Robotics

Lab session 1

POLITECNICO
MILANO 1863

Paolo Cudrano
paolo.cudrano@polimi.it

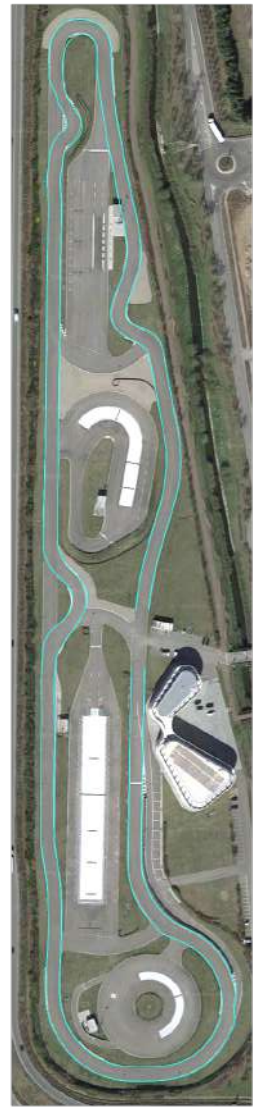
AIRLAB
ARTIFICIAL INTELLIGENCE AND ROBOTICS LAB

ABOUT ME

Paolo Cudrano, PhD student in Computer Sc. and Eng.

Artificial Intelligence and Robotics Lab (AIRLab)

Contact: paolo.cudrano@polimi.it



Research interests:

- Autonomous driving
- Computer Vision
- Machine Perception
- Deep learning

Course program

Introduction

- Middlewares in robotics
- Introduction to ROS
- ROS commands

~2 lectures

ROS development

- ROS building tools
- Topics
- Services
- Parameters
- Timers
- Message filters

~2 lectures

ROS tools

- Launch file
- ROS bags
- tf
- rviz
- Rospy

~1 lecture

ROBOTIC MIDDLEWARES

ROBOTICS

POLITECNICO
MILANO 1863

MIDDLEWARE ORIGINS

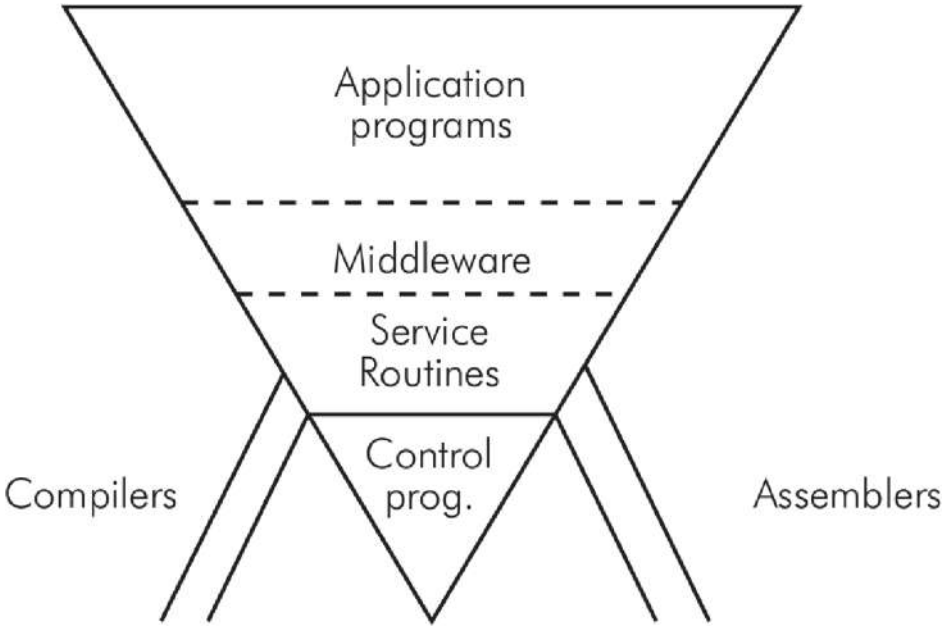
The origins

1968 introduced by d'Agapeyeff

80's wrapper between legacy systems and new applications

Nowadays: widespread in different domain fields (including Robotics)

Some (non robotics) examples: Android, SOAP, Web Services, ...



MIDDLEWARE ORIGINS

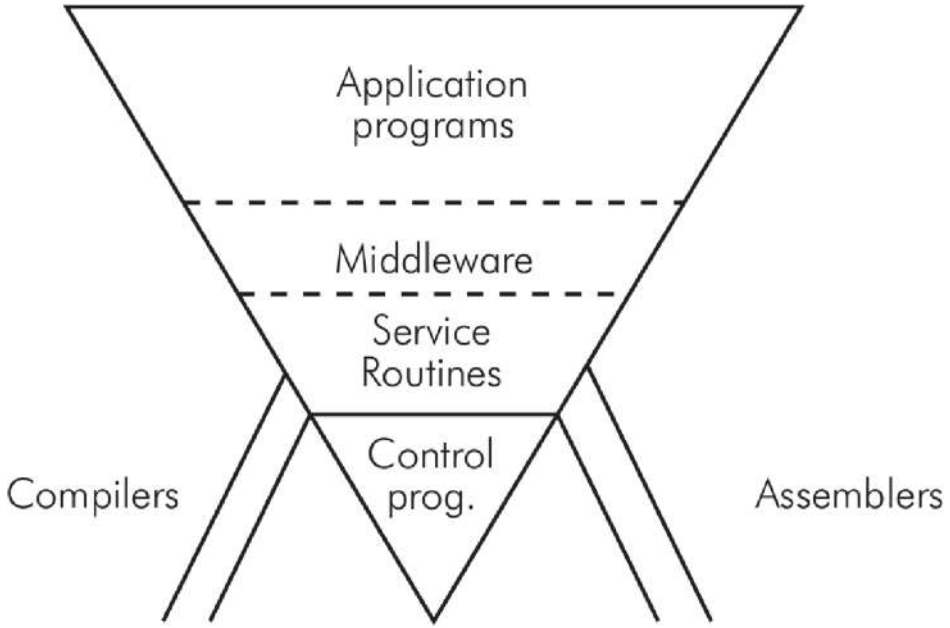
The Middleware idea

Well-known in software engineering

It provides a computational layer

A bridge between the application and the low-level details

It is not a set of API and library



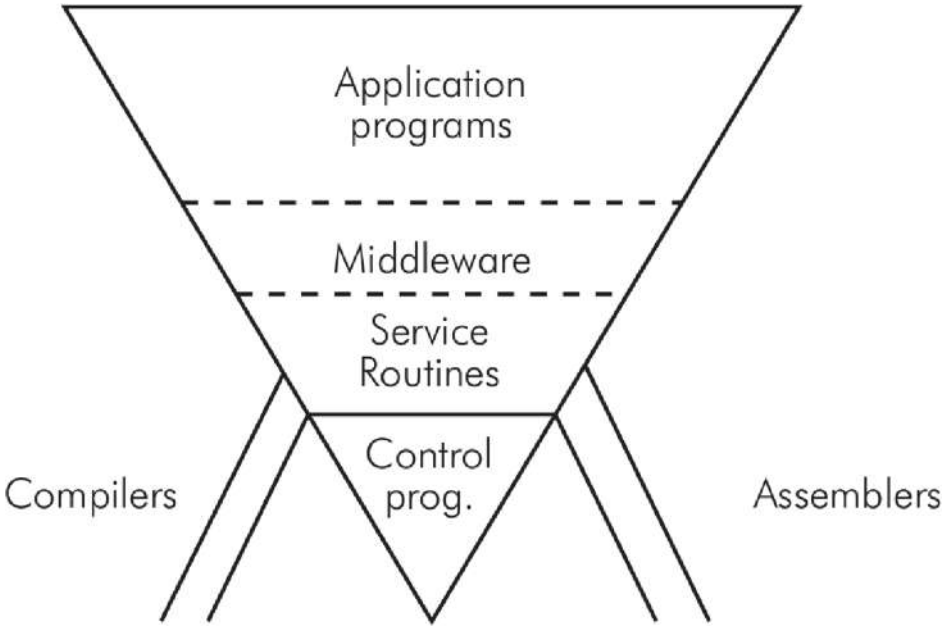
MIDDLEWARE ORIGINS

Issues in developing real robots

Cooperation between hardware and software

Architectural differences in robotics systems

Software reusability and modularity



MIDDLEWARES MAIN FEATURES

Portability*:* provides a common programming model regardless the programming language and the system architecture.

Manage the complexity: low-level aspects are handled by libraries and drivers inside the middleware. It (should) reduce(s) the programming error and decrease the development time.

Reliability: middleware are tested independently. They permit to develop robot controllers without considering the low level details and using robust libraries.

ROBOT MIDDLEWARES: A LIST

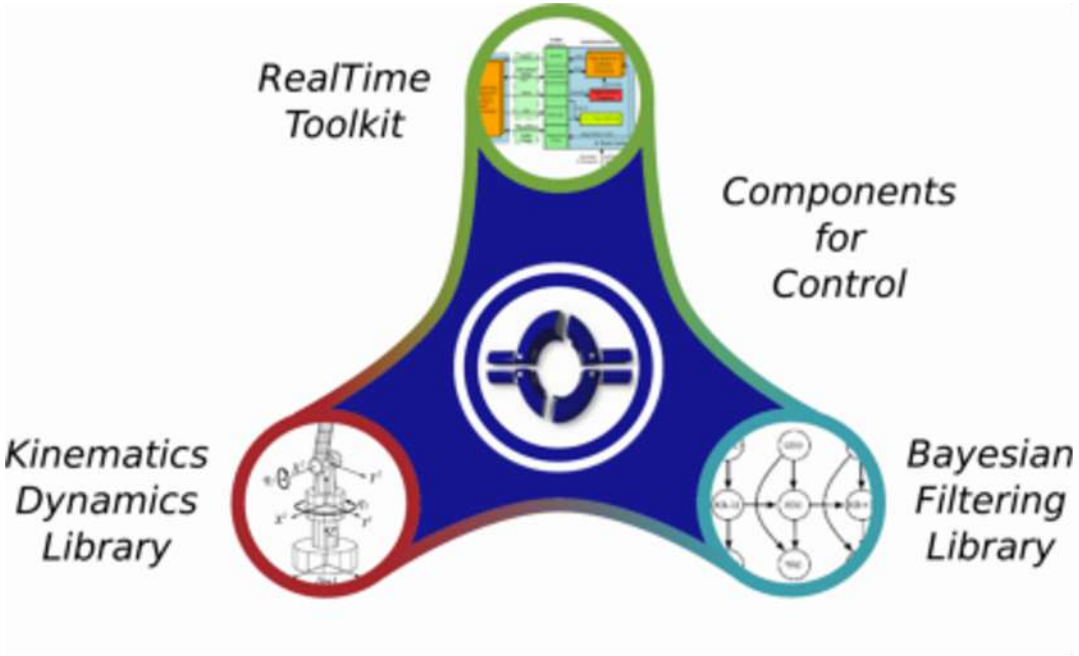
Several middleware have been developed in recent years:

OROCOS	[Europe]
ORCA	[Europe]
YARP	[Europe / Italy]
BRICS	[Europe]
OpenRTM	[Japan]
OpenRave	[US]
ROS	[US]
…	

Let’s see their common features and main differences

OROCOS: OPEN ROBOT CONTROL SOFTWARE

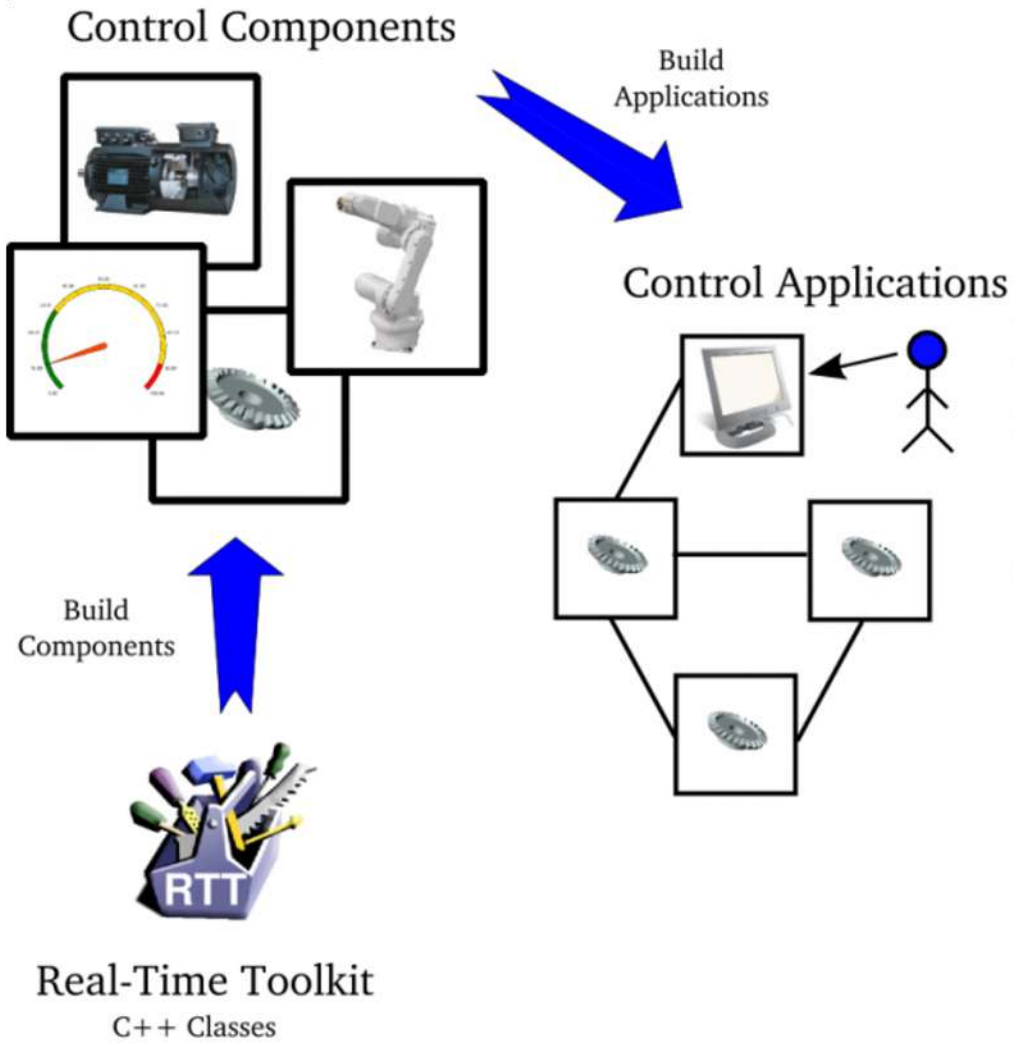
The project started in December 2000 from an initiative of the mailing list EURON then it become an European project with 3 partners: K.U. Leuven (Belgium), LAAS Toulouse (France), KTH Stockholm (Sweden)



OROCOS requirements:

- Open source license
- Modularity and flexibility
- Not related to robot industries
- Working with any kind of device
- Software components for kinematics, dynamics, planning, sensors, controller
- Not related to a unique programming language

OROCOS STRUCTURE



Real-Time Toolkit (RTT)

- infrastructure and functionalities for real-time robot systems
- component-based applications

Component Library (OCL)

- provides ready-to-use components, e.g., device drivers, debugging tools, path planners, task planners

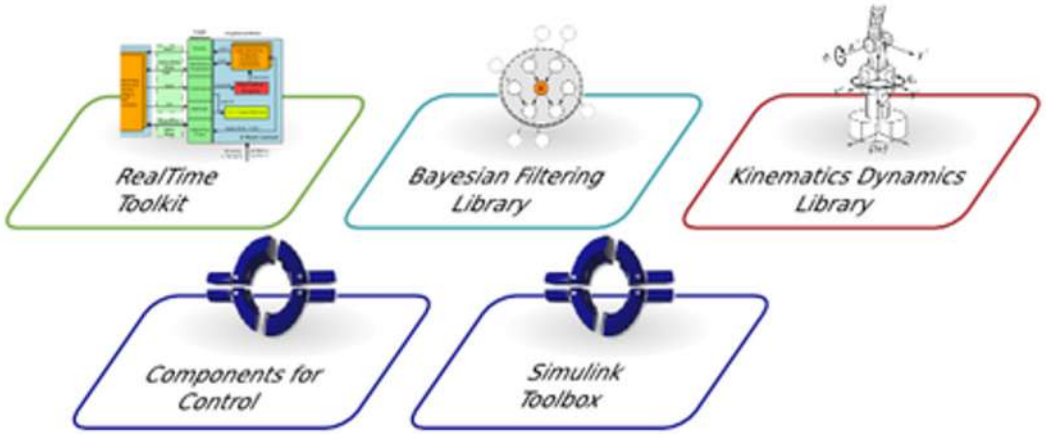
OROCOS STRUCTURE

Bayesian Filtering Library (BFL)

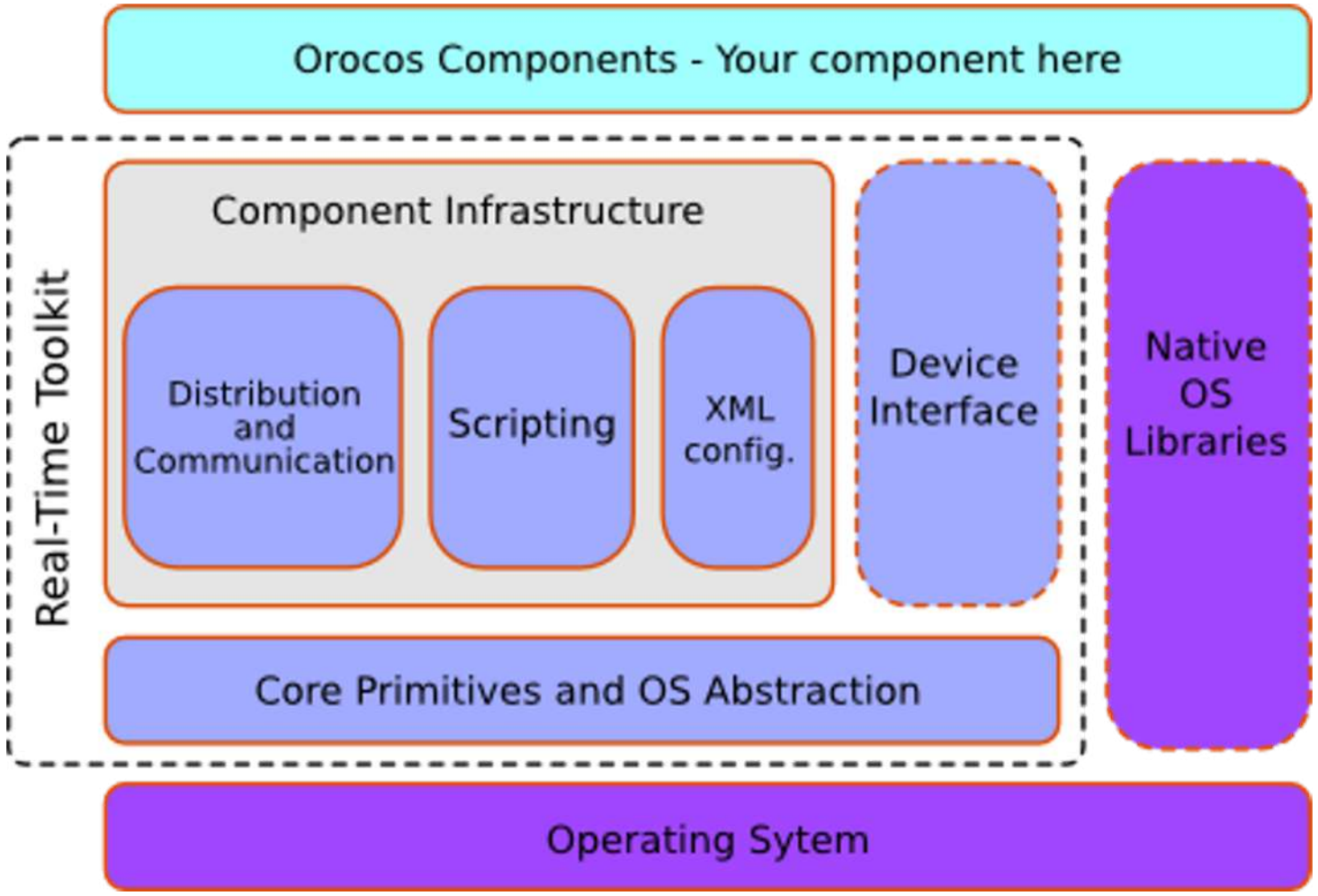
- application independent framework, e.g., (Extended) Kalman Filter, Particle Filter

