Robotics

Lab session 4

POLITECNICO
MILANO 1863

Paolo Cudrano
paolo.cudrano@polimi.it

RECAP

Last lecture:

- ROS development (part 1)
    - ROS code organization (workspace and packages)
    - Creating our workspace: ~/robotics/
    - Creating our first package: pub_sub
    - Implementing our first (two) nodes: pub, sub
    - Building our new package

OUTLINE

Today:

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

ROS NAMES AND REMAPPING

ROBOTICS

POLITECNICO
MILANO 1863

ROS NAMES

At runtime, all resources in the computation graph (nodes, parameters, topics, services) are provided with a Graph Resource Name (or **name)**

Names provide encapsulation:

- each resource is defined within a **namespace**
- **r**esources can create resources within their namespace
- resources can access resources within or above their own namespace

ROS NAMES

There are four types of Graph Resource Names in ROS:

- Base `base`
- Relative `relative/name`
- Global (to be avoided) `/global/name`
- Private `~private/name`

Notice: names with no namespace qualifiers whatsoever are *base* names (typically done for initializing node names)

ROS NAMES

By default, resolution is done relative to the node's namespace

E.g., Node `/ws/node1` (namespace: `/ws`)

In the implementation of `node1`:

- `n.subscribe("topic", 1, cb);` → resolved as `/wg/topic` (relative)
- `n.subscribe("/topic", 1, cb);` → resolved as `/topic` (global)
- `n.subscribe("~/topic", 1, cb);` → resolved as `/wg/node1/topic` (private)

ROS NAMES - REMAPPING

Sometimes we may need to change a resource name without changing directly its code definition. For this reason, any name within a ROS Node can be remapped when the node is launched

E.g.,

```
rosrun turtlesim turtle_teleop_key /turtle1/cmd_vel:=/turtle2/cmd_vel
```

Important: remapping allows us to write more general and portable code, using **generic, relative resource names** in our node implementations, and remapping them properly when running our node in each particular context

ROS NAMES - REMAPPING

ROS defines also special keywords for remapping particular aspects of a node,

such as:

- __name: special reserved keyword for "the name of the node." It lets you remap the node name without having to know its actual name. It can only be used if the program that is being launched contains one node.

```
rosrun turtlesim turtlesim_node __name:=turtlesim_1
rosrun turtlesim turtlesim_node __name:=turtlesim_2
```

LAUNCH FILE

ROBOTICS

POLITECNICO
MILANO 1863

LAUNCH FILE

When working on big projects, it is useful to create a launch file.

With only one command, the launch file 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 file with extension `.launch`

The launch file is an XML file with root tags `<launch></launch>`

Inside these tags, you can start all your nodes using:

```
<node pkg="package_name" type="node_type" name="node_name"/>
```

Name of the executable

Runtime name, specified in `ros::init`

which is equivalent to running from command line:

```
rosrun package_name node_type
```

In ROS terminology, a **node type** is the name of the executable of a node

The `name` attribute allows us to remap the **node name** (i.e., the runtime name)

LAUNCH FILE – SIMPLE EXAMPLE

Example of a simple launch file, `pub_and_sub.launch`:

```
<launch>
    <node pkg="pub_sub" type="pub" name="my_publisher" />
    <node pkg="pub_sub" type="sub" name="my_subscriber" output="screen" />
</launch>
```

This will run the nodes `pub` and `sub` at the same time and print to screen the output of `sub` (by default, screen output is disabled for nodes ran by launch files).

We can launch it from command line with:

```
roslaunch pub_sub pub_and_sub.launch
```

(In general: `roslaunch package_name launch_file_name.launch`)

LAUNCH FILE - NAMESPACES

We can also regroup some nodes under a specific **namespace** using the tag group:

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

Namespaces allow us to start multiple nodes with the same node name, because they live in different namespaces

E.g.

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

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

LAUNCH FILE - REMAPPING

Sometimes we may need to change a topic name without changing directly its code.

This can be done from command line

```
rosrun turtlesim turtle_teleop_key /turtle1/cmd_vel:=/turtle2/cmd_vel
```

To accomplish this task from a launch file we use the tag `remap` inside a node:

```
<node ...>
    <remap from="original_name" to="new_name"/>
</node>
```

LAUNCH FILE – TURTLESIM EXAMPLE

Create a new launch file `multi_turtle.launch` containing 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 – TURTLESIM EXAMPLE

This code starts two `turtlesim` and connects them together, i.e. the commands on topic `cmd_vel` for `turtlesim1` will also be redirected to `turtlesim2`

We can then open the `teleop_key` node, adding it to the `turtlesim1` namespace:

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

If we want to open the `teleop_key` node in a new terminal window, we can add to its node tag one of the following attributes:

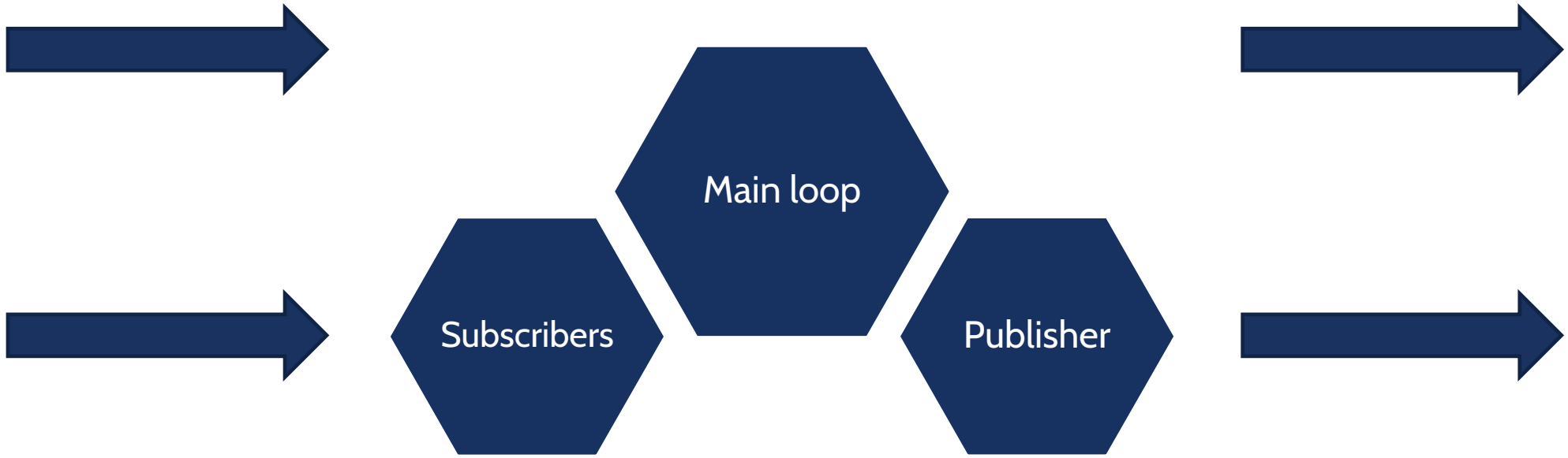
`launch-prefix="gnome-terminal -e”`	for Terminal
`launch-prefix="terminator -x”`	for Terminator

CUSTOM MESSAGES

ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE



IMPORTANT! TECHNICAL NOTE FOR THE LECTURE

We will now add several functionalities to our basic `pub_sub` package.

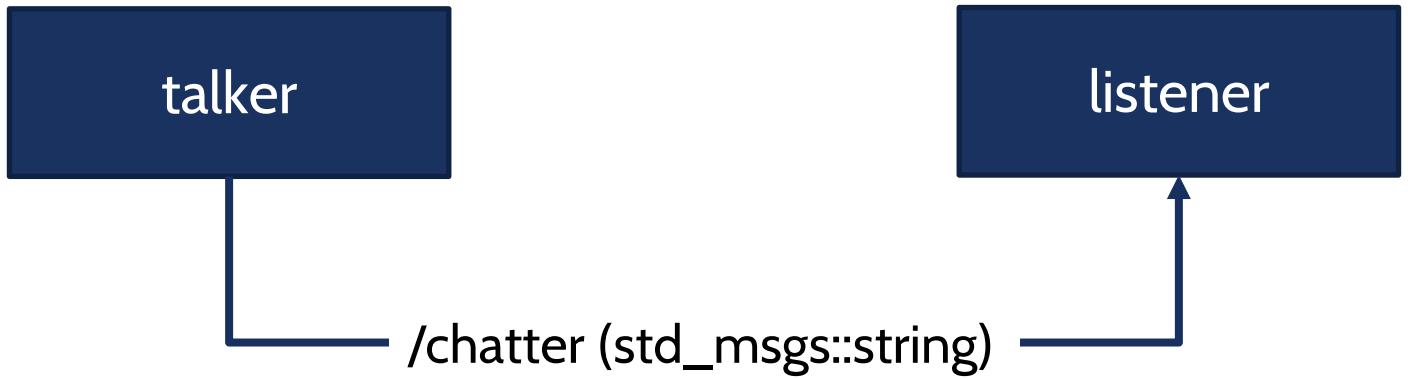
In the slides, these are described as incremental changes on the same `pub_sub` package from last lecture.

In class, however, it is not easy to implement the changes live, so you can find each incremental update ready in our shared folder, as: `pub_sub_v2`, `pub_sub_v3`, ...

