# 5) UNDERSTAND THE Nav2 STACK

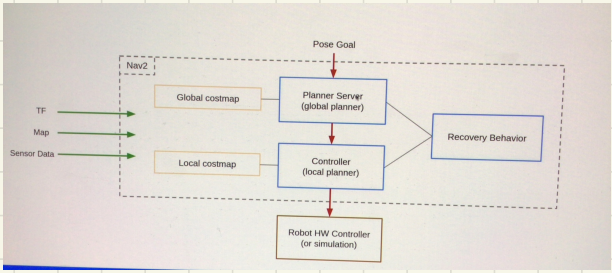
Once seen the two steps of Navigation stack { SLAM for map generation / How to navigate

↓

Let's see

- what is going on in Nav2 stack
- what are main components
- what is the architecture

← understand it by running it and focusing on one component per time




↙ at the end, a better grasp of the architecture will be given

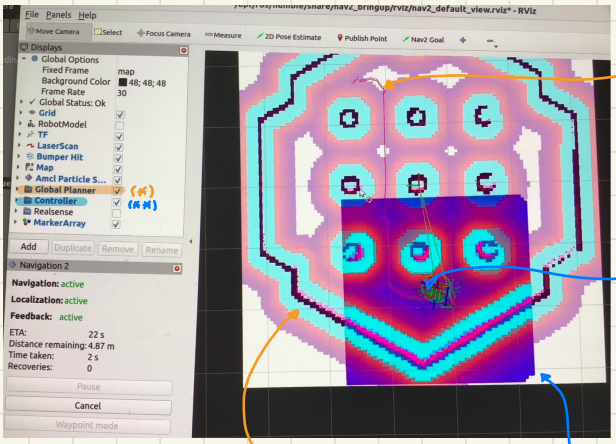
## GLOBAL / LOCAL PLANNER AND COSTMAPS

by examining the Nav2 with turtlebot as done during section (4)

↓

What happens when we give a Nav2 goal?

How Global and Local Planner works and interacts with each other?




(pink one) path to take to reach destination (global plan)

(blue one) local plan

the GLOBAL PLANNER (✗)

LOCAL PLANNER (controller) (✗✗)

When we send Nav2 Goal, this is sent to Global Planner: it will compute the path based on the entire map, to reach destination as fast as possible with best possible path!

(the Geobal planner)

To compute this path, it will use a GLOBAL COSTMAP (the colored one above the black/white pixel map)

↓

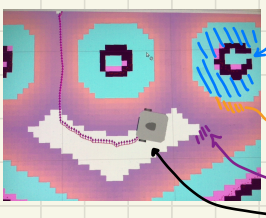
map in which each pixel have a cost,

- obstacles have highest cost

with some margin around obstacles (to avoid going close to it)

↓

around obstacles we have safety margin to avoid collisions




in the costmap we have different color shade to represent different cost

⇒ the closer we are to an obstacle and the higher is the cost

To compute the path, GLOBAL PLANNER will add all pixels it has to go through and find the minimal cost path.
Try to go as much as possible on free space and the one far from obstacle (this ensure manouvers far from obstacle)

This GLOBAL planner get updated at low frequency (~1 ÷ 5 Hz)
BUT this Geobal planner is not what really control Robot motion...
↳ the path is sent to Local Planner, which CONTROL motion.

↙

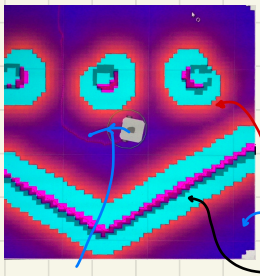
Local Planner has its own Local costmap.
After receiving geobal path, the controller try to follow it, and it is characterized by high frequency update (~ 20 ÷ 100 Hz)

- Analogy: driving a car with GPS: the GPS compute plan and update with low frequency (Geobal plan).
  Then, you will control the car, deading velocity, steering etc with high frequency, following GPS as much as possible.
  You also take into account possible correction of that global plan