Kinematics & Dynamics Library (KDL)

- real-time kinematics & dynamics computations



OROCOS RTT FRAMEWORK



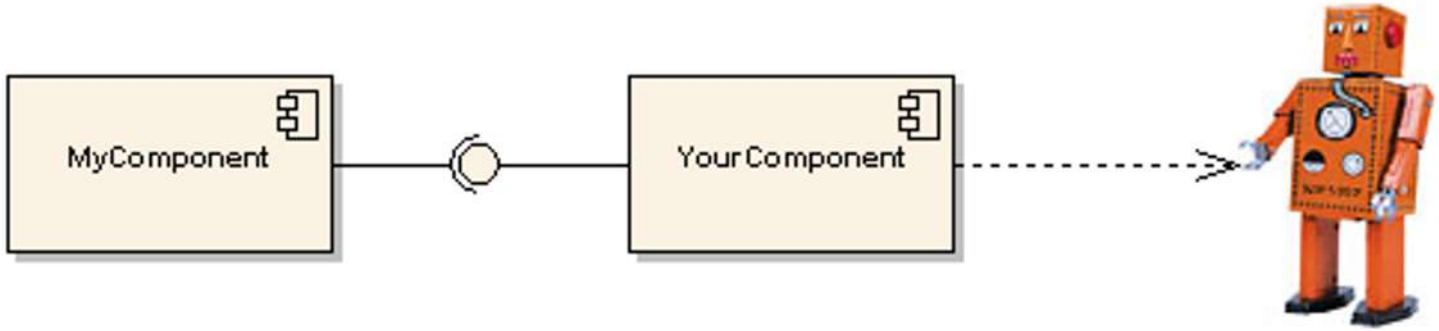
ORCA: COMPONENTS FOR ROBOTICS

The aim of the project is to focus on software reuse for scientific and industrial applications

Key properties:

- **Enable software reuse** defining commonly-use interfaces
- **Simplify software reuse** providing high-level libraries
- **Encourage software reuse** updated software repositories

ORCA defines itself as “unconstrained component-based system”



ORCA AND ICE

The main difference between OROCOS and ORCA is the communication toolkit; OROCOS uses CORBA while ORCA uses ICE

- ICE is a modern framework developed by ZeroC
- ICE is an open-source commercial communication system
- ICE provides two core services
  - IceGrid registry (Naming service): which provides the logic mapping between different components
  - IceStorm service (Event service): which constitute the publisher and subscriber architecture

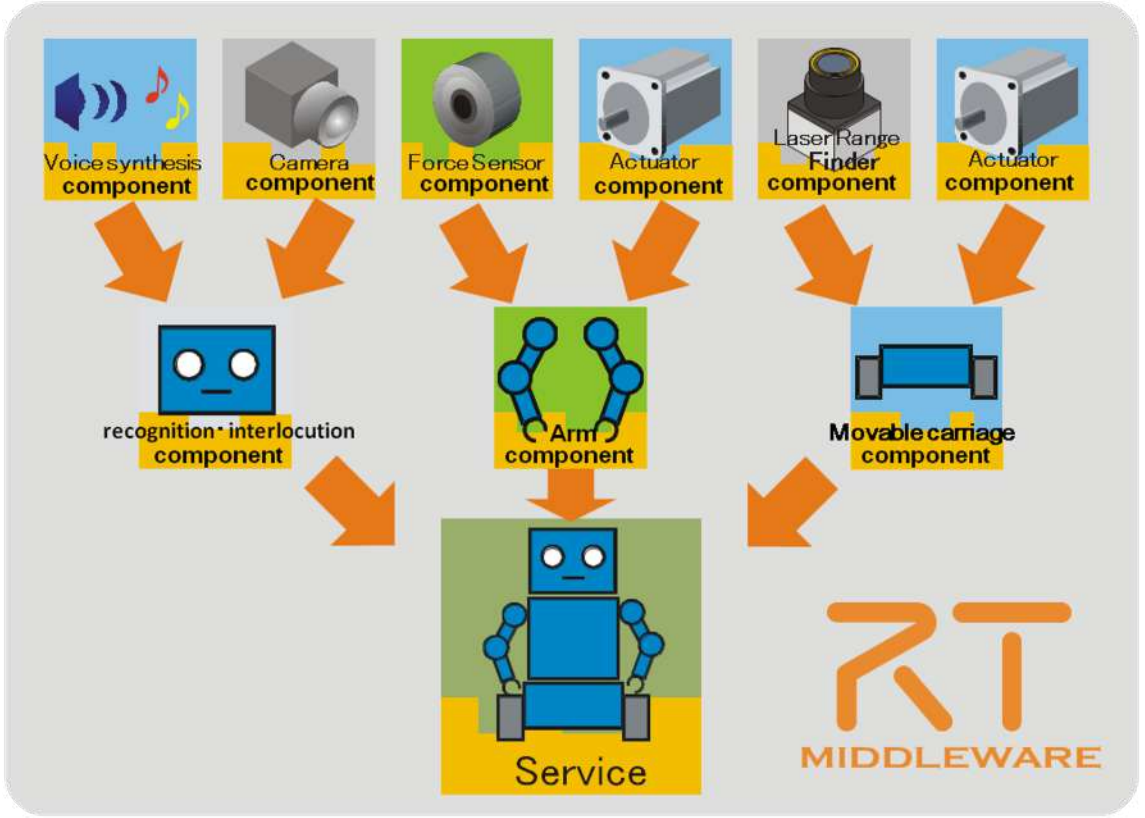
"A component can find the other components through the IceGrid registry and can communicate with them through the IceStorm service."

RT MIDDLEWARE

RT-Middleware (RTM) is a common platform standard to construct the robot system by combining the software modules of the robot functional elements (RTC):

- Camera component
- Stereovision component
- Face recognition component
- Microphone component
- Speech recognition component
- Conversational component
- Head and arm component
- Speech synthesis component

OpenRTM-aist (Advanced Industrial Science & Technology) is based on the CORBA technology to implement RTC extended specification

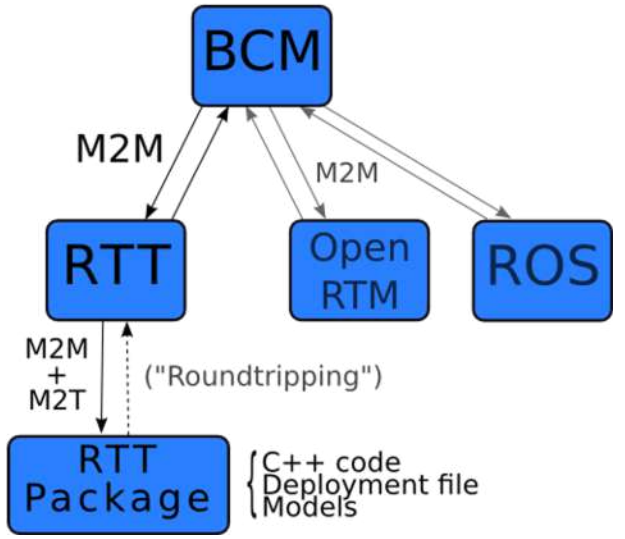


BRICS: BEST PRACTICES IN ROBOTICS

Aimed at find out the "best practices" in the developing of the robotic systems:

- Investigate the weakness of robotic projects
- Investigates the integration between hardware & software
- Promote model driven engineering in robot development
- Design an Integrated Development Environment for robotic projects (BRIDE)
- Define showcases for the evaluation of project robustness with respect to BRICS principles.

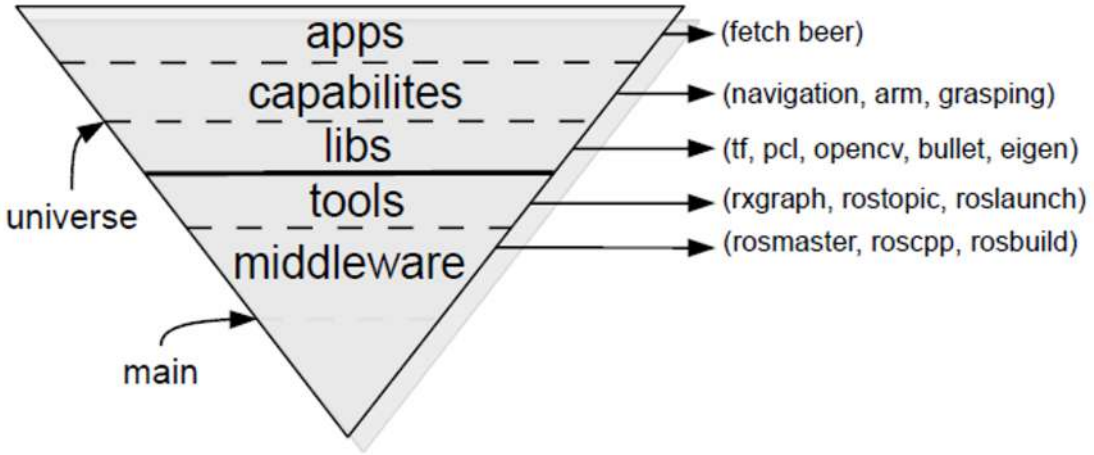
BRICS
BEST PRACTICE IN ROBOTICS



"The prime objective of BRICS is to structure and formalize the robot development process itself and to provide tools, models, and functional libraries, which help accelerating this process significantly."

ROS: ROBOT OPERATING SYSTEM

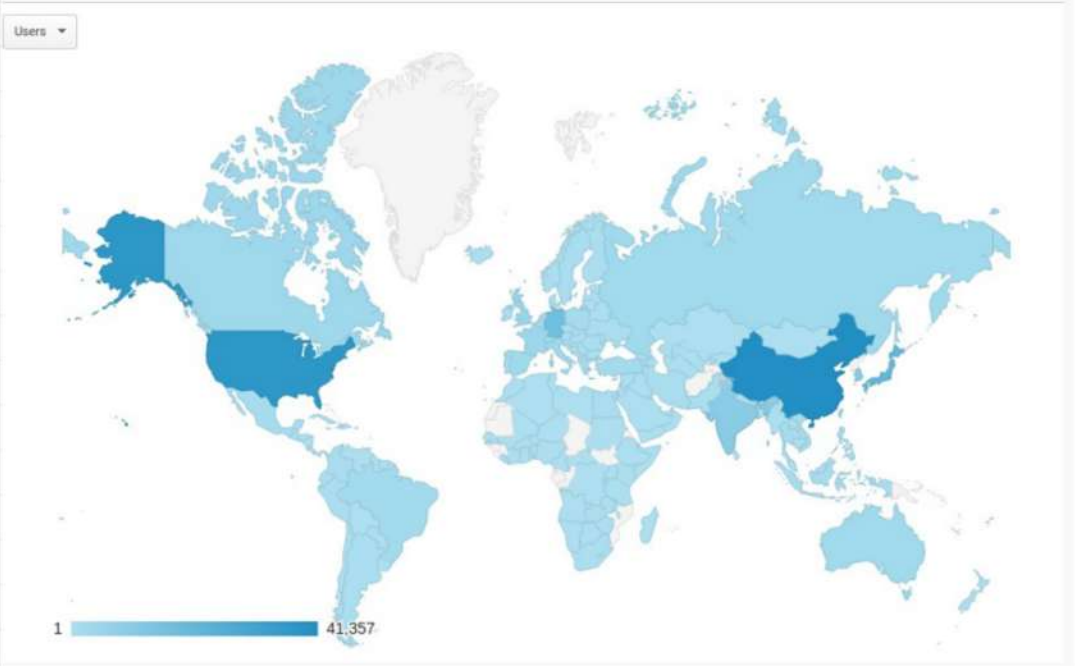
Presented in 2009 by Willow Garage, is a meta-operating system for robotics with a rich ecosystem of tools and programs



WHY ROS?

1.	China	41,357	(19.88%)
2.	United States	36,531	(17.56%)
3.	Japan	19,738	(9.49%)
4.	Germany	15,525	(7.46%)
5.	South Korea	9,382	(4.51%)
6.	India	9,345	(4.49%)
7.	United Kingdom	4,972	(2.39%)
8.	Taiwan	4,856	(2.33%)
9.	France	4,056	(1.95%)
10.	Canada	3,854	(1.85%)
11.	Singapore	3,516	(1.69%)
12.	Italy	3,464	(1.66%)
13.	Russia	3,207	(1.54%)
14.	Australia	3,114	(1.50%)
15.	Spain	3,080	(1.48%)
16.	Hong Kong	2,941	(1.41%)
17.	Brazil	2,548	(1.22%)
18.	Turkey	2,253	(1.08%)
19.	Netherlands	1,822	(0.88%)
20.	Poland	1,820	(0.87%)
21.	Philippines	1,711	(0.82%)
22.	Thailand	1,585	(0.76%)
23.	Switzerland	1,478	(0.71%)
24.	(not set)	1,429	(0.69%)
25.	Indonesia	1,345	(0.65%)

wiki.ros.org visitor locations:



Source: Google Analytics
Site: wiki.ros.org in July 2019

ROS has grown to include a large community of users worldwide

The community of developer is one of the most important characteristics of ROS

A LOT OF RESOURCES

ROS Wiki

- Archive for the existing ROS component
- Installation and configuration guides
- Information about the middleware itself
- Lots of tutorials

ROS Q&A

- For specific problems
- Thousand of already answered questions
- Active community
- Like *Stack Overflow* for ROS

SOME NUMBERS

ROS wiki:

- pages: 17058
- edits: 14,7/day
- views: 44794/day

ROS Q&A:

- total Q: 30243
- total A: 21697
- avg Q: 17,2/day

ROS deb:

- total DL: 8441279
- unique DL: 7582
- unique IP: 113345

ROBOT AND RESEARCH

Total number of papers citing
ROS: an open-source Robot Operating System
(Quigley et al., 2009)
5875 (22% increase)
6703 (scholar)

ROS Industrial

ROS INTRODUCTION

ROBOTICS

POLITECNICO
MILANO 1863

ROS: ROBOT OPERATING SYSTEM

ROS main features:

- Distributed framework
- Reuse code
- Language independent
- Easy testing on Real Robot & Simulation
- Scaling

ROS Components

- File system tools
- Building tools
- Packages
- Monitoring and GUIs
- Data Logging

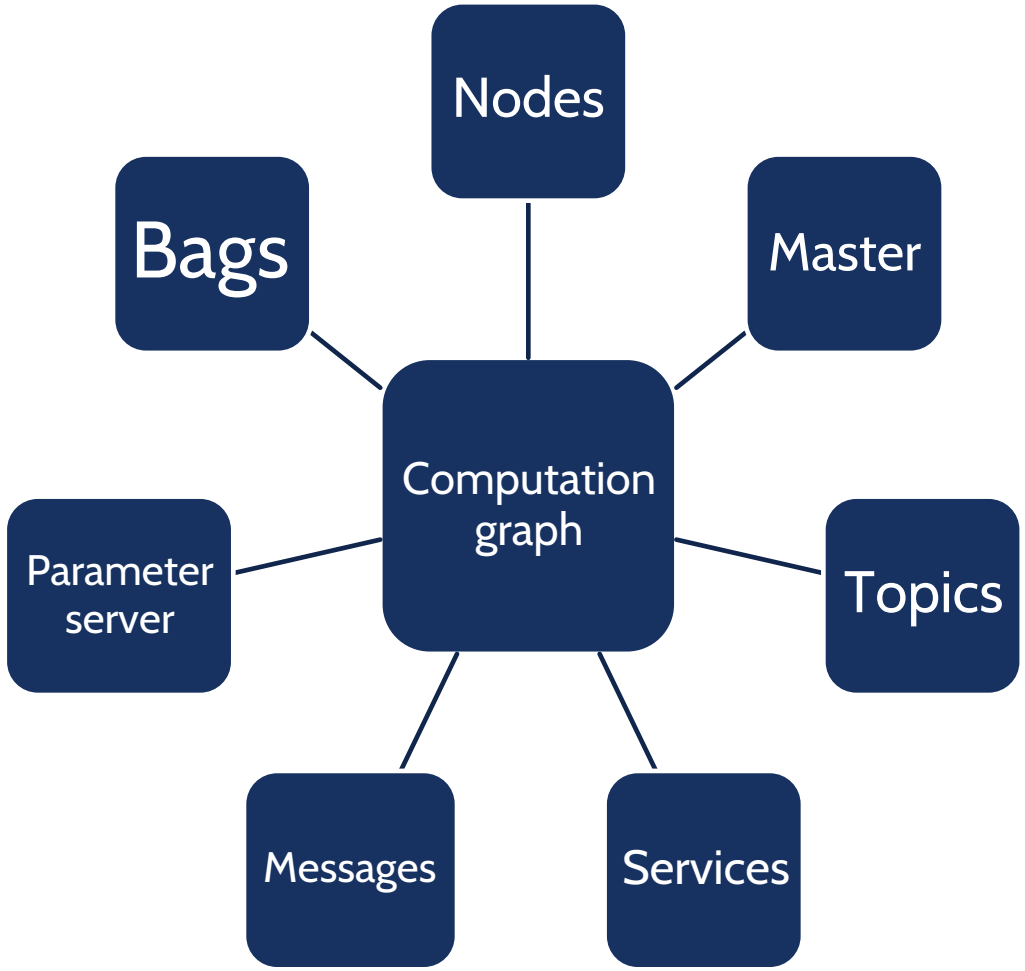
ROS

ROS COMPUTATION GRAPH

ROBOTICS

POLITECNICO
MILANO 1863

ROS STRUCTURE: COMPUTATIONAL GRAPH



The *Computation Graph* is the peer-to-peer network of ROS processes that are processing data together.

