BAGS & TF

ROBOTICS

POLITECNICO
MILANO 1863

ROSBAG (again)

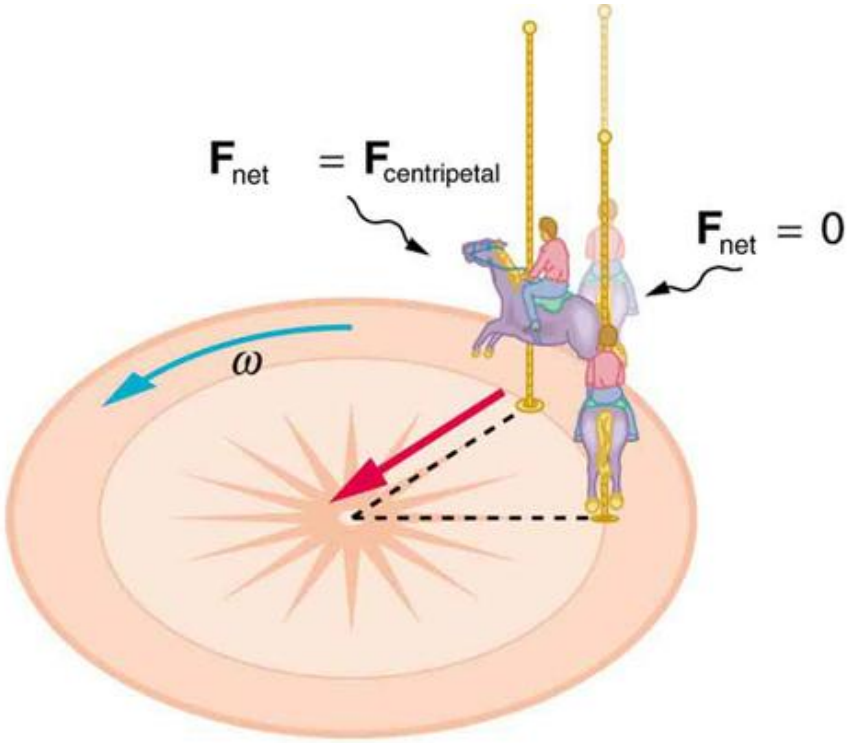
https://goo.gl/GonArW

Download the file robotics.bag

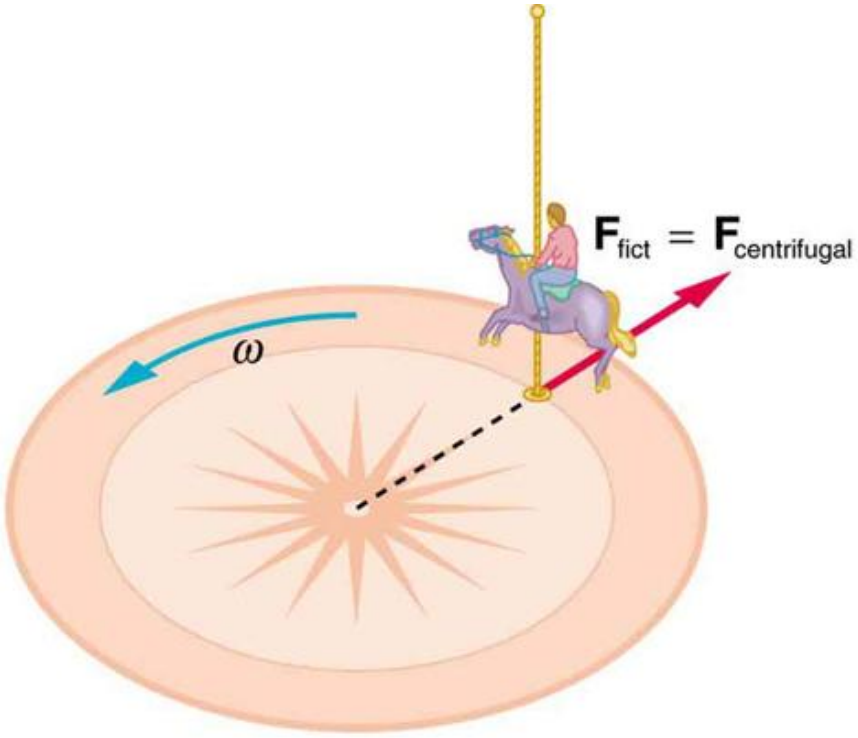
Also download the fibonacci folder

IN PHYSICS: AN EXAMPLE

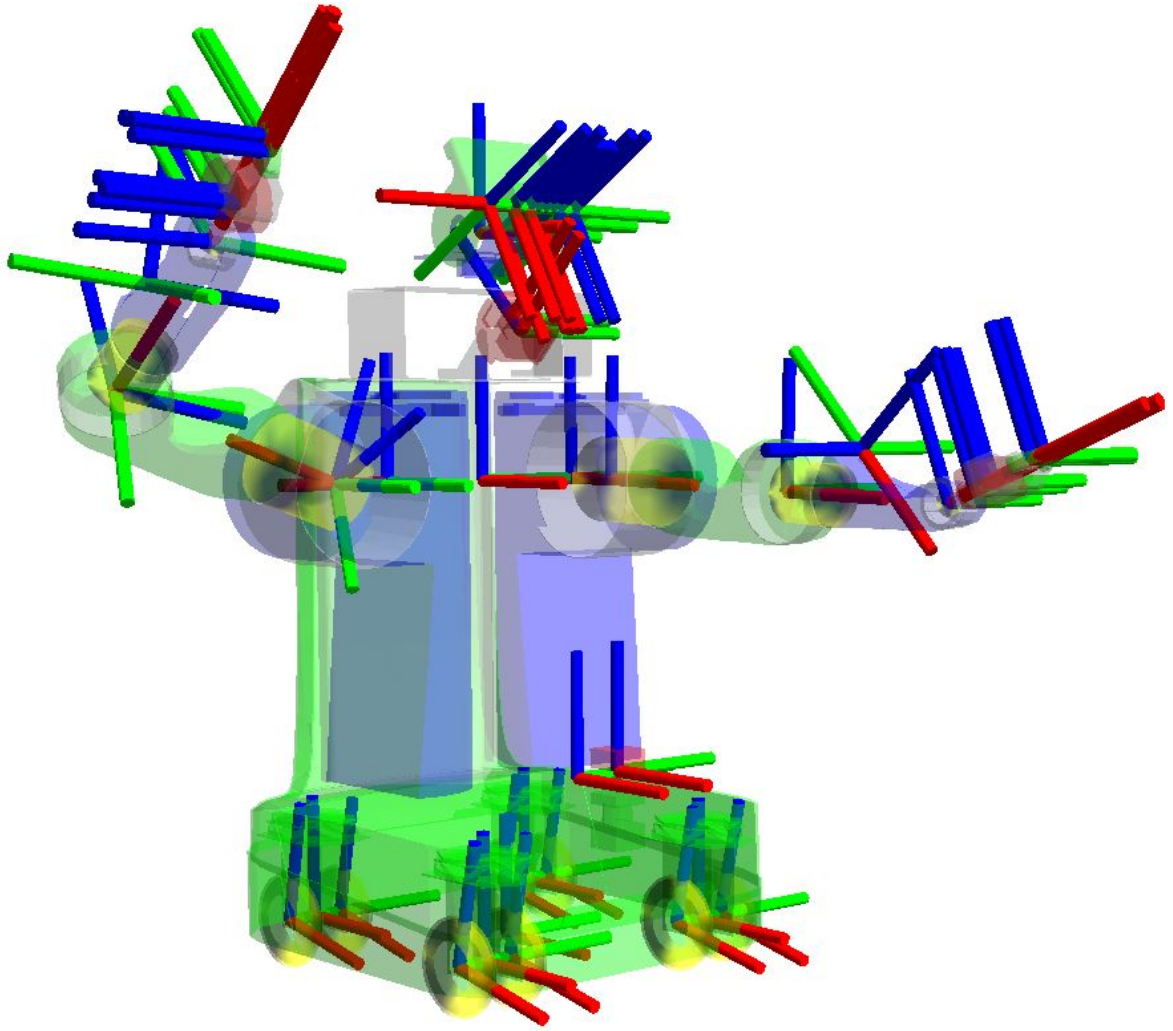
Reference System is everything



V
S



IN ROBOTICS



IN ROBOTICS

For manipulators:

- A moving reference frame for each joint
- A base reference frame
- A world reference frame

For autonomous vehicles:

- A fixed reference frame for each sensor
- A base reference frame
- A world reference frame
- A map reference frame

The frames are described in a tree and each frame comes with a transformation between itself and the father/child
The world frame is the most important, but the others are used for

FROM ONE FRAME TO ANOTHER

How is it possible to convert form a frame to another? ***Math***, lot of it.

In a tree of reference frames:

Define a roto-translation between parent and child

Combine multiple roto-translation to go from the root to the

TF: TRANSFORMATION FRAMES

When the full transformation tree is available

Does all the hard work for us!

Interpolation, transformation, tracking

Keep track of all the dynamic transformation for a limited period of time

Decentralized

Provides position of a point in each possible reference frame

TF TREE TOOLS

ROS offers different tools to analyze the transformation tree:

