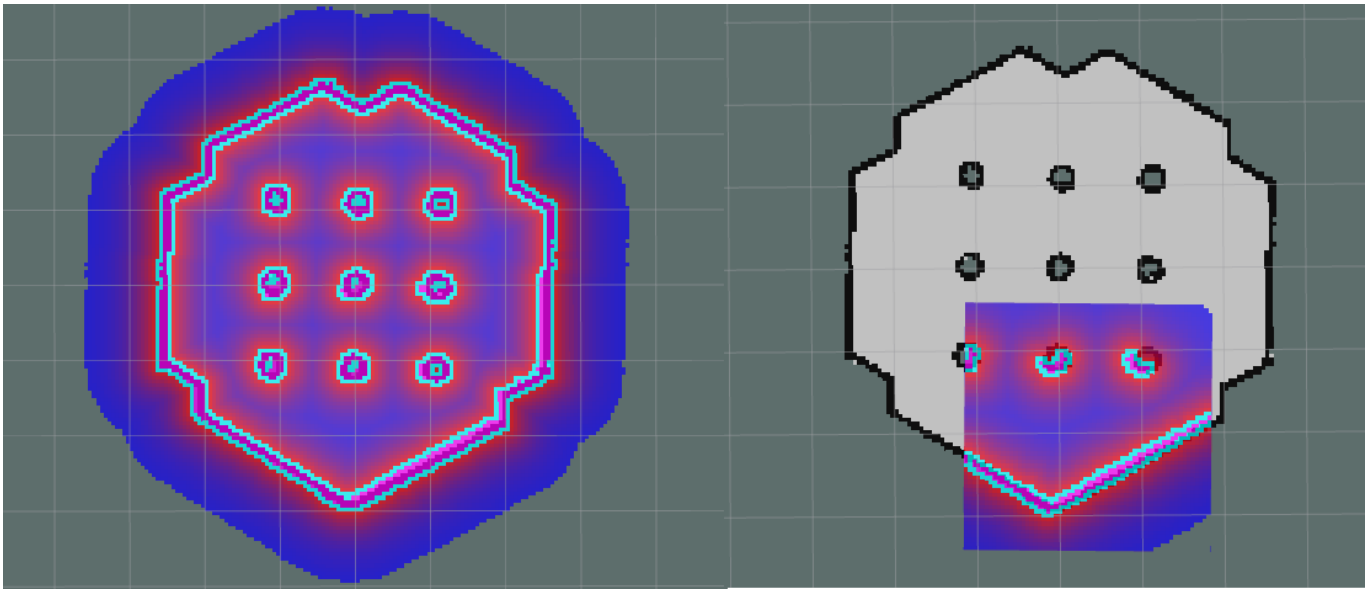
1. Un obiettivo sulla mappa viene inviato al *nav stack*. Questo è fatto attraverso un'azione ROS che invia un goal del tipo `MoveBaseGoal`, il quale specifica la posa obiettivo (posizione + orientamento) in termini di coordinate rispetto ad un sistema di riferimento preimpostato (in genere quello solidale alla mappa).
2. Il *nav stack* per mezzo del **global planner** utilizza un algoritmo di *path-planning* che pianifica, in base alle informazioni della mappa, il miglior percorso per raggiungere il goal a partire dalla posa corrente.
3. Il tracciato viene inviato al **local planner**, il quale cerca di guidare il robot lungo il percorso. È compito del *local planner* utilizzare le informazioni provenienti dai sensori al fine di evitare eventuali ostacoli non precedentemente mappati che potrebbero trovarsi lungo la strada del veicolo. Qualora il *local planner* non fosse in grado di superare un ostacolo e dovesse rimanere bloccato, potrà essere inviata una nuova richiesta di ricalcolo del percorso al *global planner*.
4. Quando il robot si trova in un certo intorno di tolleranza dal goal, l'azione termina e il task di movimento si ritiene terminato con successo.