NODES

Executable unit of ROS:

- Scripts for Python
- Compiled source code for C++

Process that performs computation

Nodes exchange information via the graph

Meant to operate at fine-grained scale

A robot system is composed by various nodes

```
rosrun package_name node_name
```

```
rosrun turtlesim turtlesim_node
```

MASTER

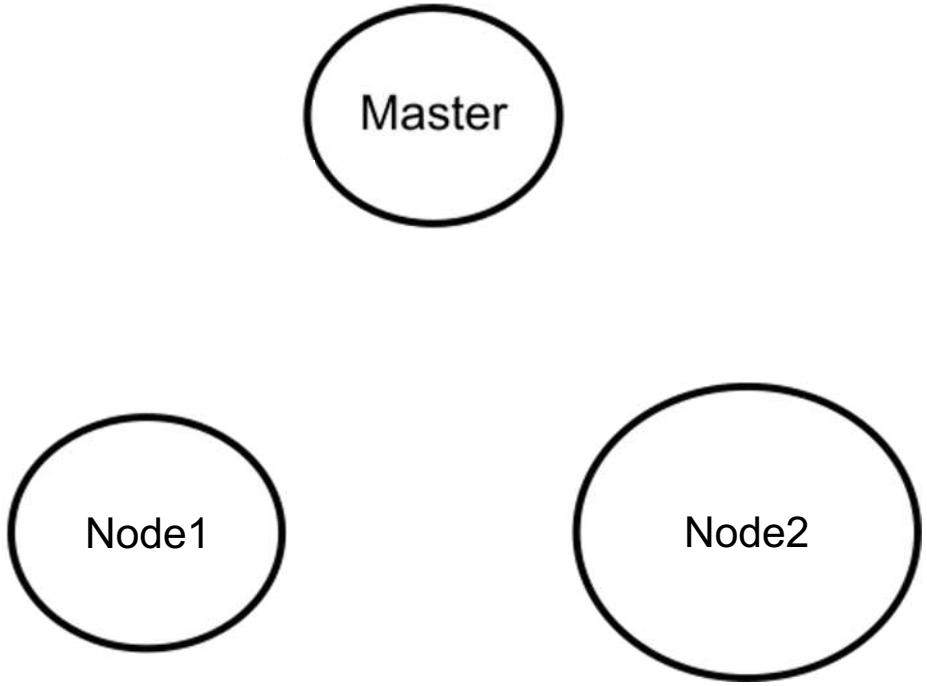
Provides naming and registration services

Essential for nodes interactions

One master for each system, even on distributed architectures

Enables individual ROS nodes to locate one another

One of the functionalities provided by `roscore`



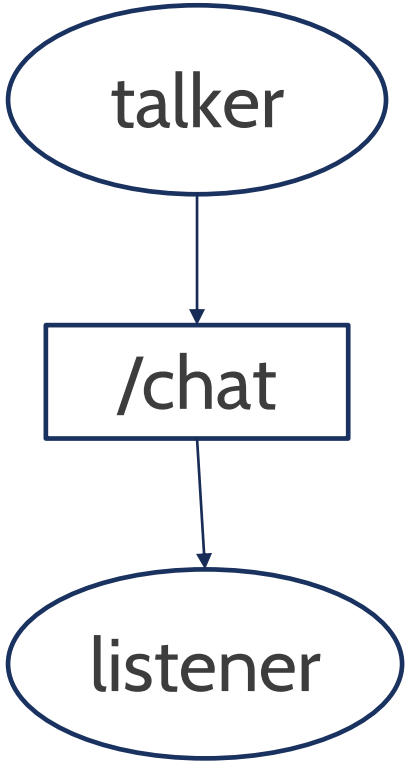
TOPICS

Named channels for communication

Implement the publish/subscribe paradigm

No guarantee of delivery

Have a specific message type



TOPICS

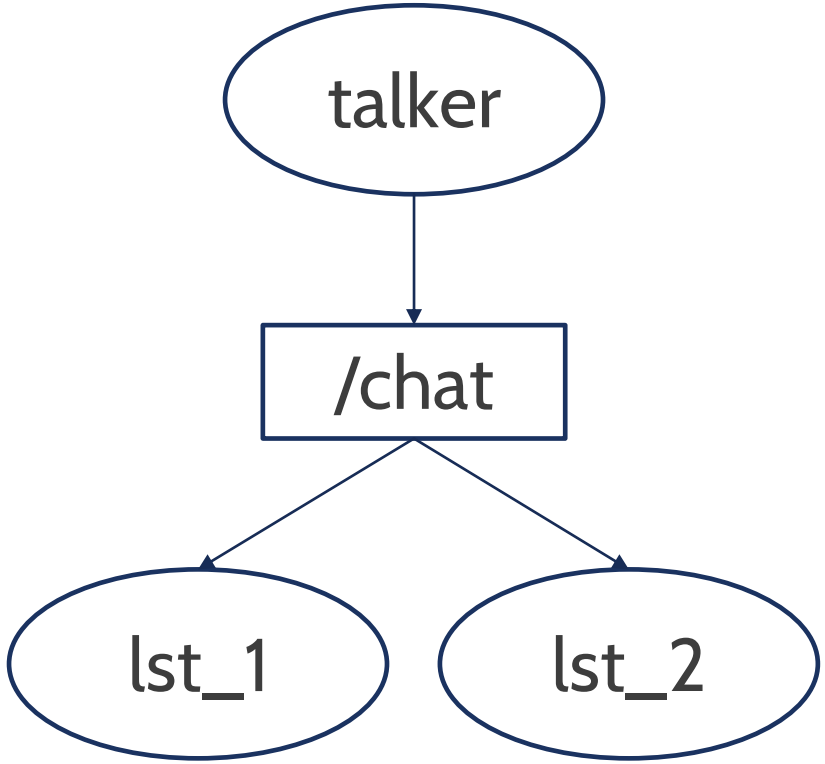
Named channels for communication

Implement the publish/subscribe paradigm

No guarantee of delivery

Have a specific message type

Multiple nodes can publish messages on a topic



TOPICS

Named channels for communication

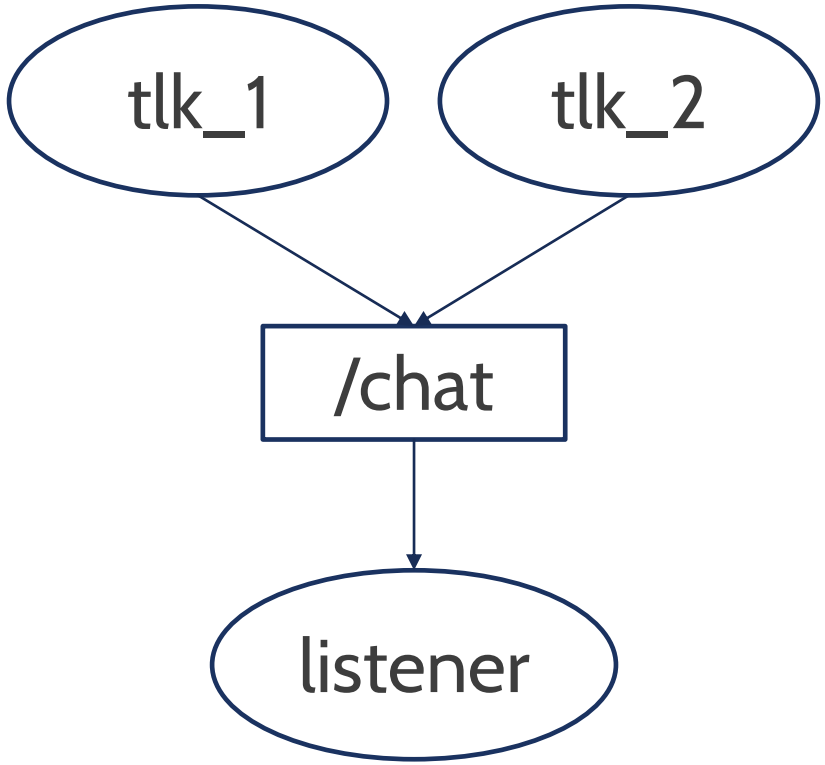
Implement the publish/subscribe paradigm

No guarantee of delivery

Have a specific message type

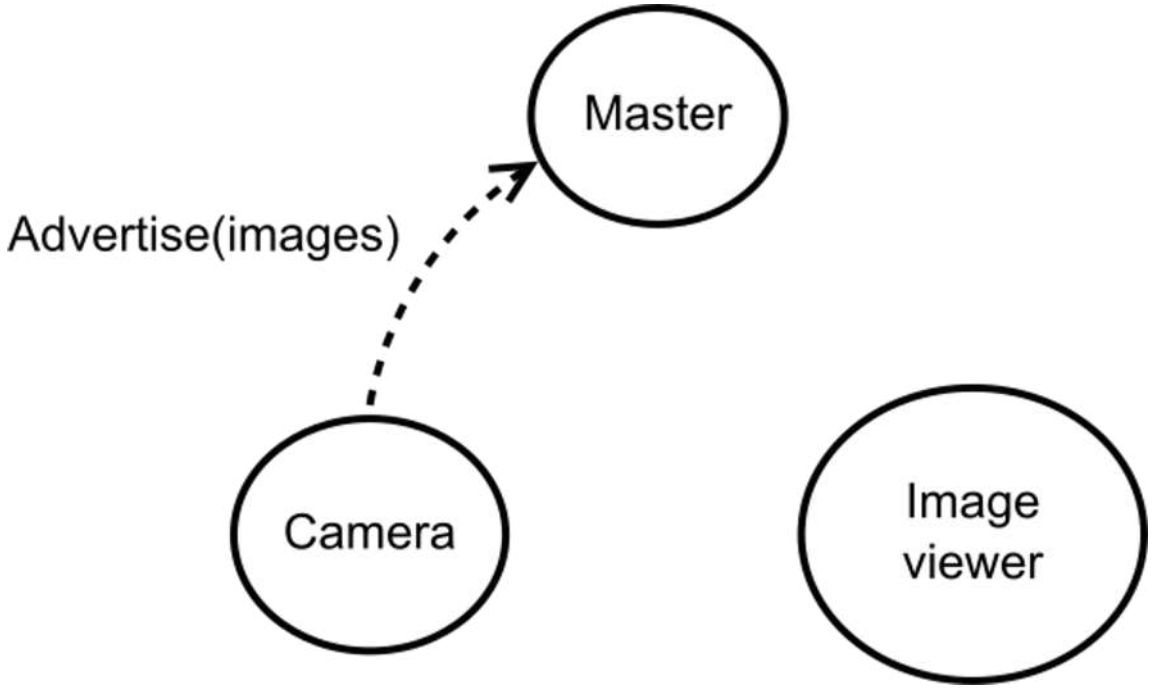
Multiple nodes can publish messages on a topic

Multiple nodes can read messages from a topic

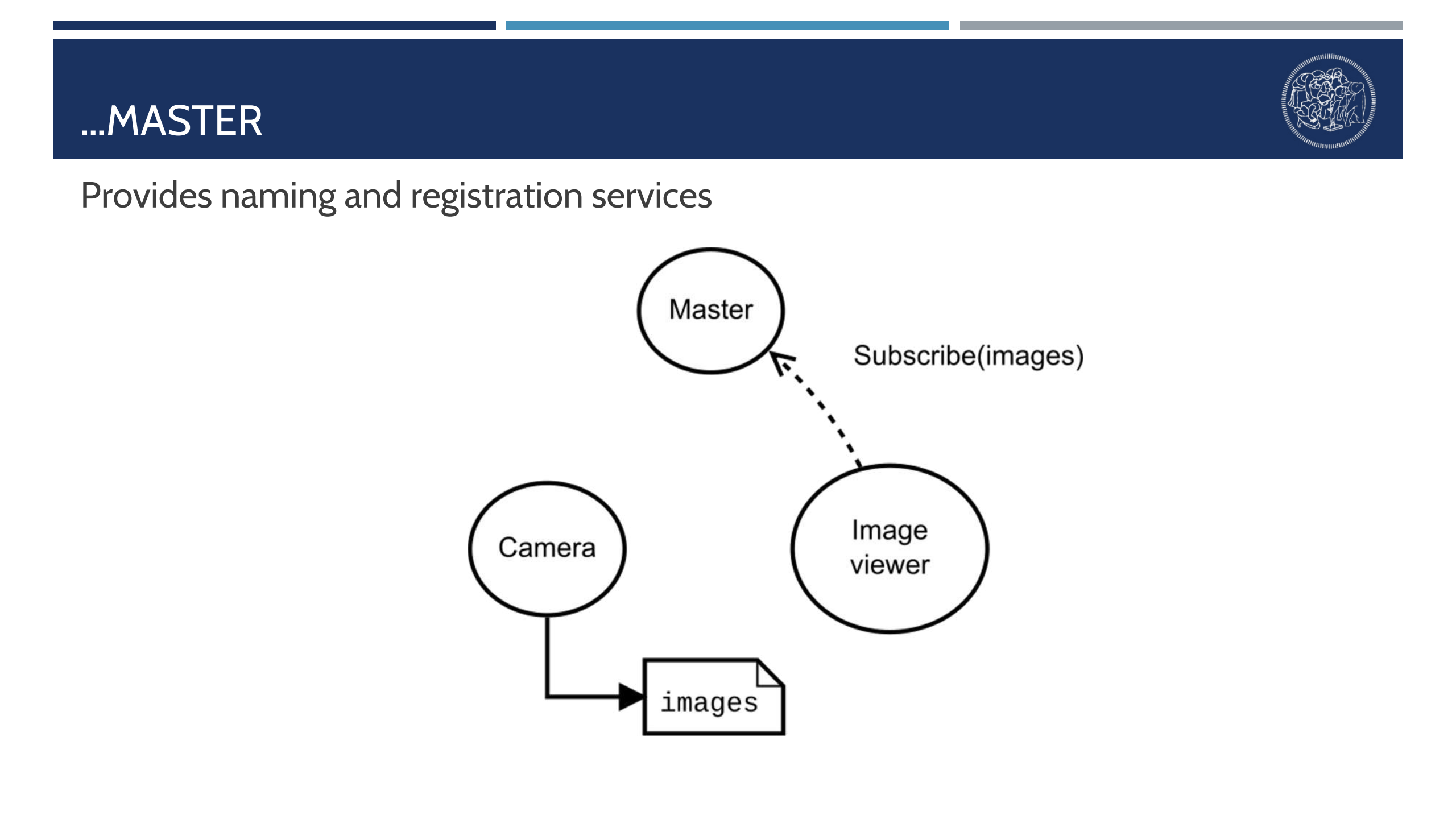


...MASTER

Provides naming and registration services



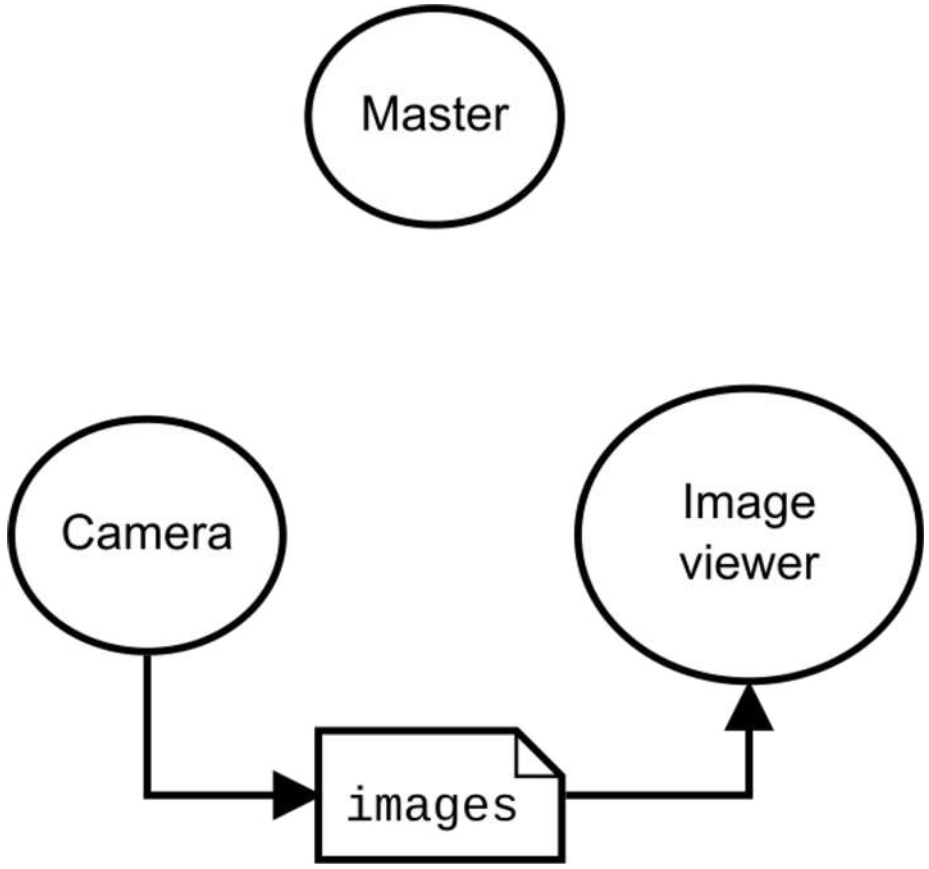
...MASTER



Provides naming and registration services

...MASTER

Provides naming and registration services



MESSAGES

Messages are exchanged on topics

They define the type of the topic

Various already available messages

It is possible to define new messages using a simple language

Existing message types can be used in new messages together with base types

std_msgs/String.msg

```
string data
```

std_msgs/Header.mgs

```
uint32 seq
time stamp
string frame_id
```

sensor_msgs/Joy.msg

```
std_msgs/Header header
float32[] axes
int32[] buttons
```

MESSAGES

Messages are exchanged on topics

They define the type of the topic

Various already available messages

It is possible to define new messages using a simple language

Existing message types can be used in new messages together with base types

Quick recap:

