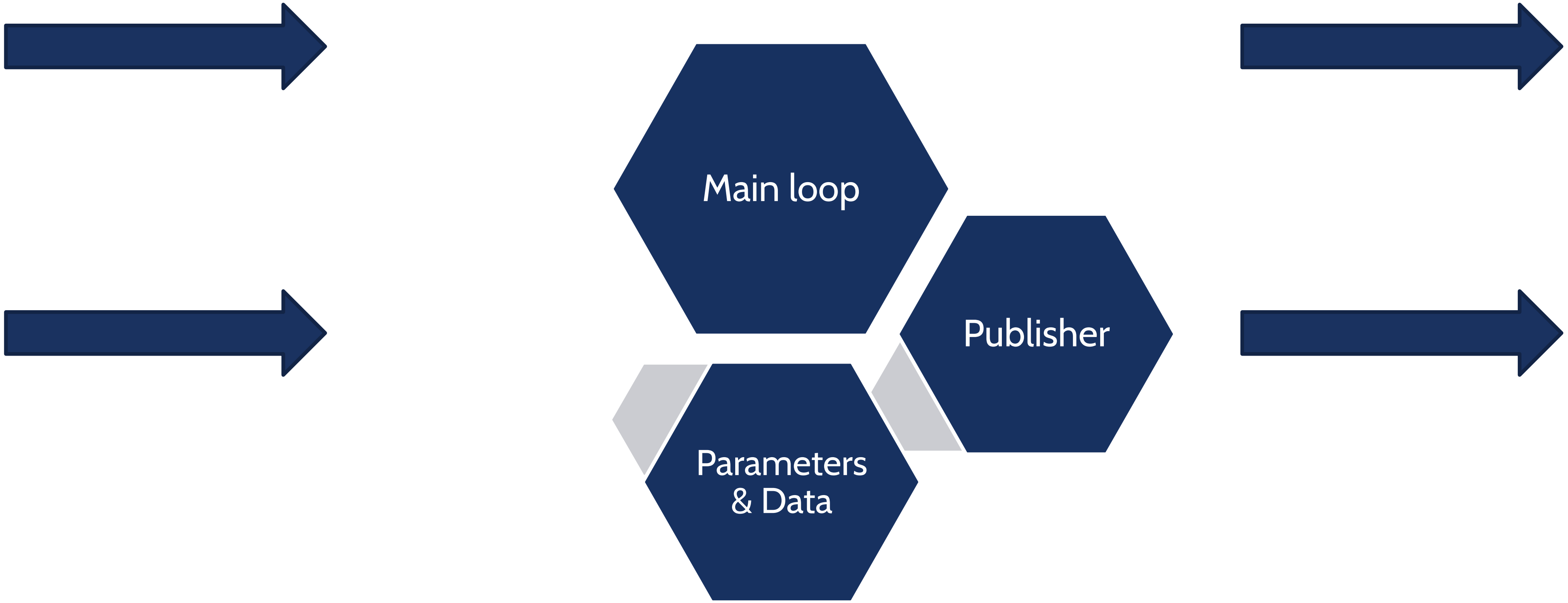
PARAMETERS

ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE



USING PARAMETERS

2 ways to use parameters:

- Look at the value before entering main loop
- Add callback to parameters change

3 ways to set parameters:

- command line
- launch file
- rqt_reconfigure

PARAMETER CHECK BEFORE MAIN LOOP

Similar code tu publisher/subscriber example:

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>
```

Standard include

PARAMETER CHECK BEFORE MAIN LOOP

Similar code tu publisher/subscriber example:

```
int main(int argc, char **argv){

    ros::init(argc, argv, "param_first")
    ros::NodeHandle n;

    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("parameter",
1000);
    std::string name;
    (...)
```

ros initialization

Publisher creation

PARAMETER CHECK BEFORE MAIN LOOP

Similar code tu publisher/subscriber example:

```
n.getParam("/name", name);
```

get parameter value

```
ros::Rate loop_rate(10);
```

set loop rate

PARAMETER CHECK BEFORE MAIN LOOP

Similar code tu publisher/subscriber example:

```
while (ros::ok()){
        std_msgs::String msg;
     msg.data = name;
     ROS_INFO("%s", msg.data.c_str());
     chatter_pub.publish(msg);
     ros::spinOnce();
     loop_rate.sleep();

  }
```

Main loop, not different from previous example

PARAMETER CHECK BEFORE MAIN LOOP

Add the new file to CMakeLists.txt, as we did in pub/sub example