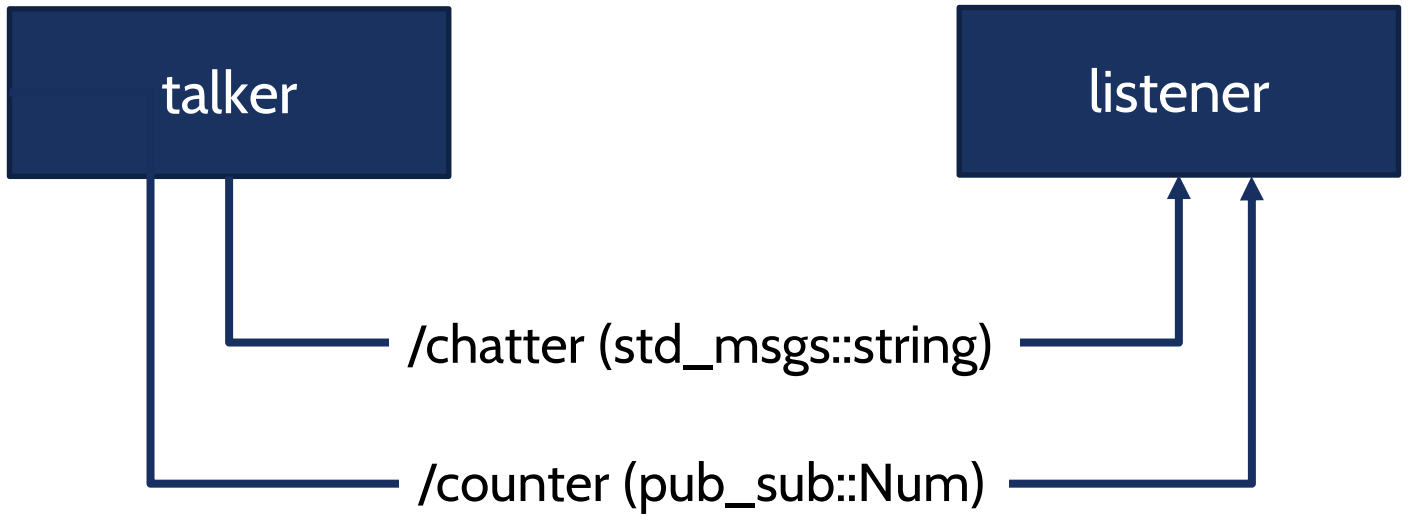
However, we CANNOT copy all of them together in your workspace, as ROS prevents us from having multiple packages and executables with the same name.

Instead, we will substitute each updated version to the previous one, keeping only 1 `pub_sub` package in our workspace at any given time.

Last lecture: pub_sub



Objective: pub_sub_v2



CREATE A CUSTOM MESSAGE

Custom messages are saved in the `msg/` folder of our package

First, create the folder inside the `pub_sub` package:

```
mkdir msg
```

Next, create the msg file:

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

(i.e., create a file named `msg/Num.msg` containing the line of code `int64 num`)

SET THE MESSAGE DEPENDENCIES

Before using the new message, we make sure they are converted into source code

To do this, open the `package.xml` file and uncomment these two lines:

```
<build_depend>message_generation</build_depend>

<exec_depend>message_runtime</exec_depend>
```

BUILDING WITH CUSTOM MESSAGE: GENERATION

Next, we edit the `CMakeLists.txt` to tell ROS to:

- generate the custom message: create header files and internal functions
- use the custom message in a node (our pub executable)

We set `message_generation` as required dependency for our package (needed to build the custom message)

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

BUILDING WITH CUSTOM MESSAGE: GENERATION

We tell catkin which are our custom message files and instruct it to build them

```
add_message_files(
  FILES
  Num.msg
)

generate_messages(
  DEPENDENCIES
  std_msgs
)
```

Custom message file

Dependencies for our custom message file
(we use int64, which is defined in std_msgs)

BUILDING WITH CUSTOM MESSAGE: GENERATION

We set `message_runtime` as export dependency for our package (needed to use the custom message from other packages):

```
catkin_package(
   CATKIN_DEPENDS message_runtime
 )
```

BUILDING WITH CUSTOM MESSAGE: USAGE

To use our custom message, we should tell catkin to include the message header files it will generate. Thankfully, catkin helps us setting their path inside its `catkin_INCLUDE_DIRS` variable. Therefore, we just add:

```
include_directories(include ${catkin_INCLUDE_DIRS})
```

BUILDING WITH CUSTOM MESSAGE: USAGE

Then, we should also specify that the publisher executable depends on the compiled custom message, with this dependency:

```
add_dependencies(pub pub_sub_generate_messages_cpp)
```

Package name

However, catkin helps us by setting `pub_sub_generate_messages_cpp` inside its `catkin_EXPORTED_TARGETS` variable, so we just need to write:

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

Notice: this tells catkin that the targets in `${catkin_EXPORTED_TARGETS}` must be built before building `pub`

BUILDING WITH CUSTOM MESSAGE: USAGE

We do the same for the subscriber:

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

BUILDING WITH CUSTOM MESSAGE

Now we can build our package, calling

```
catkin_make
```

from the root of our workspace (`~/robotics/`)

We can then test if ROS finds our new message, calling:

```
rosmsg show pub_sub/Num
```

PUB-SUB WITH CUSTOM MESSAGES

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

We start with `pub.cpp`

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 counter_pub = n.advertise<pub_sub::Num>(”counter", 1000);
```

PUB-SUB WITH CUSTOM MESSAGES

Last, we create a message of type `pub_sub::Num` and assign `count` to it

```
pub_sub::Num count_msg;
count_msg.num = count;

counter_pub.publish(count_msg)
```

PUB-SUB WITH CUSTOM MESSAGES

The changes to the `sub.cpp` file are similar

First, include the new message:

```
#include "pub_sub/Num.h”
```

Then, we subscribe to this topic

```
ros::Subscriber sub_counter = n.subscribe("counter", 1000,
            counterCallback);
```

PUB-SUB WITH CUSTOM MESSAGES

Finally, add the subscriber callback:

```
void counterCallback(const pub_sub::Num::ConstPtr& msg) {
  ROS_INFO("I counted: [%d]", msg->num);
}
```

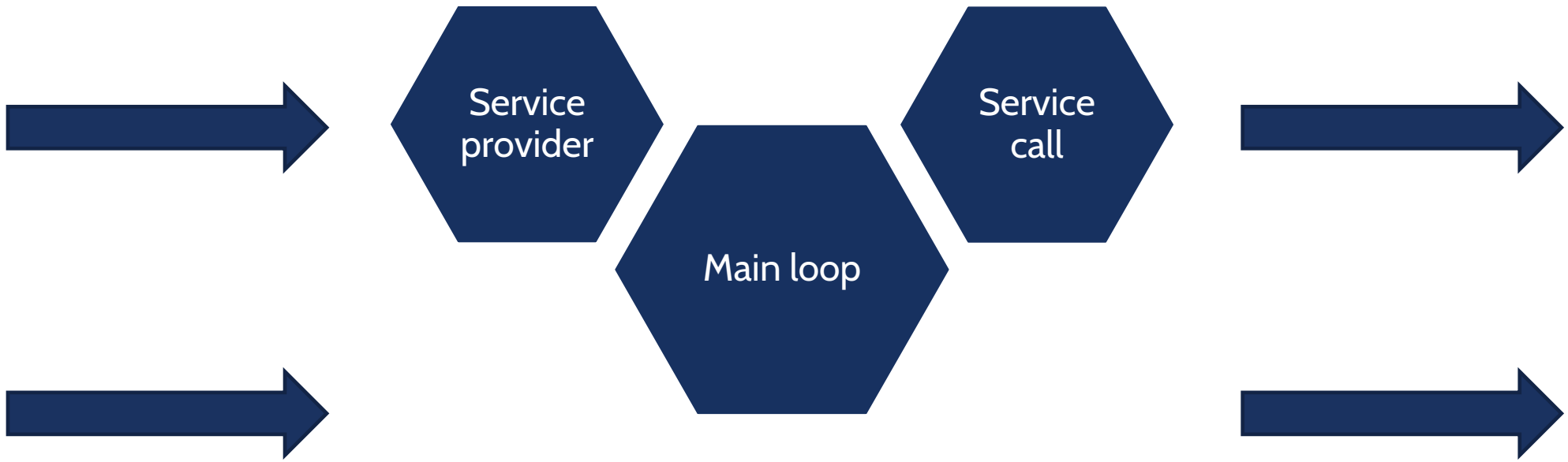
Now we can compile and test both the publisher and the subscriber

SERVICES

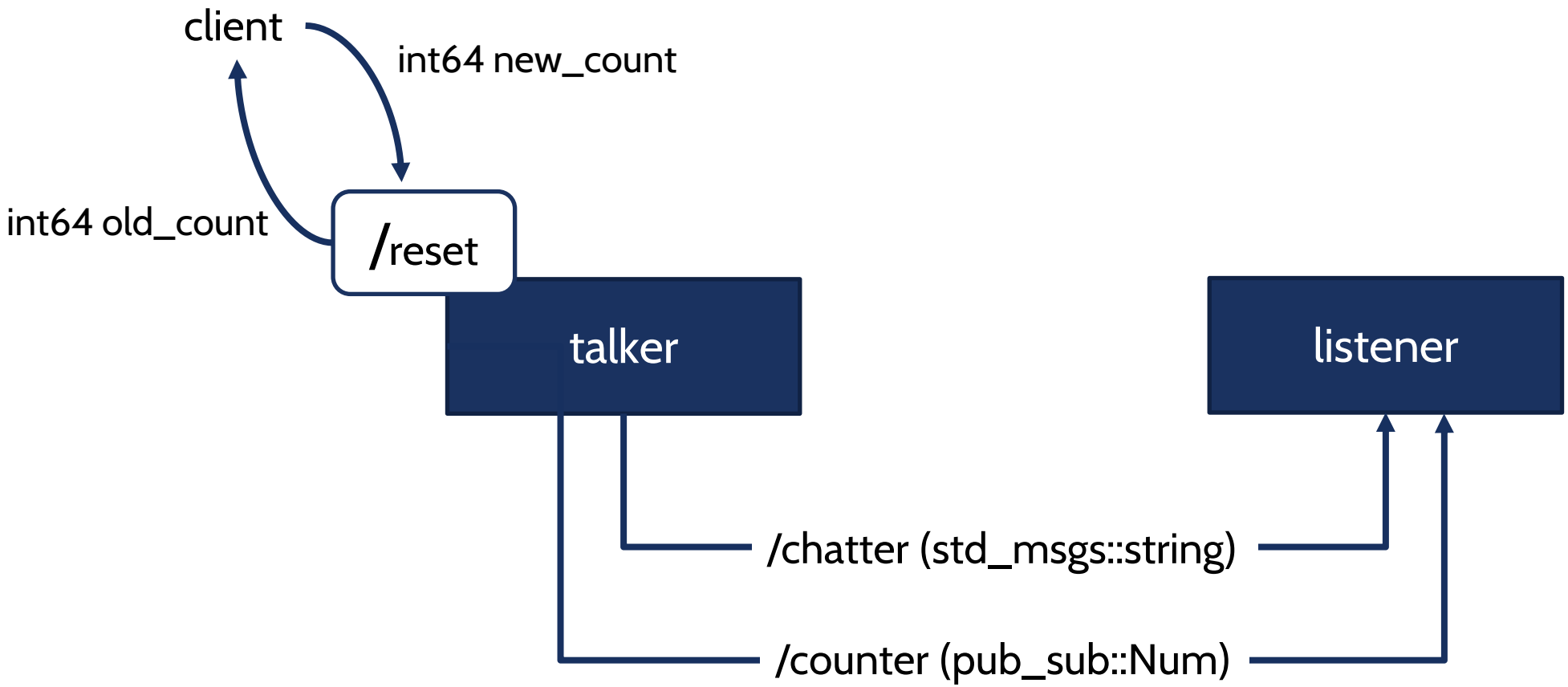
ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE



Objective: pub_sub_v3



SERVICES

The service creation process is similar to the custom messages

First, we create a `srv` folder where we insert the definition of the service, in our example we create the file `Reset.srv`

```
int64 new_count
---
int64 old_count
```

SERVICES (Server)

Then, we setup the service server in our publisher node.

