Robotics

Lab session 5

POLITECNICO
MILANO 1863

Paolo Cudrano
paolo.cudrano@polimi.it

RECAP

Last lecture:

- ROS development (part 2)
    - ROS names and remapping
    - Launch files
    - Custom messages
    - Services
    - Parameters
        - Static
        - Dynamic …
    - Timers

OUTLINE

Today:

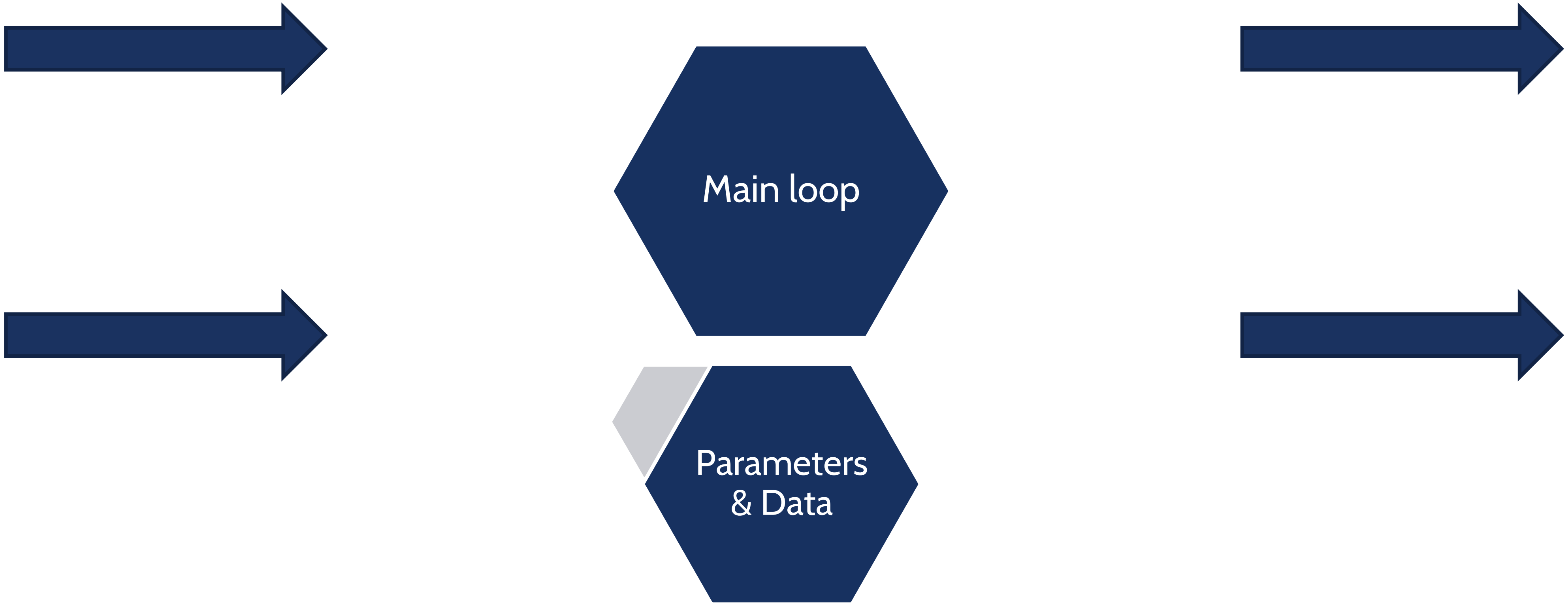
- ROS development (part 2)
  - ROS names and remapping
  - Launch files
  - Custom messages
  - Services
  - Parameters
    - Static
    - … Dynamic
  - Timers
- ROS callbacks and good practices
- ROS tools
  - TF
  - rviz
  - rqt_plot and plotjuggler