In ROS2 Nav2 stack: { Global planner ~ 1Hz (GPS)
Local planner ~ 20Hz (car driver)

also, Global planner has smaller update frequency because it takes more computation power for this path
(IF running at high freq, it takes too much resources)

the LOCAL COST MAP



↓ works with same principle of the GLOBAL one, but only on the ROBOT surroundings

red higher cost

blue lower cost

light blue are obstacle safe zone to avoid

⇒ Local planner will follow the path, by avoiding Higher cost pixels

Local plan currently following ⇒ Global path updated a bit, and Local planner update with high frequency and take best short term decision to control robot motion

↓

the local planner will control by sending a velocity command to the robot

# PARAMETERS

Focus on a few of Nav2 parameters (some of them allow to tune costmap)

↓

We will see just some, because there are a LOT to define Nav behavior

Focus on the most important / usefull

(Once understood this ones... it will be easy to adapt the knowledge to others)

To find the parameters:

on a new terminal `rqt`

↓

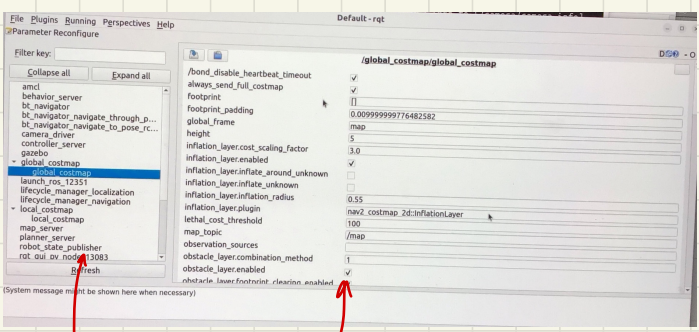
GUI with lots of plugings: > Plugins > Configuration > Dynamic Reconfigure

It will open a list on the left, we can see • global_costmap

↓

global_costmap

for example:




↙ all live parameters for global costmap

we have many parameters section

and for each section a lot of parameters...

Let's see the most important

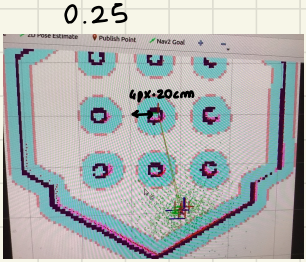
• publish_frequency : 1.0 [Hz] (frequency of update of global costmap)

• inflation_layer.inflation_radius: [m]

0.55

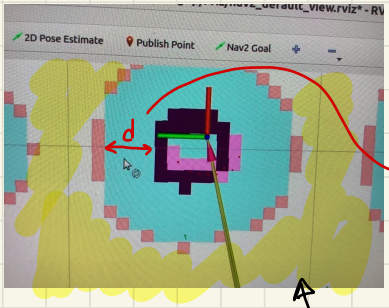
→

0.25




the map has more white space... this "inflation radius" represent the radius from obstacle where the pixel still have cost

↳




inflation radius: radius from the obstacles where we still have cost in the pixel

5 cm/pixel discretization of map

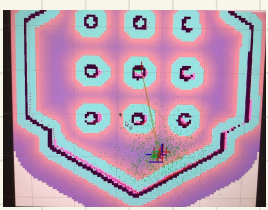
d ≃ 4 pixels : 4.5 = 20 cm of space considered as obstacle around (light blue)

+ 0.25 cm

inflation radius → pixel cost up to 5 pixels radius from obstacles

The same around the walls...
we have pixel cost up to 5 pixel far from it.
While all the other space is considered with 0 cost free space

0.8

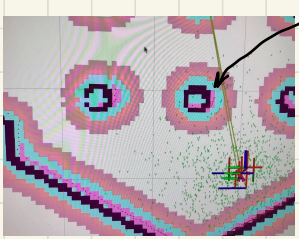


everything has higher cost,
up to 0.8 ~ 80 cm = 16 pixels radius of cost

- robot_radius : 0.22 [m]

used to compute where the Robot can go

IF I update to 0.1



obstacles light blue area get updated to smaller areas

⇒ this means that robot can traverse closer to the obstacle

It is good norm to use the correct "radius" of the Robot model used for navigation (22 cm is waffle turtlebot 3 radius)

