ROS core components

ROBOTICS

POLITECNICO
MILANO 1863

Today schedule

- Launch files
- Custom messages
- Services
- Timers
- Parameters/dyn reconfigure

ROS on Docker

You should have compiled the docker image for robotics (L1/L2)

To start the image:

- **Docker compose:** `docker-compose -f robotics.yaml up`
- **Docker run:** `docker run --net=ros --env="DISPLAY=novnc:0.0"  -it -v  C:\Users\Simone\Documents\Robotics:/home/simone/robotics robotics /bin/bash`
- In both scenario you should properly set the mounting options (under volumes in docker compose, -v option in docker run)
- **Connect to running container:** `docker exec -it --user your_username robotics_app /bin/bash`

ROS on Docker (GUI)

- You have to setup the network (only once): `docker network create ros`
- You start the vnc server docker image (provided script or command line):
  - `./vnc.sh`
  - `docker run -d --rm --net=ros --env="DISPLAY_WIDTH=1920" --env="DISPLAY_HEIGHT=1080" --env="RUN_XTERM=no" --name=novnc -p=8080:8080 theasp/novnc:latest`
- Go to http://localhost:8080/vnc.html to see the desktop

TMUX Commands recap (if you work from terminal)

- tmux new -s session_name	create a new session
- ctrl+b -> %	split vertically
- ctrl+b -> “	split horizontally
- ctrl+b -> arrow	move to a different terminal
- ctrl+b -> d	exit session
- tmux a	attach to last session
- tmux a -t session_name	attach to session
- tmux kill-session -t session_name	kill session

LAUNCH FILE

ROBOTICS

POLITECNICO
MILANO 1863

LAUNCH FILE

When working on big project it's useful to create a launch file which with only one command will:

- start roscore
- start all the nodes of the project together
- set all the specified parameters

To create a launch file cd to the pub_sub package and create a launch folder

$ mkdir launch

LAUNCH FILE

Inside the launch folder create a launcher.launch file

the launch file is an XML file, the root tags are

<launch></launch>

inside these tags you can start all your nodes using:

```
<node pkg="package" type="file_name" name="node_name"/>
```

when we started a node from the command line we used:

$ rosrun package file_name

the name attribute allow us to specify inside the launch file the name of the node

LAUNCH FILE

We can also regroup some nodes under a specific namespace using the tags:

```
<group ns="turtlesim1"></group >
```

Namespaces allow us to start multiple node with the same name, because they lives in different namespace

Sometimes we may need to change some topics name without changing directly the package code, to accomplish this task we use:

```
<remap from="original" to="new"/>
```

LAUNCH FILE

Inside the launch file paste this code:

```
<launch>

  <group ns="turtlesim1">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <group ns="turtlesim2">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>

</launch>
```

LAUNCH FILE

This code starts two turtlesim and connect them together, the command from cmd vel to turtlesim1 will be redirected also to turtlesim2

But we still have to run in a new terminal window the teleop_key node

So we also have to add

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

inside the turtlesim1 namespace

If we want to open a node in a new terminal we can add the attribute:

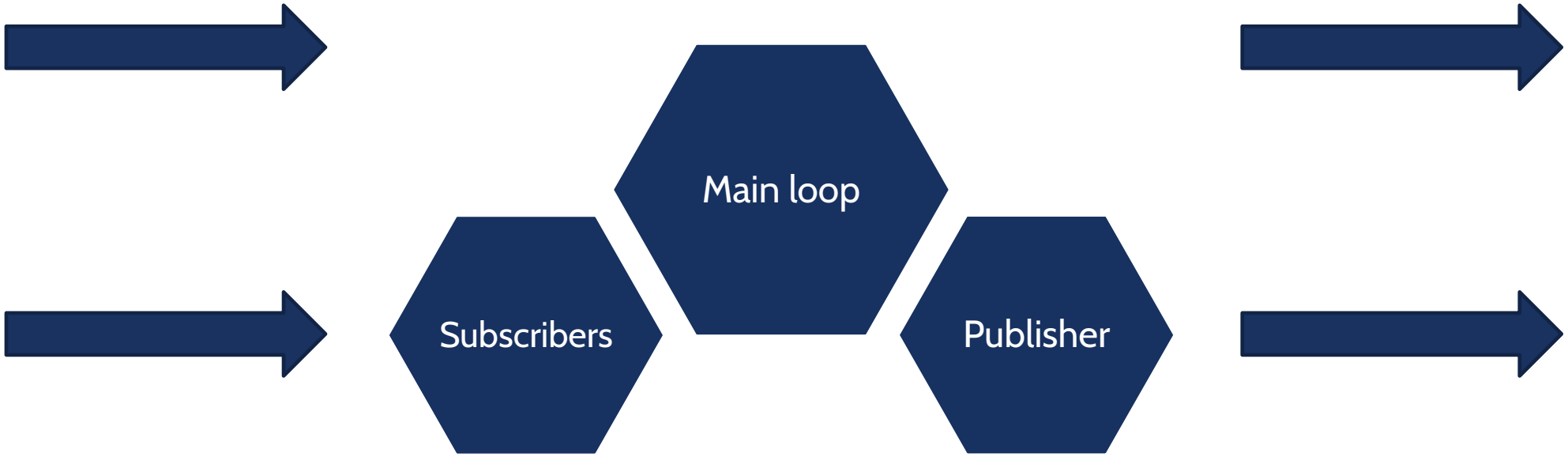
```
launch-prefix="xterm -e"
```

CUSTOM MESSAGES

ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE



CREATING A MESSAGE

Custom messages definitions must be created in the msg folder of our package.

First, we create the folder inside the pub_sub package:

mkdir msg

Next, we create the msg file:

echo "int64 num" > msg/Num.msg

CREATING A MESSAGE

Before using the new message, we must specify that they must be converted into source code for C++

We open the package.xml file and uncomment the following lines:

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

CREATING A MESSAGE

Next, we edit the CMakeLists.txt file to build the custom messages together with the package

We first specify the dependency to message_generation in find_package

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
  )
```

CREATING A MESSAGE

Then, we export the message_runtime dependency, uncommenting the corresponding line and adding message_runtime:

```
catkin_package(
   CATKIN_DEPENDS message_runtime
 )
```

We also must specify that the publisher package depends on the custom message

```
add_dependencies(publisher pub_sub_generate_messages_cpp)
```

CREATING A MESSAGE

Lastly, we specify the custom message definition: uncomment the following lines and add the path to the custom msg file (Num.msg) and its dependencies:

```
add_message_files(
 FILES
 Num.msg
 )
```

```
generate_messages(
   DEPENDENCIES
  std_msgs
 )
```

CREATING A MESSAGE

Now we can compile our code calling catkin_make in the root directory of our workspace

We can test if ROS finds our new message by calling:

rosmsg show pub_sub/Num

USING CUSTOM MESSAGES

To test our new message type, we modify the publisher-subscriber nodes

We first open the pub.cpp file

We include the custom message, adding:

```
#include "pub_sub/Num.h"
```

Then, we modify the publisher object, changing the message type:

```
ros::Publisher chatter_pub = n.advertise<pub_sub::Num>("publisher", 1000);
```

USING CUSTOM MESSAGES

Lastly, we modify the message creation. In particular, we create a message of type pub_sub::Num and assign a number to the num field:

```
static int i=0;
i=(i+1)%1000;
pub_sub::Num msg;
msg.num =i;
```

Now we can build our package and look at the published topic using:

```
$ rostopic echo /publisher
```

USING CUSTOM MESSAGES

The changes to the sub.cpp file are similar:

First, include the new message type

```
#include "pub_sub/Num.h"
```

Then change the type of the message received by the callback:

void pubCallback(const pub_sub::Num::ConstPtr& msg)

USING CUSTOM MESSAGES

Last update the print function:

```
ROS_INFO("I heard: [%d]", msg->num);
```

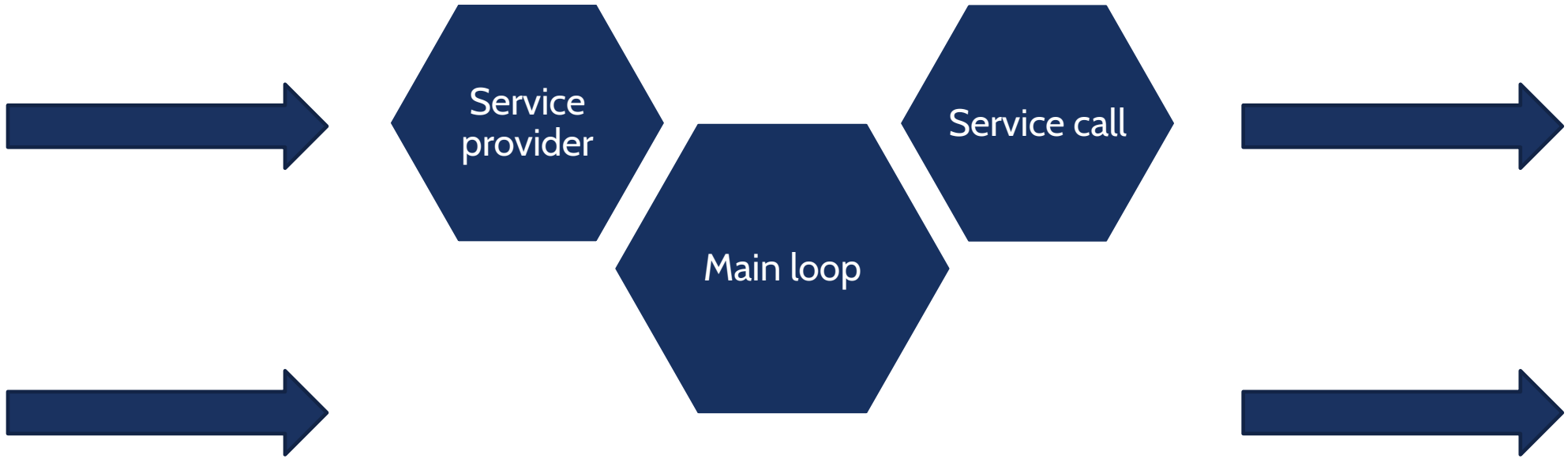
Now we can build and test both the publisher and the subscriber

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:

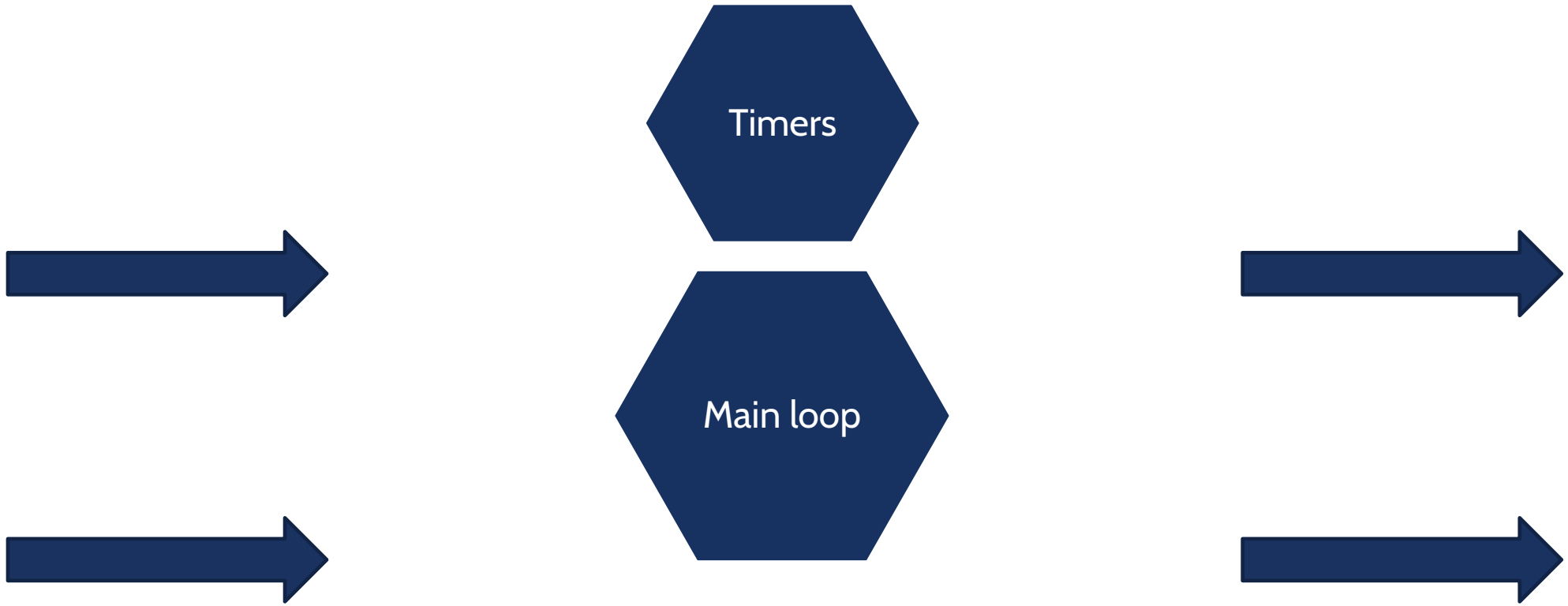
```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