```
#include ”pub_sub/Reset.h"
```

Include the header file generated starting from the Reset.srv

SERVICES (Server)

In the initialization part of the main, we create the service server:

```
ros::ServiceServer service =

      n.advertiseService<pub_sub::Reset::Request,

                         pub_sub::Reset::Response>(

          "reset",

          boost::bind(&reset_callback, &count, _1, _2)

      );
```

Name of the service

Callback function using boost::bind to pass additional arguments (we pass count, as pointer)

SERVICES (Server)

In the callback function, differently from the subscriber, we have two fields, one for the request and one for the response:

```
bool reset_callback(int *count, pub_sub::Reset::Request &req,
                    pub_sub::Reset::Response &res) {
```

Pointer to `count` in main()

Type of the service

Pointers to request and response

SERVICES (Server)

In the callback, we reset the count variable to the new count and return the old count. We also print to screen their values.

```
bool reset_callback(int *count, pub_sub::Reset::Request &req,
                                pub_sub::Reset::Response &res) {
  res.old_count = *count;
  *count = res.new_count;
  ROS_INFO(”Request to reset count to %ld - Responding with old count: %ld”,
           (long int) req.new_count, (long_int)res.old_count);
  return true;
}
```

SERVICES (Client)

We can call the service from command line, with

```
rosservice call /reset 0
```

Optionally, we can also write a client node: `client.cpp`

We add similar includes:

```
#include "ros/ros.h”
#include ”pub_sub/Reset.h"
```

SERVICES (Client)

We initialize ROS and check if the client node was properly launched, passing the new count as command line argument

```
int main(int argc, char **argv) {
  ros::init(argc, argv, "reset_client");

  if (argc != 2) {
    ROS_INFO("usage: client new_count");
    return 1;
  }
```

SERVICES (Client)

Then, we create the node handle and a service client using the service type and its name. We create the service object and the request

```
ros::NodeHandle n;

ros::ServiceClient client = n.serviceClient<pub_sub::Reset>("reset");

pub_sub::Reset srv;
srv.request.new_count = atoll(argv[1]);
```

SERVICES (Client)

Last we try calling the server and if we get a response we print it

```
if (client.call(srv)) {
  ROS_INFO("Old count: %ld", (long int)srv.response.old_count);
}
else {
  ROS_ERROR("Failed to call service reset");
  return 1;
}
```

SERVICES (CMakeLists.txt)

We also have to do some changes in the CMakeLists.txt

If not already there, add "`message_generation`" to the `find_package`

Then, add the service file

```
add_service_files(
  FILES
  Reset.srv
)
```

SERVICES (CMakeLists.txt)

Next, if not already there, we have to set

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

and

```
catkin_package(CATKIN_DEPENDS message_runtime)
```

SERVICES (CMakeLists.txt)

Lastly, to make sure that the header file are generated, before compiling the nodes we add

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

after creating the pub target (if not already there)

We also create a client target

```
add_executable(reset_client src/client.cpp)
add_dependencies(reset_client ${catkin_EXPORTED_TARGETS})
target_link_libraries(reset_client ${catkin_LIBRARIES})
```

SERVICES (Package.xml)

Finally, we edit the `Package.xml` to add the new dependencies

Insert (if not already there):

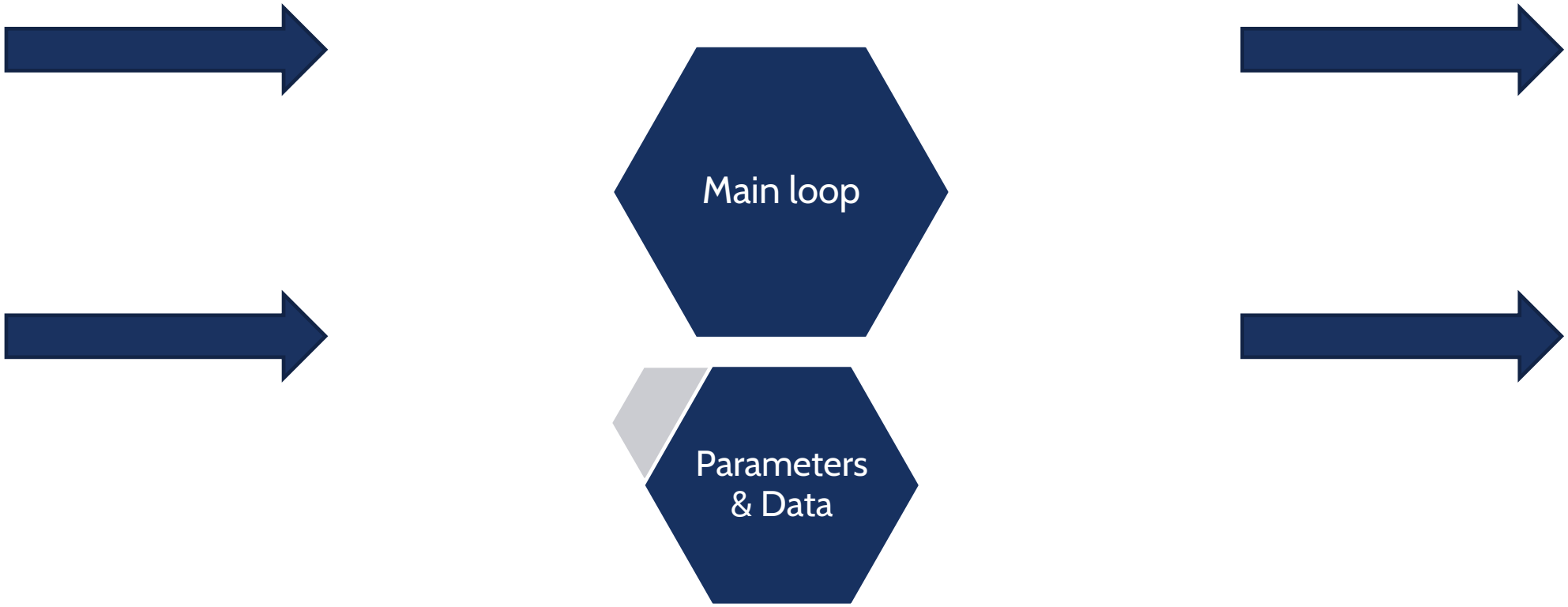
```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

