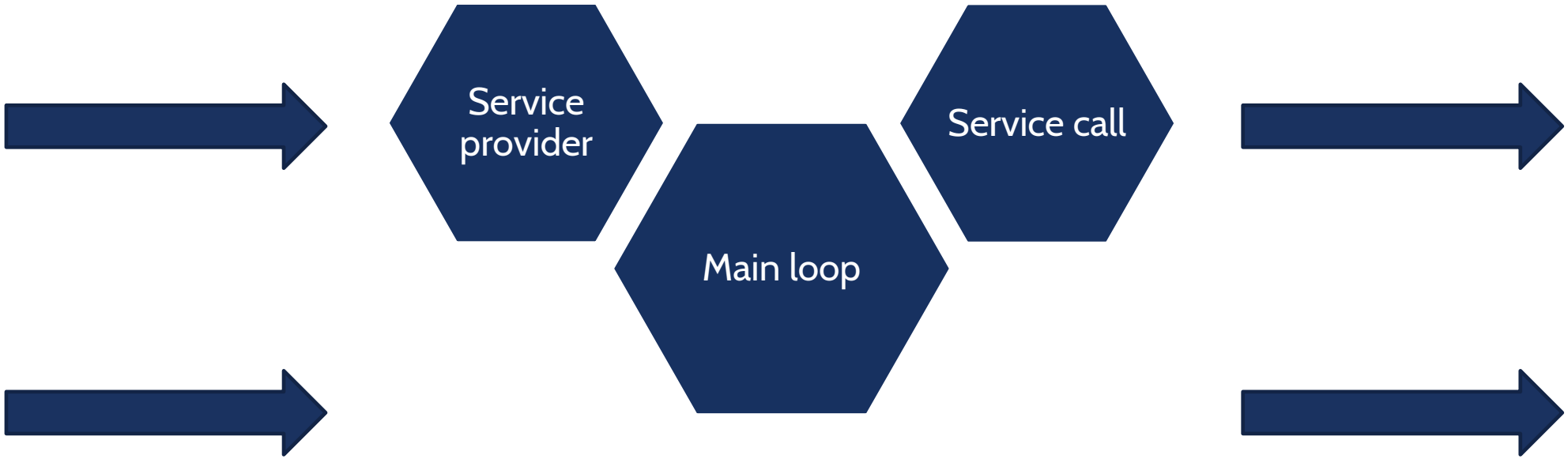
SERVICES

ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE



SERVICES

The service creation process is similar to the custom messages, first we create a srv folder where we insert the structure of the service, in our example we create the file AddTwoInts.srv

```
int64 a
int64 b
---
int64 sum
```

SERVICES (Server)

Than we create the service server, create a file add_two_ints.cpp in the src folder

```
#include "ros/ros.h"
#include "service/AddTwoInts.h"
```

Standard ROS include

Include the header file generated from the AddTwoInts.src

SERVICES (Server)

Standard main where we initialize ROS and create the node handle

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

SERVICES (Server)

Next we create the service server:

```
ros::ServiceServer service = n.advertiseService("add_two_ints", add);
```

Name of the service

Callback function

SERVICES (Server)

And we start spinning

```
  ROS_INFO("Ready to add two ints.");
  ros::spin();

  return 0;
}
```

SERVICES (Server)

Last we write the callback function, differently from the subscriber we have two fields, one for the inputs and one for the outputs:

```
bool add(service::AddTwoInts::Request &req,service::AddTwoInts::Response &res)
```

Type of the service

Pointer to the input

Pointer to the output

SERVICES (Server)

Inside the callback we compute the output value, print some information for debug and return:

```
res.sum = req.a + req.b;
ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
ROS_INFO("sending back response: [%ld]", (long int)res.sum);
return true;
```

SERVICES (Client)

Now we can write the client, as for the server we have to include the service header

```
#include "ros/ros.h"
#include "service/AddTwoInts.h"
```

SERVICES (Client)