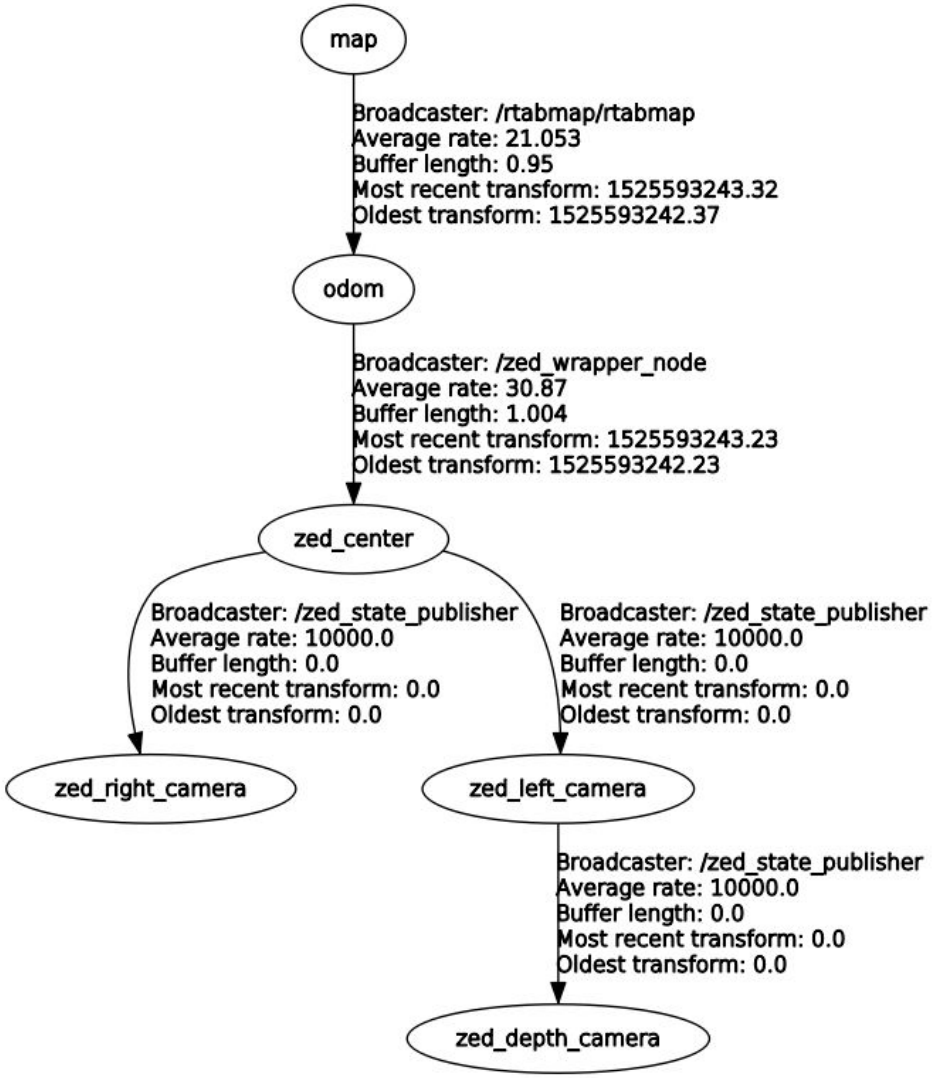
-rosrun rqt_tf_tree rqt_tf_tree

shows the tf tree at the current time

-rosrun tf view_frames

listen for 5 seconds to the /tf topic and create a pdf file with the tf tree

HOW TF_TREE SHOULD LOOK LIKE



TRY IT OUT (using bags)

Can’t provide you 50 cameras

But ROS has bags, cd to the folder where you downloaded the bag from slide 1

Bags can be played simply using (**remember to start roscore**):

$ rosbag play bagname.bag

But ROS offers also a decent gui to take a look at the bag file while playing it:

$ rqt_bag

TRY IT OUT (using bags)

Open your bag file and select the messages you want to publish (right click->Publish), in our case we select all of them

Now you can play the bag using the play button

Next open rqt_tf_tree:

$ rosrun rqt_tf_tree rqt_tf_tree

you should see a tf tree similar to the one from slide 9

if not restart the bag from the beginning (static transform are published in the first frames)

VISUALIZE TF (using rviz)

Tf tree allow us to take a look to the tf tree, how different transformation are connected together.

To properly visualize the data we will use rviz.

Keeping the bag running open rviz

$ rviz

First we have to set the Fixed Frame in the top of the left tab; using the drop down tab select “map”

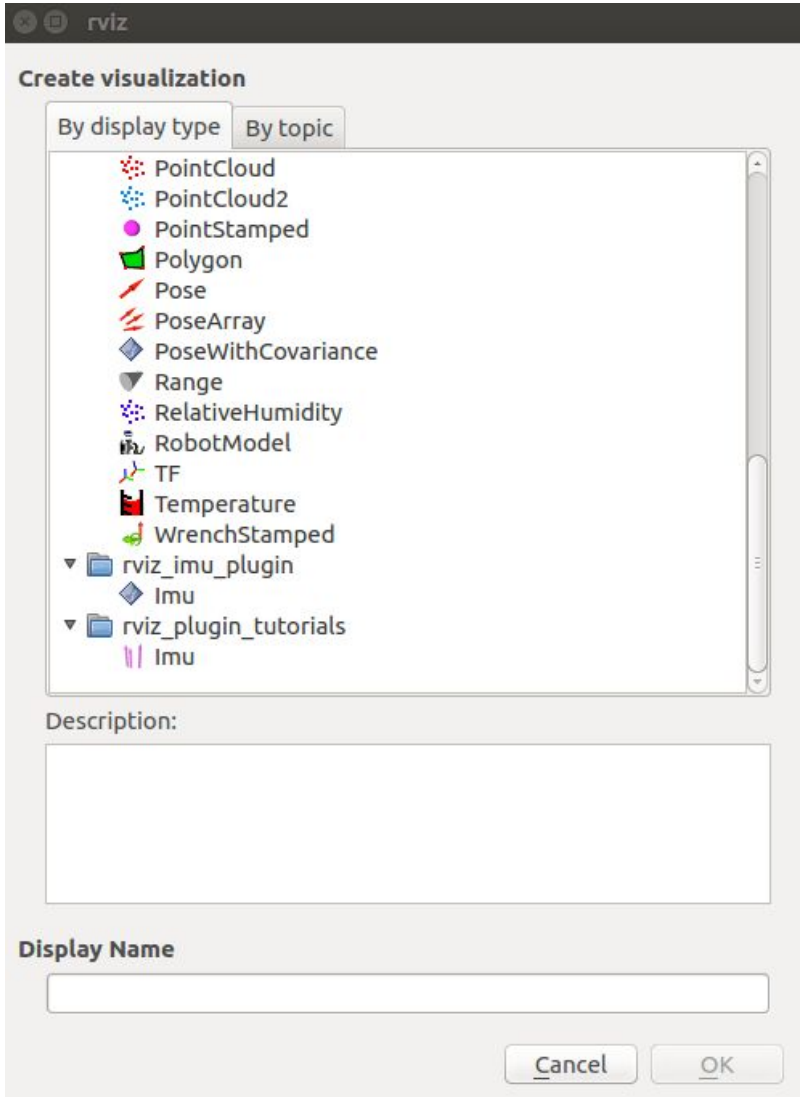
VISUALIZE TF (using rviz)