PARAMETERS

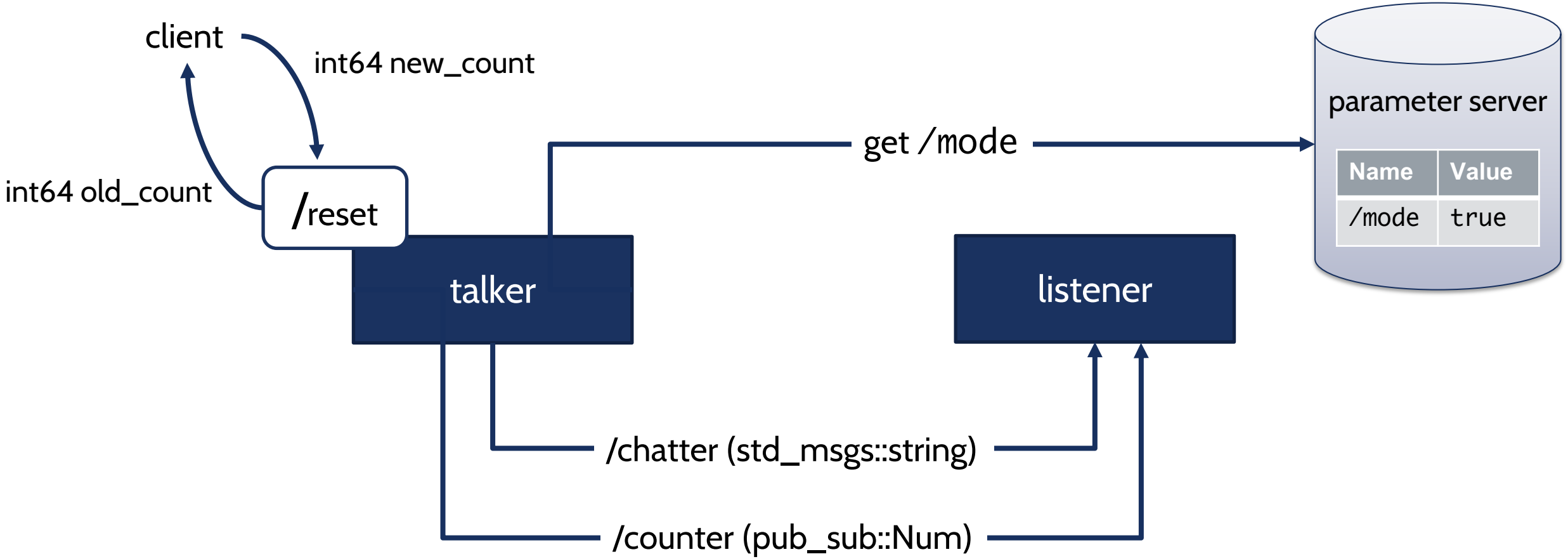
ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE



Objective: pub_sub_v4



PARAMETERS

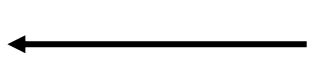
Main usage:

- Parameters set up before starting the node (or in launch file)
- Node looks at parameters before entering its main loop

RETRIEVING A PARAMETER (BEFORE THE MAIN LOOP)

During initialization (**before main loop**):

```
bool mode;
n.getParam("/mode", mode);
```

 get parameter value

Important: if you change the value while the node is running (in its main loop), the change will have no effect because **the node looks at the value only during the initialization**

SETTING PARAMETERS FROM LAUNCH FILE

A good practice is to set parameters directly in your launch file, in order to avoid initializing them from command line every time you want to run the node.

