ROS DEVELOPMENT

ROBOTICS

POLITECNICO
MILANO 1863

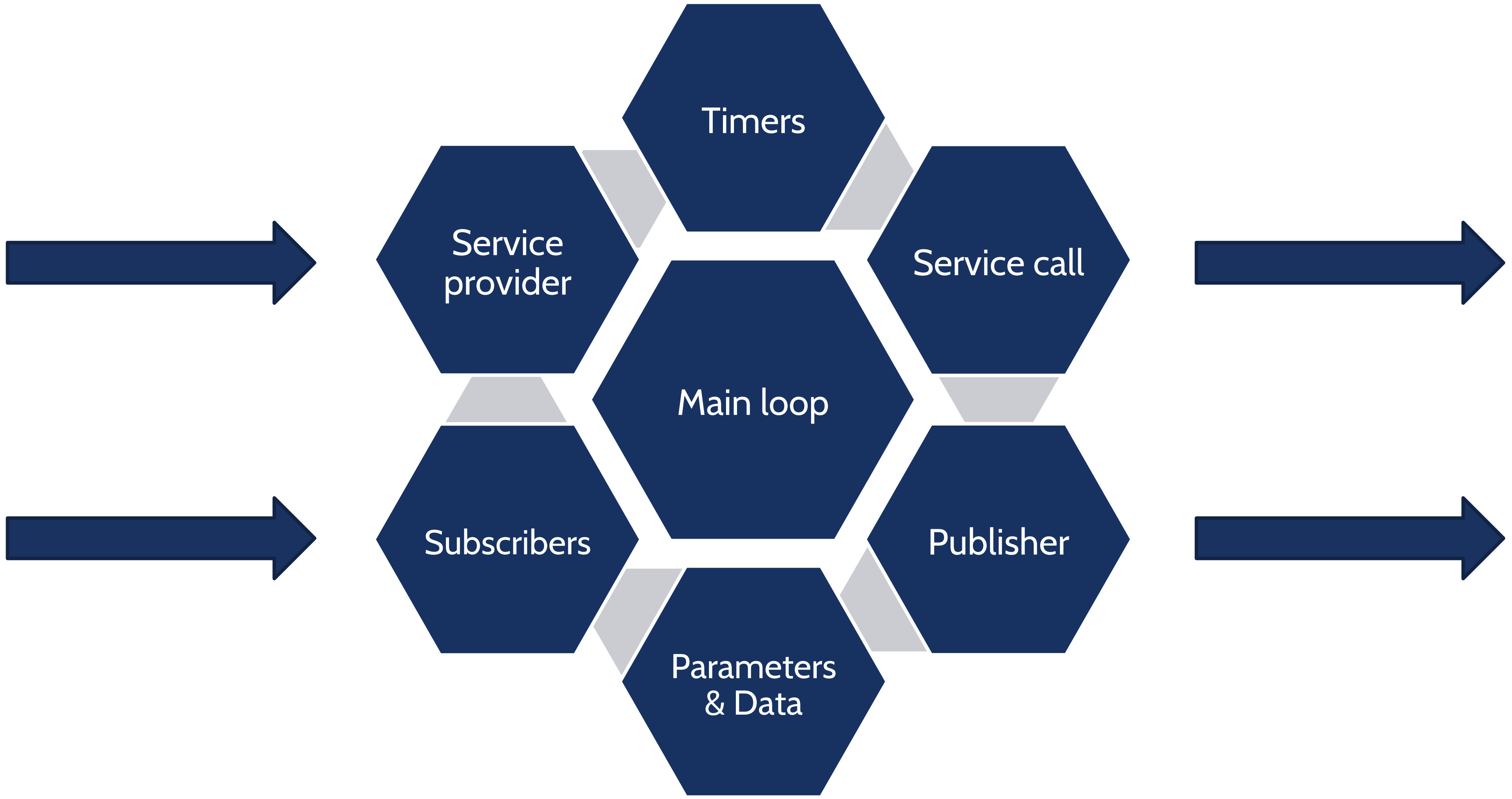
EVERYTHING HAPPENS IN NODES

Nodes are the main and atomic element of ROS. Each node is an indipendent process.

How do we create a node?

Write code in C++ or Python

INSIDE THE NODE



INITIALIZATION

Any node has to be registered to the ROS master using an unique identifier

The actual node is initialized using an handler

Each executable has an **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 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();
}
```

PARAMETERS

Stored in the parameter server and retrieved at the beginning of the execution

Adjustable at runtime using dynamic reconfigure

Global parameters and relative parameters (in the node namespace)

```
if(!nh.getParam("/global_name", global_name)) { /* :( */ }
if(!nh.getParam("relative_name", relative_name)) { /* :( */ }
nh.param<std::string>("param_name", default_param, "default_value");
```

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 everywhere

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);

sub = nh.subscribe("topic_name", 10, &class::callback, this);

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

TIMER

Used to execute something after a specific time (repeatable)

When the timer ends a callback function get called

Tied to ROS internal clock

```
ros::Timer timer = nh.createTimer(ros::Duration(0.5), callback);
timer = nh.createTimer(ros::Duration(0.5), &class::callback, this);
void [class::]callback(const ros::TimerEvent& t)
```

SERVICE PROVIDER (SERVER)

Answer to a service call and execute some logic associated with the content of the call

On declaration connect to the callback with the implemented logic

The answer of the service is already in the callback