PARAMETERS: DYNAMIC RECONFIGURE

ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE



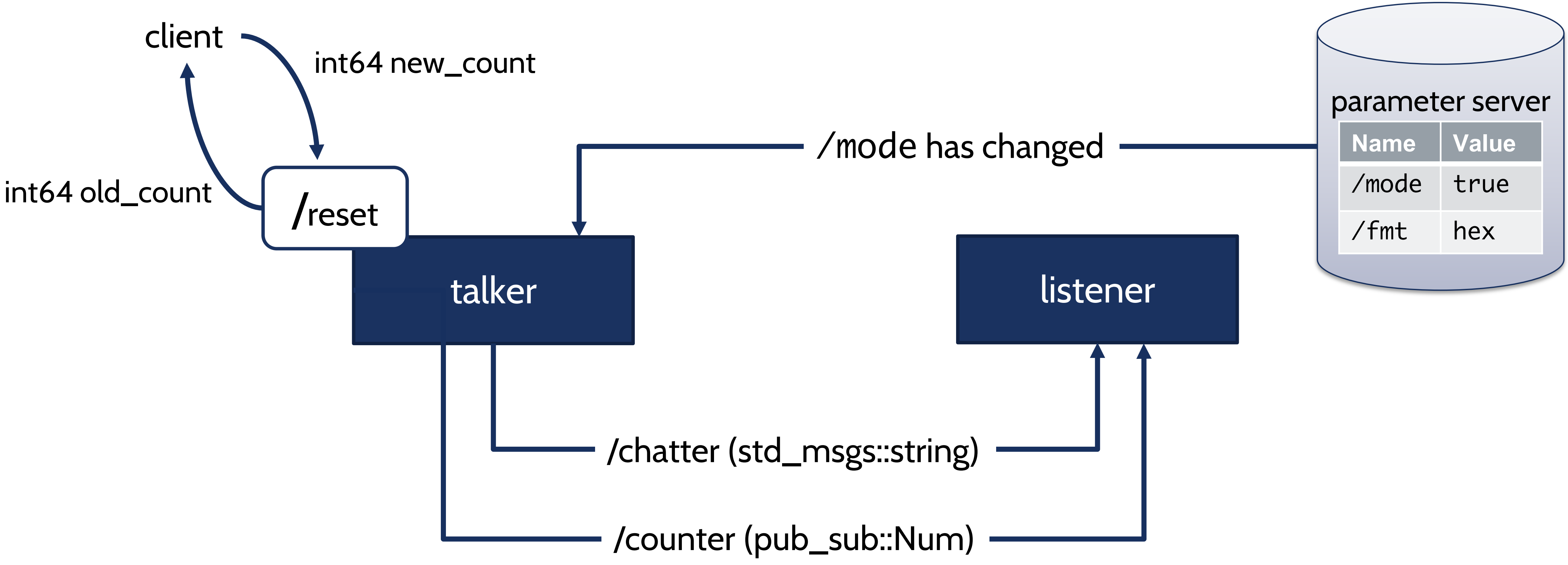
DYNAMIC RECONFIGURE

The previous example allowed us to set the parameter value only once

If we plan to change the value while the node is running, it is not recommended to insert the call to `getParam()` inside the main loop, as it is resource-consuming and inefficient

Instead, we can use **dynamic reconfigure**, which uses callbacks to notify us when a watched parameter has changed

Objective: pub_sub_v5



DYNAMIC RECONFIGURE

First, we create a folder `cfg` and, inside, a file `parameters.cfg`

Then, we make this file executable:

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

Now we can start writing our configuration file

`cfg` files are written in Python

DYNAMIC RECONFIGURE

In `parameters.cfg`:

```
#!/usr/bin/env python

PACKAGE = "parameter_test"

from dynamic_reconfigure.parameter_generator_catkin import *

gen = ParameterGenerator()
```

Set the package of the node

Import line 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)
```

For example:

```
gen.add("int_param",    int_t,    0, "An Integer parameter", 50,  0, 100)
gen.add("double_param", double_t, 1, "A double parameter",    .5, 0,   1)
gen.add("str_param",    str_t,    2, "A string parameter",  "Hello World")
gen.add("bool_param",   bool_t,   3, "A Boolean parameter",  True)
```

In our case:

```
gen.add(”mode",   bool_t,   0, ”Mode selecting which topic to publish",  True)
```

DYNAMIC RECONFIGURE

We can also create multiple choice parameters using enumerations

First, create an enum using a list of const. To create a constant:

```
const_1 = gen.const (“name”, type, value, “description”)
```

Then, create the enum:

```
my_enum = gen.enum([const_1, const_2, ...], “description”)
```

Lastly, add the enum to the generator

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

DYNAMIC RECONFIGURE

In our case, we create a parameter `fmt` with three possible values:

```
fmt_enum = gen.enum([  gen.const(”Decimal", int_t, 0, ”Decimal format"),
                       gen.const(”Binary", int_t, 1, ” Binary format"),
                       gen.const(”Hexadecimal", int_t, 2, ”Hexadecimal format"),
                      ”Enum of formats")

gen.add(”fmt", int_t, 1, ”Format of count", 1, 0, 2, edit_method=fmt_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

Notice: the prefix value is the string used by catkin to name the corresponding header file. In our C++ code, we can then include it as “`prefixConfig.h`”

DYNAMIC RECONFIGURE

In our case, we can write the following to also terminate the configuration:

```
exit(gen.generate(PACKAGE, ”pub_sub", "parameters"))
```

DYNAMIC RECONFIGURE

We can now modify the C++ code of our publisher node

We add the include

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

Include the previously generated file

DYNAMIC RECONFIGURE

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

  ros::init(argc, argv, ”pub_sub");

  dynamic_reconfigure::Server<pub_sub::parametersConfig> dynServer;
```

Create the parameter server specifying the type of config

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

Create the callback

DYNAMIC RECONFIGURE

```
f = boost::bind(&param_callback, &mode, &fmt, _1, _2); _1, _2);
```

Bind the callback

Pass `mode` and `fmt` as pointers

```
dynServer.setCallback(f);
```

Set the server callback

DYNAMIC RECONFIGURE

```
void callback(bool *mode, int* fmt,
                pub_sub::parametersConfig &config, uint32_t level) {
```

Create the callback

Pointer to the parameters structure

Value of the level bitmask

The level bitmask can be used to check which parameter has changed

DYNAMIC RECONFIGURE

In the callback, we print the values of all the parameters and set the new mode and/or `fmt`

```
ROS_INFO("Reconfigure Request: %s %d - Level %d",
            config.mode?"True":"False",
            config.fmt,
            level);