14 base types

32 std_msgs

29 geometry_msgs

26 sensor_msgs

...and more

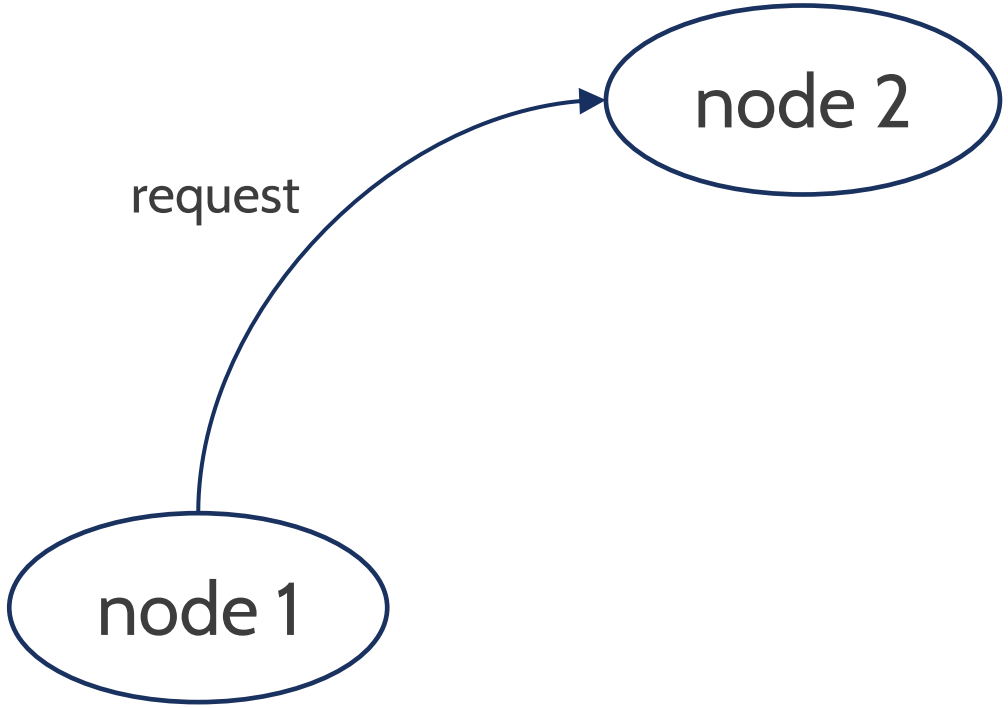
SERVICES

Work like remote function calls

Implement the client/server paradigm

Code waits for service call to complete

Guarantee of execution



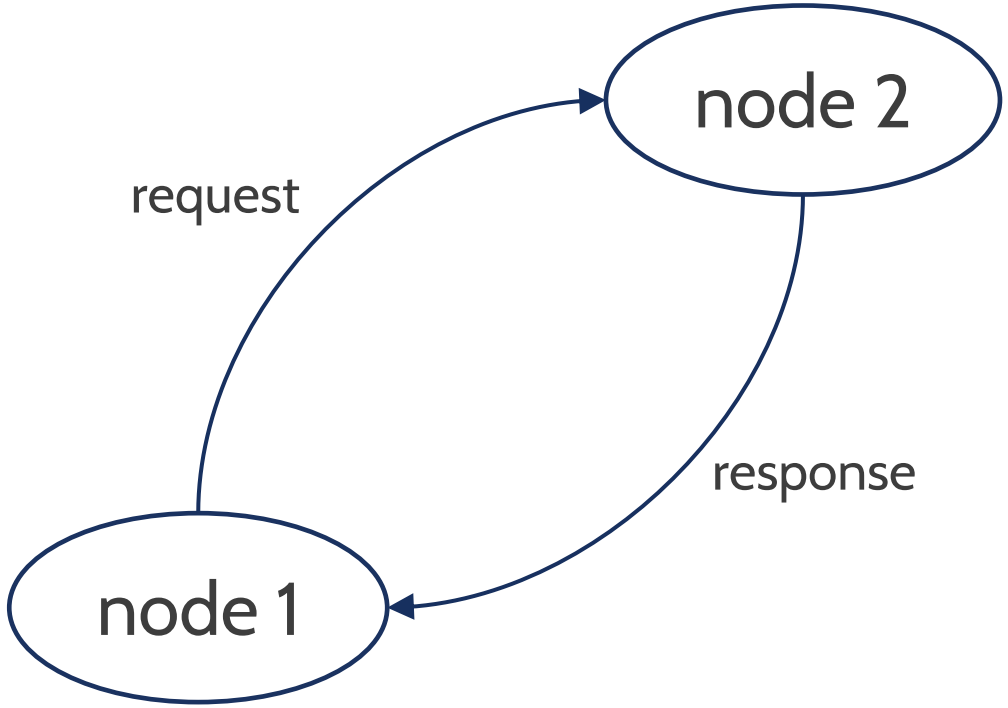
SERVICES

Work like remote function calls

Implement the client/server paradigm

Code waits for service call to complete

Guarantee of execution



SERVICES

Work like remote function calls

Implement the client/server paradigm

Code waits for service call to complete

Guarantee of execution

Use of message structures

example/AddTwoInt.srv

```
int64 A
int64 B
---
int64 Sum
```

request

response

PARAMETER SERVER

Shared, multivariable dictionary that is accessible via network

Nodes use this server to store and retrieve parameters at runtime

Not designed for performance, not for data exchange

Connected to the `master`, one of the functionalities provided by `roscore`

name	value
/gains/P	10.0
/gains/I	1.0
/gains/D	0.1
use_sim_time	True

```
rosparam [set|get] name value

rosparm set use_sim_time True

rosparam get use_sim_time
  > True
```

PARAMETER SERVER

Shared, multivariable dictionary that is accessible via network

Nodes use this server to store and retrieve parameters at runtime

Not designed for performance, not for data exchange

Connected to the master, one of the functionalities provided by roscore

Available types:

- 32-bit integers
- Booleans
- Strings
- Doubles
- ISO8601 dates
- Lists
- Base64-encoded binary data

BAGS

File format (**.bag*) for storing and playing back messages

Primary mechanism for data logging

Can record anything exchanged on the ROS graph (messages, services, parameters, actions)

Important tool for analyzing, storing, visualizing data and testing algorithms.

```
rosbag record -a
rosbag record /topic1 /topic2

rosbag play ~/bags/fancy_log.bag

rqt_bag ~/bags/fancy_log.bag
```

ROSCORE

`roscore` is a collection of nodes and programs that are pre-requisites of a ROS-based system

Must be running in order for ROS nodes to communicate

Launched using the `roscore` command.

Elements of `roscore`:

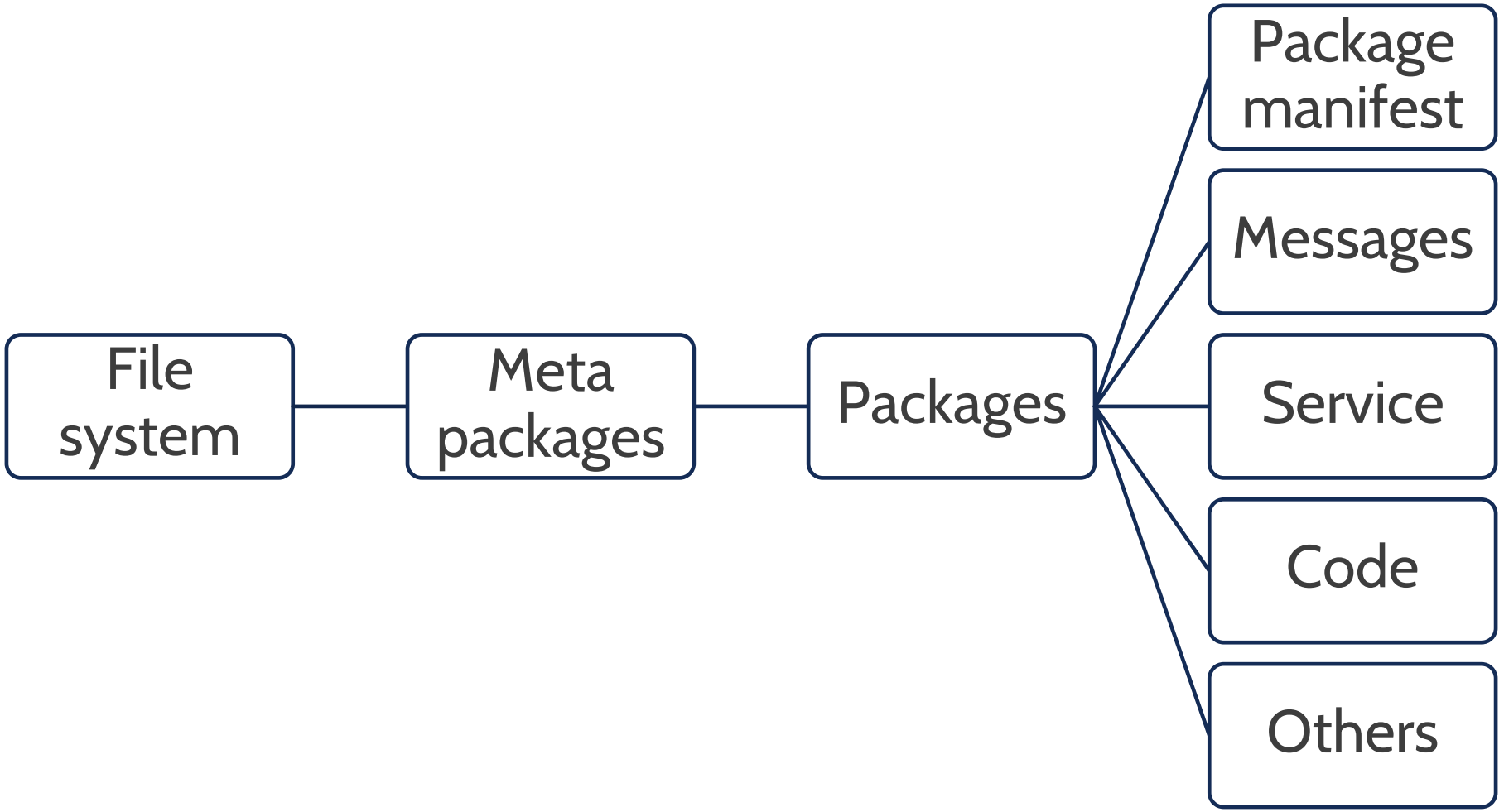
- a ROS Master
- a ROS Parameter Server
- a rosout logging node

ROS FILESYSTEM

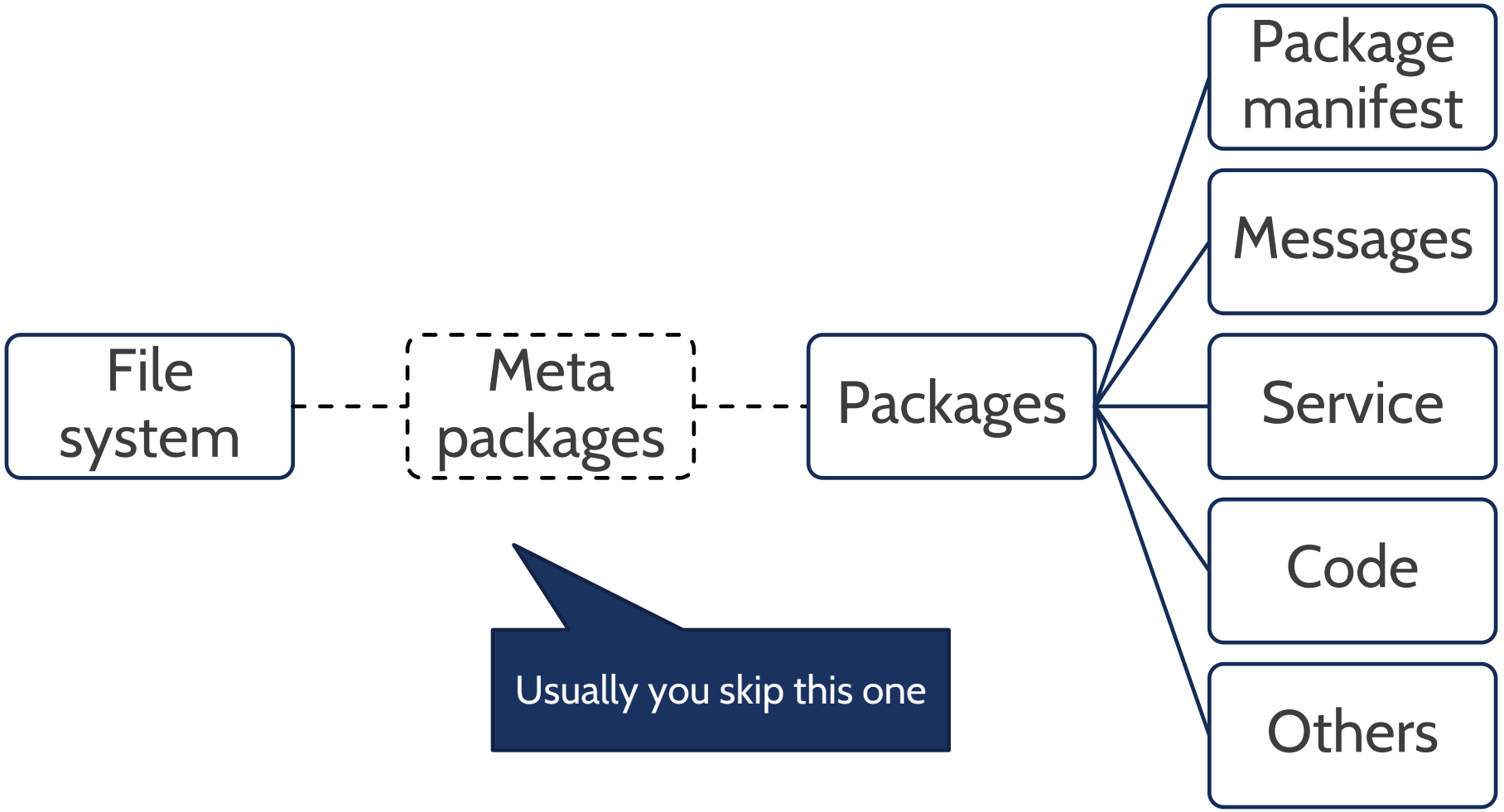
ROBOTICS

POLITECNICO
MILANO 1863

ROS FILE SYSTEM



ROS FILE SYSTEM



PACKAGES AND METAPACKAGES

PACKAGES

- Atomic element of ROS file system
- Used as a reference for most ROS commands
- Contains nodes, messages and services
- `package.xml` used to describe the package
- Mandatory container

METAPACKAGES

- Aggregation of logical related elements
- Not used when navigating the ROS file system
- Contains other packages
- `package.xml` used to describe the package
- Not required

STRUCTURE OF A PACKAGE

Folder structure:

`/src`, `/include`, `/scripts` (coding)

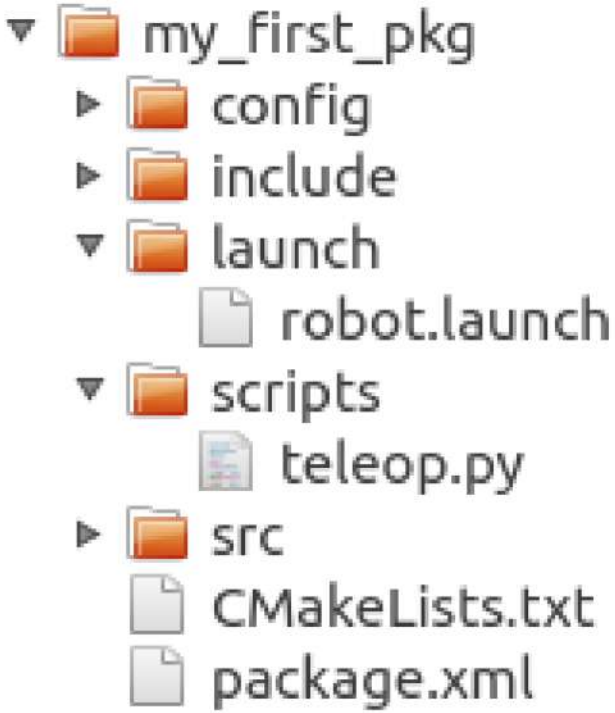
`/launch` (launch files)

`/config` (configuration files)

Required files:

`CMakeList.txt`: Build rules for catkin

`package.xml`: Metadata for ROS



ROS INSTALLATION

ROBOTICS

POLITECNICO
MILANO 1863

INSTALLATION GUIDELINES

If your machine currently runs:

- Linux
  - Ubuntu 18.04 (suggested) → You're good, install ROS Melodic
  - Ubuntu 20.04 → You should be good, install ROS Noetic (advanced packages might not work properly)
  - Another distro or Ubuntu version → Install Ubuntu 18.04 (you probably already know how to do it ☺)
- Windows
  - Dual boot Windows + Ubuntu 18.04 on your internal disk (suggested) → several guides online, e.g. this
  - Install Ubuntu 18.04 on an external disk (caution: experimental) → guide (but use Ubuntu 18.04)
  - Virtual Machine (strongly discouraged) → e.g. VirtualBox + several guides online, e.g. this
- macOS
  - Virtual Machine (suggested, still bad) → e.g. VirtualBox + several guides online, e.g. this
  - [[Previous to 2019: you might try installing Ubuntu 18.04 on an external disk (at your own risk!) → guide]]

INSTALLATION GUIDELINES

If your machine currently runs:

Make backups!