Next we initialize ROS and check if the node was properly started passing the two integers to sum

```
 int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_client");
  if (argc != 3)
  {
    ROS_INFO("usage: add_two_ints_client X Y");
    return 1;
  }
```

SERVICES (Client)

Then we create the node handle and a service client using the service type and its name. Next we create the service object and set the input fields

```
 ros::NodeHandle n;
ros::ServiceClient client = n.serviceClient<service::AddTwoInts>("add_two_ints");
  service::AddTwoInts srv;
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);
```

SERVICES (Client)

Last we try calling the server and if we get a response we print it

```
if (client.call(srv))
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
  }
  else
  {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }
```

SERVICES (CMakeLists.txt)

We also have to do some changes in the CMakeLists.txt; first add

“`message_generation`” on the find_package function

Then add the service file

```
add_service_files(
  FILES
  AddTwoInts.srv
)
```

SERVICES (CMakeLists.txt)

Next we also have to set:

```
generate_messages(
   DEPENDENCIES
   std_msgs
)
```

And:

```
catkin_package(CATKIN_DEPENDS message_runtime)
```

SERVICES (CMakeLists.txt)

Last, to make sure that the header file are generated before compiling the nodes we add:

```
add_dependencies(add_two_int ${catkin_EXPORTED_TARGETS})
add_dependencies(client ${catkin_EXPORTED_TARGETS})
```

After the add_executable and target_link_libraries call

SERVICES (Package.xml)

We also have to edit the Package.xml to add the new dependencies, insert:

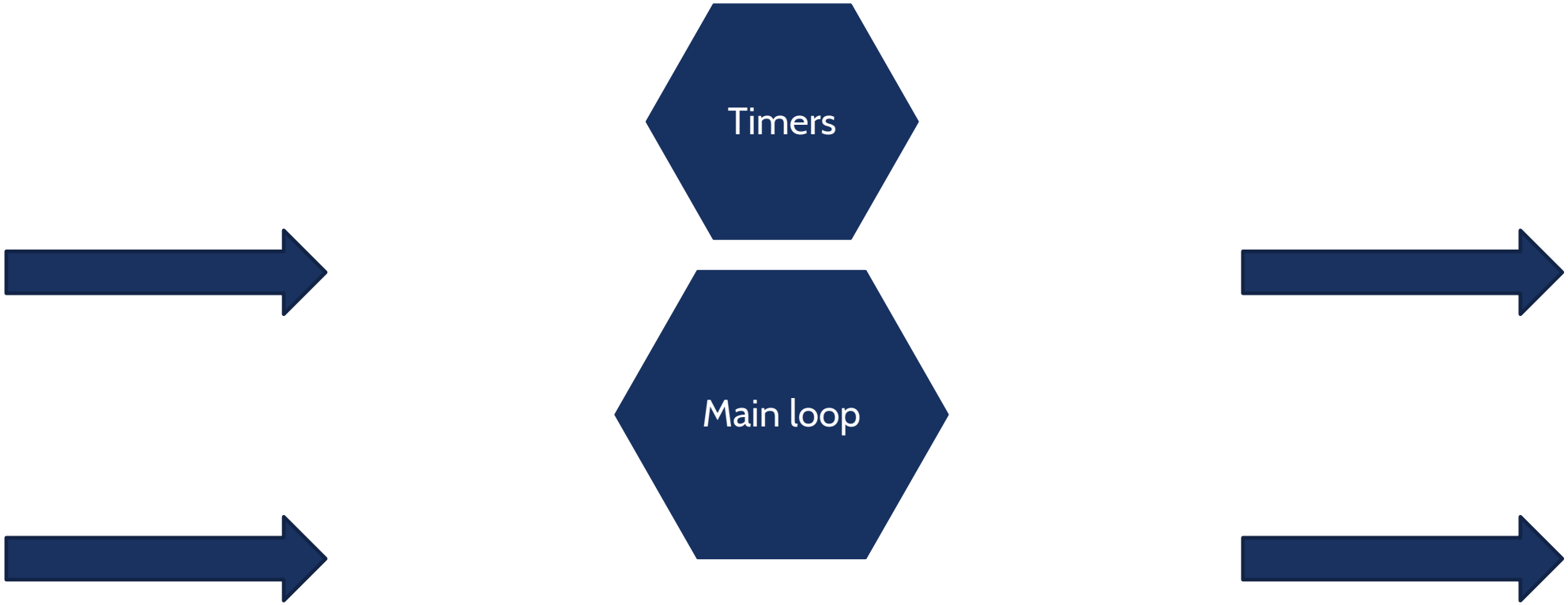
```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

TIMERS

ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE



TIMERS

Timers are similar to subscriber, we setup a callback which will be called at timer data rate

Create a file in your src folder called pub.cpp

TIMERS