```
ros::ServiceServer s = nh.advertiseService(“service”, callback);
s = nh.advertiseService(“service”, &class::callback, this);
bool [class::]callback(pack::srv_type::Request& req,
                       pack::srv_type::Response& res);
```

SERVICE PROVIDER (SERVER)

Generates the call for a specific service

On declaration is connected to the a service identified by a name

Can be called everywhere in the code

May result in a bad call

```
ros::ServiceClient cl = nh.serviceClient<pack::srv_type>(“service”);

pack::srv_type srv;

/* fill the service */

if (cl.call(srv)) { /* :) */ } else { /* :( */ }
```

CREATING THE WORKSPACE

ROS uses a custom compiling environment called **Catkin**

cmake/make with specific flags

Requires a workspace with a specific structure

Easy to setup and easy to use

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/
catkin_make
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

WORKSPACE STRUCTURE

All your stuff goes here!

Source space (/src):

- contains the source code of catkin packages.
- Subfolder of this are the ROS packages you want to add to your system

Build space (/build):

- space where cmake is invoked 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

BUILDING YOUR CODE

```
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)
```

BUILDING YOUR CODE

```
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 depending on your code!

BUILDING YOUR CODE

```
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)
```

Only if you have custom messages!

WRITING A PUBLISHER NODE

First create a package inside you src folder:

```
$ catkin_create_pkg pub_sub std_msgs rospy roscpp
```

Next cd to the new pub_sub/src folder and create a c++ file:

```
$ gedit pub.cpp
```

WRITING A PUBLISHER NODE

First write some includes:

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

WRITING A PUBLISHER NODE

We are still writing c++ code, so we have to write a min function

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


}
```

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

WRITING A PUBLISHER NODE

As previously explained the first thing to do when we write a ROS node is call ros::init():

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

And next create a node handle:

```
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, but in this case we want to control the loop frequency:

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

WRITING A PUBLISHER NODE

Next we write the main loop:

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

while (ros::ok()) is just a better way to write while(1), it'll handle interrupt, stop if a new node with the same name is create 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 during the publisher creation is std_msgs::String

WRITING A PUBLISHER NODE

Now that we have a message we can call publish it:

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

Last we call:

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

which will wait until the time previously specified has passed, and then restart the loop

WRITING A PUBLISHER NODE

Before compiling our code we have to add it to the CMakeLists.txt file

It already has some code, generated by the create_package command; first add at the end of the file:

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

to add the new file we have created

WRITING A PUBLISHER NODE

Then add:

```
target_link_libraries(publisher ${catkin_LIBRARIES})
```

to specify the correct library, and:

```
add_dependencies(publisher pub_sub_generate_messages_cpp)
```

to specify the dependencies from message generation

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

```
$ catkin_make
```

WRITING A PUBLISHER NODE

If everything went well you can start roscore:

```
$ roscore
```

and start your node:

```
$ rosrun pub_sub publisher
```

you can check the published topic with:

```
$ rostopic echo /publisher
```

WRITING A SUBSCRIBER NODE

The subscriber node has a similar structure to the publisher, create a file called sup.cpp:

```
#include "ros/ros.h"
#include "std_msgs/String.h"
int main(int argc, char **argv)
{

  ros::init(argc, argv, "sub");
  ros::NodeHandle n;
}
```

WRITING A SUBSCRIBER NODE

But this time inside 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 called every time a new message is received

WRITING A SUBSCRIBER NODE

We are also not interested in cycle at a predetermined speed, so we will simply call:

```
 ros::spin();


  return 0;
```

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 pointer to the received message, in our case a std_msg::String

WRITING A SUBSCRIBER NODE

As we did with the publisher node we have to add it to the CMakeLists.txt file:

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

Now we can compile it and test the two nodes together

CREATE A MESSAGE

Messages are saved in the msg folder of the package; first create the folder inside the pub_sub package:

```
$ mkdir msg
```

Next create the msg file:

```
$ echo "int64 num" > msg/Num.msg
```

CREATE A MESSAGE

Before using the new message we have to make sure they are converted into source code for c++, open the package.xml file and uncomment those two line:

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

CREATE A MESSAGE

Next we have to edit the CMakeLists.txt file, first add message_generation dependency to find_package

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

CREATE A MESSAGE

Then export the message_runtime dependency uncommenting the following call and adding message_runtime:

```
catkin_package(
  CATKIN_DEPENDS message_runtime
 )
```

CREATE A MESSAGE

Last we have to add the message file and generate them, so uncomment the "add_message_files" and "generate_messages" and add to the first our msg file

```
add_message_files(
 FILES
 Num.msg

 )
```

```
generate_messages(
   DEPENDENCIES
  std_msgs
 )
```

CREATE A MESSAGE

Now we can compile our code calling catkin_make in the root directory of the workspace and test if ros finds our new message calling:

```
$ rosmsg show pub_sub/Num
```

USING CUSTOM MESSAGES

To test our new message we will modify the publisher-subscriber nodes open the pub.cpp file

first we include the custom message adding:

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

then we modify the publisher object, changing the type of the message:

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

USING CUSTOM MESSAGES

Last 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 compile our code 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

```
#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 compile and test both the publisher and the subscriber

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 node 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 launchfile is a 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 launchfile 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"