Now to visualize the position of the camera use the Add button

from the first tab of the window select TF

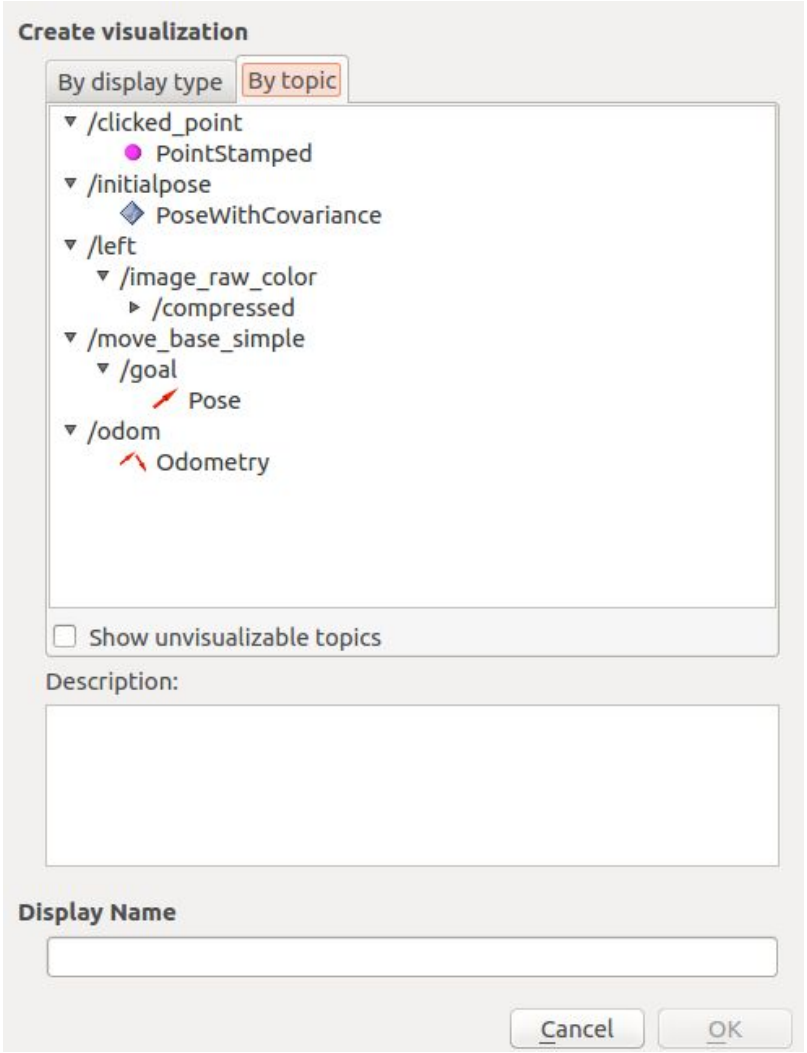
Now in the left tab you can edit preferences for the TF visualization

under frames uncheck everything but map and zed_center to show the camera movement



VISUALIZE TF (using rviz)

From the Add menu you can also select using the “By Topic” tab the odometry topic, which will also show the position of the camera and the left camera stream to get an idea of the camera movement



WRITE THE TF PUBLISHER

Now that we got an idea regarding how tf works and why it’s useful we can take a look on how to write a tf broadcaster

Usually to do this you need a robot,

we could still us a bag publishing odometry,

but turtlesim is still a good option.

THE IDEA

Subscribe to /turtlesim/pose

convert the pose to a transformation

publish the transformation referred to a world frame

add 4 static transformation for the 4 turtle’s legs

WRITE THE TF PUBLISHER

Create a package called tf_turtlebot inside you catkin environment adding the roscpp, std_msgs and tf dependencies:

$ catkin_create_pkg tf_turtlebot std_msgs roscpp tf

now cd to the package src folder and create the file tf_publisher

$ gedit tf_publisher.cpp

SUBSCRIBE AND PUBLISH IN THE SAME NODE

In all the previous example the node was simply subscribing or publishing.