Esempi di algoritmi di *path planning* sono i seguenti:

- Algoritmo dei cammini minimi di Dijkstra
- Algoritmo di ricerca dei cammini A*
- Metodo dei campi di potenziale artificiale

Gli algoritmi di pianificazione possono essere basati ad esempio sulle mappe di occupazione o su grafi, o su entrambi. I metodi basati su *occupancy grid* dividono l'area in celle (come i pixel di un'immagine) e assegnano ad ognuna di esse un peso che può essere binario (libera o occupata) o intero (grado di occupazione: prossimità ad un ostacolo). Dopo tale fase preliminare, una delle celle viene marcata come traguardo mentre un'altra come stato iniziale (posizione attuale del robot). Gli algoritmi di pianificazione hanno come scopo quello di individuare un numero di celle libere contigue che conducano il robot dalla sua posizione attuale al goal finale. La risoluzione di tale problema può essere ottenuta utilizzando algoritmi di ricerca basati su grafi che modellano la mappa di occupazione precedentemente costruita: due celle contigue sono connesse per mezzo di un arco pesato che rappresenta la difficoltà di transizione dall'una all'altra
Il nodo `move_base` è l'entità principale del sistema di navigazione di ROS, questo ha l'onere di costruire una *cost map*: una mappa dei costi in cui a ogni cella viene associato un peso che rappresenta la distanza di

questa dagli ostacoli, in cui un valore del peso maggiore corrisponde alla maggior vicinanza dall'ostacolo. Il nodo `move_base` utilizza due *cost map*: una locale, per stabilire i comandi di riferimento da dare al robot, e una globale per le traiettorie a lungo raggio. Entrame le *cost map* sono disponibili su topic: `/move_base/global_costmap/costmap`, `/move_base/local_costmap/costmap`, ed ambedue sono messaggi di tipo `nav_msgs/OccupancyGrid`.



I parametri utilizzati per costruire le *cost map* sono conservati all'interno di appositi file `*.yaml`. Un file contiene i parametri comuni sia per la mappa locale che per quella globale, altri due sono creati per i parametri indipendenti delle due mappe.
Prendendo in esame il TurtleBot 3 Waffle Pi, i parametri comuni sono specificati nel file `costmap_common_params_waffle_pi.yaml`:

```
obstacle_range: 3.0
raytrace_range: 3.5

footprint: [[-0.205, -0.155], [-0.205, 0.155], [0.077, 0.155], [0.077,
-0.155]]

inflation_radius: 1.0
cost_scaling_factor: 3.0

map_type: costmap
observation_sources: scan
scan: {sensor_frame: base_scan, data_type: LaserScan, topic: scan, marking:
true, clearing: true}
```

Una spiegazione esaustiva di tutti i parametri presenti può essere trovata nella documentazione ufficiale presente sul sito http://wiki.ros.org/costmap_2d, di seguito se ne riporta una breve presentazione dei principali:

```
obstacle_range: 3.0
```

rappresenta la massima distanza in metri a cui inserire un ostacolo all'interno della mappa, gli ostacoli al difuori di tale range non vengono presi in considerazione nel processo di pianificazione.

```
raytrace_range: 3.5
```

Questo parameto definisce la distanza limite dopo la quale si può considerare lo spazio totalmente libero.

```
footprint: [[-0.205, -0.155], [-0.205, 0.155], [0.077, 0.155], [0.077,
-0.155]]
```

Definisce la forma della proiezione del robot sul piano *xy*: questo viene modellato dall'insieme dei vertici che rappresentano il poligono che lo racchiude; tale contorno è usato per calcolare eventuali collisioni con gli oggetti presenti nella mappa.

```
inflation_radius: 1.0
cost_scaling_factor: 3.0
```

Sono parametri utilizzati per la propagazione del costo a partire da una cella occupata verso le altre. Il costo decresce in maniera omogenea con la distanza.

```
map_type: costmap
```