- Linux
  - Ubuntu 18.04 (suggested) → You're good, install ROS Melodic
  - Ubuntu 20.04 → You should be good, install ROS Noetic (advanced packages might not work properly)
  - Another distro or Ubuntu version → Install Ubuntu 18.04 (you probably already know how to do it ☺)
- Windows
  - Dual boot Windows + Ubuntu 18.04 on your internal disk (suggested) → several guides online, e.g. this
  - Install Ubuntu 18.04 on an external disk (caution: experimental) → guide (but use Ubuntu 18.04)
  - Virtual Machine (strongly discouraged) → e.g. VirtualBox + several guides online, e.g. this
- macOS
  - Virtual Machine (suggested, still bad) → e.g. VirtualBox + several guides online, e.g. this
  - [[Previous to 2019: you might try installing Ubuntu 18.04 on an external disk (at your own risk!) → guide]]

INSTALLATION

This instruction are for:

Ubuntu 18.04 (strongly suggested)

INSTALLATION

Check on ROS site for updated commands:

http://wiki.ros.org/melodic/Installation/Ubuntu

Initial setup for sources and keys for downloading the packages

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc)
main" > /etc/apt/sources.list.d/ros-latest.list'
```

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key
C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

```
google.
```

INSTALLATION

Update the packaged index

```
sudo apt-get update
```

Choose between the four pre-packaged ROS installation

Desktop-Full Install: `sudo apt-get install ros-melodic-desktop-full`

Desktop Install: `sudo apt-get install ros-melodic-desktop`

ROS-Base: `sudo apt-get install ros-melodic-ros-base`

INSTALLATION

How to install single packages:

```
sudo apt-get install ros-melodic-PACKAGE
```

Example

```
sudo apt-get install ros-melodic-slam-gmapping
```

To find the exact name of a package you can use the usual aptitude search:

```
apt-cache search ros-melodic
```

INITIALIZATION

`rosdep` enables you to easily install system dependencies and it's required by some ROS packages

```
sudo rosdep init
rosdep update
```

To use `catkin` (the compiling environment of ROS) you need to define the location of your ROS installation.

In each new terminal type:

```
source /opt/ros/melodic/setup.bash
```

Or put it inside your .bashrc

```
echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

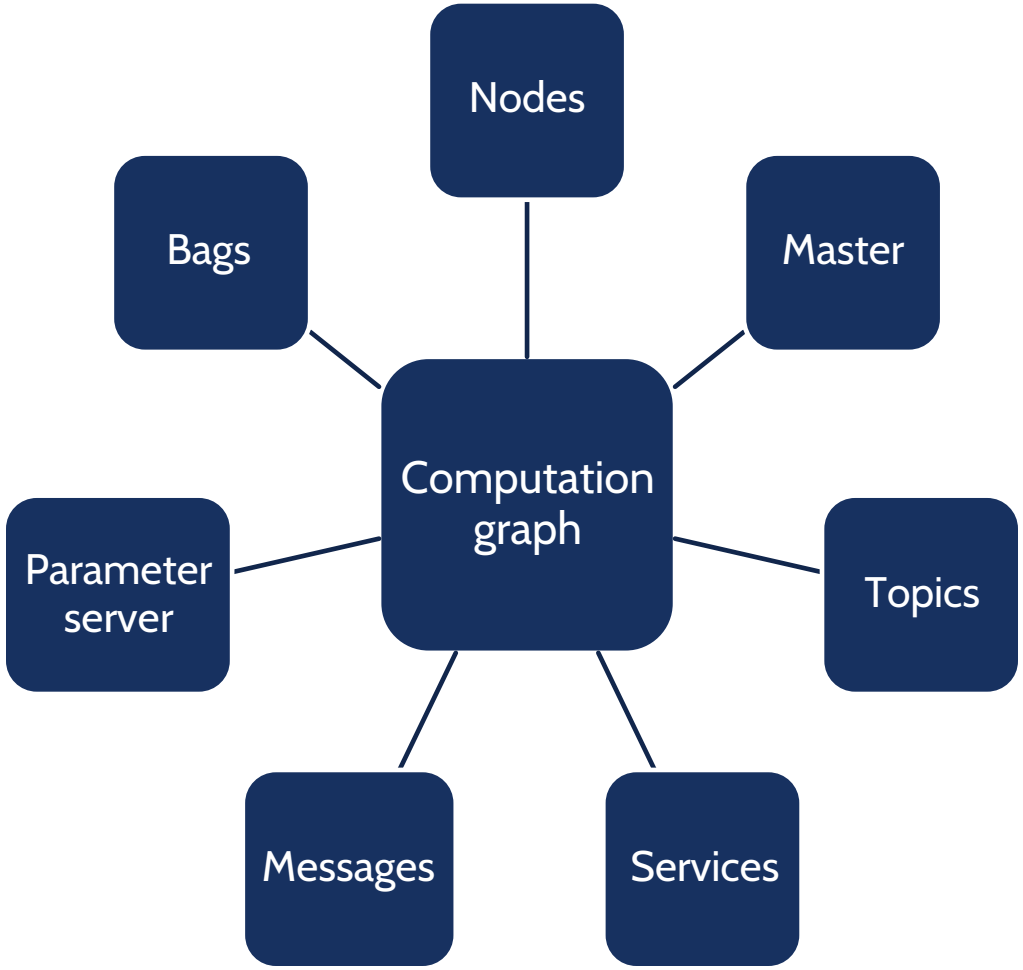
Robotics

Lab session 2

POLITECNICO
MILANO 1863

Paolo Cudrano
paolo.cudrano@polimi.it

RECAP

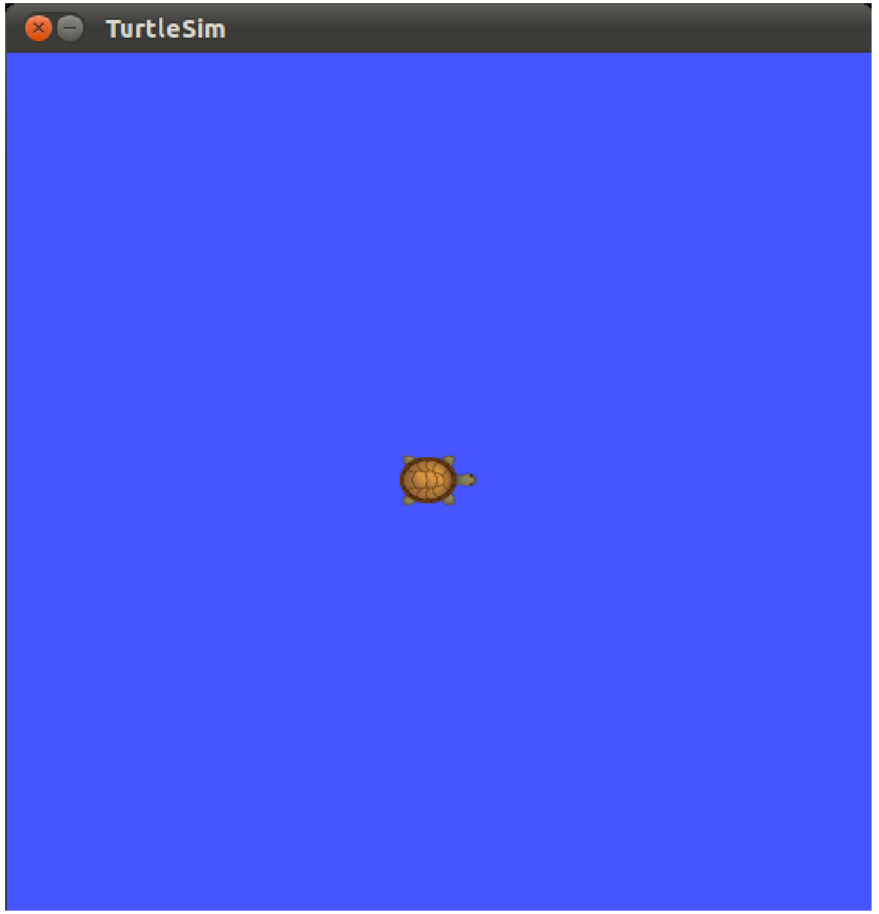

Nodes
Bags
Master
Computation graph
Parameter server
Topics
Messages
Services

TURTLESIM

ROBOTICS

POLITECNICO
MILANO 1863

TURTLESIM



Ros `desktop-full` comes with a lot of tutorials and tools

Before creating our own package and writing our own code, we will learn how to navigate the ROS file system and test how the ROS network works using the `turtlesim` package

`turtlesim` is a tool made for teaching ROS and ROS packages.

FILE SYSTEM TOOLS

Change directory in the ROS file system

```
roscd [package_name[/subdir]]
```

E.g.

```
roscd roscpp && pwd          /opt/ros/kinetic/share/roscpp
roscd roscpp/srv             /opt/ros/kinetic/share/roscpp/srv
roscd robby_roboto           ~/catkin_ws/src/robby_roboto
```

FILE SYSTEM TOOLS

Getting information about installed packages

```
rospack <subcommand> [options] [package]
```

Subcommands (among the others):

`depends [package]`	package dependencies
`find [package]`	find package directory
`list`	list available packages
`profile`	scan all workspace and index packages

E.g.

```
rospack find roscpp     /opt/ros/kinetic/share/roscpp
rospack list            <several packages>
```

STARTING THE MIDDLEWARE

To start the ROS middleware just type in a terminal

```
roscore
```

Now it is possible to display information about the elements currently running

```
rosnode list
rostopic list
rostopic echo /rosout
rosservice list
rqt_graph
```

DEALING WITH NODES

Getting information about running nodes

rosnode <command> [other_commands]

subcommands (among the others)

ping	test connectivity to node
info	print information about node
kill	kill a running node
cleanup	purge registration information of unreachable nodes

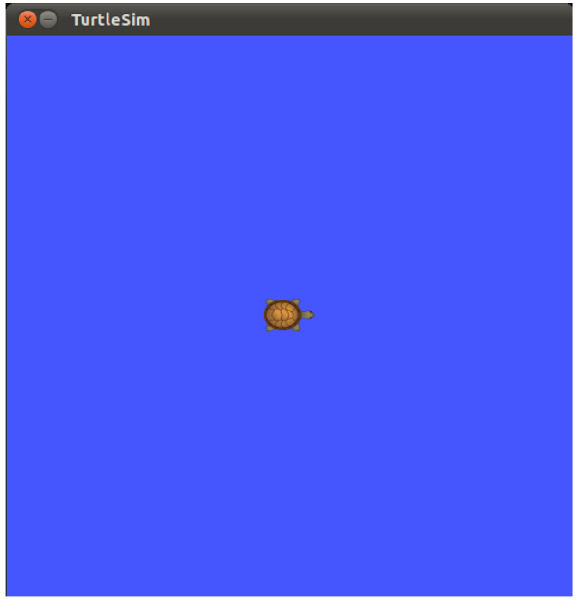
```
rosnode list
rosnode info /rosout
```

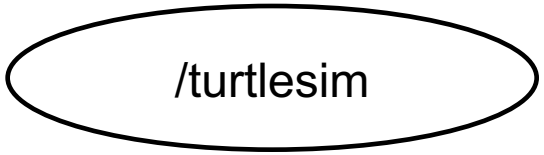
STARTING ROS NODES

To start a ROS node type in a terminal

rosrun [package_name] [node_name]

```
rosrun turtlesim turtlesim_node
rosnode ping /turtlesim
rosnode info /turtlesim
```

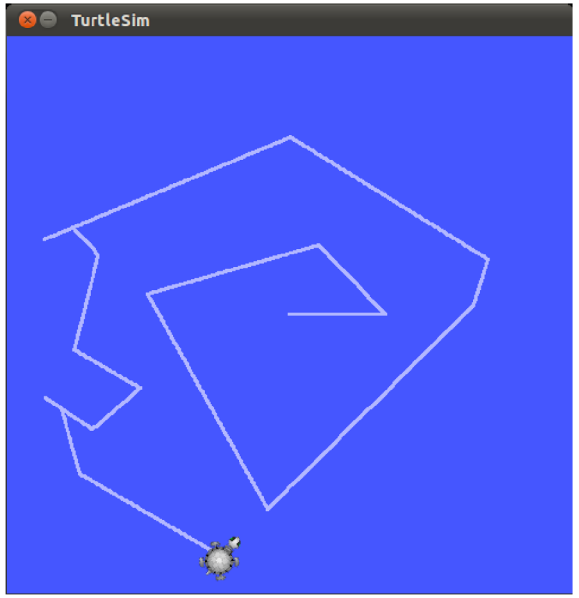




STARTING ROS NODES

In a new terminal

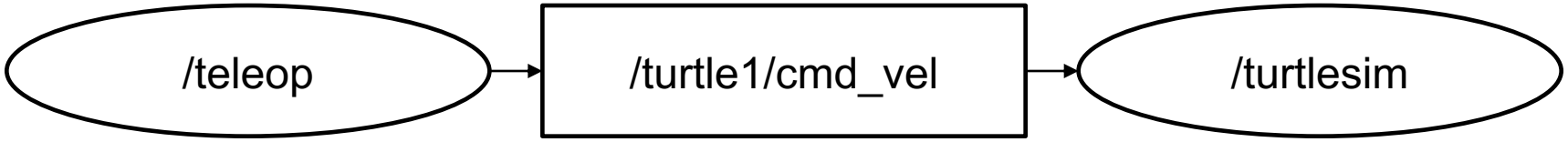
```
rosrun turtlesim turtle_teleop_key
```



Notes:

`turtle_teleop_key` is publishing the key strokes on a topic

`turtlesim` subscribes to the same topic to receive the key strokes



DEALING WITH TOPICS

To show the running node type in a terminal

```
rqt_graph
```

To plot published data on a topic

```
rqt_plot /turtle1/pose/x /turtle1/pose/y
rqt_plot /turtle1/pose/x:y
```

To monitor a topic on a terminal type

```
rostopic echo /turtle1/cmd_vel
```

DEALING WITH TOPICS

Getting information about ROS topics

`rostopic <command> [topic_name]`

Subcommands (among the others)

`echo`	print messages to screen
`find`	find topics by type
`hz`	display publishing rate of topic
`info`	print information about active topic
`list`	list active topics
`pub`	publish data to topic
`type`	print topic type

DEALING WITH TOPICS

Getting information about ROS topics

```
rostopic type [topic_name]

rostopic type /turtle1/cmd_vel
```

Publishing ROS topics

```
rostopic pub [topic] [msg type] [args]
$ rostopic pub my_topic std_msgs/String "hello there“

$ rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x: 2.0, y:
0.0, z: 0.0}, angular: {x: 0.0,y: 0.0,z: 0.0}}'
```

DEALING WITH TOPICS

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0,
0.0]' '[0.0, 0.0, 1.8]'
```

The -1 option force rostopic to publish the message only once, if you want to publish the message at a specific frequency you will use:

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0,
0.0]' '[0.0, 0.0, 1.8]'
```

Where the -r 1 option specify that the message will be published at 1hz frequency

MESSAGES AND MESSAGE TYPES

Getting information about msg files

```
rosmsg <command> [msg_file]
```

Subcommands (among the others)

`show`	display the fields in the msg
`list`	display names of all msg
`package`	list all the msg in a package
`packages`	list all packages containing the msg

```
rosmsg show Pose
rosmsg package nav_msgs
```

DEALING WITH SERVICES

Calling services from command line and getting information:

`rosservice <command> [other_commands]`

Subcommands (among the others)

`list`	print information about active services
`node`	print name of node providing a service
`call`	call the service with the given args
`args`	list the arguments of a service
`type`	print the service type
`find`	find services by service type

```
rosservice call /reset
rosservice type /reset
```

BAGS

`rosbag` is a tool for recording from, and playing back to, ROS topics

Subcommands (among the others)

`record`	record a bag file with the contents of specified topics
`info`	summarize the contents of a bag file
`play`	play back the contents of one or more bag files
`check`	determine whether a bag is playable in the current system, or if it can be migrated
`fix`	repair the messages in a bag file so that it can be played in the current system
`filter`	convert a bag file using Python expressions
`compress`	compress one or more bag files
`decompress`	decompress one or more bag files
`reindex`	reindex one or more broken bag files

ROSBAG COMMAND

- Record a bag:

```
rosbag record -O <filename>.bag (-a | <topic names>)
```

Record all topics

Record only specified topics

E.g. `rosbag record –O file.bag -a`

`rosbag record –O file.bag /topic1 /topic2 ...`

Remember that, to run `rosbag`, you need an active ROS session (`roscore` should be running)

Always monitor your bag size!
Sometimes logging all the topics is not the best idea because you will produce more data/sec than your max disk writing speed (e.g., if you are working with cameras)

ROSBAG COMMAND

- Get info about a bag:

```
rosbag info <filename>.bag
```

- Play a bag:

```
rosbag play –clock [-l] <filename>.bag
```

Use a simulated time

Loop (optional)

INTRO TO ROS DEVELOPMENT

ROBOTICS

POLITECNICO
MILANO 1863

EDITORS / IDEs

ROBOTICS

POLITECNICO
MILANO 1863

IDEs

Integrated Development Environment (IDE):
text editor + tools to help writing software

No official IDE for ROS, but several for C++

On ROS wiki you can find guides on how to properly configure the major ones:

http://wiki.ros.org/IDEs

Feel free to use any Editor or IDE

In class we will use basic text editors, e.g.:

Sublime Text

CLion

Eclpse

NetBeans

VS Code

Sublime Text

ROSED

`rosed` is part of the rosbash suite

Allows the user to edit files referenced using their package name, rather than typing their full path

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

```
rosed roscpp Logger.msg
```

The default editor is `vim`

You can edit the .bashrc file to set a different editor (`nano, emacs, gedit, …`)

Roboware

Based on Visual Studio

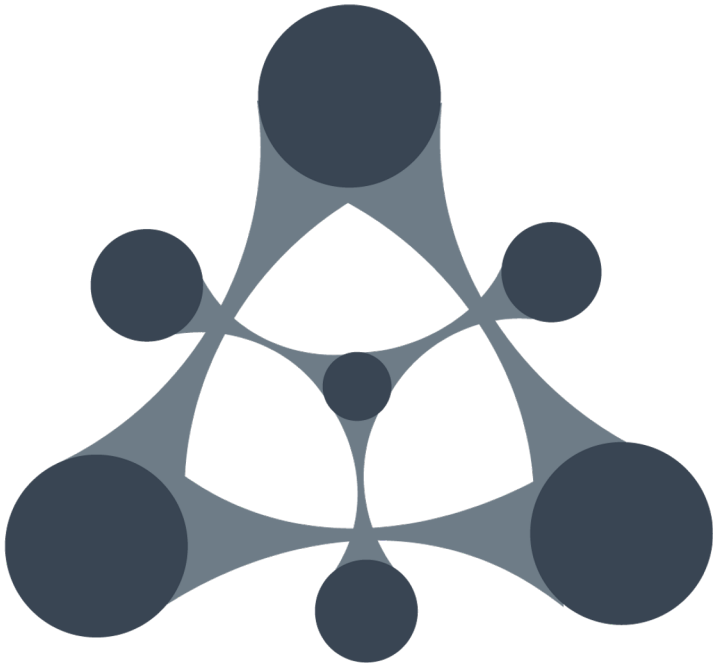
Designed for ROS

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

Currently discontinued (unofficially)



