First ROS Node

ROBOTICS

POLITECNICO
MILANO 1863

Today schedule

- Workspaces, code organization, compiling
- Create a ROS package
- Writing the first node
- Adding useful components to the node

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

ROS CODE ORGANIZATION

ROBOTICS

POLITECNICO
MILANO 1863

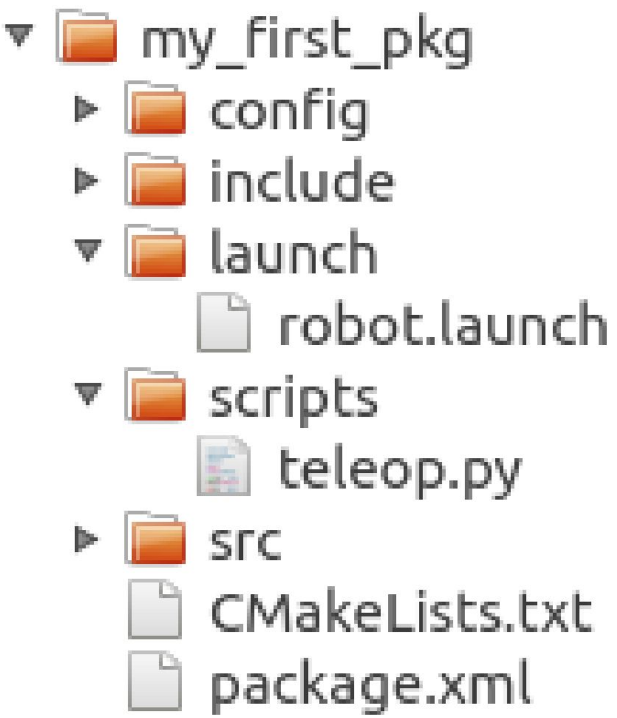
ROS Packages

Software in ROS is organized in **packages**.

Packages are the most **atomic unit of build and** the **unit of release**, i.e., a package is the smallest individual thing you can build in ROS and it is the way software is bundled for release.

They provide useful functionality in an easy-to-consume manner so that software can be easily **reused**.

A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module.



ROS WORKSPACE

A **workspace** is a folder where you modify, build, and install ROS packages.

Required by ROS build system: catkin

```
workspace_folder/
 ├── build/
 ├── devel/
 └── src/
      ├── package1/
      |    ├── CMakeLists.txt
      |    ├── package.xml
      |    ├── config/
      |    ├── launch/
      |    ├── include/
      |    ├── src/
      |    └── scripts/
      |
      ...
      └── packageN/
```

BUILDING YOUR CODE

ROBOTICS

POLITECNICO
MILANO 1863

BUILD SYSTEMS

After implementing some code, we will need to compile it.

Calling the compiler manually

e.g.gcc main.cpp function.cpp -o run

is complicated and time-consuming in non-trivial projects.

BUILD SYSTEMS

After implementing some code, we will need to compile it.

Calling the compiler manually

e.g.gcc main.cpp function.cpp -o run

is complicated and time-consuming in non-trivial projects.

Build systems (or build tools) take care of:

- Locating the source code
- Locating external libraries
- Managing code dependencies
- Dealing with the compiler

...

Popular build systems: CMake, GNU Make, Apache Ant (Java), Gradle, ...

CATKIN

ROS is a very large collection of loosely federated packages,

i.e., lots of independent packages which depend on each other and use:

- various programming languages (C++, Python),
- various programming tools,
- various code organization conventions.

Potentially very different building requirements!

CATKIN

ROS is a very large collection of loosely federated packages,

i.e., lots of independent packages which depend on each other and use:

- various programming languages (C++, Python),
- various programming tools,
- various code organization conventions.

Potentially very different building requirements!

