Chapter 1: Topics, Frames and TF

1.1 Understanding Topics in ROS 2

ROS 2 (Robot Operating System 2) enables communication between different software components called **nodes**. These nodes often need to exchange information — like sensor data, commands, or localization results. This is done using **topics**.

A **topic** is a named communication channel used for streaming data from publishers to subscribers.

Key Concepts:

- **Publisher**: A node that sends messages.
- **Subscriber**: A node that receives messages.
- **Message**: A data structure (like a packet) containing relevant data, defined by ROS message types.

1.2 Coordinate Frames and Their Importance

To understand a robot's position in the world or to relate one part of the robot to another (like a sensor to a wheel), we need a consistent way to describe positions and orientations. This is done using **coordinate frames**.

A **coordinate frame** is a mathematical 3D system with:

- An **origin**: the point (0, 0, 0)
- **Axes**: three mutually perpendicular vectors:
  - X-axis (usually forward)
  - Y-axis (usually left)
  - Z-axis (usually upward)

Each part of a robot — its base, wheels, sensors, arms — can have its own frame. These frames are often linked through transformations.

Coordinate Frame Example

Imagine a robot with a camera mounted on top and a LiDAR at the front. We can define:

- `base_link`: Frame at the center of the robot.
- `camera_link`: Frame where the camera is mounted.
- `lidar_link`: Frame where the LiDAR is mounted.

Each of these frames is positioned and oriented relative to one another.

Transform Between Frames

To convert a point from one frame to another, you use a **transformation**. This includes:

- **Translation**: shifting the origin to a new location.
- **Rotation**: rotating the coordinate axes.

Suppose a point is defined in `lidar_link`, and we want to express it in `base_link`. We need a transformation $T_{base \leftarrow lidar}$.

Mathematically:

A point p_{lidar} in lidar frame becomes:

$$p_{base} = R \, . \; p_{lidar} + \mathrm{t}$$

Where:

- : Rotation matrix (3×3)
- : Translation vector $[x, y, z]^T$

In homogeneous coordinates:

$$\begin{bmatrix} \mathbf{p}_{base} \\ 1 \end{bmatrix} = \begin{bmatrix} R & \mathbf{t} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p}_{lidar} \\ 1 \end{bmatrix}$$

This makes it easy to chain transformations through multiple frames.

Static vs Dynamic Frames

- **Static Frame**: The relative position never changes (e.g., sensor fixed to the robot).
- **Dynamic Frame**: The relative position changes over time (e.g., a moving joint).

NOTE: Go to the below link to learn more about tf2.
https://docs.ros.org/en/humble/Tutorials/Intermediate/Tf2/Tf2-Main.html#tf2

Common Coordinate Frames:

Frame	Description
base_link	Center of the robot
odom	Origin of robot's motion estimate
map	Fixed global reference frame
camera_link	Frame attached to camera sensor

NOTE: In the map co-ordinates frame, the axis representing True North is the y-axis and the x-axis represents east.
NOTE: The axis facing the front of the frames like base_link or any other sensor frame is the x-axis.

1.3 The Role of TF and TF2 in Frame Transformations

TF is a transform library in ROS. It keeps track of how frames are positioned relative to each other over time.

Why We Need TF:

- You have a LiDAR sensor mounted on your robot.
- The robot moves over time.
- You want to convert LiDAR data from lidar_link to map frame.

TF2 helps you do this by:

- Broadcasting frame positions over time.
- Listening to frame data and computing transforms.
- Making it easy to convert data between frames.

Mathematical Transform:

A transform includes **translation** and **rotation**.