- resolution 0.05 [m]

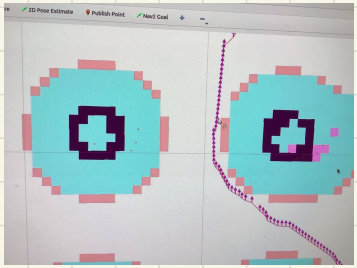
specify map pixel resolution (~5 cm)

- In `local_costmap` we have same parameters as global_costmap

for example • `inflation_radius`: 1.0 [m]

IF I update to 0.5 or smaller it get updated as for geobal one

IF this value is smaller in both, the robot will navigate closer to the obstacles,




this will be a faster root and going through free space with low cost

the choice of inflation_radius require a good trade off

- the lower: easy path computation but higher probability to hit obstacles in real life narrow environments
- the higher: less hitting probability but more complex path finding

- In `controller_server` ("local planner")

you will have usefull parameters like

- `max_vel` (change acceleration) [m/s]
- `controller_frequency`: frequency at which local planner (controller) is sending command velocity to the robot [Hz]

usually ~ 10 ÷ 100 Hz

- `goal_tolerance` when destination reached, how far it can be from destination (some tolerance)

...

⇒ from this description, experiment with the others!

# RECOVERY BEHAVIOR

Briefly recap: when sending a Nav2 Goal

↓

IF everything is IDEAL ← that's all we need!

- Global planner compute (path) based on Global costmap
  ↓
  path to local planner (controller), responsible to make robot move, and follow path in Local environment

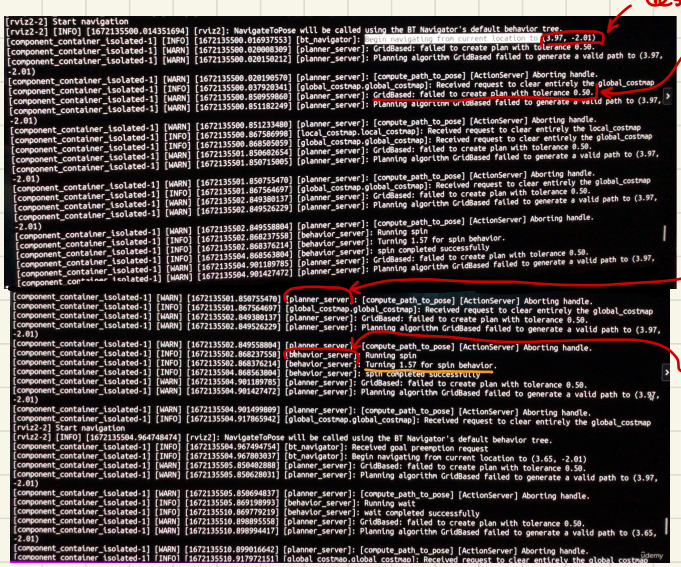
↳ BUT, in a real environment, the ROBOT sometimes doesn't manage to reach the destination, because:
- Non valid Goal
- dynamic obstacle unavoidable
- path too narrow
- algorithm fails to follow path

In those cases, Navigation stack will start the RECOVERY BEHAVIOR

↓

this is a behavior pre-defined for the ROBOT, that will try to fix the current issue, so that Robot can continue to move and reach destination.

When unreachable destination is sent:




...

until we get a "Goal failed"

What happens is that :

GLOBAL PLANNER : [planner_server] try to compute path and send to local planner

IF there is an ISSUE and global planner can't generate the path OR local planner can't follow that path

↓

recovery server will be called, and start a recovery behaviour

THIS RECOVERY BEHAVIOR can be: turn around, go backward a bit etc..

for ex: when driving car, you try to turn a bit and go backward IF there is NO visibility, to try to clear the path

SOMETIMES it succed to replan and recover the path

The logs in terminal show us how path following is going...