```
#include "ros/ros.h"
#include <time.h>
```

Standard ROS include

Include time, only for debug purposes, not needed for timer usage

TIMERS

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

    ros::Timer timer = n.createTimer(ros::Duration(0.1), timerCallback);

    ros::spin();
    return 0;
}
```

Timer duration

Timer callback

Keep spinning

TIMERS

```
void timerCallback(const ros::TimerEvent& ev){

    ROS_INFO_STREAM("Callback called at time" <<  ros::Time::now());

}
```

Timer callback

Print to terminal

Get current time

TIMERS

Both CMakeLists.txt and Package.xml don’t require particular changes from the pub/sub example to work with timers

PUB/SUB in the same node (good practice)

ROBOTICS

POLITECNICO
MILANO 1863

PUB-SUB

Up to now a node was a publisher or a subscriber, if we want to do both tasks in the same node we will need a more elaborated structure: good practice is to create a class which contains both the publisher and the subscriber

PUB-SUB

To test our publisher-subscriber node we will subscriber to two different unsynchronized topics and re-publish them at constant rate using the last message received.

First we create a test_pub.cpp file which will publish two topics

PUB

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <stdlib.h>
#include <sstream>
int main(int argc, char *argv[])
{
    ros::init(argc, argv, "publisher");
    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
    ros::Publisher chatter_pub2 = n.advertise<std_msgs::String>("chatter2", 1000);
    ros::Rate loop_rate(100);
    int count = 0;
```

Standard ROS include
plus stdlib for random

we create two publisher

PUB

```
while (ros::ok())
    {
        std_msgs::String msg;
        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());
        if (rand() % 10 <6){
            chatter_pub.publish(msg);
        }
        if (rand() % 10 <2){
            chatter_pub2.publish (msg);
        }
```

We randomly publish a message

PUB-SUB

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

In the main function we only initialize ROS, we don’t create a NodeHandle, but we create the my_pub_sub object

PUB-SUB

```
class pub_sub
{
std_msgs::String messagio;
std_msgs::String messagio2;

private:
ros::NodeHandle n;

ros::Subscriber sub;
ros::Subscriber sub2;
ros::Publisher pub;
ros::Timer timer1;
```

All the ros code will be inside this class

Here we save the message that we want to republish

NodeHandle, publisher and subscriber are created here as private

PUB-SUB

```
public:
    pub_sub(){
    sub = n.subscribe("/chatter", 1, &pub_sub::callback, this);
    sub2 = n.subscribe("/chatter2", 1, &pub_sub::callback2, this);
    pub = n.advertise<std_msgs::String>("/rechatter", 1);
    timer1 = n.createTimer(ros::Duration(1), &pub_sub::callback1, this);
}
```

Here we set our two subscriber, the publisher and a timer at 1Hz

PUB-SUB