ROS CODE ORGANIZATION

ROBOTICS

POLITECNICO
MILANO 1863

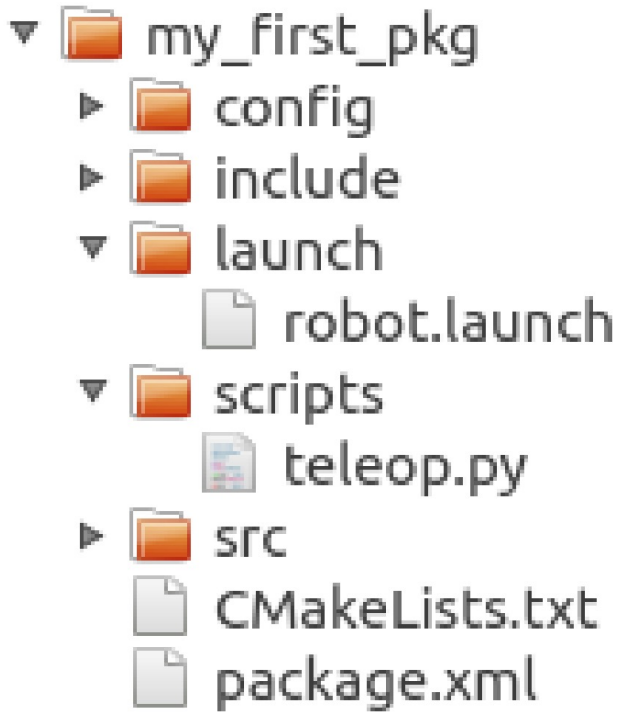
ROS Packages

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/
```

Robotics

Lab session 3

POLITECNICO
MILANO 1863

Paolo Cudrano
paolo.cudrano@polimi.it

Announcement

(Reminder) **Slides** available in shared folder:

https://bit.ly/3JiAdxn (link also on course website)

(Notice: slides for Lecture 2 updated to include “Intro to development”)

From today, during each lecture we will look at some **code**

You can download it from the same shared folder

Each compressed file will contain a single ROS package; please read the provided instructions (`Instructions.txt`) on how to import them in your workspace

RECAP

Last lecture:

- ROS commands with `turtlesim`
  - Filesystem: `roscd`, `rospack`
  - Middleware and execution: `roscore`, `rosrun`, `rosnode`, `rostopic`, `rosservice`, …
  - Tools and utils: `rosmsg`, `rossrv`, `rosbag`, `rqt_graph`, `rqt_plot`, …
- Intro to development
  - Editors / IDEs
  - ROS code organization (workspace and packages) . . .

OUTLINE

Today:

- ROS development (part 1)
  - . . . ROS code organization (workspace and packages)
  - Creating our workspace
  - Creating our first package
  - Implementing our first (two) nodes
  - Building our new package
  - Add functionalities to our first package . . .

ROS DEVELOPMENT

ROBOTICS

POLITECNICO
MILANO 1863

ROS CODE ORGANIZATION

ROBOTICS

POLITECNICO
MILANO 1863

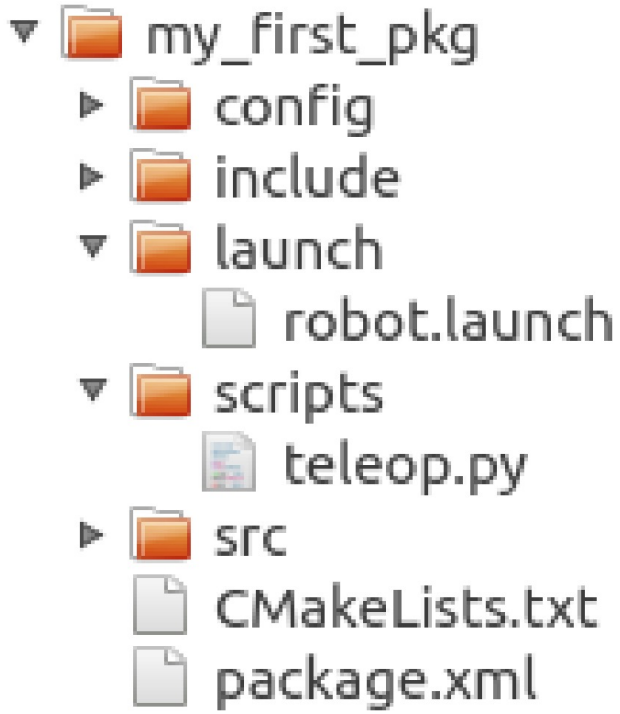
ROS Packages

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
```

Notice: you can only have 1 workspace at a time on your machine!

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 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/`
      - `CMakeLists.txt`
      - `package.xml`
      - `include/`
      - `src/`

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

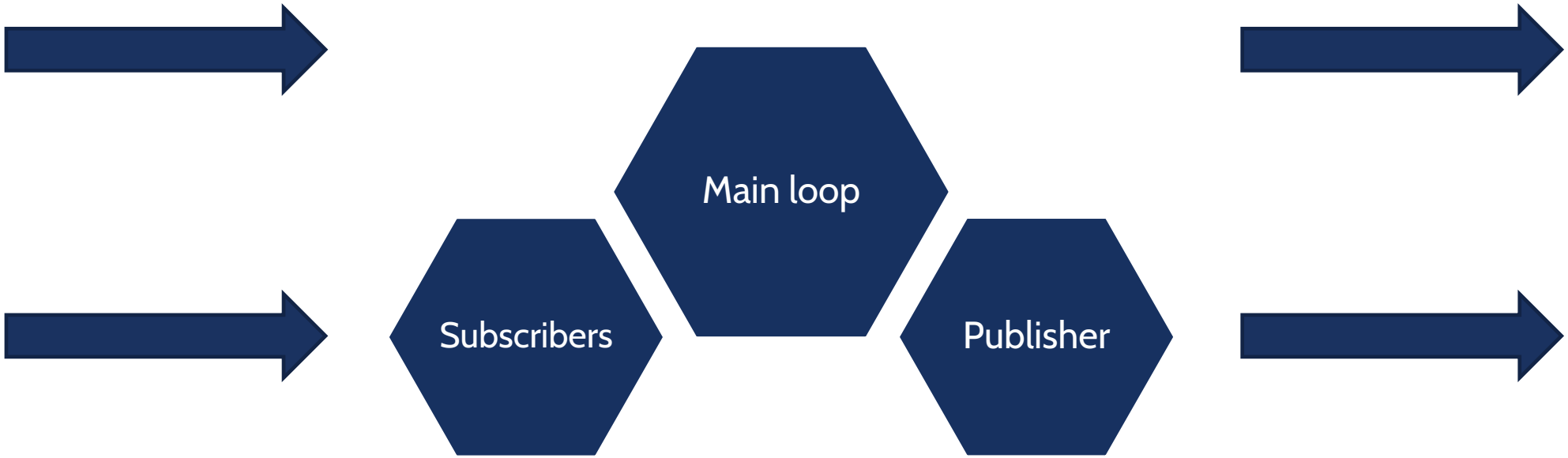
OUR FIRST NODES:
PUBLISHER-SUBSCRIBER

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

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:

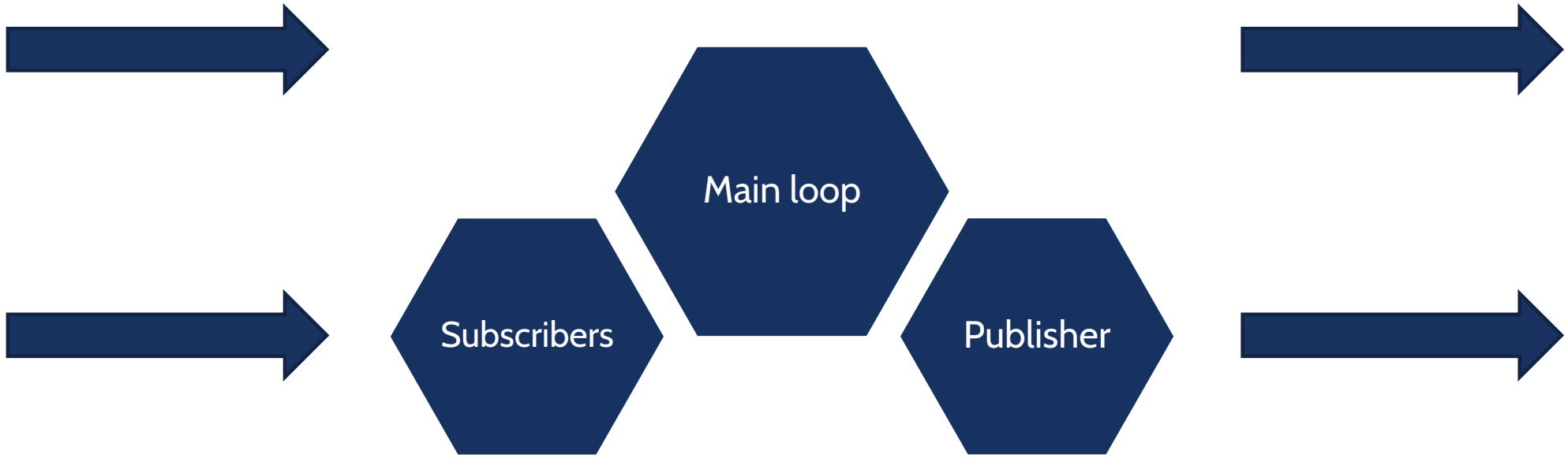
`launch-prefix="gnome-terminal -e”`	for Terminal
`launch-prefix="terminator -x”`	for Terminator

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.

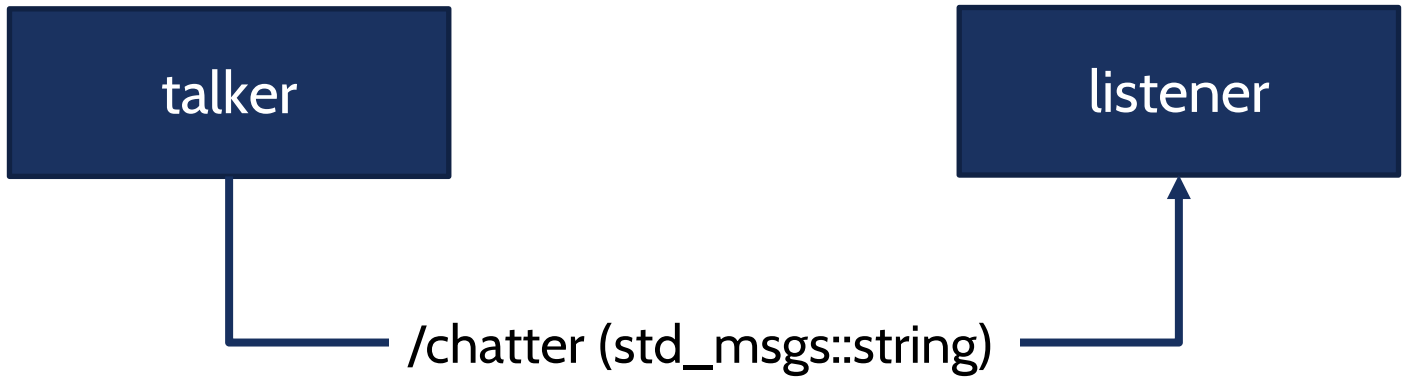
In the slides, these are described as incremental changes on the same `pub_sub` package from last lecture.

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`, ...

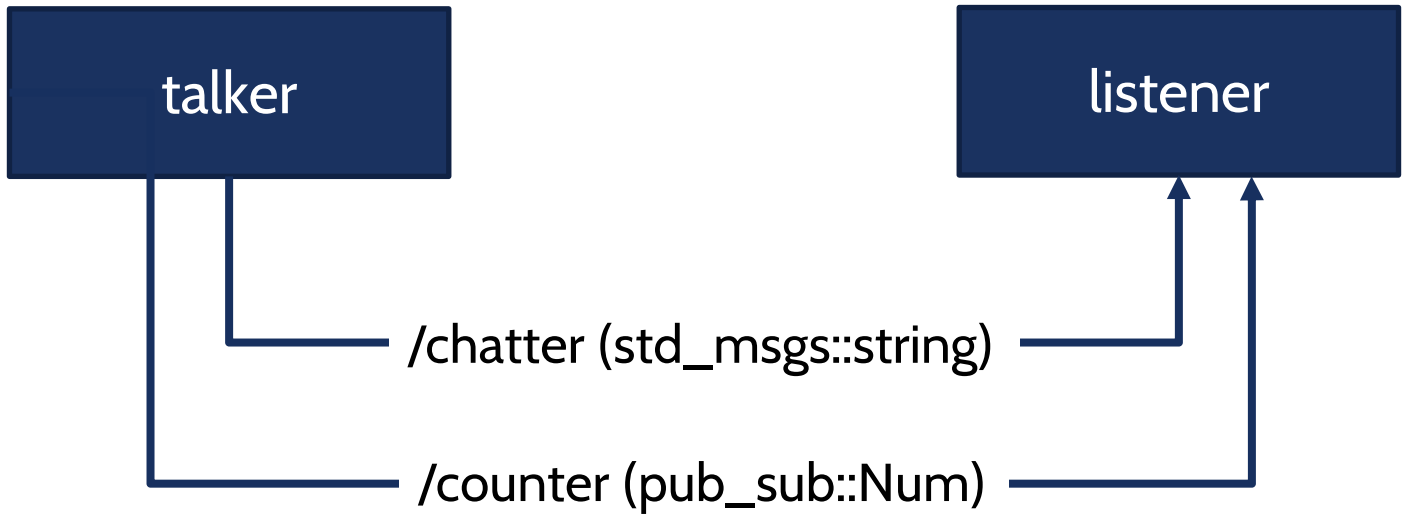
However, we CANNOT copy all of them together in your workspace, as ROS prevents us from having multiple packages and executables with the same name.

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);
}
```

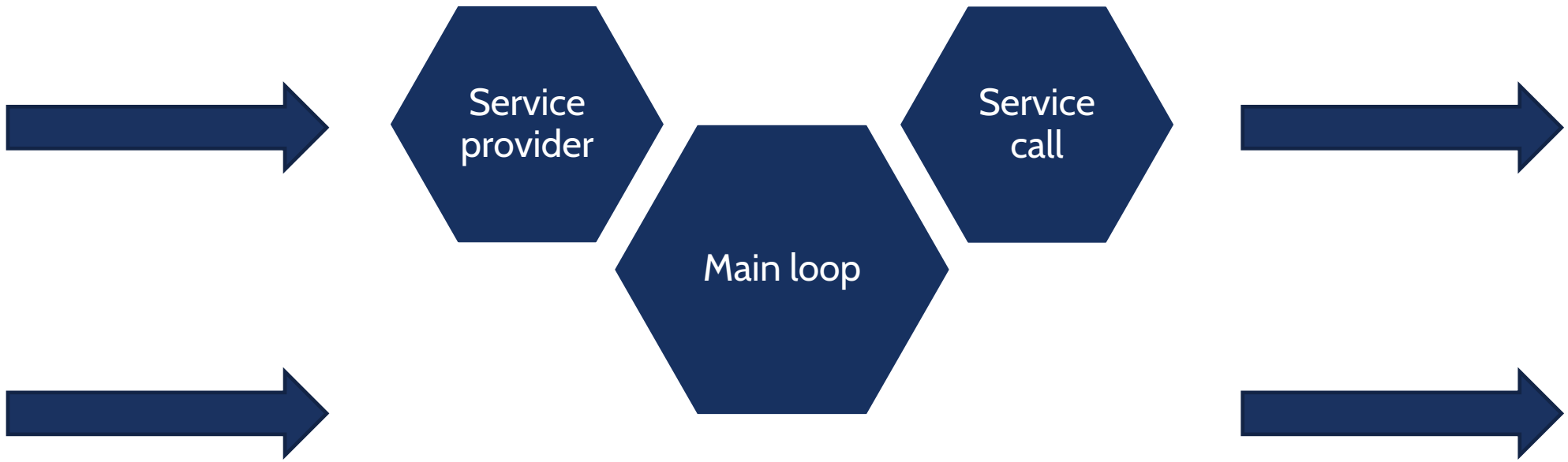
Now we can compile and test both the publisher and the subscriber

SERVICES

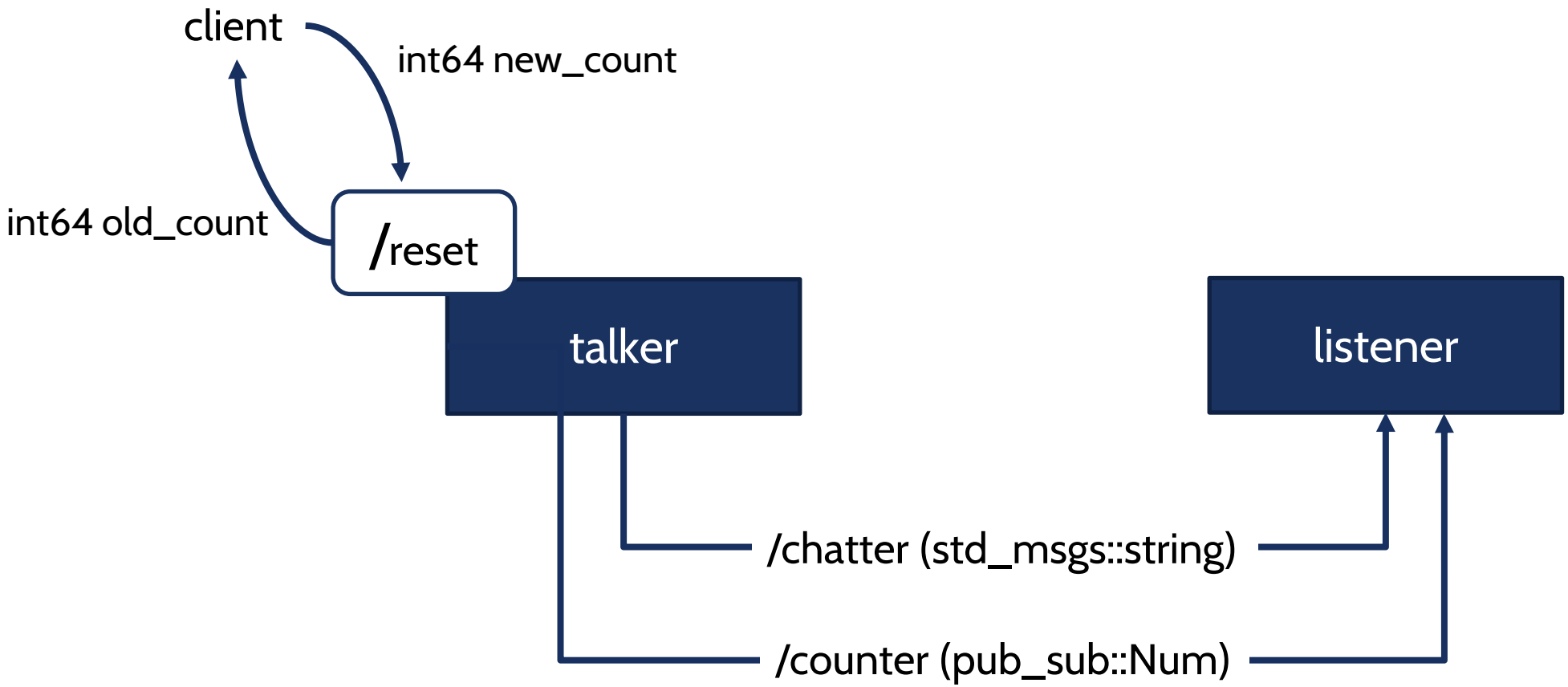
ROBOTICS

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):

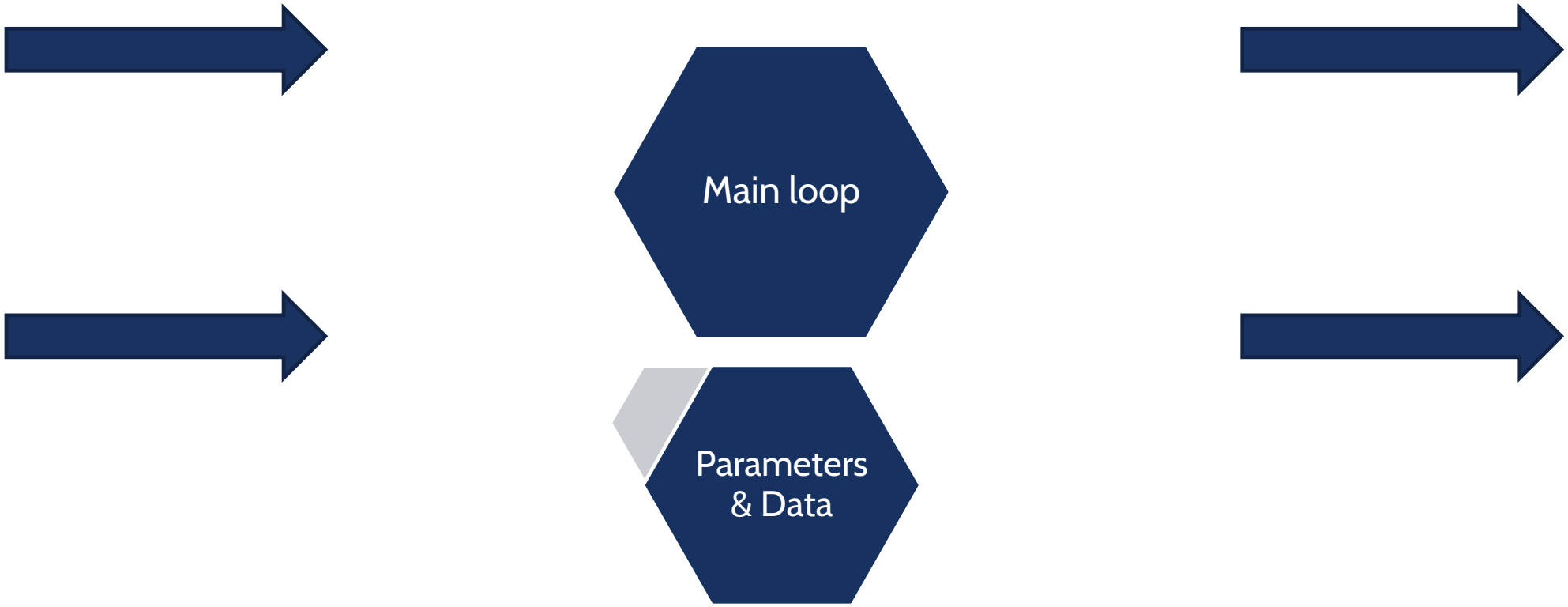
```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