```
add_executable(param_first src/param_first.cpp)
target_link_libraries(param_first ${catkin_LIBRARIES})
```

Compile the new node

PARAMETER CHECK BEFORE MAIN LOOP

Start the node:

- If no parameter is previously set the node will publish an empty string
- Set the parameter value using: “ rosparam set name "first" ”
- Now the node will publish “first” string
- If you change again the value while the node is running it will have no effect because the node looks at the value only once

SETTING PARAMETER VALUE INSIDE THE LAUNCH FILE

A good practice with parameter is to set the value directly inside the launch file, so the user doesn't have to initialize the values using command line tools, add the line:

```
<param name="name" value="value" />
```

Inside a Launch file to set a parameter

SETTING PARAMETER VALUE INSIDE THE LAUNCH FILE

Create a param_set.launch file inside a launch folder

```
<launch>
 <param name="name" value="second" />
 <node pkg="parameter_test" name="param_first" type="param_first"
output="screen"
/>

</launch>
```

Set the parameter value

Redirect the output of ROS_INFO to the terminal

DYNAMIC RECONFIGURE

Previous examples allowed us to set the parameter value only once, to change the value while the node is running it's not recommended to insert the getParam call inside the mail loop because it's resource consuming and inefficient, to achieve this task we use dynamic reconfigure

DYNAMIC RECONFIGURE

First create a cfg folder and inside a parameters.cfg file, than make it executable:

```
chmod +x parameters.cfg
```

Now we can start writing the configuration file; cfg file are not written in c++ but in python

DYNAMIC RECONFIGURE

```
#!/usr/bin/env python
PACKAGE = "parameter_test"

from dynamic_reconfigure.parameter_generator_catkin import *

gen = ParameterGenerator()
```

Set the package of the node

Import for dynamic reconfigure

Create a generator

DYNAMIC RECONFIGURE

To add a parameter we use the command:

```
gen.add (“name”, type, level, “description”,default, min, max)
```

In our case:

```
gen.add("int_param",    int_t,    0, "An Integer parameter", 50,  0, 100)
gen.add("double_param", double_t, 0, "A double parameter",    .5, 0,   1)
gen.add("str_param",    str_t,    0, "A string parameter",  "Hello World")
gen.add("bool_param",   bool_t,   0, "A Boolean parameter",  True)
```

DYNAMIC RECONFIGURE

We can also create multiple choice parameter using enum, first create an enum using a list of const; to create a constant:

```
gen.const (“name”, type, value, “description”)
```

Than create the enum:

```
my_enum = gen.enum([const_1, cosnt_2, ...], “description”)
```

Last we add the enum like previously

```
gen.add (“name”, type, level, “description”,default, min, max, edit_method = my_enum)
```

DYNAMIC RECONFIGURE

In our case we create a size parameter with four values:

```
size_enum = gen.enum([ gen.const("Small",      int_t, 0, "A small constant"),
                       gen.const("Medium",     int_t, 1, "A medium constant"),
                       gen.const("Large",      int_t, 2, "A large constant"),
                       gen.const("ExtraLarge", int_t, 3, "An extra large 
constant")],
                     "An enum to set size")

gen.add("size", int_t, 0, "A size parameter which is edited via an enum", 1, 
0, 3, edit_method=size_enum)
```

DYNAMIC RECONFIGURE

Lastly we have to tell the generator to generate the files:

```
gen.generate(“package name”, "node_name", "prefix")
```

Name of the package   Name of the node   Name of the prefix

The prefix value is the string used to create the name of the header file you will have to include, with the name `“prefixConfig.h”`

DYNAMIC RECONFIGURE

In our case we write:

```
exit(gen.generate(PACKAGE, "param_second", "parameters"))
```

Now we can write a node using those parameters, create a file "param_second.cpp" in your src folder

DYNAMIC RECONFIGURE

```
#include <ros/ros.h>
#include <dynamic_reconfigure/server.h>
```

Standard include

```
#include <parameter_test/parametersConfig.h>
```

Include the previously generated file

DYNAMIC RECONFIGURE

```
int main(int argc, char **argv) {
  ros::init(argc, argv, "param_second");
```

ROS initialization

```
  dynamic_reconfigure::Server<parameter_test::parametersConfig> server;
```

Create the parameter server specifying the type of config

```
dynamic_reconfigure::Server<parameter_test::parametersConfig>::CallbackType f;
```

Create the callback

DYNAMIC RECONFIGURE

