

EDITORS/ IDEs

ROBOTICS



POLITECNICO
MILANO 1863

ROSED



rosed is part of the rosbash suite

Allow the user to edit files using directly the package name, rather than typing the entire path

```
rosed [package_name] [filename]
```

```
rosed roscpp Logger.msg
```

The default editor is vim

You can edit the .bashrc file setting a more user friendly editor

IDEs



No official IDE by ROS

C++ editor with ROS specific plugins

On ROS wiki you can find guides on how to properly configure the plugins

<http://wiki.ros.org/IDEs>

Simply add some features like easier compiling and some debug tools



Roboware



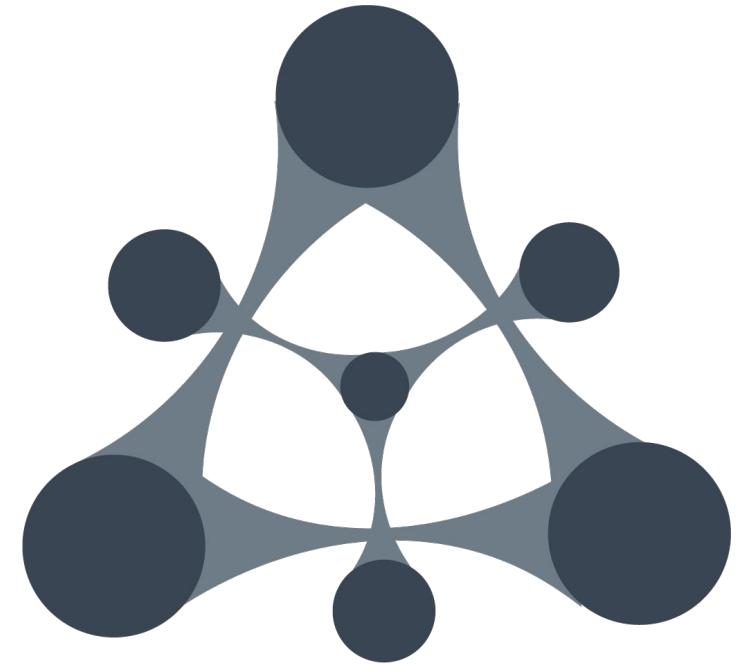
Based on Visual Studio

Designed for ROS

No need to install third parties plugin

Offers some functionalities:

- Run program directly inside Roboware
- Debugger
- Automatic file generation
- CMakeLists and Package.xml automatic update (partial)
- Integrated ros tool