ROS relies on a custom build system: **catkin**

catkin specifies a standard for building ROS packages

it becomes easier to use and share ROS code

CMake

catkin is based on CMake, a very popular build system for C++

We will not study CMake in details, and it will not be required for this course

Iff personally interested, you can learn more about it on:

- official website: https://cmake.org/
- official tutorial: https://cmake.org/cmake/help/latest/guide/tutorial/index.html
- Many resources online, especially
  - Modern CMake: https://cliutils.gitlab.io/modern-cmake/
  - More Modern CMake: https://hsf-training.github.io/hsf-training-cmake-webpage/index.html

WORKSPACE STRUCTURE

catkin (ROS) workspace:

```
workspace_folder/
├── build/
├── devel/
└── src/
    ├── package1/
    |   ├── CMakeLists.txt
    |   ├── package.xml
    |   ├── include/
    |   └── src/
    |
    ...
    └─── packageN/
```

Automatically managed by catkin

Building and meta information

WORKSPACE STRUCTURE

Source space (/src):

It contains the source code of the ROS packages you want to add to your system

Everything we do goes here!

Build space (/build):

Where CMake is used to build the catkin packages

CMake and catkin keep their cache information and other intermediate files here

Devel space (/devel):

Space where built targets are placed prior to being installed

We will never touch these

PACKAGE.XML

It contains **metadata** about the package and its dependencies

Basic version generated with catkin_create_pkg
Requires editing for additional functionalities (we will see them)

Example:

```
<?xml version="1.0"?>
<package format="2">
  <name>package_name</name>
  <version>0.0.0</version>
  <description>The package_npackage</description>
  <maintainer email=”user@todo.todo">username</maintainer>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_export_depend>roscpp</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <exec_depend>roscpp</exec_depend>
  <exec_depend>std_msgs</exec_depend>
</package>
```

CMAKELISTS.TXT

Responsible for preparing and executing the **build process**

Concept coming from CMake

Easy to configure, as catkin does most of the work

Basic example:
(continued on next slide)

CMAKELISTS.TXT

```
cmake_minimum_required(VERSION 2.8.3)
project(package_name)
find_package(catkin REQUIRED COMPONENTS roscpp std_msgs genmsg)
add_message_files(FILES custom_message.msg)
add_service_files(FILES custom_service.srv)
generate_messages(DEPENDENCIES std_msgs)
catkin_package()

include_directories(include ${catkin_INCLUDE_DIRS})
add_executable(executable_name src/source_code.cpp)
target_link_libraries(executable_name ${catkin_LIBRARIES})
add_dependencies(executable_name package_name_generate_messages_cpp)
```

CMAKELISTS.TXT

```
cmake_minimum_required(VERSION 2.8.3)
project(package_name)
find_package(catkin REQUIRED COMPONENTS roscpp std_msgs genmsg)
add_message_files(FILES custom_message.msg)
add_service_files(FILES custom_service.srv)
generate_messages(DEPENDENCIES std_msgs)
catkin_package()

include_directories(include ${catkin_INCLUDE_DIRS})
add_executable(executable_name src/source_code.cpp)
target_link_libraries(executable_name ${catkin_LIBRARIES})
add_dependencies(executable_name package_name_generate_messages_cpp)
```

This is what you have to change every time

CMAKELISTS.TXT

```
cmake_minimum_required(VERSION 2.8.3)
project(package_name)
find_package(catkin REQUIRED COMPONENTS roscpp std_msgs genmsg)
add_message_files(FILES custom_message.msg)
add_service_files(FILES custom_service.srv)
generate_messages(DEPENDENCIES std_msgs)
catkin_package()

include_directories(include ${catkin_INCLUDE_DIRS})
add_executable(executable_name src/source_code.cpp)
target_link_libraries(executable_name ${catkin_LIBRARIES})
add_dependencies(executable_name package_name_generate_messages_cpp)
```

This is needed only if you have custom messages

CREATING OUR WORKSPACE

ROBOTICS

POLITECNICO
MILANO 1863

WORKSPACE CREATION

We create a folder named “robotics” in our /home and initialize it as a ROS workspace
This will be our ROS workspace for the entire course

```
mkdir -p ~/robotics/src
cd ~/robotics/
catkin_make
```

To tell ROS where your workspace is, open the file ~/.bashrc with a text editor and paste the following on a new line (if not already present):

```
source ~/robotics/devel/setup.bash
```

If already present, do not replicate this line!