```
void callback(const std_msgs::String::ConstPtr& msg){
messagio=*msg;
}
void callback2(const std_msgs::String::ConstPtr& msg){
messagio2=*msg;
}
void callback1(const ros::TimerEvent&)
{
    pub.publish(messagio);
    pub.publish(messagio2);
    ROS_INFO("Callback 1 triggered");
}
```

First message callback

Second message callback

Timer callback which publish the two messages

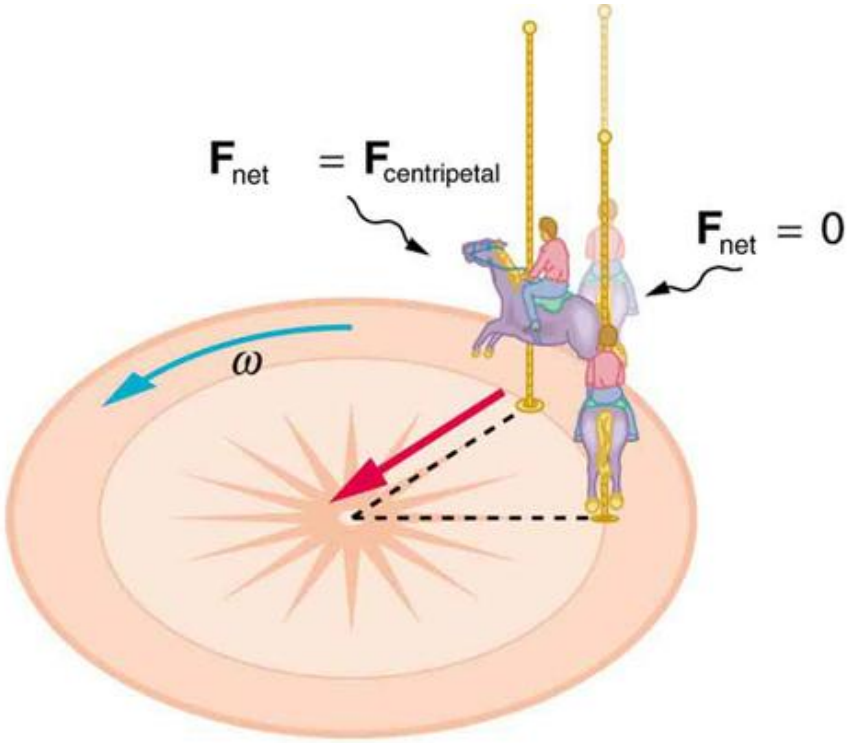
TF

ROBOTICS

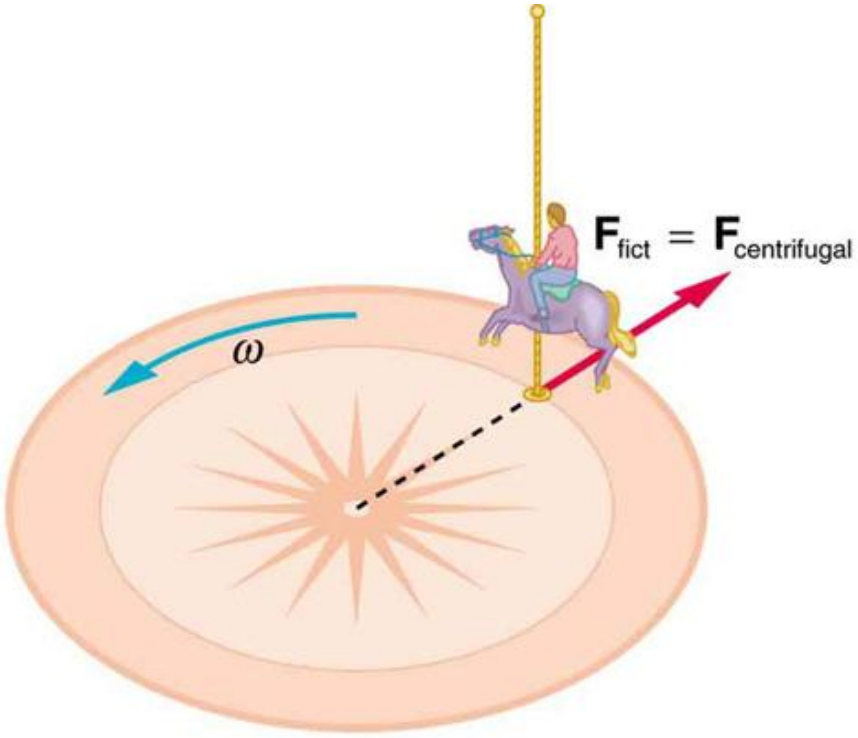
POLITECNICO
MILANO 1863

IN PHYSICS: AN EXAMPLE

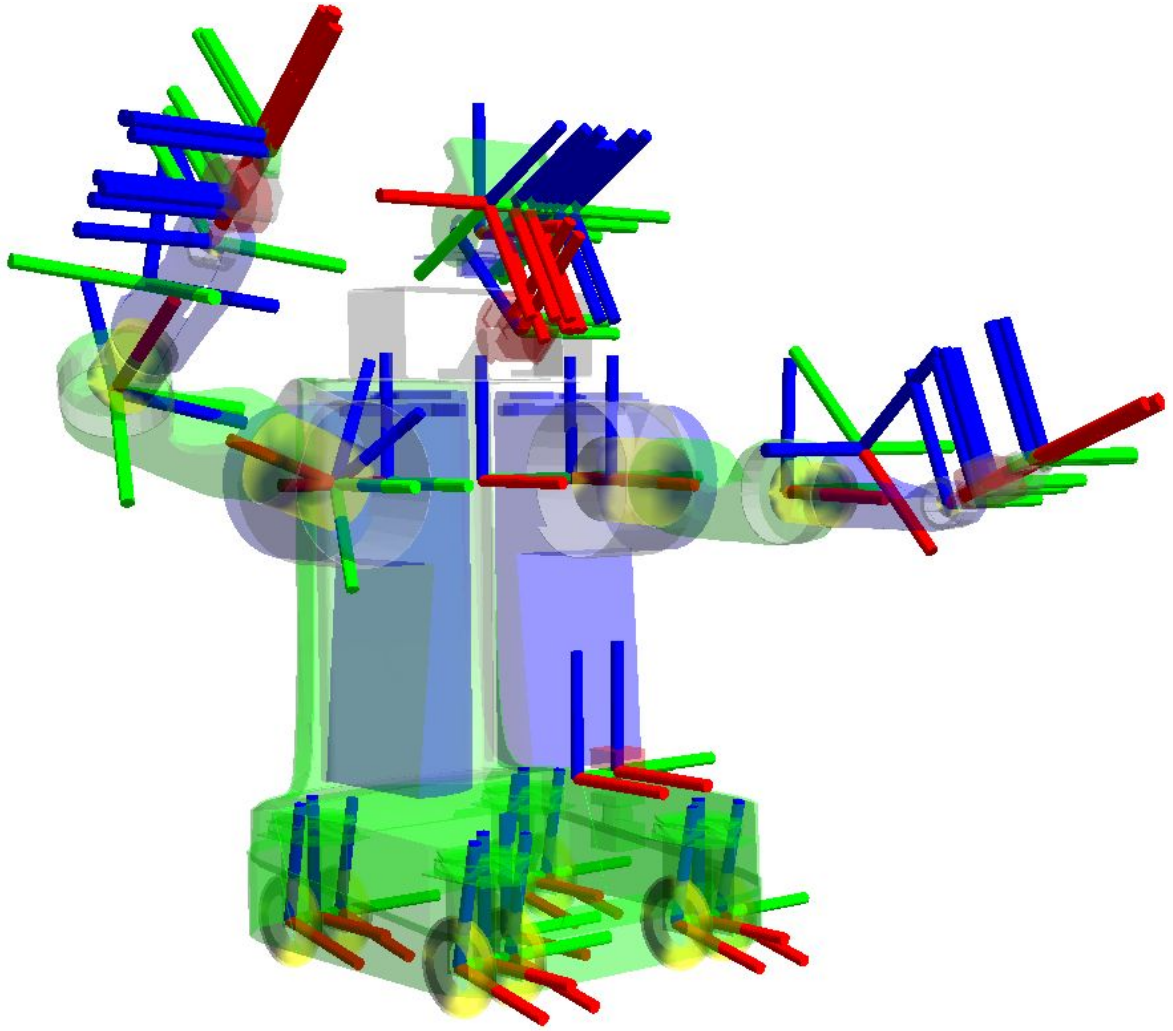