*mode = config.mode;
*fmt = config.fmt;
```

DYNAMIC RECONFIGURE

We also have to edit the `CMakeLists.txt`,

Add to the find_package: `dynamic_reconfigure`

Add the `.cfg` file:

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

To make sure the header file is built before compiling our node, use (if not already there):

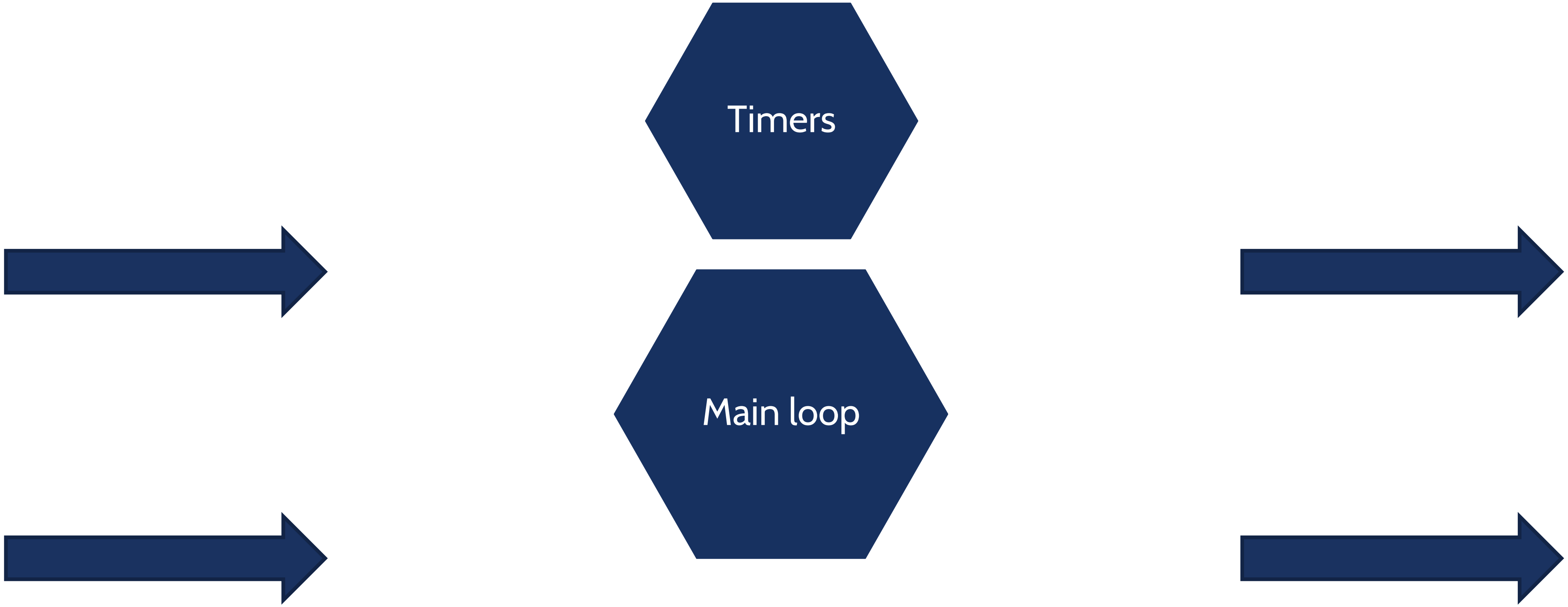
```
add_dependencies(pub ${catkin_EXPORTED_TARGETS})
```

TIMERS

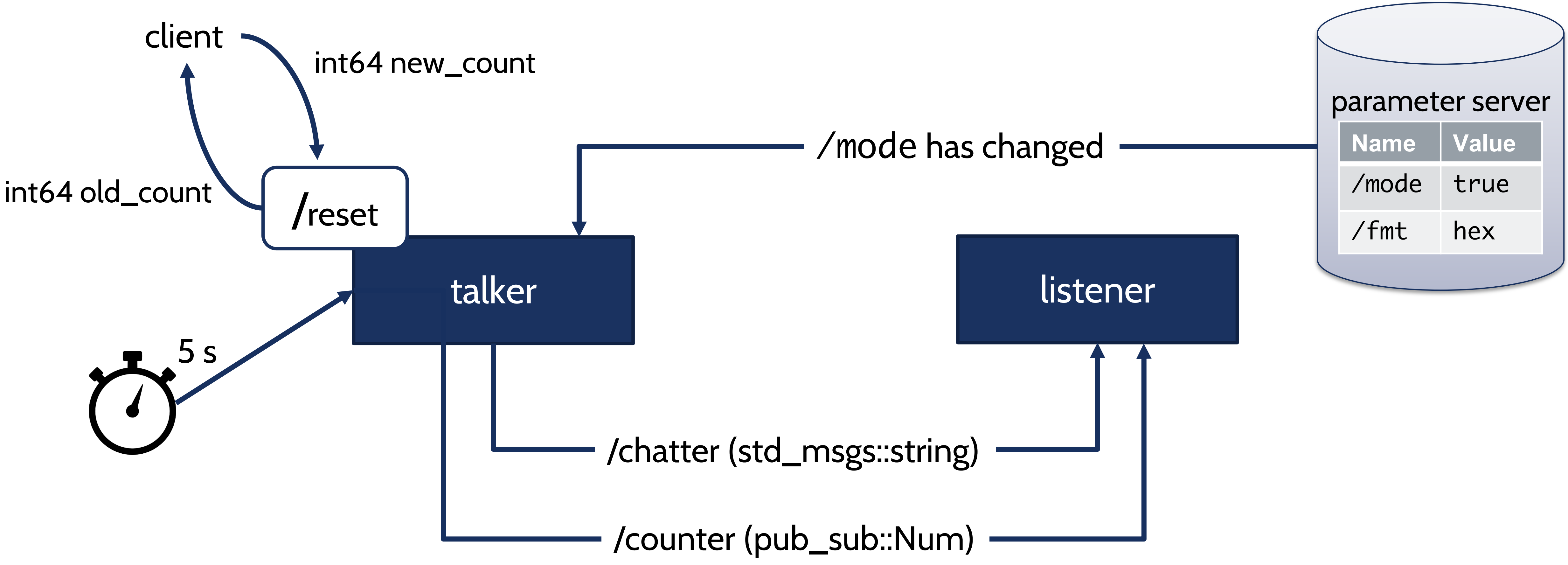
ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE



Objective: pub_sub_v6



TIMERS

We setup a callback which will be called when the timer expires (repeatedly or only once)

In main initialization

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

Timer duration

Timer callback

`true`: one-shot
`false`: repeat (default)

TIMERS

Timer callback

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

  ROS_INFO_STREAM("Publisher: timer callback called at time: " << ros::Time::now());

}
```

Print to terminal

Get current time

TIMERS

`CMakeLists.txt` and `package.xml` do not require any changes

CALLBACKS AND GOOD PRACTICES

ROBOTICS

POLITECNICO
MILANO 1863

CALLBACKS IN ROS

We have seen that ROS makes extensive use of callbacks to provide functionalities