Then run in your terminal:

```
source ~/.bashrc
```

CREATING OUR FIRST PACKAGE

ROBOTICS

POLITECNICO
MILANO 1863

PACKAGE CREATION

Command to create a new package

catkin_create_pkg [package_name] [dependency_1] [...] [dependency_n]

Before running the script, cd to your src directory (robotics/src).

Then run:

catkin_create_pkg pub_sub_test std_msgs rospy roscpp

roscpp and rospy are package dependencies required to use C++ and Python respectively

std_msgs is required to use standard message types

Notice: before creating a package, you must have a ROS workspace!

PACKAGE CREATION

The script should have created the following structure:

- robotics/
  - src/
    - pub_sub_test/
      - CMakeLists.txt
      - package.xml
      - include/
      - src/

To build the new package, cd to the ROS workspace and run catkin_make

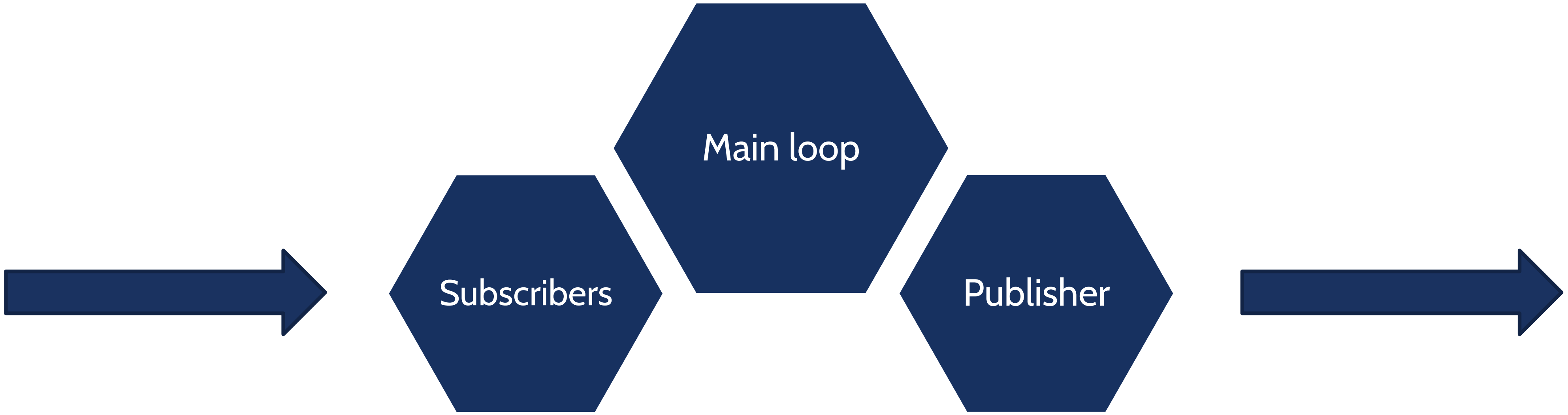
OUR FIRST NODES:
PUBLISHER-SUBSCRIBER

ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE

INSIDE THE NODE: PUB-SUB



INITIALIZATION

Any node must be registered to the ROS master using a unique identifier

The actual node is initialized using a handler

Each executable has a **unique name**

Each executable may have multiple handlers

```
void ros::init(argv, argc, std::string node_name, uint32_t options);
ros::init(argc, argv, "my_node_name");
ros::init(argc, argv, "my_node_name", ros::init_options::AnonymousName);

ros::NodeHandle nh;
```

MAIN LOOP

Each ROS node loops waiting for something to do

At each loop it checks:

- is there a message waiting to be received?
- is there a completed timer?
- is there a parameter to be reconfigured?

Two ways to implement the main loop:

- Automatically, no developer intervention
- Manual, specific sleep time and execution at each loop

```
ros::spin();
```

```
ros::Rate r(10); //10 hz
while (ros::ok()) {
  /* some execution */
  ros::spinOnce();
  r.sleep();
}
```

PUBLISHER

Used to publish messages on a ROS topic

On declaration, connect the publisher to a topic and define the type of the message

Can be called from anywhere

The frequency of the messages are not set

```
ros::Publisher pub = nh.advertise<std_msgs::String>("topic_name", 5);
std_msgs::String str;
str.data = "hello world";
pub.publish(str);
```