In this case we need to both subscribe to the turtlesim pose and publish the tf

The classic way to solve this problem is to create a class which handles all the work

WRITE THE TF PUBLISHER

First we write some standard include:

```
#include "ros/ros.h"
#include "turtlesim/Pose.h"
#include <tf/transform_broadcaster.h>
```

SUBSCRIBE AND PUBLISH IN THE SAME NODE

Then we write the main function:

```
int main(int argc, char **argv)
{
 ros::init(argc, argv, "subscribe_and_publish");
 tf_sub_pub my_tf_sub_bub;
 ros::spin();
 return 0;
}
```

Notice that we still have to initialize ros, but we are not creating the node handle here, instead we instantiate an object of class tf_sub_pub

SUBSCRIBE AND PUBLISH IN THE SAME NODE

Now we have to create our class:

```
class tf_sub_pub
{
    public:
    tf_sub_pub(){
    }
    private:

};
```

WRITE THE TF PUBLISHER

First we declare as private the node handle:

```
ros::NodeHandle n;
```

Then we create the subscriber and the tf broadcaster:

```
tf::TransformBroadcaster br;
ros::Subscriber sub;
```

WRITE THE TF PUBLISHER

Now we can call the subscribe function inside the class constructor:

```
sub = n.subscribe("/turtle1/pose", 1000, &tf_sub_pub::callback, this);
```

Then we write the callback function:

```
void callback(const turtlesim::Pose::ConstPtr& msg){
}
```

WRITE THE TF PUBLISHER

Inside the callback we create a transform object:

```
tf::Transform transform;
```

and populate it using the data from the message (we are in a 2D environment):

```
transform.setOrigin( tf::Vector3(msg->x, msg->y, 0) );
tf::Quaternion q;
q.setRPY(0, 0, msg->theta);
transform.setRotation(q);
```

WRITE THE TF PUBLISHER

Last we publish the transformation using the broadcaster; the stampedtransform function allow us to create a stamped transformation adding the timestamp, our custom transformation, the root frame and the child frame:

```
br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world", "turtle"));
```

THE CODE

```
#include "ros/ros.h"
#include "turtlesim/Pose.h"
#include <tf/transform_broadcaster.h>
class tf_sub_pub
{
public:
        tf_sub_pub(){
        sub = n.subscribe("/turtle1/pose", 1000, &tf_sub_pub::callback, this);
}
void callback(const turtlesim::Pose::ConstPtr& msg){
tf::Transform transform;
transform.setOrigin( tf::Vector3(msg->x, msg->y, 0) );
tf::Quaternion q;
q.setRPY(0, 0, msg->theta);
transform.setRotation(q);
br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world", "turtle"));
}
private:
ros::NodeHandle n;
tf::TransformBroadcaster br;
ros::Subscriber sub;
};
int main(int argc, char **argv)
{
 ros::init(argc, argv, "subscribe_and_publish");
 tf_sub_pub my_tf_sub_bub;
 ros::spin();
 return 0;
}
```

WRITE THE TF PUBLISHER

Now as usual we have to add this new file to the CMakeLists file. We specified the dependencies during the package creation, so we only need to add the lines:

```
add_executable(tf_turtlebot
 src/tf_publisher.cpp
)
add_dependencies(tf_turtlebot ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
target_link_libraries(tf_turtlebot
 ${catkin_LIBRARIES}
)
```

TESTING

Now we can cd to the root of the environment and compile everything

Before adding the legs transformation we can test our code:

run turtlesim, turtlesim_teleop and our node, then open rviz to visualize the tf

$ roscore

$ rosrun turtlesim turtlesim_node

$ rosrun turtlesim turtle_teleop_key

$ rosrun tf_turtlebot tf_turtlebot

$ rviz

ADD STATIC TF

After properly testing our code we can add the other tf.

But the legs tf are fixed from the turtlebot body, so we don’t need to write a tf broadcaster like we did, we can simply run them using the static transform node

We don't’ want to manually start four tf in four different terminals, so we will create a launch file:
create a folder launch and a file called launch.launch

ADD STATIC TF

The launch file will have as usual the <launch> tags and the node we previously wrote:

```
<launch>
<node pkg="tf_turtlebot" type = "tf_turtlebot" name = "tf_turtlebot"/>
</launch>
```

We can also add the two turtlesim node:

```
<node pkg="turtlesim" type = "turtlesim_node" name = "turtlesim_node"/>
<node pkg="turtlesim" type = "turtle_teleop_key" name = "turtle_teleop_key"/>
```

ADD STATIC TF

Now we will add the four static tf specifying in the args field the position (x,y,z) and the rotation as a quaternion (qx,qy,qz,qw) then the root frame, the cild frame and the update rate:

```
<node pkg="tf" type="static_transform_publisher" name="back_right" args="0.3 -0.3 0 0 0 0 1 turtle FRleg 100" />
<node pkg="tf" type="static_transform_publisher" name="front_right" args="0.3 0.3 0 0 0 0 1 turtle FLleg 100" />
<node pkg="tf" type="static_transform_publisher" name="front_left" args="-0.3 0.3 0 0 0 0 1 turtle BLleg 100" />
<node pkg="tf" type="static_transform_publisher" name="back_left" args="-0.3 -0.3 0 0 0 0 1 turtle BRleg 100" />
```

Now we will only need to call the launch file to start all the nodes:

$ roslaunch tf_turtlebot launch.launch

ADD STATIC TF

Now run rqt_tf_tree to show the tf tree and rviz for the visual rappresentation of the turtle position

If you want to see the published tf you can use rostopic echo, but also:

$ rosrun tf tf_echo father child

$ rosrun tf tf_echo \world \FRleg

ACTIONLIB

ROBOTICS

POLITECNICO
MILANO 1863

WHAT IS ACTIONLIB

Node A sends a request to node B to perform some task

Service	**Action**
Small execution time	Long execution time
Requesting node can wait	Requesting node cannot wait
No status	Status monitoring
No cancellation	Cancellation