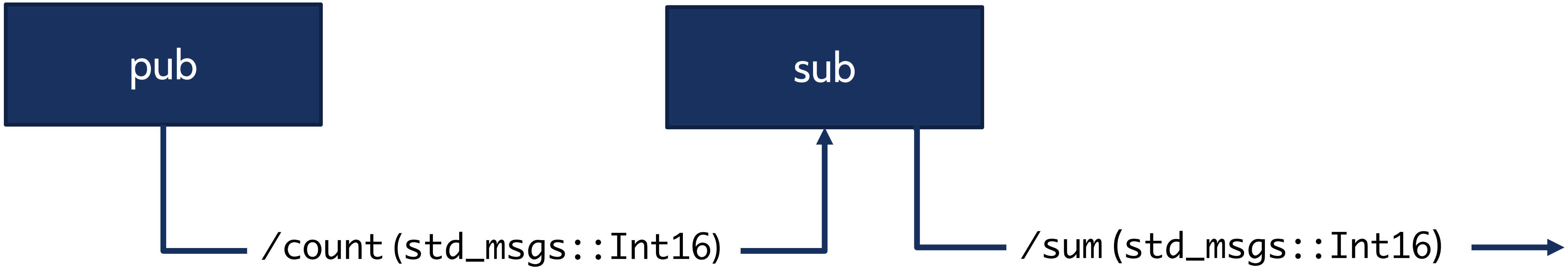
Most often, these callbacks are required to:

- Change the internal state of the node
  - E.g., subscriber callback updates internal variables
- Exploit variables defined in the main() function to provide functionalities
  - E.g., subscriber callback processes the received message and republishes the modified data on another topic

CALLBACKS IN ROS

Example:

- pub publishes an incremental integer on `/count`
- sub subscribes to `/count`, computes a cumulative sum of the numbers received and republishes the result in `/sum`



CALLBACKS IN ROS

Up to now, we would have some difficulties in implementing this cleanly. Indeed,

```
void countCallback(const std_msgs::Int32::ConstPtr& msg) {
  sum = sum + msg->data; // ERROR --> cannot see main::sum
}

int main(int argc, char **argv) {
  [...]
  int sum = 0;
  ros::Subscriber sub = n.subscribe("count", 1000, countCallback);
  [...]
}
```

CALLBACKS IN ROS

To allow callbacks to access variables defined externally (in the main() function), we would like to pass these variables as arguments.

E.g.,

```
void countCallback(int sum,
                   const std_msgs::Int32::ConstPtr& msg)
```

However, ROS calls these callbacks passing a predetermined set of arguments

E.g., for a subscriber, whenever a new message arrives, ROS calls the callback function with 1 single argument, representing the received message.

E.g., countCallback(<new message>); ← No sum argument passed by ROS!

CALLBACKS IN ROS: BOOST::BIND

To be able to pass additional arguments to callbacks, we can use the `boost::bind` function (part of the Boost library). Usage:

- Callback prototype:

```
void countCallback(int *sum,
                   const std_msgs::Int32::ConstPtr& msg)
```

- In `main()`:

Explicitly specify message type

```
ros::Subscriber sub =
    n.subscribe<std_msgs::Int32>(
        "count", 1000, boost::bind(&countCallback, &sum, _1)
    );
```

Callback using boost

CALLBACKS IN ROS: BOOST::BIND

General usage:

```
boost::bind(<ref to callback>, <list of args>)
```

`args` will be passed to our callback in the specified order.

`args` can be:

- A variable to be passed as "extra argument"
- A placeholder _n (e.g., `_1, _2`, ...), indicating the position of the n-th argument that will be passed by ROS

Notice: if you expect to modify the variables passed as argument, remember to pass them by reference (i.e., as pointers)!

CALLBACKS IN ROS: BOOST::BIND

Boost will internally create a function with prototype macthing what ROS expects; inside this function, it will call your callback, passing the extra arguments.

Notice: this is a simplification!

Eg. with a subscriber, boost creates an internal function like:

```
void boostInternalCallback(const std_msgs::Int32::ConstPtr& msg) {
     countCallback(sum, msg);
}
```

The pointer we passed to boost (boost saved it internally)

When a new message arrives, ROS will call the boost internal callback as

```
     boostInternalCallback( <new message> );
```

which, in turn, will call our callback with the extra argument!

CALLBACKS IN ROS: BOOST::BIND

Using `boost::bind` is the solution we have adopted so far.

It is effective and it requires a localized intervention in the code (no refactoring, just change the small parts of code)

However it could also be a bit cumbersome (think about having many extra arguments . . .)

CALLBACKS IN ROS: CLASSES

The ultimate answer to solve this problem (and write everything more cleanly) is to use a class for our entire node!

In general, it is a **good practice** to use classes for our nodes

CALLBACKS IN ROS: CLASSES

In `sub.cpp`, we create a class `Subscriber`:

```
class Subscriber {
  private:
    ros::NodeHandle n;
    ros::Subscriber sub;
    ros::Publisher pub;
    int sum;
```

Member variables, or **fields**

These variables can be seen from anywhere inside the class!
We declare as private everything we don't need to access from outside the class (typically all member variables)

CALLBACKS IN ROS: CLASSES

We add its **member functions**, or **methods:**

```
public:

  Subscriber() {

    this->sub = this->n.subscribe("count", 1000,
                                  &Subscriber::countCallback, this);

    this->pub = this->n.advertise<std_msgs::Int32>("sum", 1000);

    this->sum = 0;

  }
```

We declare as public every member variable or function that needs to be accessed from outside the class.
The constructor must be public.

Constructor
Every initialization goes here

`this->` operator should be used to access every member variables or functions from within the class

We must pass `this` as last argument to any ROS function expecting a callback

CALLBACKS IN ROS: CLASSES

```
public:
```

(No need to repeated it in our code)

```
  void main_loop() {
    ros::Rate loop_rate(10);
    while (ros::ok()) {
      ROS_INFO("Current sum: %d", this->sum);
      ros::spinOnce();
      loop_rate.sleep();
    }
  }
```

Main loop function

It's just a regular function, which we will call after initialization, when we want the main loop to start executing

CALLBACKS IN ROS: CLASSES

```
public:
```

(No need to repeated it in our code)

```
  void countCallback(const std_msgs::Int32::ConstPtr& msg) {
    ROS_INFO("Received: %d", msg->data);
    this->sum = this->sum + msg->data;

    std_msgs::Int32 sum_msg;
    sum_msg.data = this->sum;
    this->pub.publish(sum_msg);
  }
```

Callback function
No need for extra arguments as it can already see every member variables inside the class

```
};
```

End of class definition

CALLBACKS IN ROS: CLASSES

We can add our main function, where the execution will start:

```
int main(int argc, char **argv) {
  ros::init(argc, argv, "callbacks_sub");
  Subscriber my_subscriber;
  my_subscriber.main_loop();
  return 0;
}
```

Call to `ros::init` at beginning of execution

Create an instance (object) of `Subscriber` class. The constructor is internally called.