PARAMETERS

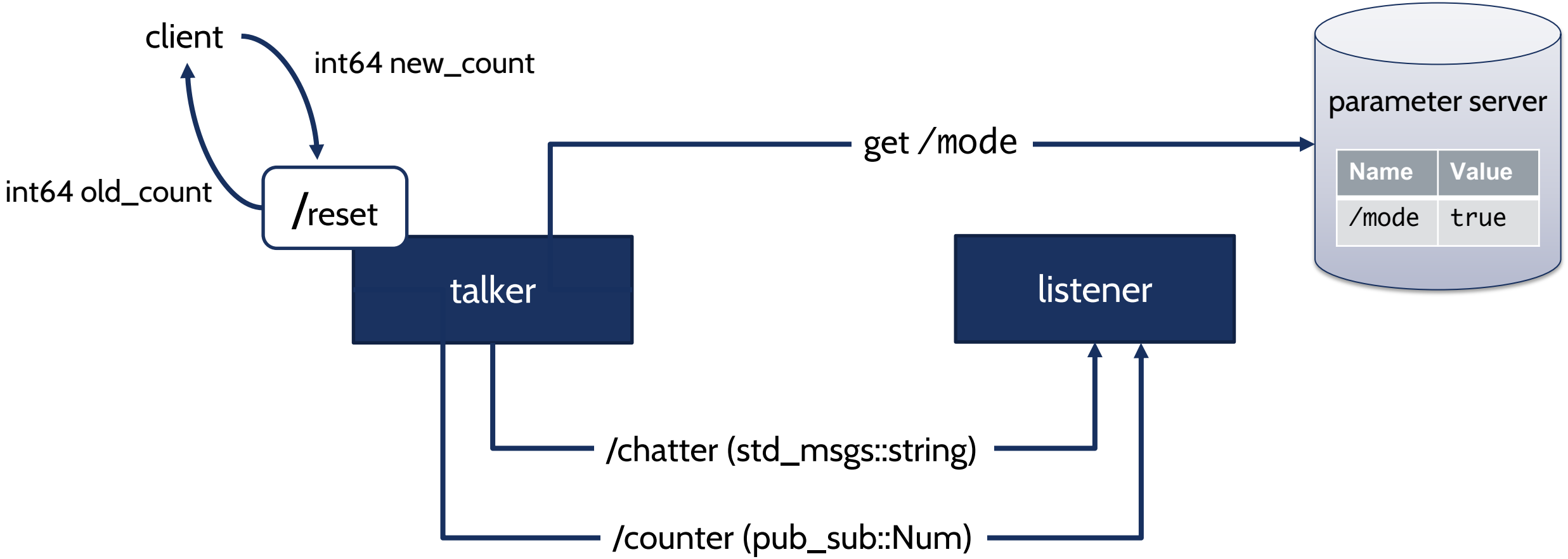
ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE



Objective: pub_sub_v4



PARAMETERS

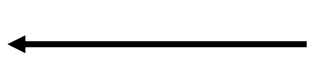
Main usage:

- 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>
```

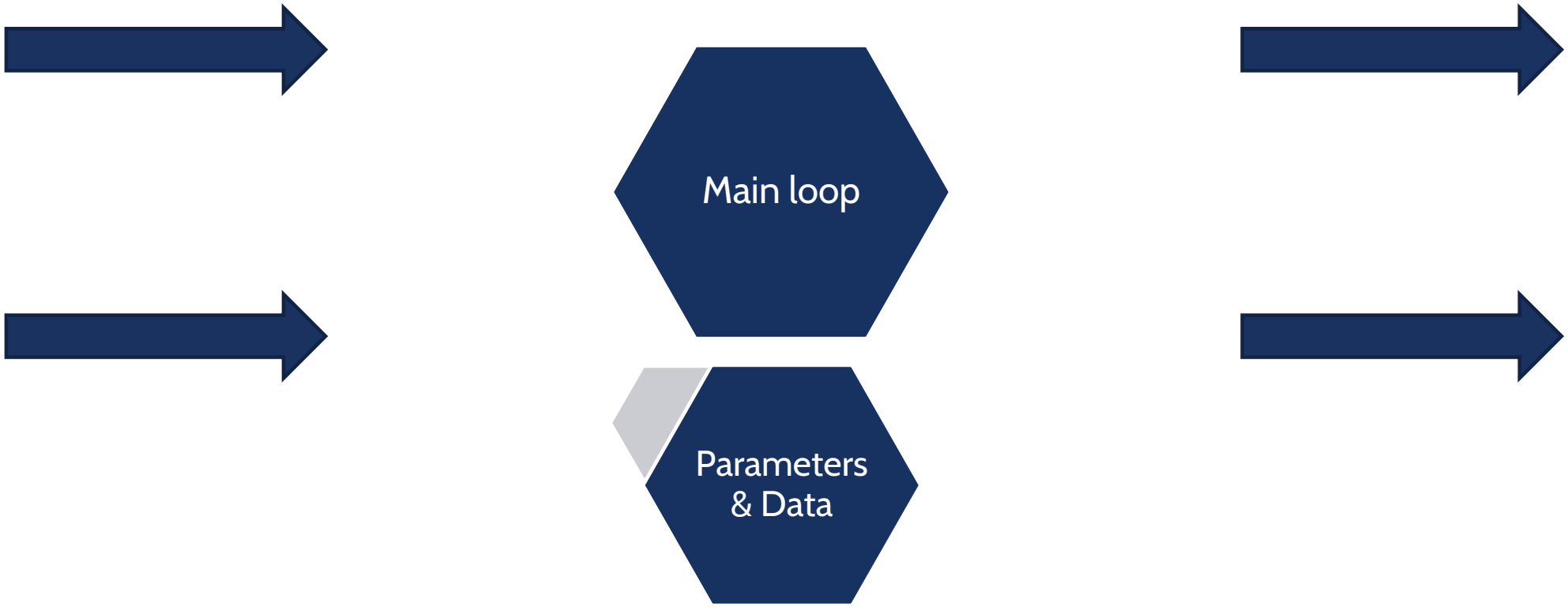
Set the parameter value

PARAMETERS: DYNAMIC RECONFIGURE

ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE



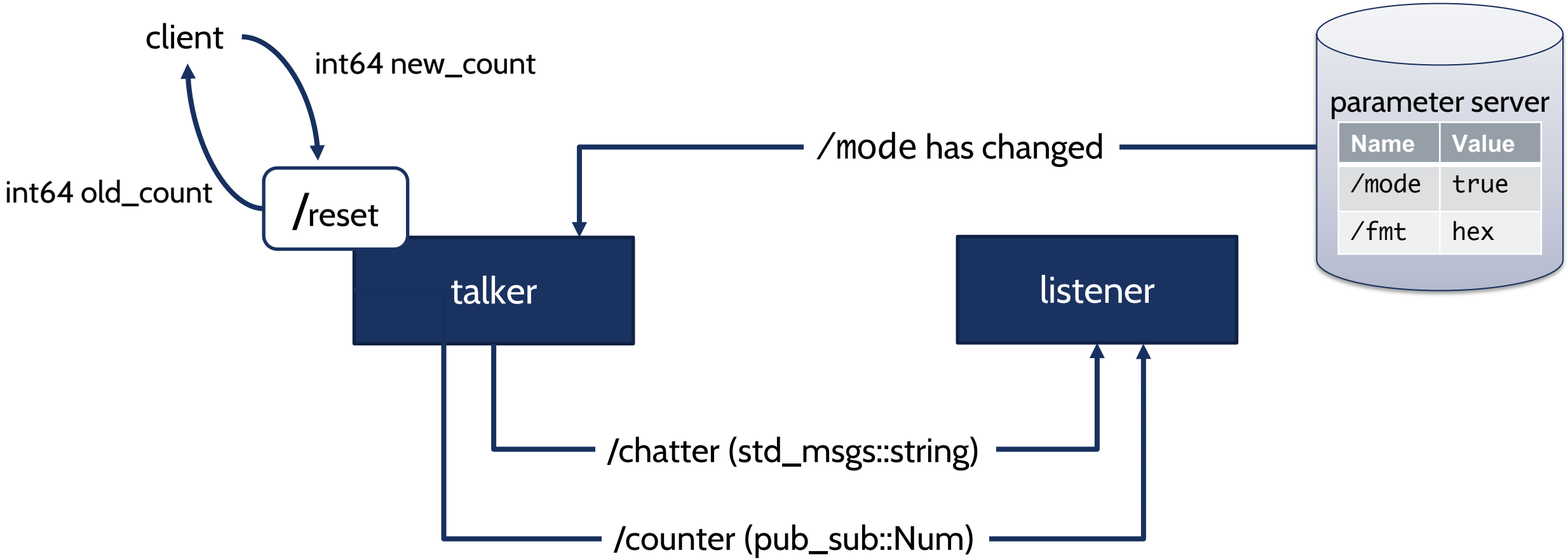
DYNAMIC RECONFIGURE

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:

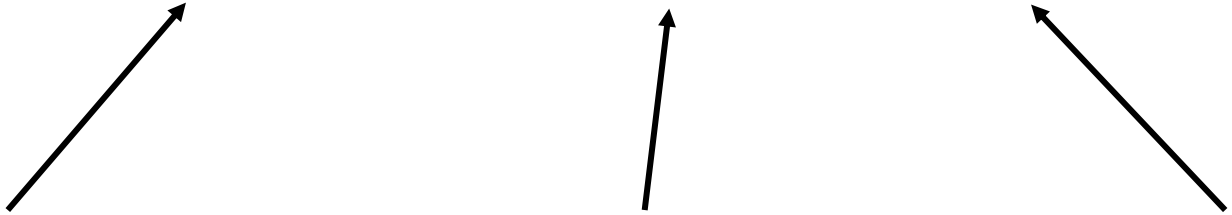
```
fmt_enum = gen.enum([  gen.const(”Decimal", int_t, 0, ”Decimal format"),
                       gen.const(”Binary", int_t, 1, ” Binary format"),
                       gen.const(”Hexadecimal", int_t, 2, ”Hexadecimal format"),
                      ”Enum of formats")

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);
```

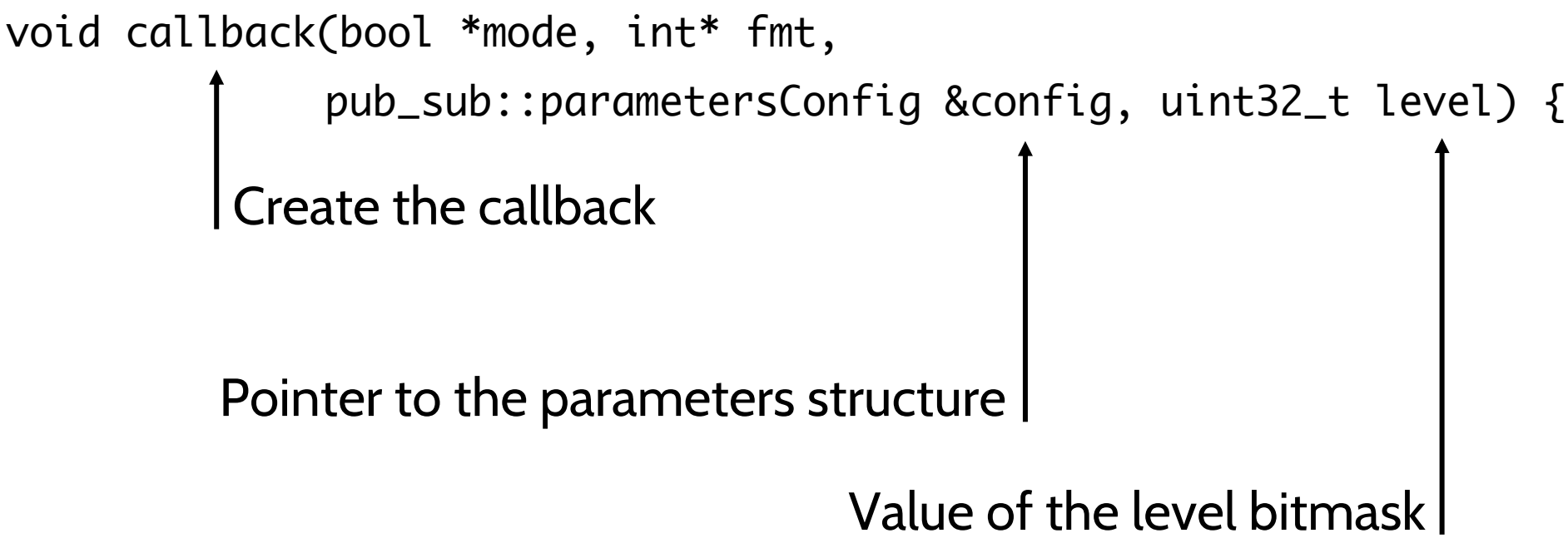
Bind the callback

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})
```

Robotics

Lab session 5

POLITECNICO
MILANO 1863

Paolo Cudrano
paolo.cudrano@polimi.it

RECAP

Last lecture:

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

OUTLINE

Today:

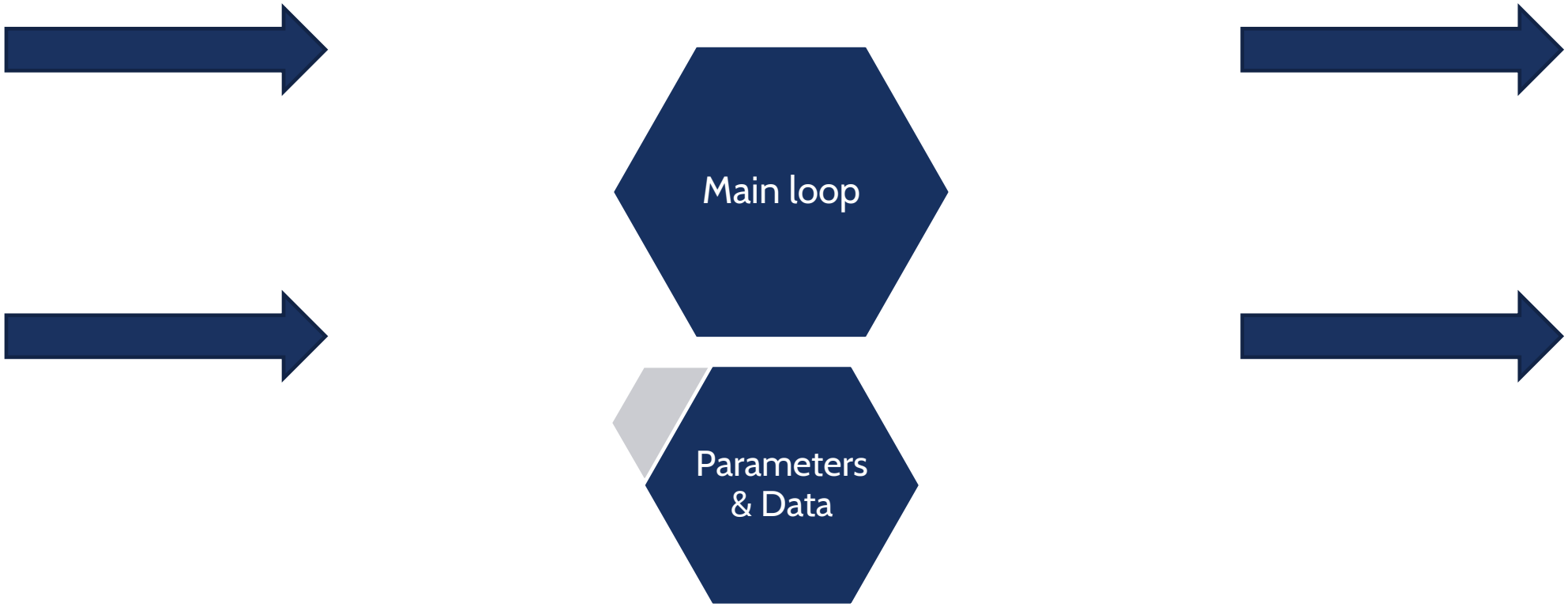
- ROS development (part 2)
  - ROS names and remapping
  - Launch files
  - Custom messages
  - Services
  - Parameters
    - Static
    - … Dynamic
  - Timers
- ROS callbacks and good practices
- ROS tools
  - TF
  - rviz
  - rqt_plot and plotjuggler