"Passing new path to controller" try to do something else until "Running backup"

↓

try to find new path... When issue with global planner / local planner behavior_server is called → start a series of recovery behaviors to try to clear the map and find valid path

IF it doesn't succeed, at a certain point the Goal will be Aborted

# TFs AND IMPORTANT FRAMES

Another important "component" to understand before the details of Nav2 stack Architecture are (TF) (NOT Navigation specific. Used for any Robot developed in ROS)

⇓

Just quick introduction of TF to understand what is it and how are used in navigation

"TransForm" visible in Rviz, in the TF section...

⇓

Problem to solve: In a Robotic application we have many frames representing:
- different robot parts
- origin of the map
- environment frames

Challenge:
To keep track of each frame relative to other frames

[ex: • where is the Robot relative to map origin?
• where is left-wheel pose relative to robot base?]

The default math solution is based on Rotation/translation computation in 3D. This has to be done for each frame in the Robot and environment
LOT OF COMPUTATION!

⇓

ROS 2 approach: use tf2 package!
which keeps track of each frame over time, as structured tree containing all frames.
To get this TF tree we can subscribe to /tf or see it in Rviz

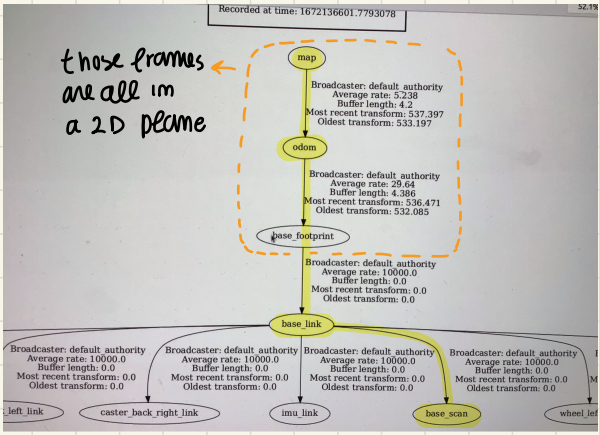
In Rviz we see the frames gets updated at runtime moving one with respect to the other.

In /tf topics we have a set of transforms published mapping parent to child relationship. As Rotation + Translation

Also, to visualize the TF tree, run

`tf2_tools view_frames`

that will export a pdf with a representation of it.




**base_footprint**: projection of the robot on the ground (being the base a bit elevated, the base is projected to z axis)

**base_link**: main robot chassis where all wheels and sensors are attached

## Required TFs for Nav2

⇓ when you have a Robot running with Nav stack you need:

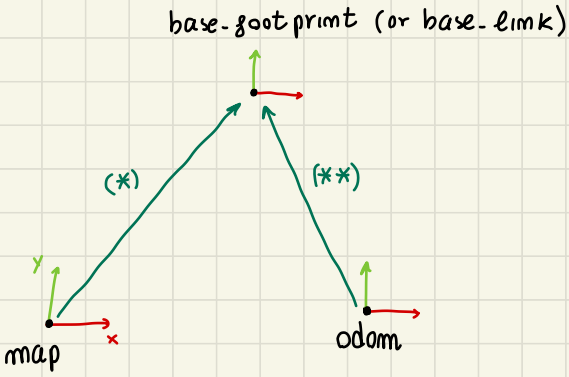
- map → odom
- odom → base_link (NOT direct, but through base_footprint)
- base_link → base_scan (where the lidar information are)

↑ NOT an issue, the odom to base_link can be still computed

usually this is fixed, being the LIDAR fixed on the main structure of the Robot

Let's focus on **map** and **odom** relation with base_link

↳




(*) • map → base_footprint : used to compute exact location of the Robot with SLAM using LaserScan from Lidar (or GPS)

ISSUE : the location can be precise over long time, but in short term is unstable (jump from one point to another, uncertain)
→ correct in long term but noisy in short term

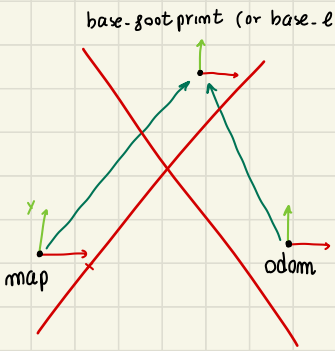
(**) • odom → base_footprint :
"odometry" = estimation of robot position using robot internal measurements (for ex. recorded wheels speed)

The location with odometry will be smooth in short term, but will drift over time (error cumulate)
ex. when estimating the distance you walk based on step size... you accumulate error over time!