Start the main loop

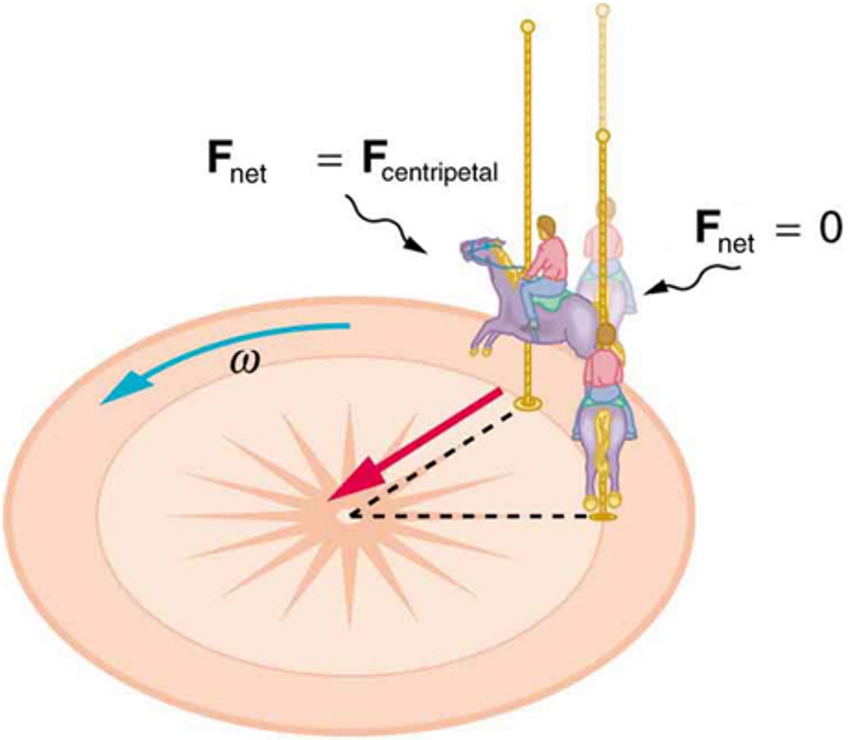
TF

ROBOTICS

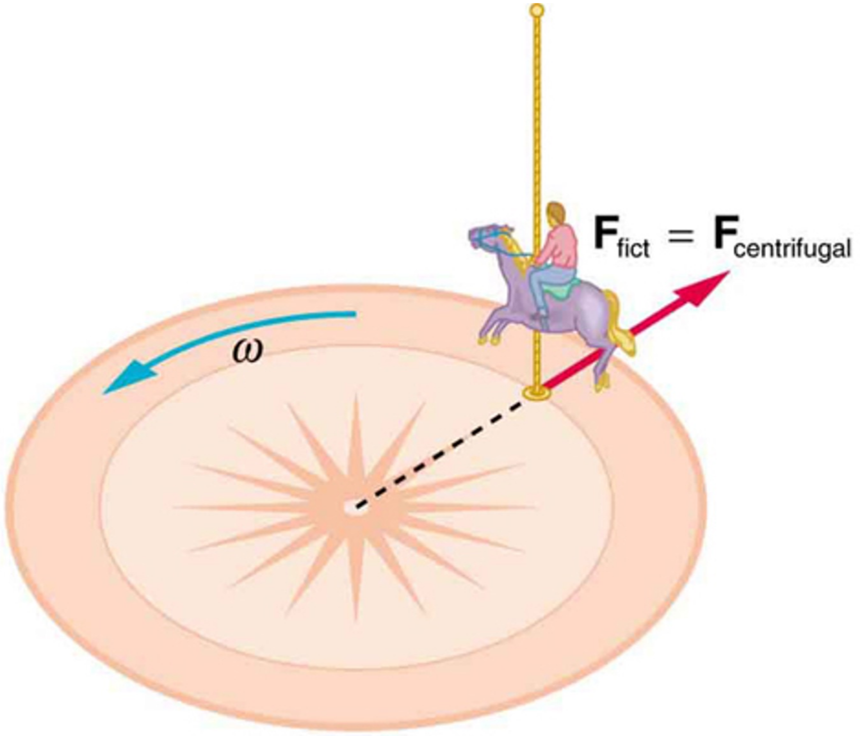
POLITECNICO
MILANO 1863

REFERENCE SYSTEMS: IN PHYSICS

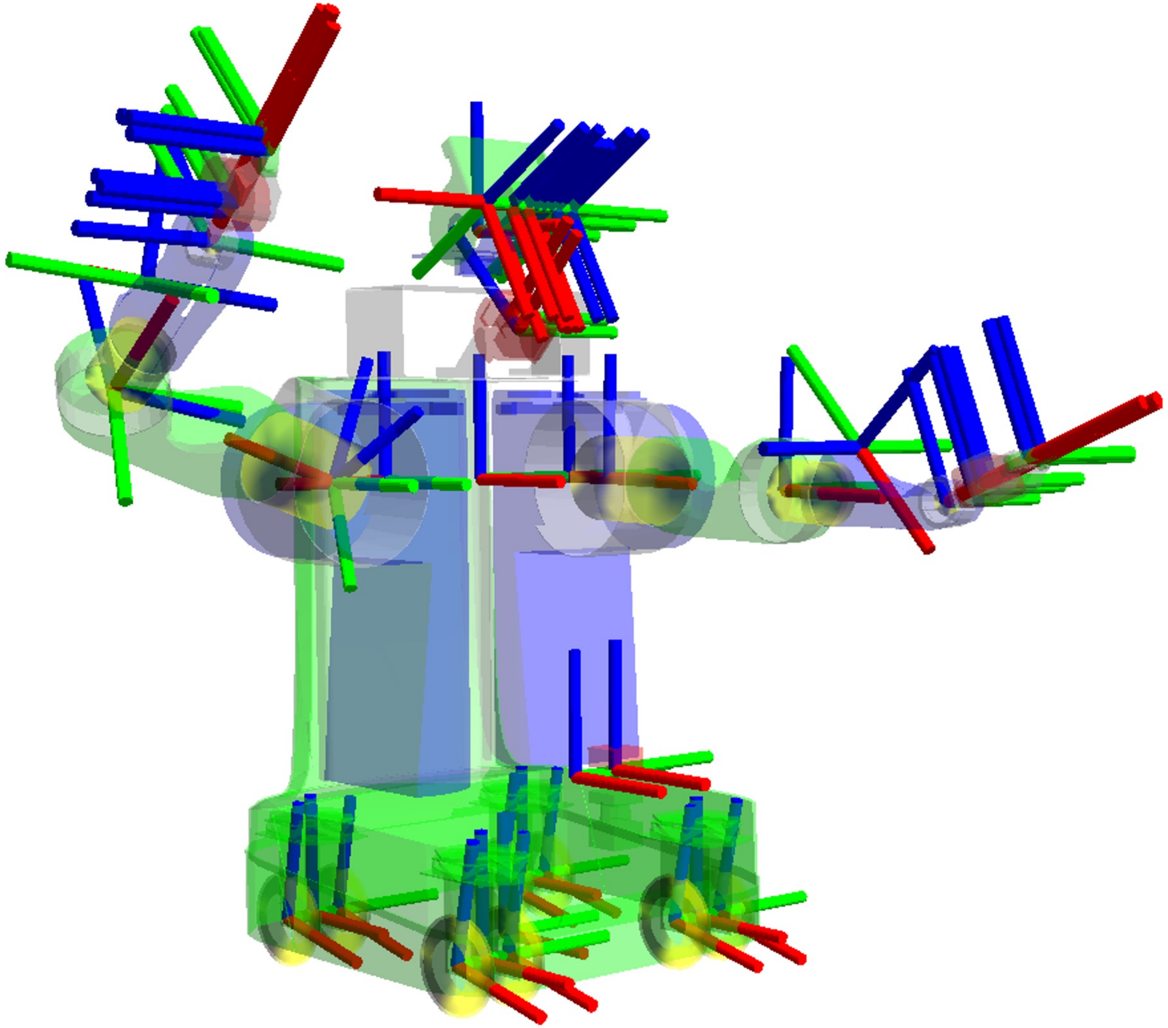
Reference System is everything



VS



REFERENCE SYSTEMS: 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

How is it possible to convert form one frame to another?

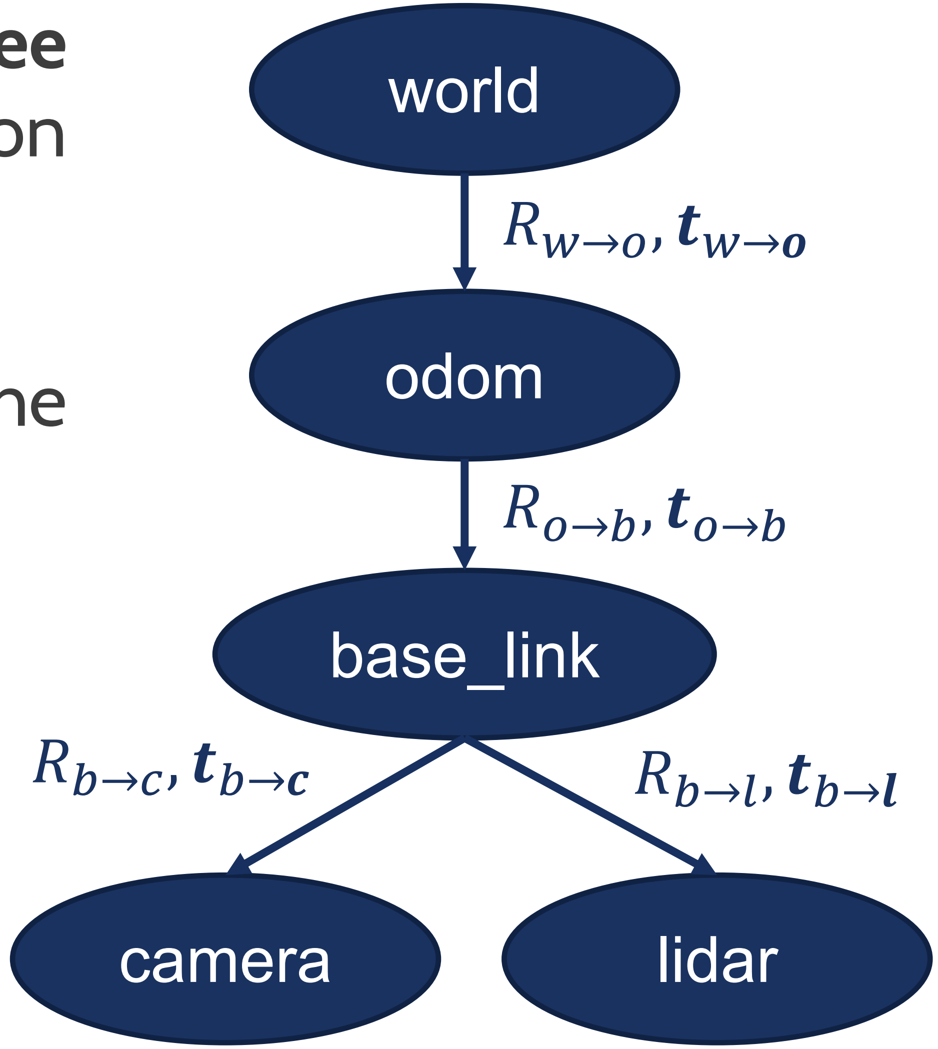
Math, lot of it.

IN ROBOTICS

To simplify things, frames are described in a **tree**
Each frame comes with a transformation between itself and its father

The **world frame** is the most important, but the others are used for simplicity

In a tree of reference frames, we define a roto-translation between parent and child
To compute the transformation between two frames, we can simply combine the roto-translation found trasversing the tree



TF: TRANSFORMATION FRAMES

If we specify the transformation tree, ROS does all the hard work for us, thanks to **TF**!