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

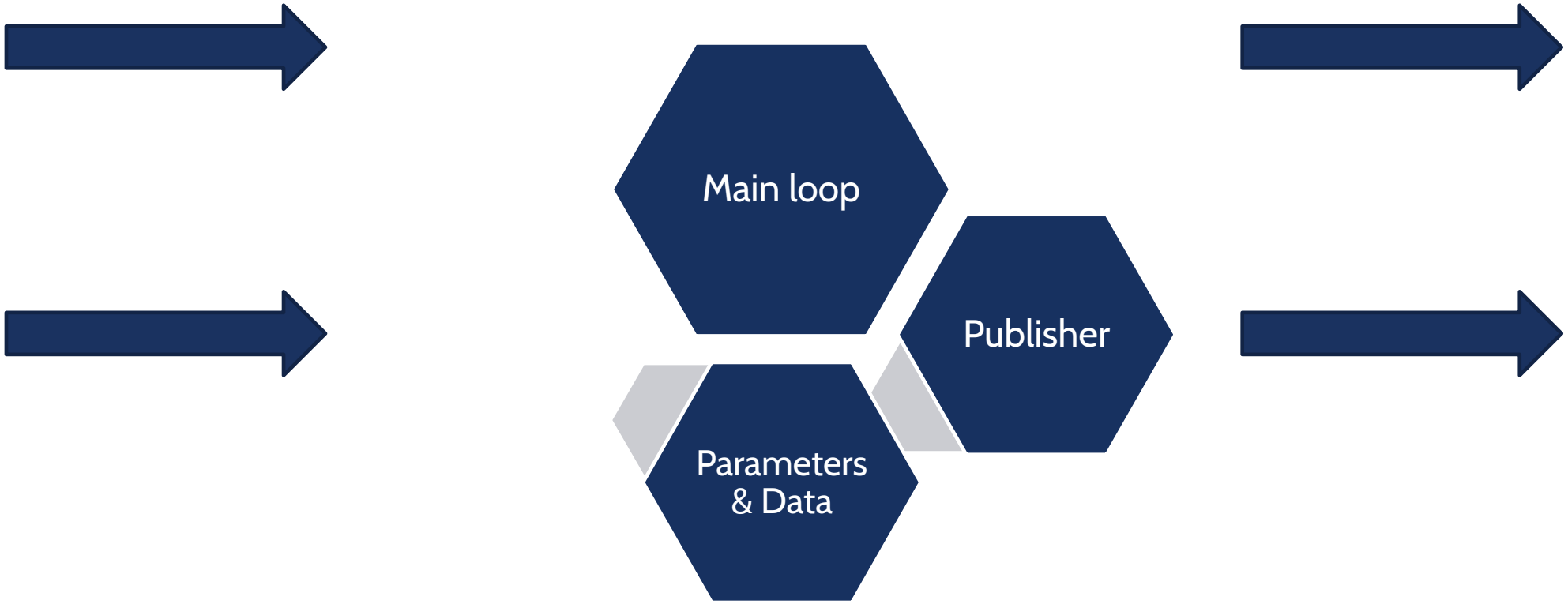
Both CMakeLists.txt and Package.xml don’t require particular changes from the pub/sub example to work with timers

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:

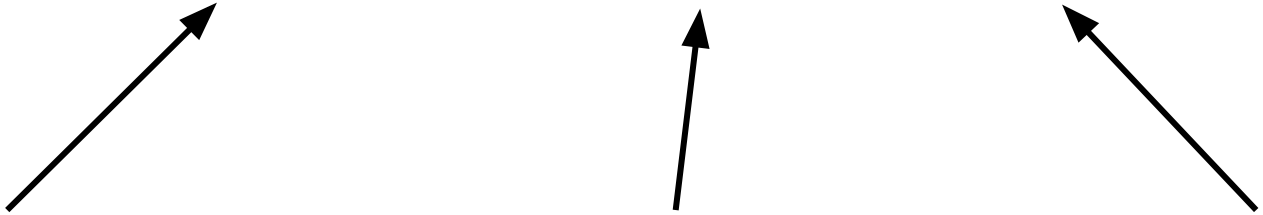
```
size_enum = gen.enum([ gen.const("Small",      int_t, 0, "A small constant"),
                       gen.const("Medium",     int_t, 1, "A medium constant"),
                       gen.const("Large",      int_t, 2, "A large constant"),
                       gen.const("ExtraLarge", int_t, 3, "An extra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



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");

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

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

ROS initialization

Create the parameter server specifying the type of config

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