Nav2 stack: it take the best of both, combining map and odom to have precise location of the Robot over time (map) with smooth/linear location NOT jumping around (odom)

⇓

due to how TF are designed in ROS: any frame can have at most one parent! (and many children)

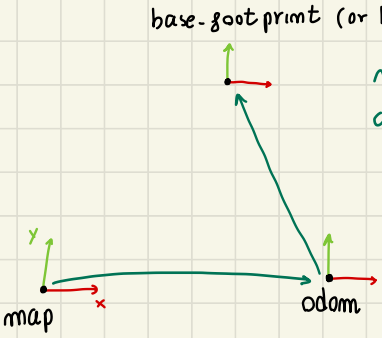



← it has two parent ?

NOT possible in this way

⇓

⇒ It has been decided to structure TF tree as:




map as odom parent and odom as parent of base_footprint

even if this map → odom relationship seems to NOT have any sense

↪ this is a workaround to make TF works with one parent per link

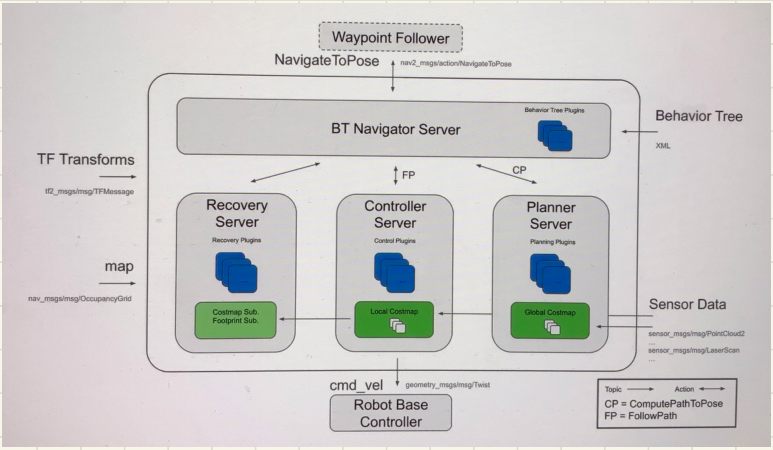
the important aspect is:
- map for long term precise location
- odom for smooth location in short term

↳ it may happen that odom "jump around" relative to map, but this will be compensated for base_footprint (odom to base_footprint adapt automatically to compensate map to odom noise)

robot will not "jump around" ←

# Nav 2 ARCHITECTURE

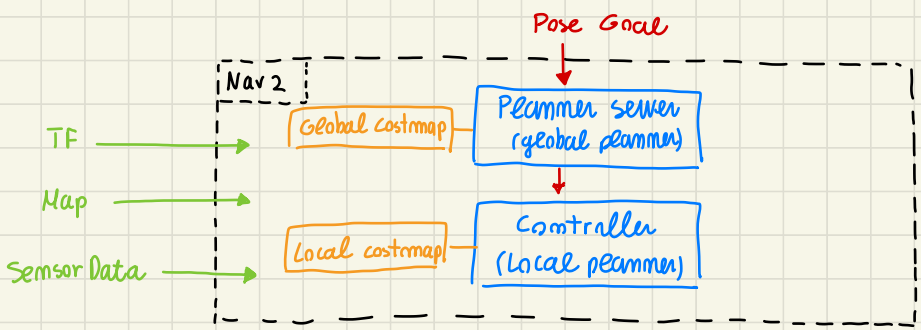
We can now put all together the main components of the stack in the global Architecture.