To set a parameter from the launch file, add to the file the line:

```
<param name="name" value="value" />
```

SETTING PARAMETERS FROM LAUNCH FILE

In pub_and_sub.launch:

```
<launch>

 <param name=”mode" value=”true" />

 <node pkg=”pub_sub” type=“pub” name=”my_publisher">
 <node pkg=”pub_sub” type=“sub” name=”my_publisher" output="screen"/>

</launch>
```

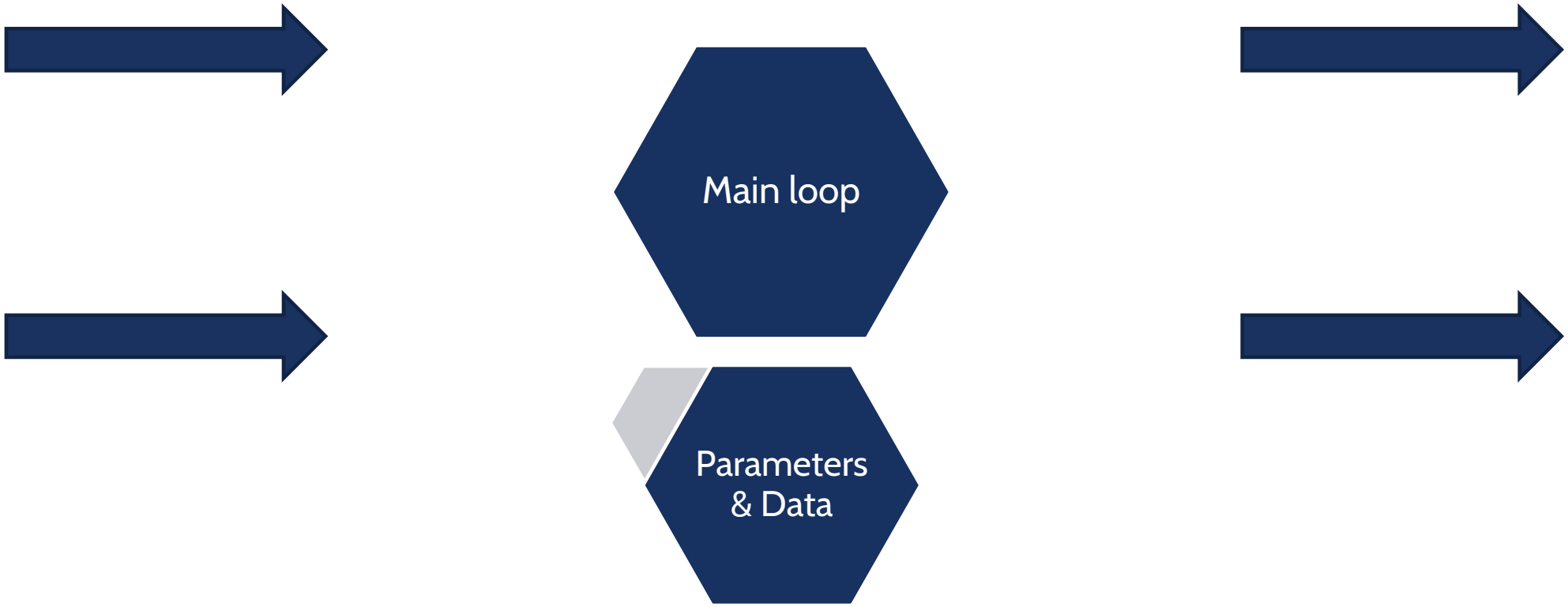
Set the parameter value

PARAMETERS: DYNAMIC RECONFIGURE

ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE



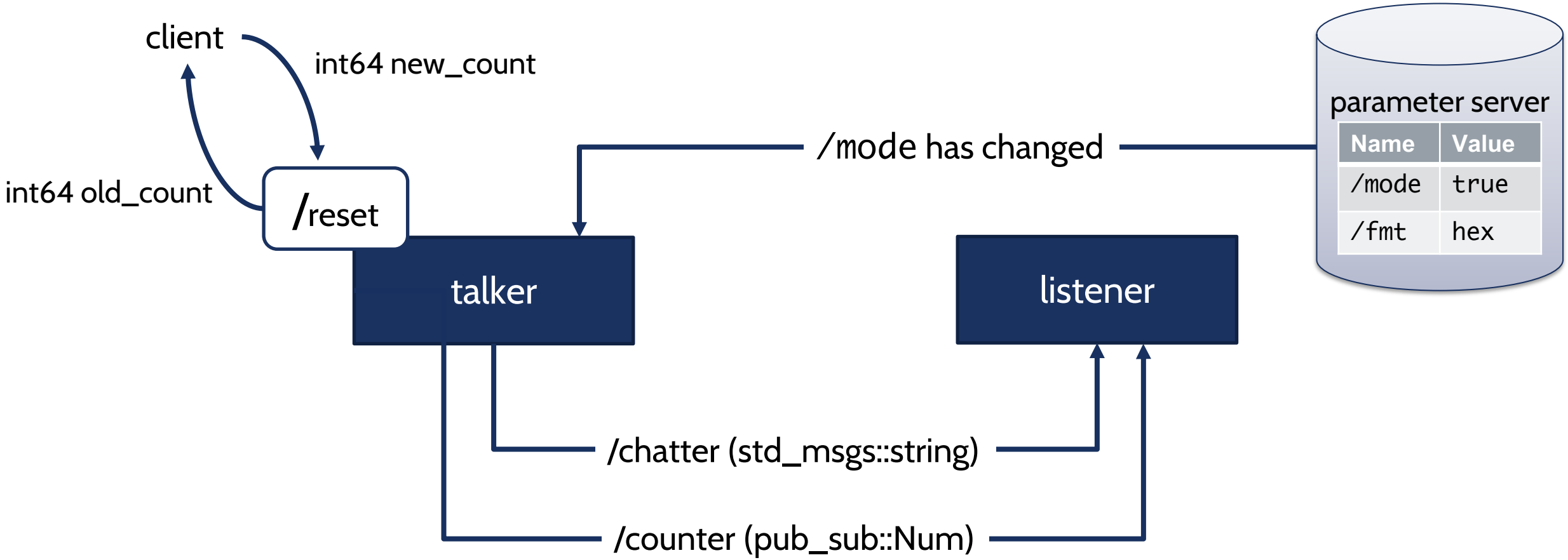
DYNAMIC RECONFIGURE

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:

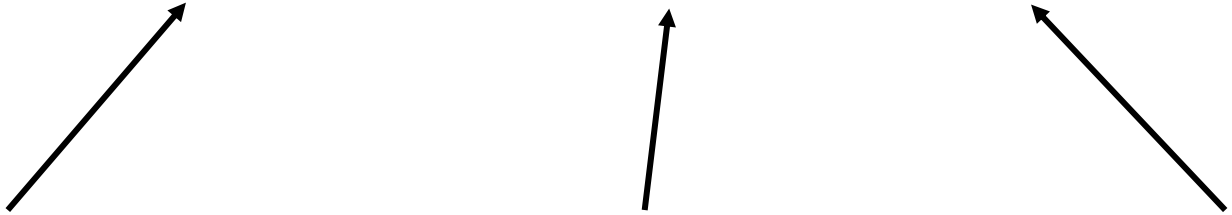
```
fmt_enum = gen.enum([  gen.const(”Decimal", int_t, 0, ”Decimal format"),