We can do interpolation, transformation, tracking, …

TF is a ROS tool that:

- keeps track of all the dynamic transformation for a limited period of time
- is decentralized
- provides the position of a point in each possible reference frame

COMMON FRAMES

For each transformation we specify in our transformation tree, we have to specify

- the name of the father frame
- the name of the child frame
- the rototranslation between the two

Frame names can be arbitrary, but some particular frames are usually named following a convention:

- world → the root frame, fixed
- map → used in navigation, mostly fixed, but can be realigned over time
- odom → frame in which we express the odometry of the vehicle
- base_link → the main frame of the robot, tipically in its center of gravity

TF TREE TOOLS

ROS offers different tools to analyze the transformation tree:

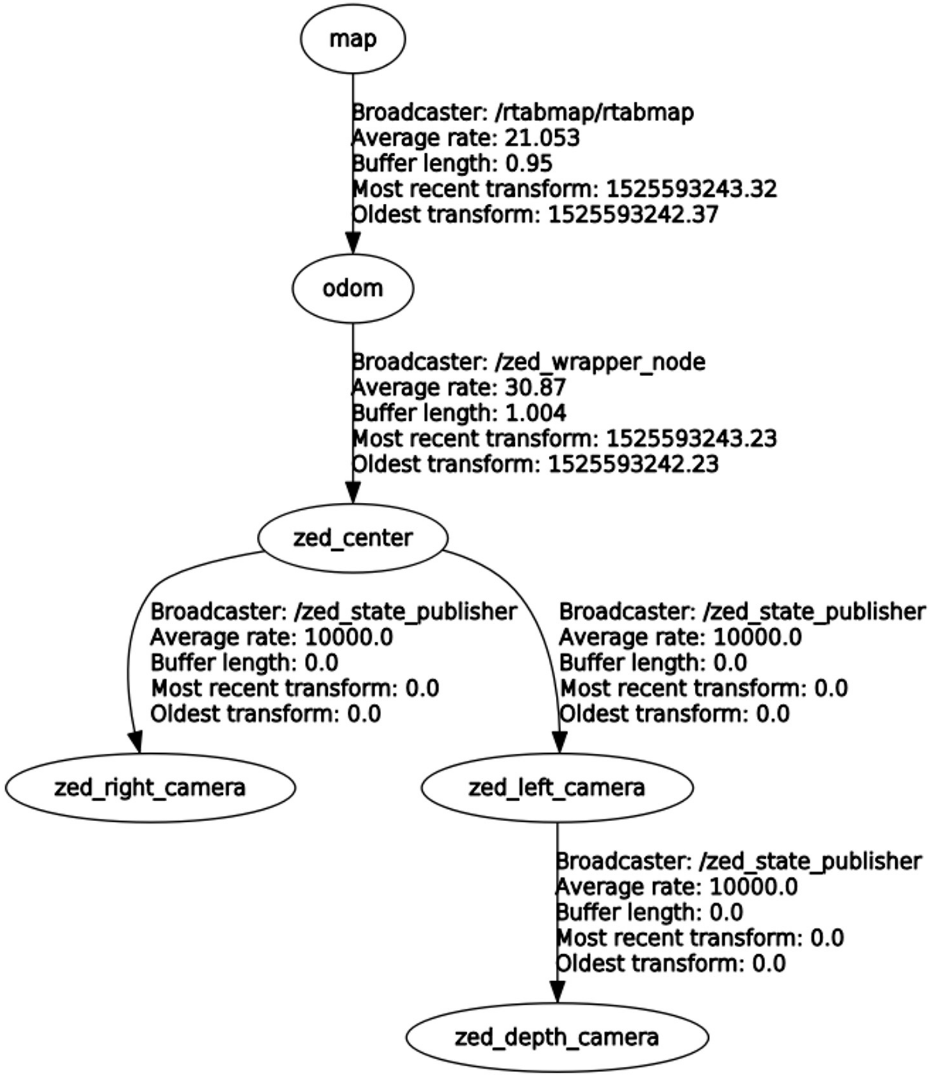
- `rosrun rqt_tf_tree rqt_tf_tree`

  visualizes the tf tree at the current time

- `rosrun tf view_frames`

  listens for 5 seconds to the `/tf` topic and creates a pdf file with the tf tree

HOW TF_TREE LOOKS LIKE IN PRACTICE



TF AND TF2

ROS originally shipped with TF

Then several bug fixed, to the point that a new version of TF was created: TF2

Some legacy packages still require TF, otherwise always use TF2

In class, we will only see TF2.

At home, you can find TF example in our package as well.

WRITING A TF BROADCASTER

Now that we got an idea of how tf works and why it’s useful, we can take a look at how to specify our TF tree

We do so using a **tf broadcaster**

Since we do not have a physical robot here,
we will use turtlesim for our example

WRITING A TF BROADCASTER

Turtlesim provides us with a topic indicating its pose /turtlesim/pose

(we can see it as a very precise odometry)

We will use this topic to broadcast a **(dynamic) transformation**

`world` → `turtle`, which will change as the turtle moves around

We can also add a **static transformations** from turtle to each of its 4 legs:

`turtle → FRleg, turtle → FLleg, turtle → BRleg, turtle → BRleg`

WRITING A TF BROADCASTER

We create a package called `tf_examples` inside the `src` folder of our catkin environment, adding dependencies `roscpp std_msgs tf2 tf2_ros turtlesim`

```
$ catkin_create_pkg tf_examples roscpp std_msgs tf2 tf2_ros
turtlesim
```

Then we can create a `broadcaster_tf2.cpp`

WRITING A TF BROADCASTER

In `broadcaster_tf2.cpp`:

Includes:

```
#include "ros/ros.h"
#include "turtlesim/Pose.h"
#include <tf2/LinearMath/Quaternion.h>
#include <tf2_ros/transform_broadcaster.h>
#include <geometry_msgs/TransformStamped.h>
```