- Translation: $\boldsymbol{t} = [x , y , z]^T$
- Rotation (quaternion): *[x, y, z, w]*

To convert a point $\mathbf{p}_A$ from frame A to B:

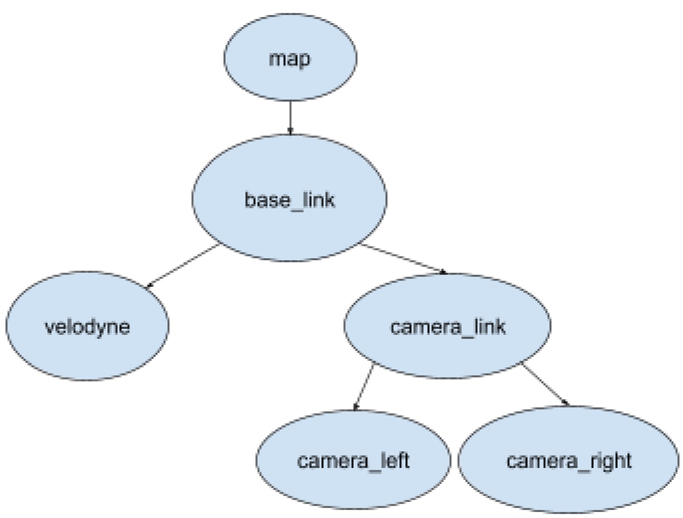
$$\mathbf{p}_B = T_{AB} \cdot \mathbf{p}_A$$

Where $T_{AB} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$

- is a rotation matrix from quaternion.
- is the translation vector.

TF Tree:

TF frames form a tree, with one root (usually `map`):



Chapter 2: Robot State Publisher and Types of TF Broadcasters

2.1 robot_state_publisher

This package reads the robot's URDF and joint states, and publishes the correct transforms between links using TF2. It eliminates the need to manually publish transforms between robot joints.

Workflow:

- You create a URDF or xacro file.
- Your robot node publishes `/joint_states`.
- `robot_state_publisher` publishes all transforms using TF.

This enables you to visualize the entire TF tree in RViz and use those transforms in other systems.

Example Launch:

```
<node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher"/>
```

2.2 Static and Dynamic Transform Publishers

Static Publisher:

- Used for unchanging transforms (e.g. base_link to camera_link).
- CLI:
  ```
  ros2 run tf2_ros static_transform_publisher 0 0 1 0 0 0 1 base_link
  camera_link
  ```

Dynamic Publisher:

- Used when transforms vary over time.
- Implemented in code using `TransformBroadcaster` (Python/C++).

Python Example:

```python
br = TransformBroadcaster(self)
t = TransformStamped()
t.header.stamp = self.get_clock().now().to_msg()
t.header.frame_id = 'odom'
t.child_frame_id = 'base_link'
# Fill in t.transform.translation and t.transform.rotation
br.sendTransform(t)
```

2.3 TF Lookup

When you need to convert data from one frame to another (e.g. sensor to map), use `TransformListener` and `Buffer`.

Python Example:

1. ```python
   self.tf_buffer = Buffer()
   self.tf_listener = TransformListener(self.tf_buffer, self)
   trans = self.tf_buffer.lookup_transform('map', 'base_link',
   rclpy.time.Time())
   ```

This is especially critical in localization, navigation, and perception pipelines.

Chapter 3: Localization and NDT Scan Matching

3.1 Fundamentals of Robot Localization

Localization is the ability of a robot to determine its position and orientation (collectively known as "pose") in a known environment.

In ROS 2, localization is about continuously estimating the robot's pose in the map frame.
   ```

Why is it Needed?

- Navigation: To plan and follow paths.
- Mapping: To build maps accurately.
- Sensor Fusion: To align sensor data.

Common Data Sources:

- LiDAR (produces point clouds)
- Odometry (wheel encoders, IMU)
- Pre-built maps (point clouds or occupancy grids)

The robot uses these sources to infer how far it has moved and where it is now.

3.2 Principles of Scan Matching

Scan matching is a process that aligns two sets of point data (typically from LiDAR):

- One set is the **reference** (e.g., a map).
- The other is the **current** scan.

Goal:

To find the best transformation (rotation and translation) that aligns the current scan with the reference.