↳ Offical Nav2 stack architecture in documentation

⇓

Let's simplify it a bit, building our own representation step by step




to make Nav2 stack works, we need some inputs

TF, Map (generated by Mapping before Navigation, the Map is given to the stack)

Sensor Data (such as Laser scan or camera, 3D scanner etc)

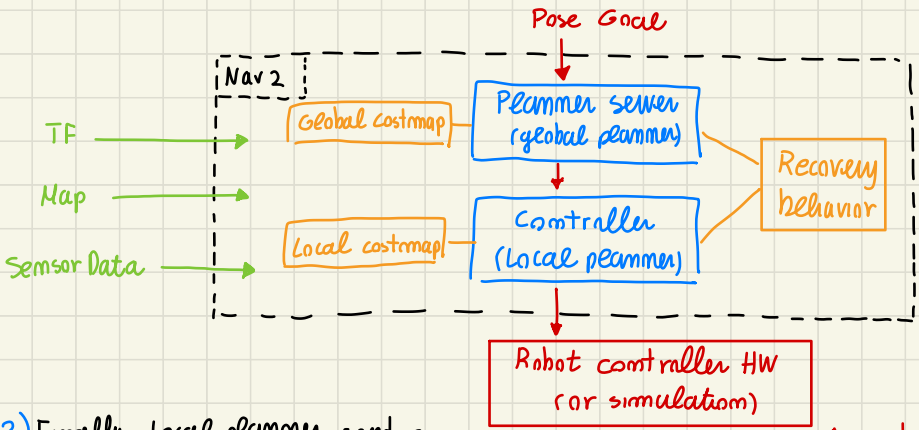
When you send a Nav Goal in terms of Pose (position + orientation)

1) global planner (called "Planner server") is responsible to find a valid path, using both Map and Global costmap

↓

once a path is found

2) Local planner (controller) try to make sure robot follow the path and reach destination. Using Local costmap AND the data from all sensors



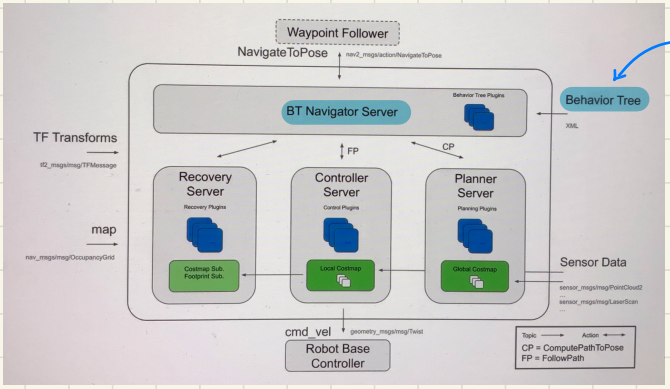

3) Finally, Local planner send a command to Robot Controller itself

↑ indipendent from Nav 2 stack!

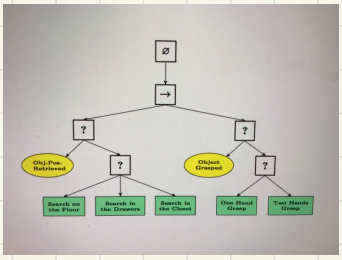
" custom controller of the Robot that take velocity command and translate it into a command sent to motor (so that Robot move)

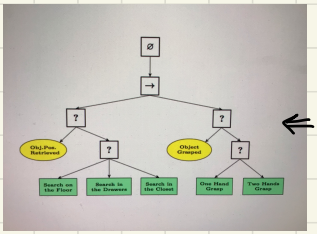
4) To complete the architecture, When path is NOT followed properly... a Recovery behavior is integrated

From this simplified Nav 2 stack Architecture, IF we look back to the offical one: It is almost the same...




this is the big difference from our simple version "Behavior Tree" which is NOT Nav 2 specific, an example of Behavior Tree:






BT example:

You want to GRASP an object

1) search for the object around until you found
2) try to grasp it (one/two hands)

In Navigation: When the stack receive Pose Goal

1) Use Planner Server to find valid path
2) Use controller to make robot follow path
3) IF issue occurs: recovery server is called, to try Recovery Behavior