WHAT IS ACTIONLIB

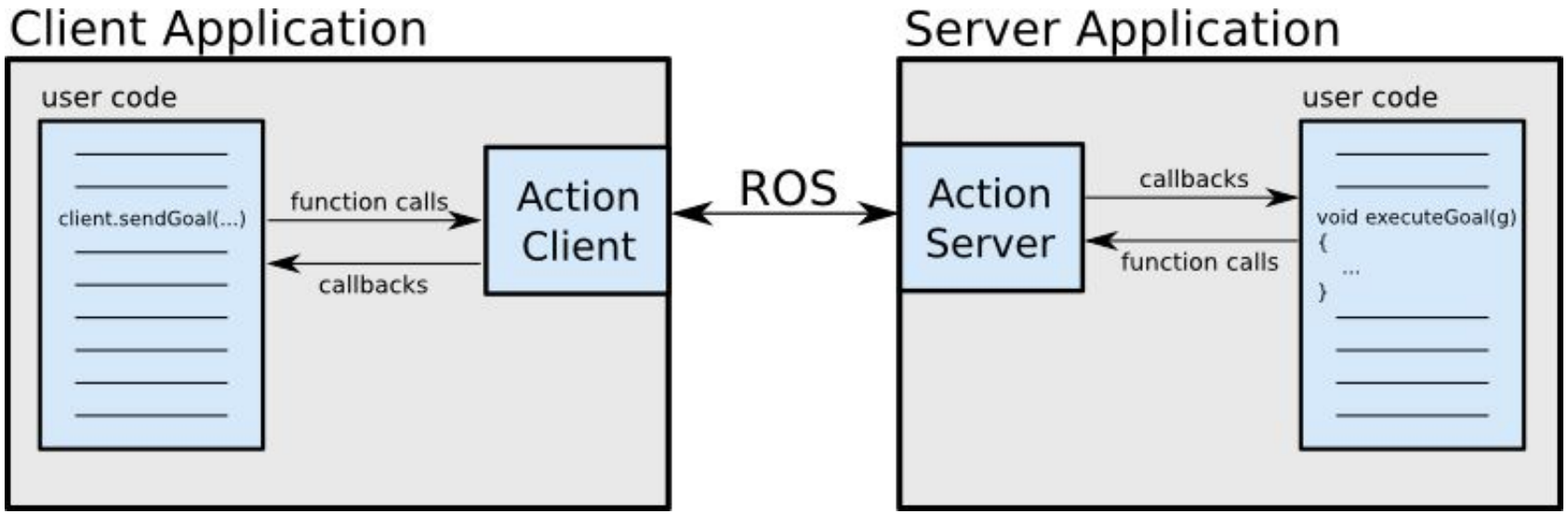
actionlib package is:

- sort of ROS implementation of threads
- based on a client/server paradigm

And provides tools to:

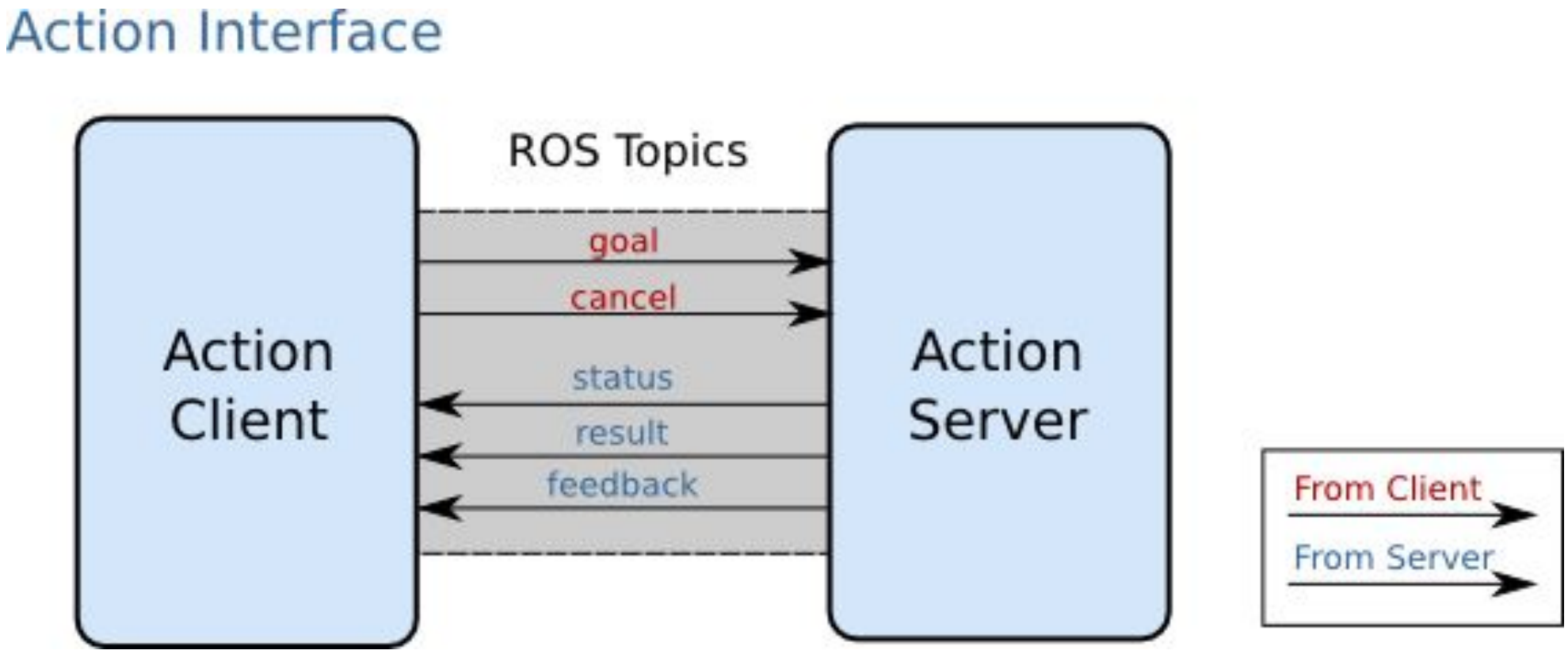
- create servers that execute long-running tasks (that can be preempted).
- create clients that interact with servers