How it Works:

1. Find corresponding points between scans.
2. Compute the transformation that minimizes distance between those pairs.

Mathematical Idea:

Minimize the following cost:

$$E(T) = \sum_i \| T(p_i) - q_i \|^2$$

Where:

- (p_i) = points in the current scan
- (q_i) = corresponding points in the map
- (T) = transformation matrix

The lower the cost the better the match.

Popular scan matching algorithms:

- **ICP (Iterative Closest Point)**
- **NDT (Normal Distributions Transform)**

3.3 Normal Distributions Transform (NDT) for Localization

NDT is an advanced scan matching method that turns a point cloud map into a set of probability distributions.

Instead of treating the map as raw points, NDT models each region (voxel) as a 3D Gaussian distribution. Incoming scan points are then matched to these distributions.

How NDT Works:

1. **Voxel Grid**: The map is divided into 3D cubes (voxels).
2. **Gaussian Modeling**:
   For each voxel containing points, compute:
   - Mean ()
   - Covariance ()
3. **Probability Evaluation**:
   For each incoming scan point (x), calculate how likely it belongs to each voxel:
   $$P(x) = \frac{1}{\sqrt{(2\pi)^3|\Sigma|}} exp\left(- \frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu)\right)$$
4. **Optimization**:
   Adjust the transformation (T) to maximize this likelihood for all scan points.

Why NDT is Useful:

- More robust to noise and sparsity.
- Handles partial overlaps better than basic methods like ICP.

- Faster convergence in structured environments.

Chapter 4: Structure of the Package and Flow of the Program

The autonomous localization system is built using a modular ROS 2 package architecture, consisting of three main sub-packages:

4.1 Sub-packages Overview and Node Structure

The localization system is structured into three primary sub-packages. Each one plays a distinct role in converting raw sensor data into a reliable global pose estimate.

---

1. Lidar Driver

- **Function**: Interfaces with the physical LiDAR hardware.
- **Output Topic**: `/lidar_points`
- **Details**: This is a vendor-specific or open-source driver (e.g., Ouster, Velodyne) that converts sensor data into ROS messages.
- **Note**: This is not part of the localization package but is a dependency.

---

2. Robot State Publisher

- **Function**: Publishes the robot's static and dynamic transforms using URDF.
- **Transform Broadcasts**:
  - `/base_link → laser`
  - `/odom → base_link`

- **Details**: It ensures that coordinate transformations are available for TF lookup and accurate sensor alignment.

---

3. NDT Localizer Package

This package contains three major C++ nodes:

a. `map_loader` — File: `map_loader.cpp`

- **Node Name**: `pointcloud_map_loader`
- **Function**: Loads a pre-recorded point cloud map (typically in PCD format) and publishes it as a static reference on the `/map` topic.
- **Key Steps**:
  - Loads a PCD file at startup using the Point Cloud Library (PCL).
  - Publishes the map once as a `sensor_msgs::msg::PointCloud2`.
- **Enhancement Notes**:
  - The file supports loading large maps efficiently.
  - It can be extended to support dynamic reloading if required (e.g., for online map switching).

---

b. `voxel_filter` — File: `voxel_grid_filter.cpp`

- **Node Name**: `voxel_filter`
- **Function**: Applies a voxel grid filter to the raw LiDAR scan to reduce point cloud density while preserving shape.
- **Input Topic**: `/lidar_points`
- **Output Topic**: `/downsampled_pointcloud`

- **Key Parameters**:
  - Leaf size (voxel resolution) for x, y, z dimensions.
- **Working**:
  - Uses PCL's `VoxelGrid` filter.
  - Converts the input ROS point cloud to PCL format, applies filtering, and repackages to ROS message.
- **Enhancement Notes**:
  - Adaptive voxel sizes could be introduced for varying speeds or environments.
  - Downsampling improves scan-matching performance with minimal loss in accuracy.