```
f = boost::bind(&callback, _1, _2);
```

Bind the callback

```
server.setCallback(f);
```

Set the server callback

```
ROS_INFO("Spinning node");
ros::spin();
return 0;
```

keep spinning

DYNAMIC RECONFIGURE

```
void callback(parameter_test::parametersConfig &config, uint32_t level) {
```

Create the callback

Pointer to the parameters structure

Value of the level bitmask

DYNAMIC RECONFIGURE

Last we print all the parameters value

```
ROS_INFO("Reconfigure Request: %d %f %s %s %d",
            config.int_param, config.double_param,
            config.str_param.c_str(),
            config.bool_param?"True":"False",
            config.size);
```

DYNAMIC RECONFIGURE

We also have to edit the CMakeLists.txt, to the find_package call

add: “`dynamic_reconfigure`”

Also add the .cfg file:

```
generate_dynamic_reconfigure_options(
  cfg/parameters.cfg
)
```

And to prevent to first create the header file and than compile our node use:

```
add_dependencies(param_second ${PROJECT_NAME}_gencfg)
```

DYNAMIC RECONFIGURE

The level bitmask can be used to get what parameter has changed, edit the parameters.cfg file and set unique values to the level field

In the param_second.cpp callback add:

```
ROS_INFO ("%d",level);
```

To print the index of the label of the level value of the changed parameter

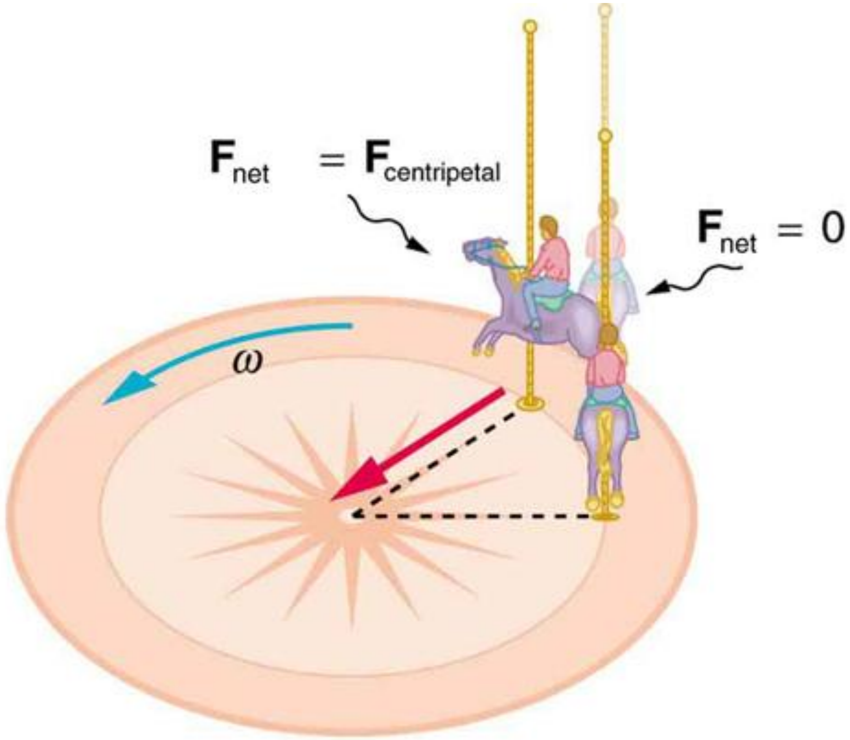
TF

ROBOTICS

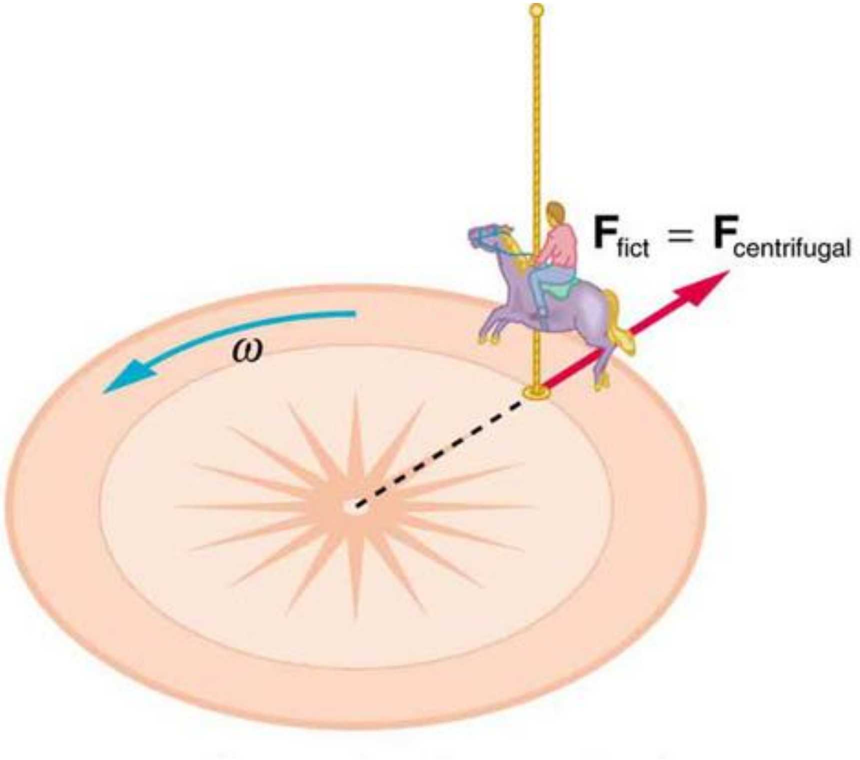
POLITECNICO
MILANO 1863

IN PHYSICS: AN EXAMPLE