                       gen.const(”Binary", int_t, 1, ” Binary format"),
                       gen.const(”Hexadecimal", int_t, 2, ”Hexadecimal format"),
                      ”Enum of formats")

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;

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

Create the parameter server specifying the type of config

Create the callback

DYNAMIC RECONFIGURE

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

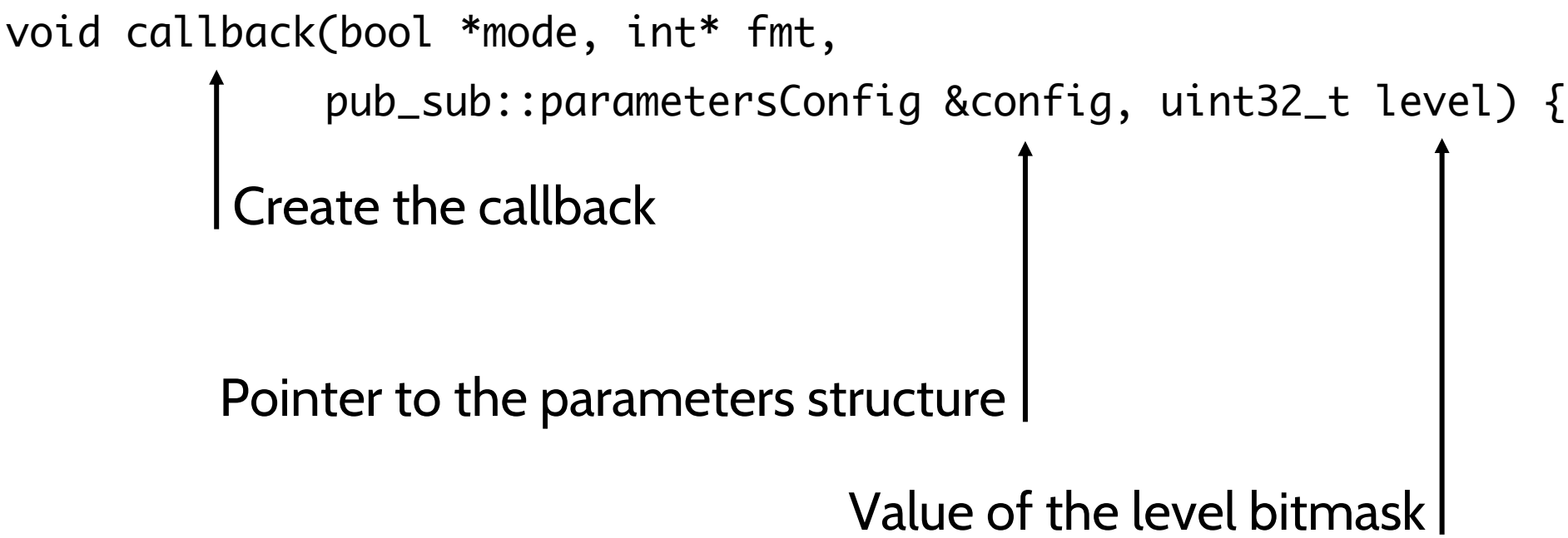
Bind the callback

Pass `mode` and `fmt` as pointers

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

Set the server callback

DYNAMIC RECONFIGURE



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})
```