c. `ndt_localizer` — File: `ndt.cpp`

- **Node Name**: `ndt_scanmatcher`
- **Function**: Performs 3D scan matching using the Normal Distributions Transform (NDT) algorithm.
- **Input Topics**:
  - `/downsampled_pointcloud` (real-time scan)
  - `/map` (static pointcloud map)
  - `/current_pose` (initial guess or predicted pose)
- **Output Topics**:
  - `/localized_pose`: Final estimated pose (geometry_msgs/PoseWithCovarianceStamped)
  - `/scan_matched_pointcloud`: Aligned scan for debugging

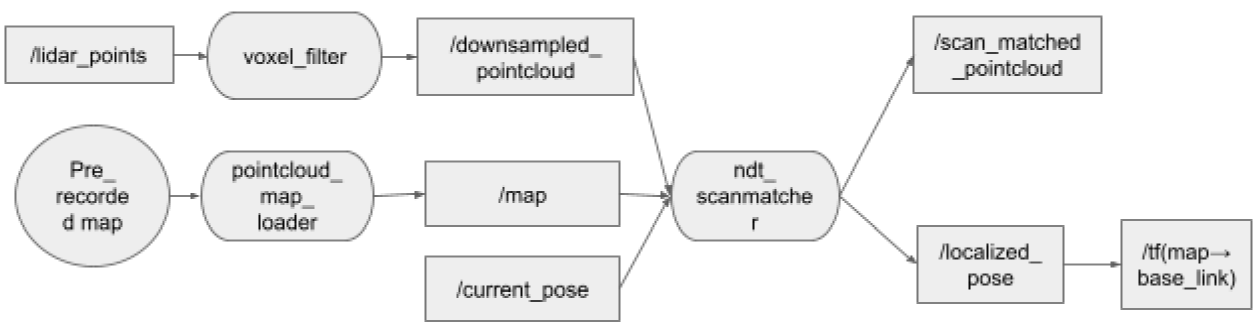
- `/tf (map → base_link)`: Broadcasted transformation

- **Key Features**:
  - Uses `pcl::NormalDistributionsTransform`.
  - Maintains and updates transformation matrices.
  - Supports initial pose injection and loop rate control.

- **Enhancement Notes**:
  - Can be extended with IMU/fusion inputs for better initial guess.
  - Loop closure modules or GPS fallbacks could be integrated.

4.2 Flow of the Program (Updated)

Based on the C++ implementation of each node, the program flow becomes more technically grounded:



1. **Raw Data Capture**:
   - `/lidar_points` is published by the driver.

2. **Preprocessing**:
   - `voxel_filter` node (from `voxel_grid_filter.cpp`) reduces the point cloud size by aggregating nearby points into voxels.

3. **Map Loading**:
    - `map_loader` node (from `map_loader.cpp`) reads a `.pcd` file into memory and publishes it on `/map`.
4. **Scan Matching**:
    - `ndt_scanmatcher` node (from `ndt.cpp`) uses the downsampled scan and map to estimate the robot's position using the NDT algorithm.
5. **Pose Output**:
    - `/localized_pose`: The final estimated pose.
    - `/tf (map → base_link)`: TF transformation broadcast.
    - `/scan_matched_pointcloud`: Debugging scan output.

This tightly-coupled pipeline ensures efficient and accurate LiDAR-based localization in real-time.

4.3 Output of the System (Reaffirmed with Code Insights)

- **Primary Output**:
    - `/localized_pose`: This pose is calculated by the `ndt_scanmatcher` using PCL's NDT method.
    - The transform is broadcasted via `tf2_ros::TransformBroadcaster`.
- **Debug Output**:
    - `/scan_matched_pointcloud`: Generated in `ndt.cpp` using the aligned point cloud.
    - This can be visualized in RViz to verify scan-map alignment.
- **TF Integration**:
    - The NDT output is converted into a `geometry_msgs::TransformStamped` and broadcasted for other components (e.g., global planner, costmaps).