WHAT IS ACTIONLIB



The ActionClient and ActionServer communicate via a "ROS Action Protocol", which is built on top of ROS messages

CLIENT-SERVER INTERACTION



CLIENT-SERVER INTERACTION

goal: to send new goals to server

cancel: to send cancel requests to server

status: to notify clients on the current state of every goal in the system.

feedback: to send clients periodic auxiliary information for a goal

result: to send clients one-time auxiliary information upon completion of a goal

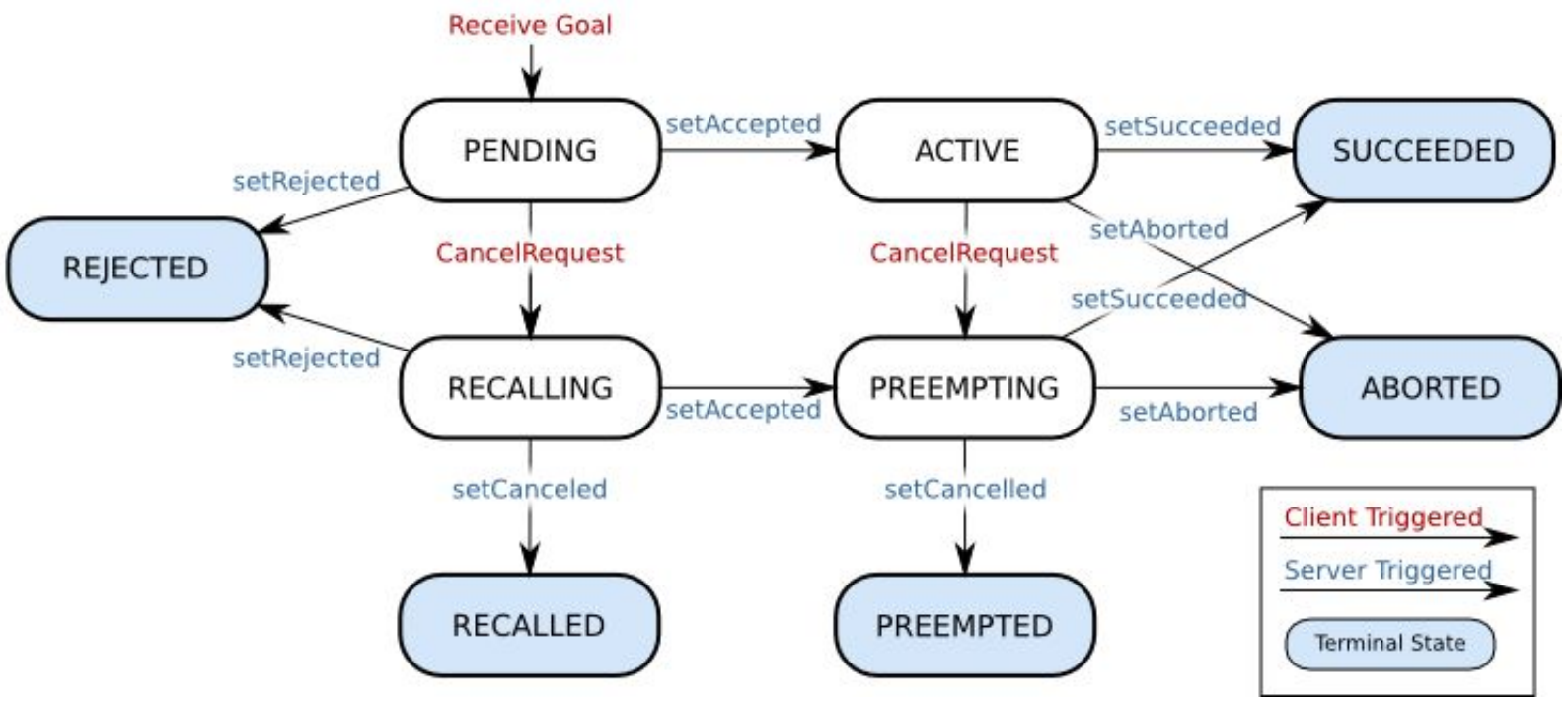
ACTION AND GOAL ID

Action templates are defined by a name and some additional properties through an .action structure defined in ROS