Reference System is everything



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:

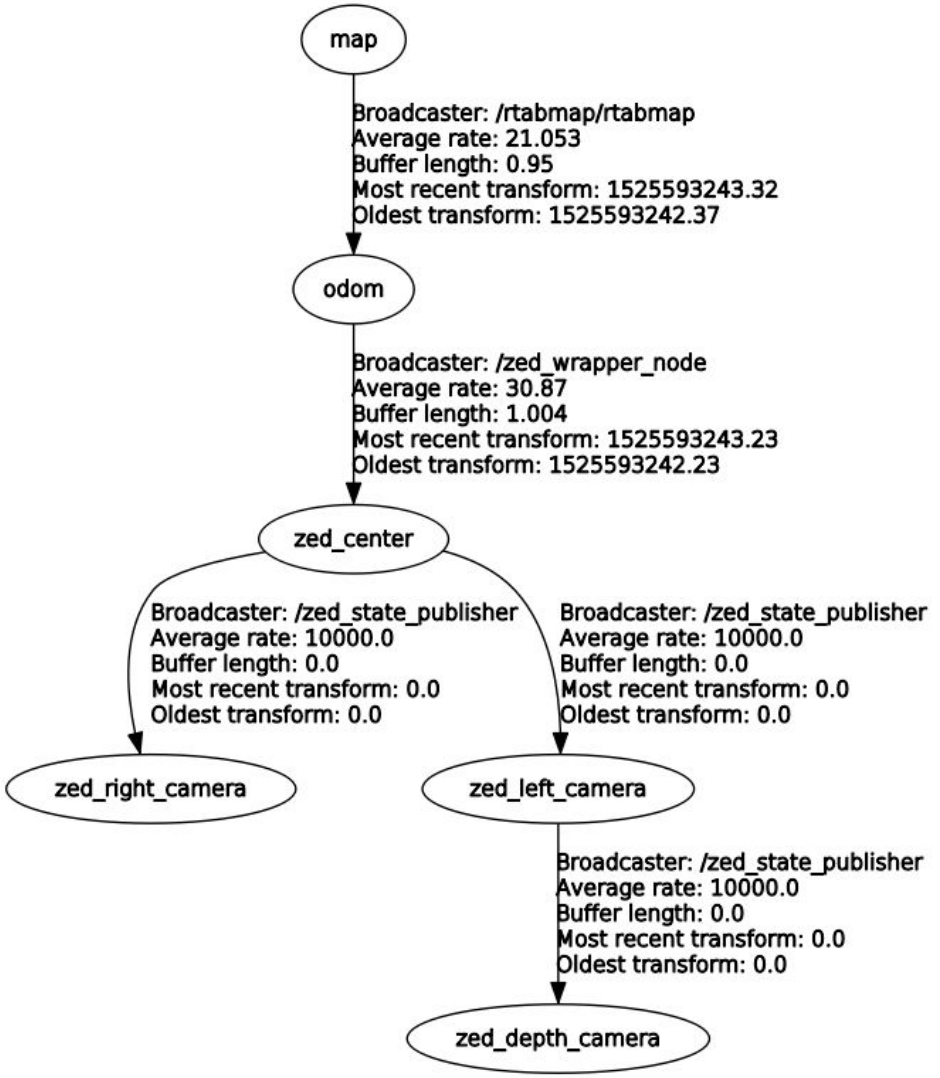
-rosrun rqt_tf_tree rqt_tf_tree

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’ 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