Specifica che la mappa utilizzata è in due dimensioni.

```
observation_sources: scan
scan: {sensor_frame: base_scan, data_type: LaserScan, topic: scan, marking:
true, clearing: true}
```

Definisce il tipo di sensori utilizzati per acquisire le informazioni dall'ambiente circostante e le loro proprietà:

- *sensor_frame* - sistema di riferimento legato al sensore;
- *data_type* - tipo di messagio pubblicato dal sensore;
- *topic* - nome del topic sul quale vengono pubblicati i dati;
- *marking* - settato a `true` se il sensore può essere utilizzato per rilevare la presenza di ostacoli;
- *clearing* - settato a `true` se il sensore può essere utilizzato per rilevare spazio libero difronte a lui.

Per costruire la *global costmap* sono utilizzati i seguenti parametri contenuti nel file `global_costmap_params.yaml`:

```
global_costmap:
  global_frame: map
  robot_base_frame: base_footprint

  update_frequency: 10.0
  publish_frequency: 10.0
  transform_tolerance: 0.5

  static_map: true
```

Analizzando le singole linee:

```
global_frame: map
robot_base_frame: base_footprint
```

i quali definiscono i sistemi di riferimento legati alla mappa e al robot.

```
update_frequency: 10.0
publish_frequency: 10.0
```

Questi parametri definiscono ogni quanto tempo la *cost map* deve essere ricalcolata e a ogni quanto viene pubblicata sul relativo topic.

```
transform_tolerance: 0.5
```

Definisce la massima latenza tollerabile (in secondi) nella pubblicazione delle trasformazioni, se il *tf tree* non è aggiornato entro il tempo massimo il *navigation stack* arresta il robot.

```
static_map: true
```

Specifica che la mappa non varia nel tempo.
Analogamente i parametri necessari alla costruzione della *local costmap* sono definiti nel file `local_costmap_params.yaml`:

```
local_costmap:
  global_frame: odom
  robot_base_frame: base_footprint
```

```
  update_frequency: 10.0
  publish_frequency: 10.0
  transform_tolerance: 0.5

  static_map: false
  rolling_window: true
  width: 3
  height: 3
  resolution: 0.05
```

I quali hanno lo stesso significato dei precedenti, ma valori diversi. Sono inoltre specificati dei parametri aggiuntivi :

```
rolling_window: true
```

per indicare che la *local costmap* utilizza una finestra a scorrimento: il robot rimane sempre al centro di tale finestra e vengono scartate tutte le informazioni inerenti a punti al di fuori della mappa, le cui dimensioni sono specificate dai parametri seguenti:

```
width: 3
height: 3
resolution: 0.05
```

Nei file `move_base_params.yaml` e `dwa_local_planner_params_waffle_pi.yaml` pososno essere trovati i parametri utilizzati dagli algoritm di pianificazione della traiettoria e di generazione dei riferimenti di velocità, di seguito si riportano alcuni

```
# The velocity when robot is moving in a straight line
  max_vel_trans:  0.26
  min_vel_trans:  0.13

  max_vel_theta: 1.82
  min_vel_theta: 0.9

  acc_lim_x: 2.5
  acc_lim_y: 0.0
  acc_lim_theta: 3.2

# Goal Tolerance Parametes
  xy_goal_tolerance: 0.05
  yaw_goal_tolerance: 0.17
```

Il nodo `move_base` può essere eseguito insieme ai suoi parametri attraverso il file `*.launch` presente nel package `turtlebot3_navigation`. Modificare la posizione iniziale nel file `amcl.launch`, impostando le

coordinate corrette utilizzate per posizionare il modello del robot su Gazebo

```
<arg name="initial_pose_x" default="-2.0"/>
<arg name="initial_pose_y" default="-0.5"/>
```