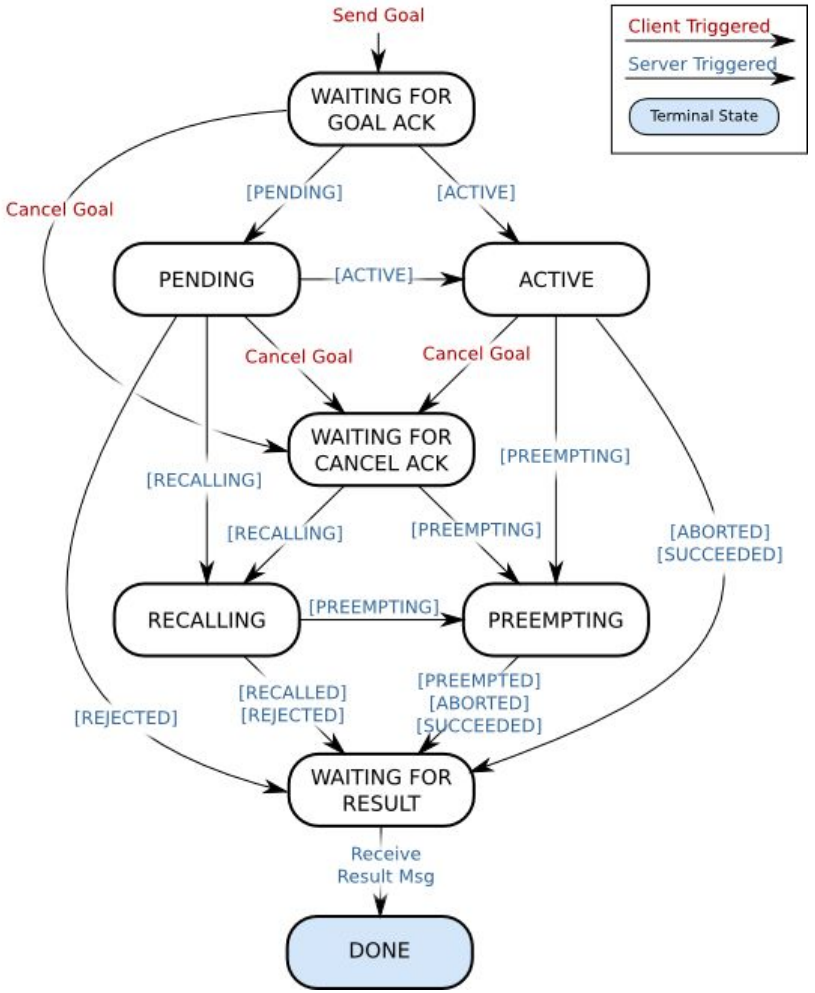
Each ***instance*** of an action has a unique Goal ID

Goal ID provides the action server and the action client with a robust way to monitor the execution of a particular instance of an action.

SERVER STATE MACHINE



CLIENT STATE MACHINE



.ACTION EXAMPLE

```
# Define the goal
uint32 dishwasher_id  # Specify the dishwasher id
---
# Define the result
uint32 total_dishes_cleaned
---
# Define a feedback message
float32 percent_complete
```

SIMPLEACTIONSERVER

```
int main(int argc, char** argv) {
  ros::init(argc, argv, "do_dishes_server");
  ros::NodeHandle n;
  Server server(n, "do_dishes", boost::bind(&exe, _1, &server), false);
  server.start();
  ros::spin();
  return 0;
}
```

SIMPLEACTIONSERVER

```
void exe(const chores::DoDishesGoalConstPtr& goal, Server* as) {
  while(allClean()) {
    doDishes(goal->dishwasher_id)
    if(as->isPreemptRequested() || !ros::ok()) {
      as->setPreempted();
      break;
    }
    as->publishFeedback(currentWork(goal->dishwasher_id))
  }
  if(currentWork(goal->dishwasher_id) == 100)
    as->setSucceeded();
}
```

SIMPLEACTIONCLIENT

```
#include <chores/DoDishesAction.h>
#include <actionlib/client/simple_action_client.h>

typedef actionlib::SimpleActionClient<chores::DoDishesAction> Client;
```

SIMPLEACTIONCLIENT

```
int main(int argc, char** argv) {
  ros::init(argc, argv, "do_dishes_client");
  Client client("do_dishes", true); // true -> don't need ros::spin()
  client.waitForServer();
  chores::DoDishesGoal goal;
  //set goal parameters
  goal.dishwasher_id = pickDishwasher();
```

SIMPLEACTIONCLIENT

```
  client.sendGoal(goal);
  client.waitForResult(ros::Duration(5.0));
  if (client.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
    ROS_INFO("Yay! The dishes are now clean");
  std::string state = client.getState().toString();
  ROS_INFO("Current State: %s\n", state.c_str());
  return 0;
}
```

TESTING

Copy the actionlib_tutorial folder inside the src folder of your catkin workspace and compile it

To start the server:

$ rosrun actionlib_tutorials fibonacci_server

The client has some parameters that can be set in the launch file, order and duration; after setting those parameters call:

$ roslaunch actionlib_tutorials launcher.launch

TESTING

You can monitor the server status simply using topics:

$ rostopic list

To get the feedback from the server:

$ rostopic echo /fibonacci/feedback