WRITING A TF BROADCASTER

Now we have to create our class:

```
class TfBroad {
public:
      TFBroad() {…}
      void callback(const turtlesim::Pose::ConstPtr& msg){…}
private:
      ros::NodeHandle n;
      tf2_ros::TransformBroadcaster br;
      geometry_msgs::TransformStamped transformStamped;
      ros::Subscriber sub;
};
```

WRITING A TF BROADCASTER

In class constructor, we subscribe to `/turtle1/pose`

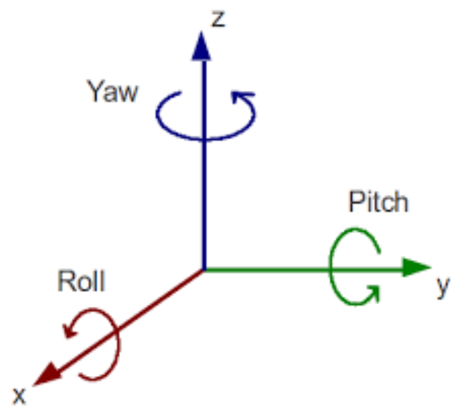
```
sub = n.subscribe("/turtle1/pose", 1000, &TfBroad::callback, this);
```

WRITING A TF BROADCASTER

Inside the callback (every time we get a new pose), we populate a transformation object using the data from the pose message (we are in a 2D environment)

Notice: TF uses quaternions for rotations, but it provides tools to specify them also through roll, pitch and yaw!

Finally, we broadcast the transformation



WRITING A TF BROADCASTER

```
void callback(const turtlesim::Pose::ConstPtr& msg) {
  transformStamped.header.stamp = ros::Time::now();
  transformStamped.header.frame_id = "world";
  transformStamped.child_frame_id = "turtle";

  transformStamped.transform.translation.x = msg->x;
  transformStamped.transform.translation.y = msg->y;
  transformStamped.transform.translation.z = 0.0;

  tf2::Quaternion q;
  q.setRPY(0, 0, msg->theta);
  transformStamped.transform.rotation.x = q.x();
  transformStamped.transform.rotation.y = q.y();
  transformStamped.transform.rotation.z = q.z();
  transformStamped.transform.rotation.w = q.w();

  br.sendTransform(transformStamped);
}
```

Setup the **header**, containing timestamp and parent/child frames

Translation

Rotation
Setup a quaternion using ROS functions to convert from roll pitch yaw

Broadcast the transform

WRITING A TF BROADCASTER

Then we write the main function:

```
int main(int argc, char **argv) {
  ros::init(argc, argv, "tf_broadcast");
  TfBroad my_tf_broadcaster;
  ros::spin();
  return 0;
}
```

WRITING A TF BROADCASTER

As usual, we have to add this new executable to the CMakeLists.

Package dependencies (already specified during the package creation):

```
find_package(catkin REQUIRED COMPONENTS roscpp std_msgs tf2 tf2_ros turtlesim)
```

Add the executable:

```
add_executable(tf2_broad src/broadcaster_tf2.cpp)
add_dependencies(tf2_broad ${catkin_EXPORTED_TARGETS})
target_link_libraries(tf2_broad ${catkin_LIBRARIES})
```

WRITING A TF BROADCASTER

We can setup a launchfile to test our node

We add a `turtlesim_node`, a `turtle_teleop_key`, and our broadcaster node:

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

  <node pkg="tf_examples" type = "tf2_broad" name = "tf_broad"/>
</launch>
```

ADD STATIC TF

Now we can add the transformations for the 4 legs of the turtle

We could use the same mechanism seen so far, adding broadcasters to our node.

However, the transformation between legs and turtle frame will always be fixed (the turtle robot is a rigid object).

For static (fixed) transformations, ROS helps us with the node `static_transform_publisher`, which we can run directly from our launchfile!

ADD STATIC TF

We add the 4 static transform, specifying as args:

- position (x,y,z) and rotation (as a quaternion qx,qy,qz,qw) of the robot
- parent frame
- child frame

```
<node pkg="tf2_ros" type="static_transform_publisher" name="back_right" args="0.3 -0.3 0 0 0 0 1 turtle FRleg " />
<node pkg="tf2_ros" type="static_transform_publisher" name="front_right" args="0.3 0.3 0 0 0 0 1 turtle FLleg " />
<node pkg="tf2_ros" type="static_transform_publisher" name="front_left" args="-0.3 0.3 0 0 0 0 1 turtle BLleg " />
<node pkg="tf2_ros" type="static_transform_publisher" name="back_left" args="-0.3 -0.3 0 0 0 0 1 turtle BRleg " />
```

ADD STATIC TF

We can visualize our tf tree running `rqt_tf_tree`

We can use rviz to visualize the motion of the turtle

We can also see all the published tf using `rostopic echo`

Or we can see specific tf transforms with `tf_echo`:

```
$ rosrun tf tf_echo father child
```

```
$ rosrun tf tf_echo \world \FRleg
```

ROS VISUALIZATION: RVIZ

ROBOTICS

POLITECNICO
MILANO 1863

ODOMETRY

Odometry messages contain estimated information about the position and velocity of the robot in space.

`nav_msgs/Odometry` definition:

```
std_msgs/Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

We can visualize certain messages using rviz, a visualization tool provided by ROS

ODOMETRY ON RVIZ

You can use rviz to display Odometry messages as arrows.
The location of the arrows represents the robot position,
The orientation of the arrow represents the current direction of movement

Type in the terminal:

```
$ rviz
```

In rviz, press on “Add“ in the “Display” panel (bottom left)

- Add an Odometry message and specify the topic name
- You can also select “By Topic” and see directly which topics are available

You can change the visualization properties from the “Display” panel (left)
Change the parameter “Keep” to display or not display past poses

ROS PLOTTING TOOLS

You can plot informations in ROS using `rqt_plot` (basic)

```
rosrun rqt_plot rqt_plot
```

Or using plotjuggler (more advanced)

```
rosrun plotjuggler plotjuggler
```