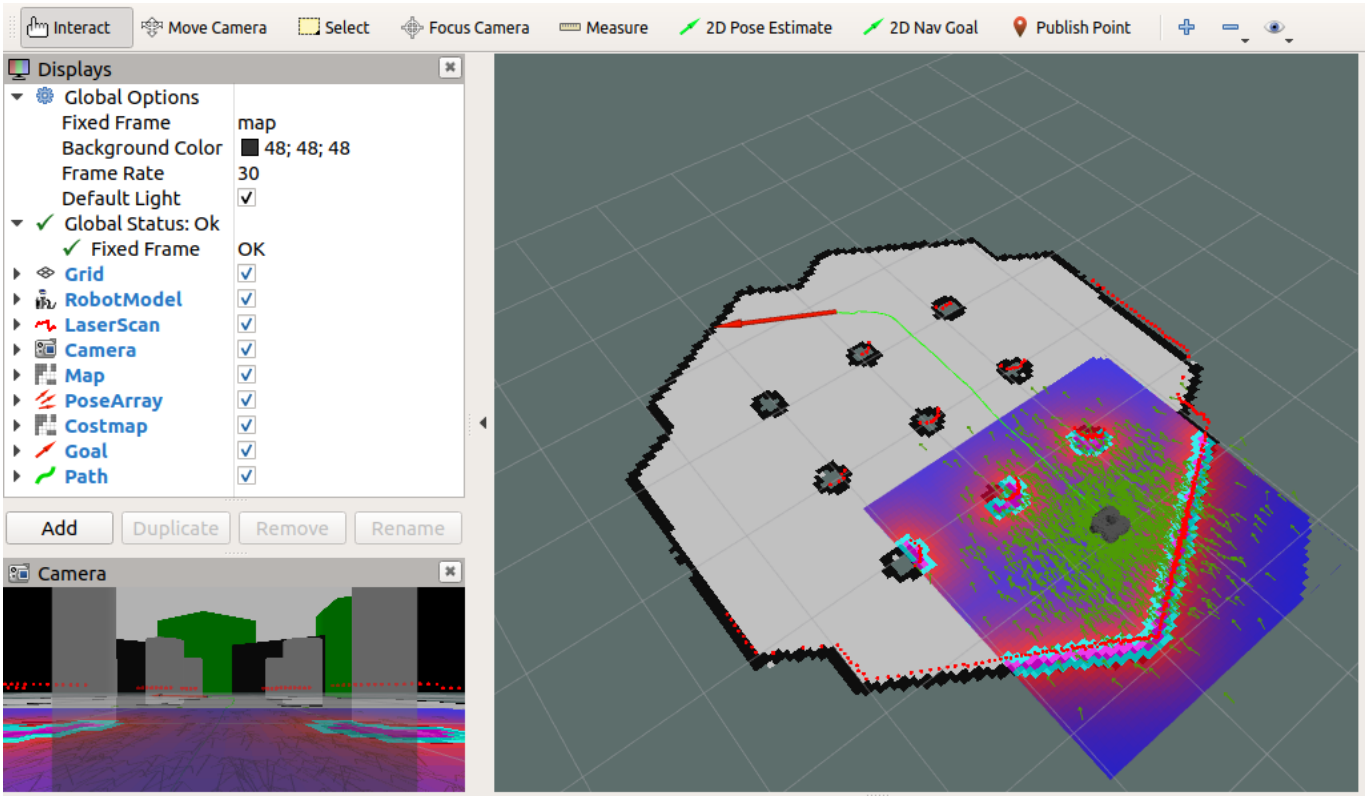
dopodiché eseguire l'ambiente di simulazione digitando

```
$ roslaunch tutlebot3_navigation turtlebot3_localization.launch
```