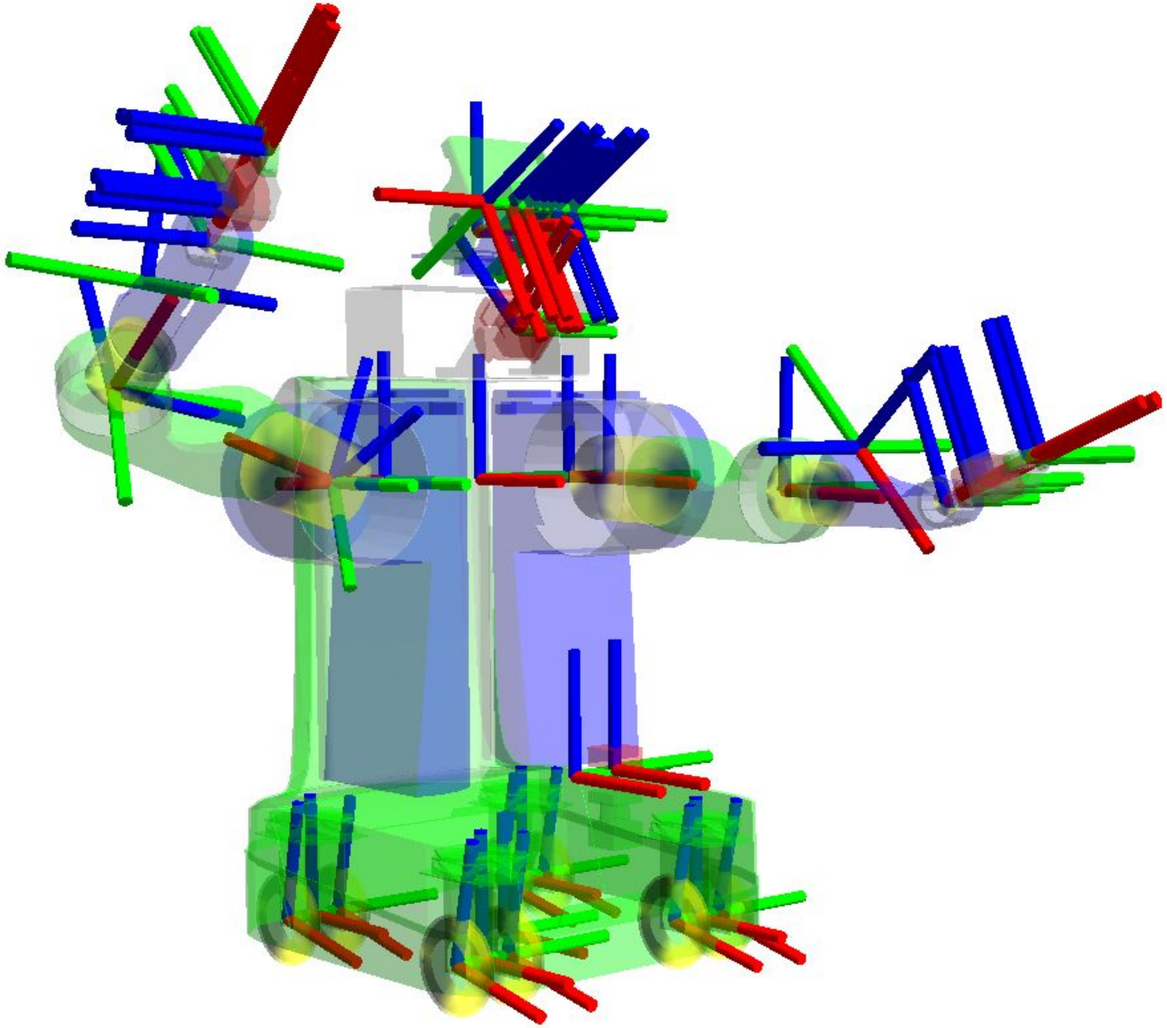
Reference System is everything



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:

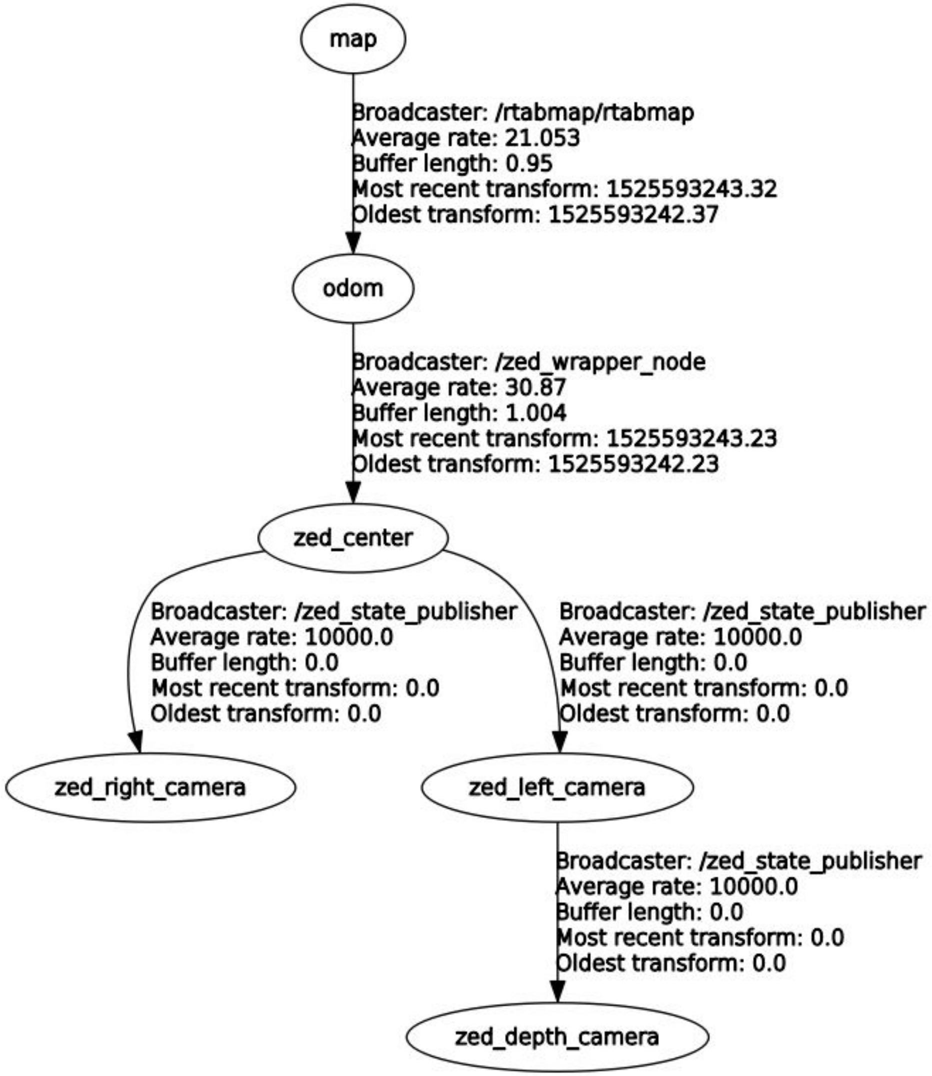
-rosrun rqt_tf_tree rqt_tf_tree

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



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.

WRITE THE TF PUBLISHER

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
```

WRITE THE TF PUBLISHER

First we write some standard include:

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

WRITE THE TF PUBLISHER

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

WRITE THE TF PUBLISHER

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'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