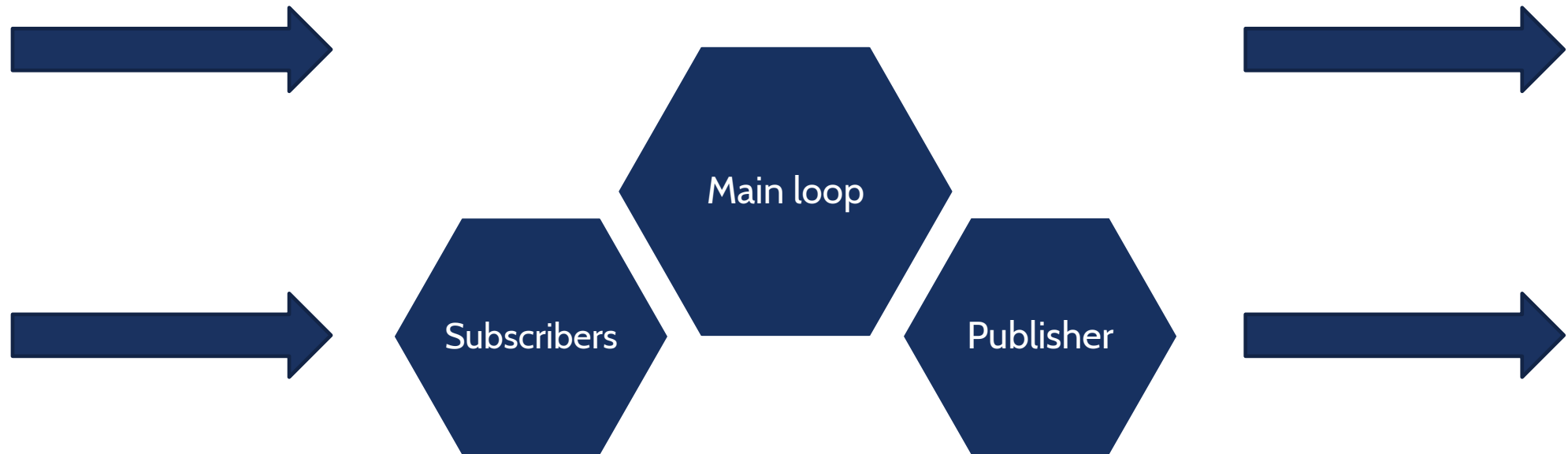
PUBLISHER-SUBSCRIBER

ROBOTICS



POLITECNICO
MILANO 1863

INSIDE THE NODE





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

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_msgs::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})
```

Now we can compile it and test the two nodes together

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 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"
```

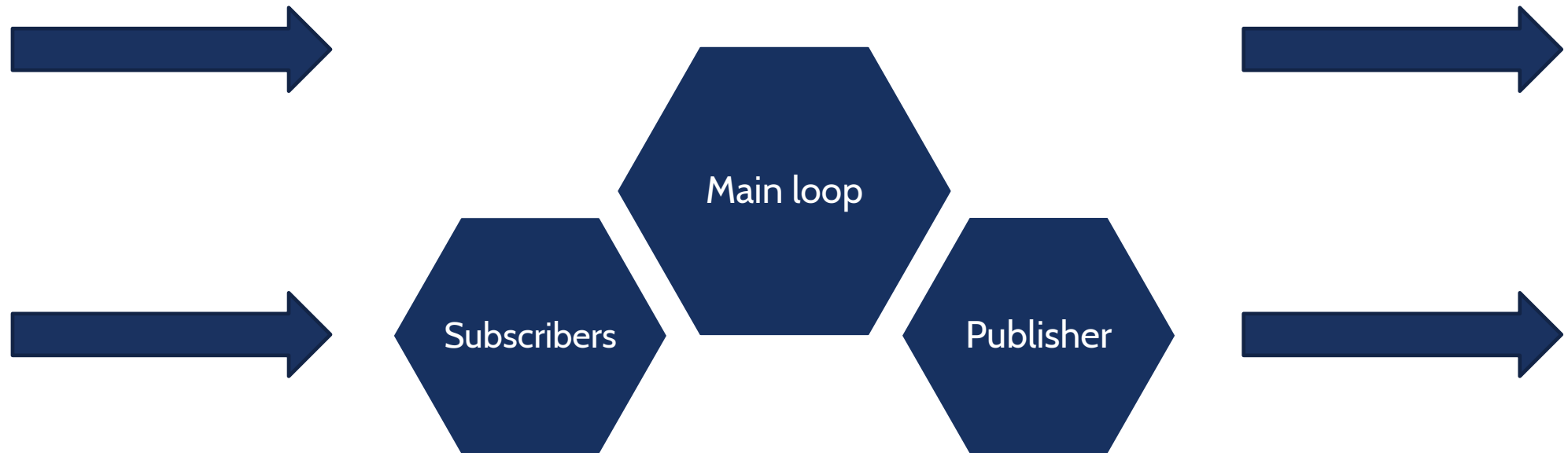
CUSTOM MESSAGES

ROBOTICS



POLITECNICO
MILANO 1863

INSIDE THE NODE





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  
)
```

Also specify that the publisher package depends from the custom message

```
add_dependencies(publisher pub_sub_generate_messages_cpp)
```

to specify the dependencies from message generation



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:

```
$ rosmmsg 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

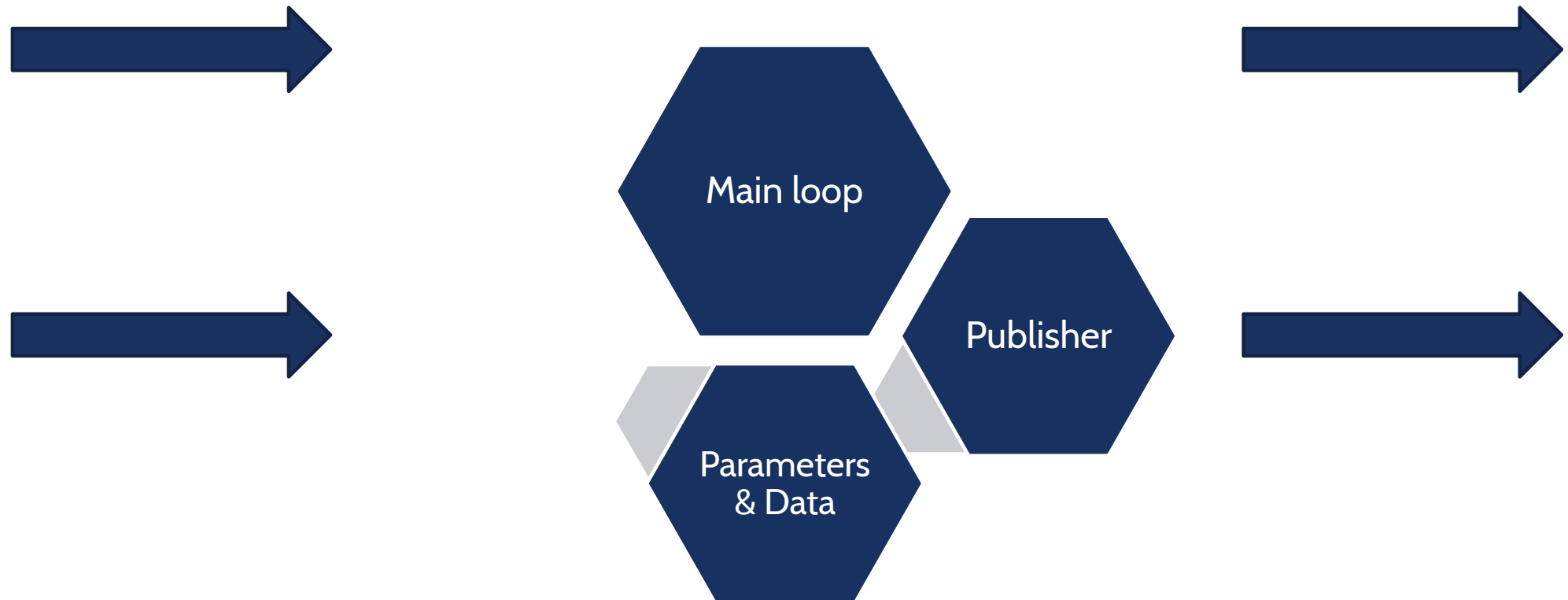
PARAMETERS

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 to publisher/subscriber example:

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

```
#include "std_msgs/String.h"
```

```
#include <sstream>
```

} Standard include



PARAMETER CHECK BEFORE MAIN LOOP

Similar code to publisher/subscriber example:

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

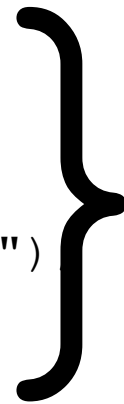
```
    ros::init(argc, argv, "param_first")
```

```
    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 to publisher/subscriber example:

`n.getParam("/name", name);` ← get parameter value

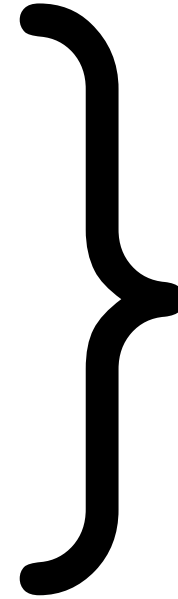
`ros::Rate loop_rate(10);` ← set loop rate



PARAMETER CHECK BEFORE MAIN LOOP

Similar code to 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" /> ← Set the parameter value
```

```
<node pkg="parameter_test" name="param_first" type="param_first"
```

```
output="screen" ← Redirect the output of ROS_INFO to the terminal
```

```
/>
```

```
</launch>
```