SUBSCRIBER

Used to read messages from a ROS topic

On declaration, connect the subscriber to a topic and define the type of the message

Call a specific function when receive a message

Operate at a given frequency

```
ros::Subscriber sub = nh.subscribe("topic_name", 10, callback);

void [class::]callback(const pack_name::msg_type::ConstPtr& msg)
```

WRITING A PUBLISHER NODE

We first create a package inside our workspace src folder:

```
catkin_create_pkg pub_sub std_msgs rospy roscpp
```

Next, we cd to the new pub_sub/src folder and create a C++ file:

```
gedit pub.cpp
```

WRITING A PUBLISHER NODE

First, we write some includes:

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
```

WRITING A PUBLISHER NODE

We are writing C++ code, so we must have a main function

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

}
```

The code for the publisher node will be written inside this function

WRITING A PUBLISHER NODE

The first thing to do when we write a ROS node is to call ros::init():

```
ros::init(argc, argv, "pub");
```

Next, we create a node handler:

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

WRITING A PUBLISHER NODE

Now we create a publisher object:

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

We have different way to create a spinner in ROS.

In this case, we want to fix the loop frequency (10 Hz):

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

WRITING A PUBLISHER NODE

Next, we create the main loop:

```
while (ros::ok()) {

}
```

while (ros::ok()) is just a better way to write while(1): it'll handle interrupts and stop if a new node with the same name is created or a shutdown command is called

WRITING A PUBLISHER NODE

Before calling the publisher node, we create our message:

```
std_msgs::String msg;
std::stringstream ss;
ss << "hello world ";
msg.data = ss.str();
```

The type of the message, as shown when creating the publisher, is

std_msgs::String

WRITING A PUBLISHER NODE

Now that we have a message, we can publish it on the chatter topic:

```
chatter_pub.publish(msg);
```

Last, we call:

```
loop_rate.sleep();
```

which will wait as much time as needed to keep the loop cycling at the specified frequency

WRITING A PUBLISHER NODE

To compile our node, we must add it to the CMakeLists.txt

We can start from a basic CMakeLists.txt automatically generated by the create_package command.

We add at the end of the file:

```
add_executable(publisher src/pub.cpp)
```

specifying that a node of *type name* publisher must be build from the source file src/pub.cpp

WRITING A PUBLISHER NODE

Then, we add:

target_link_libraries(publisher ${catkin_LIBRARIES})

to link the node executable to the needed libraries

Now we can cd to the root of our workspace and build our code using:

catkin_make

This will build every newly changed package in our workspace

WRITING A PUBLISHER NODE

If everything went well, we can start the ROS middleware:

roscore

and start our publisher node:

rosrun pub_sub publisher

We can check the messages published:

rostopic echo /publisher

WRITING A SUBSCRIBER NODE

The subscriber node has a similar structure to the publisher

We create a file called sub.cpp with includes and main function:

```
#include "ros/ros.h"
#include "std_msgs/String.h"
int main(int argc, char **argv) {
 ros::init(argc, argv, "sub");
 ros::NodeHandle n;


 return 0;
}
```

WRITING A SUBSCRIBER NODE

In the main function we create a subscriber object

```
ros::Subscriber sub = n.subscribe("/publisher", 1000, pubCallback);
```

where pubCallback is the name of the callback function

ROS will automatically call this function every time a new message is received

WRITING A SUBSCRIBER NODE

We are not interested in cycling at a fixed rate, so we simply call:

```
ros::spin();
```

`ros::spin()` will simply cycle as fast as possible, calling our callback when needed, but without using CPU if there is nothing to do