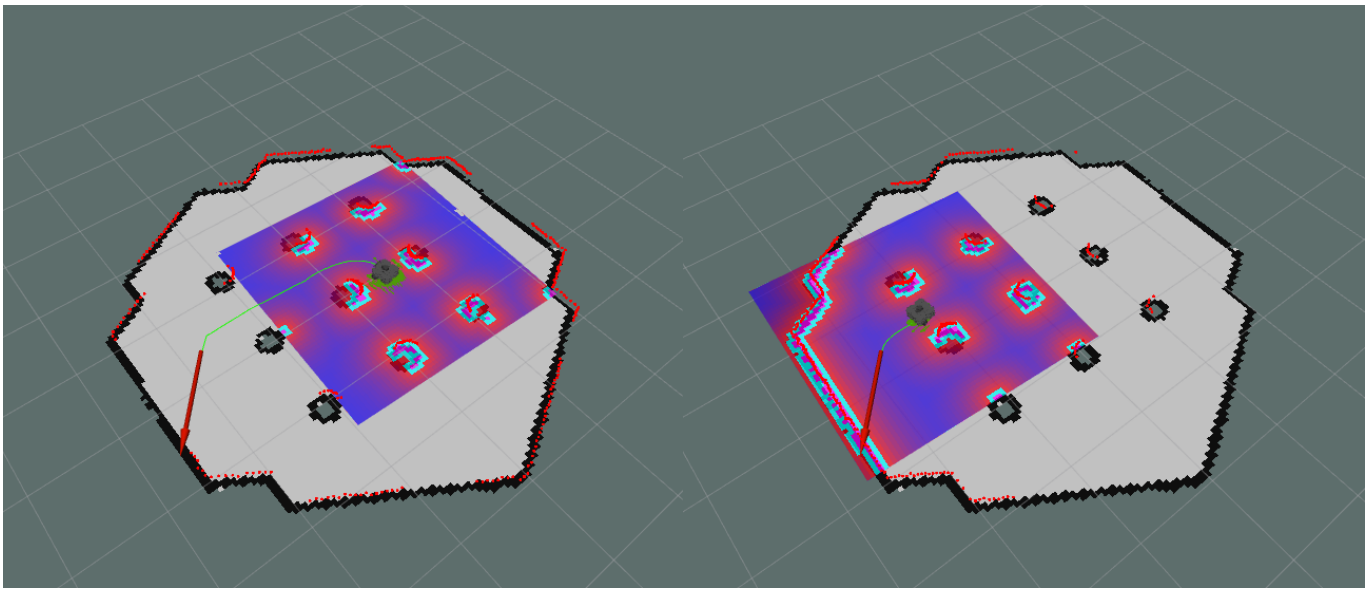
In un nuovo terminale eseguire il nodo `move_base`

```
$ roslaunch tutlebot3_navigation move_base.launch
```

Su Rviz utilizzare il bottone "Add" per aggiungere nella lista laterale un oggetto di tipo *Map* per visualizzare la *local costmap*, un oggetto *Pose* per visualizzare il pnto di arrivo desiderato, pubblicato sul topic `/move_base_simple/goal`, e un oggetto _Path_per mostrare le traiettoria desiderata calcolata dal *global planner*.



Utilizzando il pulsante "2D Nav Goal" in Rviz, è possibile pubblicare un goal per il robot e visualizzare il percorso calcolato. Durante la simulazione è possibile pubblicare un nuovo goal in qualsiasi momento cancellando il target precedente. È interessante notare come il processo di localizzazione migliori via via che il robot si muove e acquisisce dati dall'ambiente circostante confrontandoli con la mappa.



> ⚠️ **CONFLITTI DI COMANDI**
>
> Qualora si volesse migliorare la bontà della stima della posizione del robot prima che questo compia dei movimenti in autonomia, è possibile teleguidare il robot all'interno del ambiente affinché l'algoritmo di localizzazione converga alla posizione attuale del robot. Prima di commutare in modalità autonoma, assicurarsi però che il nodo `turtlebot3_teleop_key` venga arrestato (shutdown) onde evitare potenziali conflitti dei comandi di velocità pubblicati sul topic `/cmd_vel`.