WRITING A SUBSCRIBER NODE

Now we can write our callback function

```
void pubCallback(const std_msgs::String::ConstPtr& msg) {
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

the argument of the function is a constant pointer to the received message, in our case std_msg::String

WRITING A SUBSCRIBER NODE

As we did for the publisher, we add it to the CMakeLists.txt file and link it to the required libraries:

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

Now we can build it with catkin and test the two nodes together

LAUNCH FILE

ROBOTICS

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:

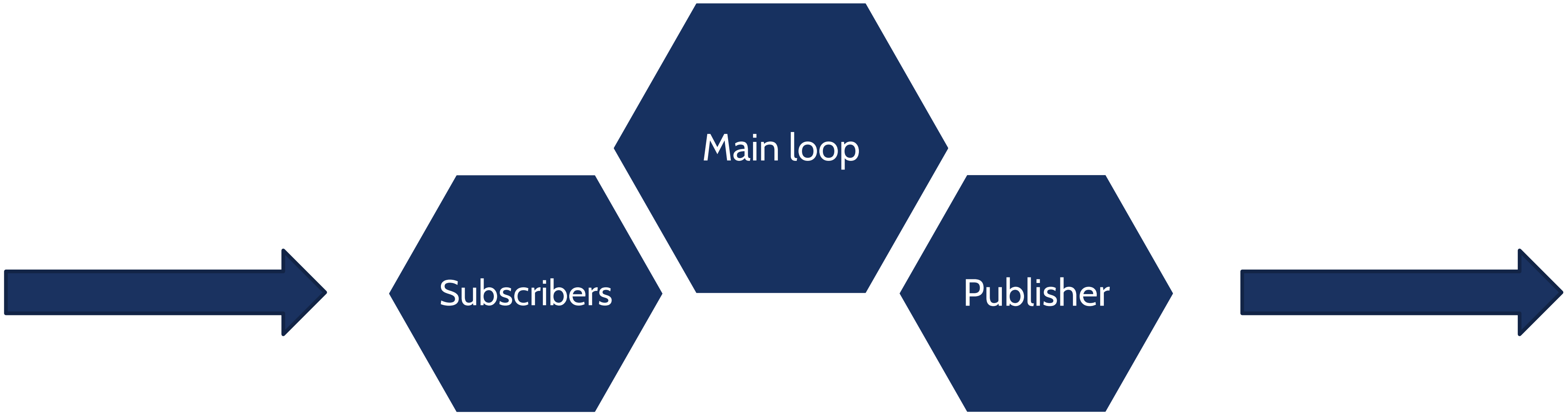
launch-prefix="xterm -e"

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);
```

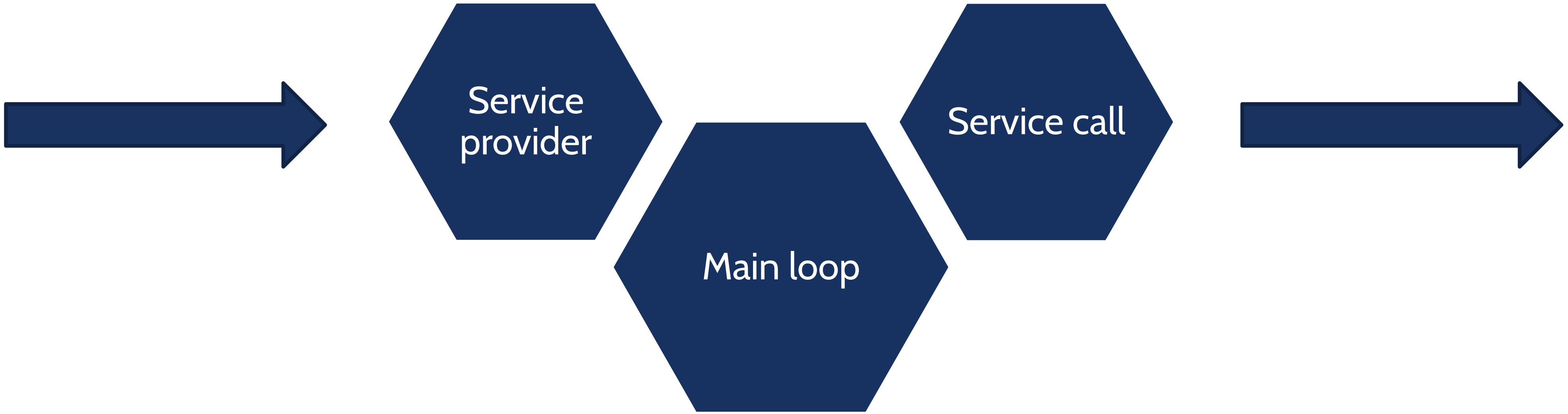
Now we can build and test both the publisher and the subscriber

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>
```