PARAMETERS: DYNAMIC RECONFIGURE

ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE



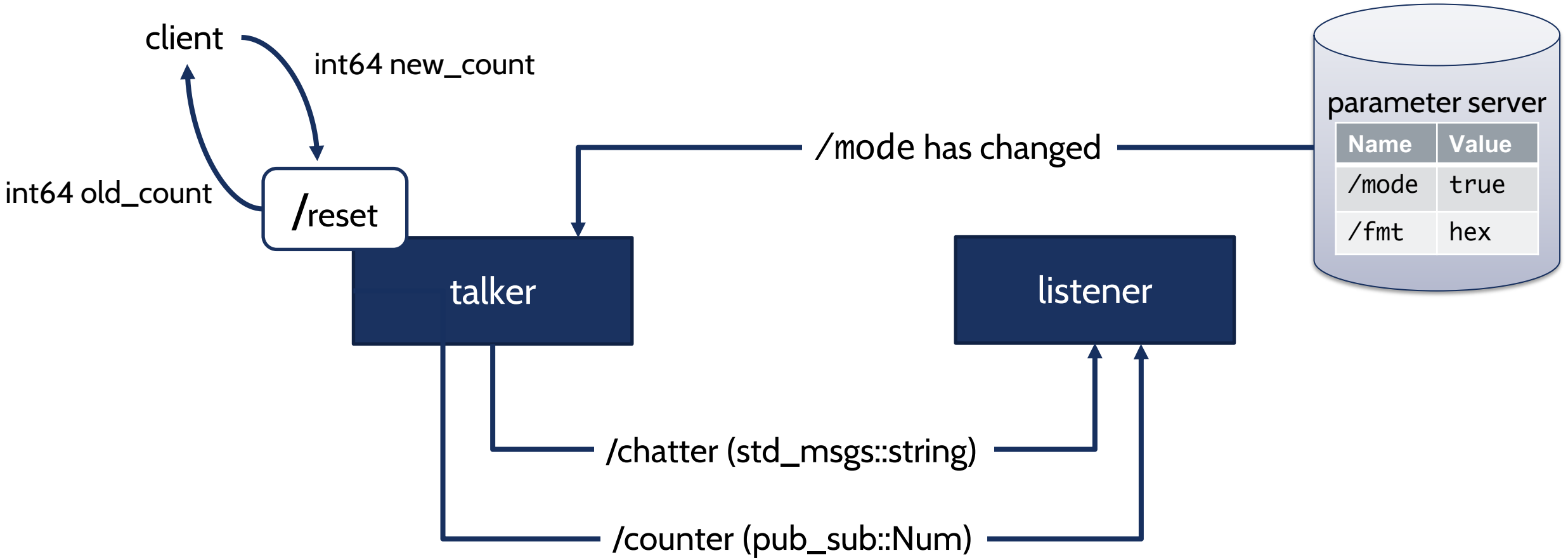
DYNAMIC RECONFIGURE

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:

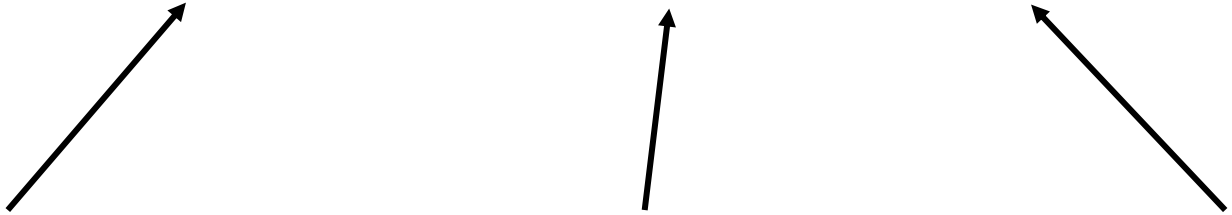
```
fmt_enum = gen.enum([  gen.const(”Decimal", int_t, 0, ”Decimal format"),
                       gen.const(”Binary", int_t, 1, ” Binary format"),
                       gen.const(”Hexadecimal", int_t, 2, ”Hexadecimal format"),
                      ”Enum of formats")

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);
```

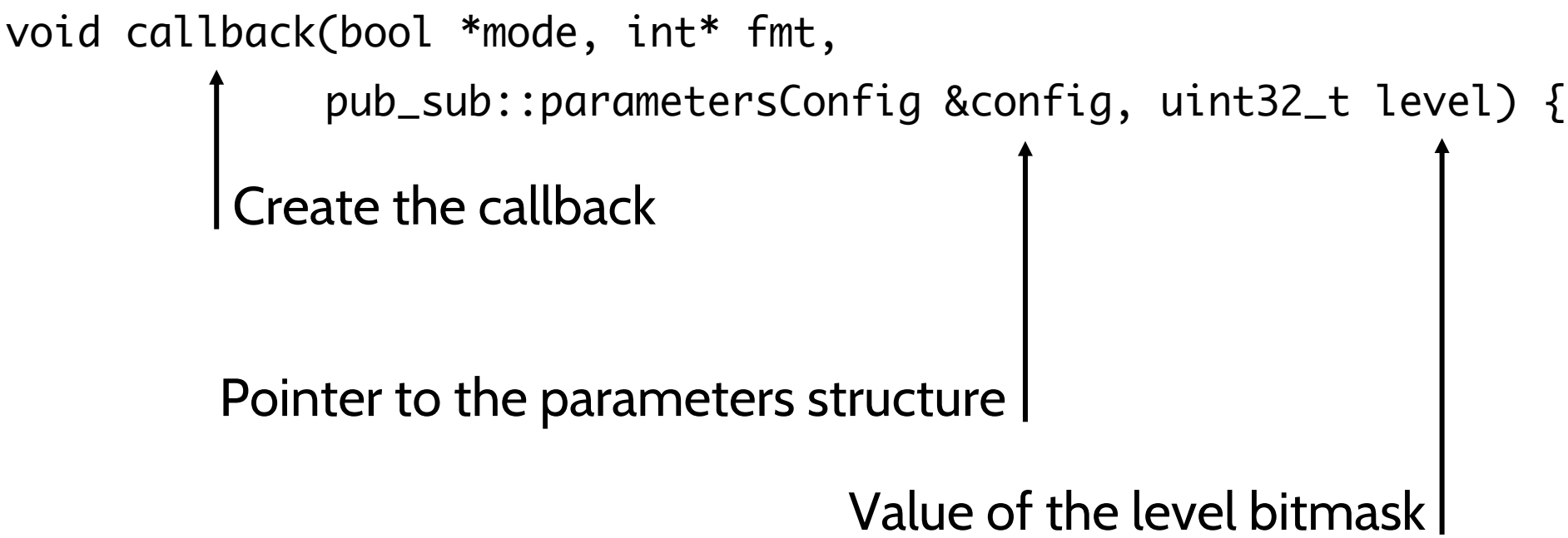
Bind the callback

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):

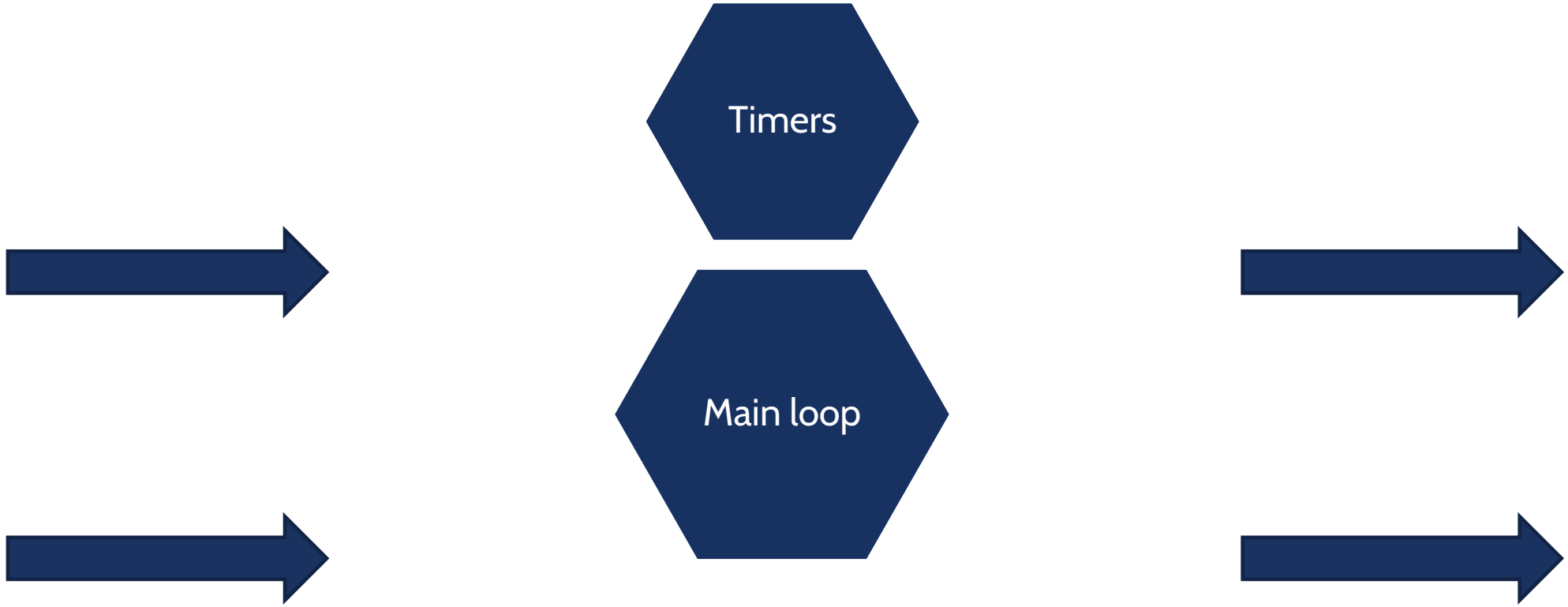
```
add_dependencies(pub ${catkin_EXPORTED_TARGETS})
```

TIMERS

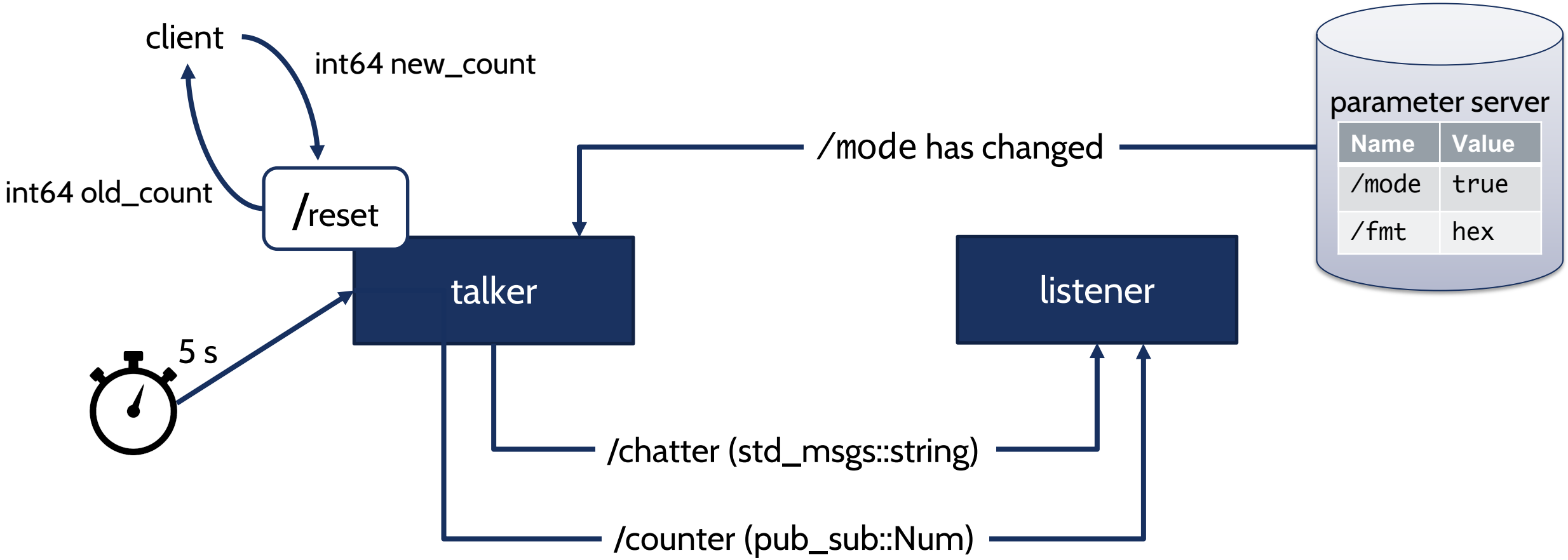
ROBOTICS

POLITECNICO
MILANO 1863

INSIDE THE NODE



Objective: pub_sub_v6



TIMERS

We setup a callback which will be called when the timer expires (repeatedly or only once)

In main initialization

```
ros::Timer timer = n.createTimer(ros::Duration(0.1), timerCallback, false);
```

Timer duration

Timer callback

`true`: one-shot
`false`: repeat (default)

TIMERS

Timer callback

```
void timerCallback(const ros::TimerEvent& ev) {

  ROS_INFO_STREAM("Publisher: timer callback called at time: " << ros::Time::now());

}
```

Print to terminal

Get current time

TIMERS

`CMakeLists.txt` and `package.xml` do not require any changes

CALLBACKS AND GOOD PRACTICES

ROBOTICS

POLITECNICO
MILANO 1863

CALLBACKS IN ROS

We have seen that ROS makes extensive use of callbacks to provide functionalities

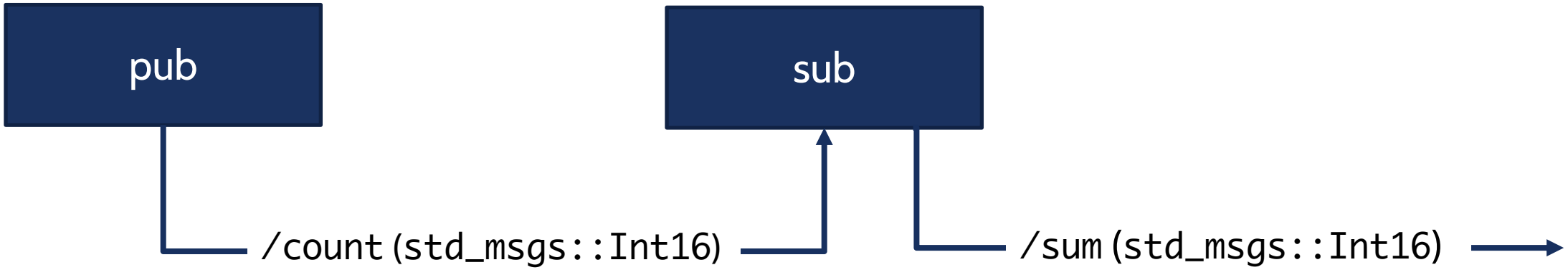
Most often, these callbacks are required to:

- Change the internal state of the node
  - E.g., subscriber callback updates internal variables
- Exploit variables defined in the main() function to provide functionalities
  - E.g., subscriber callback processes the received message and republishes the modified data on another topic

CALLBACKS IN ROS

Example:

- pub publishes an incremental integer on `/count`
- sub subscribes to `/count`, computes a cumulative sum of the numbers received and republishes the result in `/sum`



CALLBACKS IN ROS

Up to now, we would have some difficulties in implementing this cleanly. Indeed,

```
void countCallback(const std_msgs::Int32::ConstPtr& msg) {
  sum = sum + msg->data; // ERROR --> cannot see main::sum
}

int main(int argc, char **argv) {
  [...]
  int sum = 0;
  ros::Subscriber sub = n.subscribe("count", 1000, countCallback);
  [...]
}
```

CALLBACKS IN ROS

To allow callbacks to access variables defined externally (in the main() function), we would like to pass these variables as arguments.

E.g., `void countCallback(int sum,`

`const std_msgs::Int32::ConstPtr& msg)`

However, ROS calls these callbacks passing a predetermined set of arguments

E.g., for a subscriber, whenever a new message arrives, ROS calls the callback function with 1 single argument, representing the received message.

E.g., `countCallback( <new message> );` ← No `sum` argument passed by ROS!

CALLBACKS IN ROS: BOOST::BIND

To be able to pass additional arguments to callbacks, we can use the `boost::bind` function (part of the Boost library). Usage:

- Callback prototype:

```
void countCallback(int *sum,
                   const std_msgs::Int32::ConstPtr& msg)
```

- In `main()`:

```
ros::Subscriber sub =
    n.subscribe<std_msgs::Int32>(
        "count", 1000, boost::bind(&countCallback, &sum, _1)
    );
```

Explicitly specify message type

Callback using boost

CALLBACKS IN ROS: BOOST::BIND

General usage:

```
boost::bind(<ref to callback>, <list of args>)
```

`args` will be passed to our callback in the specified order.

`args` can be:

- A variable to be passed as "extra argument"
- A placeholder _n (e.g., `_1`, `_2`, ...), indicating the position of the n-th argument that will be passed by ROS

Notice: if you expect to modify the variables passed as argument, remember to pass them by reference (i.e., as pointers)!

CALLBACKS IN ROS: BOOST::BIND

Boost will internally create a function with prototype macthing what ROS expects; inside this function, it will call your callback, passing the extra arguments.

Notice: this is a simplification!

Eg. with a subscriber, boost creates an internal function like:

```
void boostInternalCallback(const std_msgs::Int32::ConstPtr& msg) {
     countCallback(sum, msg);
}
```

The pointer we passed to boost (boost saved it internally)

When a new message arrives, ROS will call the boost internal callback as

```
boostInternalCallback( <new message> );
```

which, in turn, will call our callback with the extra argument!

CALLBACKS IN ROS: BOOST::BIND

Using `boost::bind` is the solution we have adopted so far.

It is effective and it requires a localized intervention in the code (no refactoring, just change the small parts of code)

However it could also be a bit cumbersome (think about having many extra arguments . . .)

CALLBACKS IN ROS: CLASSES

The ultimate answer to solve this problem (and write everything more cleanly) is to use a class for our entire node!

In general, it is a **good practice** to use classes for our nodes

CALLBACKS IN ROS: CLASSES

In `sub.cpp`, we create a class `Subscriber`:

```
class Subscriber {

  private:

    ros::NodeHandle n;

    ros::Subscriber sub;

    ros::Publisher pub;

    int sum;
```

Member variables, or **fields**

These variables can be seen from anywhere inside the class!
We declare as private everything we don't need to access from outside the class (typically all member variables)

CALLBACKS IN ROS: CLASSES

We add its **member functions**, or **methods:**

```
public:
```

We declare as public every member variable or function that needs to be accessed from outside the class.
The constructor must be public.

```
    Subscriber() {
        this->sub = this->n.subscribe("count", 1000,
                                      &Subscriber::countCallback, this);
        this->pub = this->n.advertise<std_msgs::Int32>("sum", 1000);
        this->sum = 0;
    }
```

Constructor
Every initialization goes here

`this->` operator should be used to access every member variables or functions from within the class

We must pass `this` as last argument to any ROS function expecting a callback

CALLBACKS IN ROS: CLASSES

```
public:
```
(No need to repeated it in our code)

```
  void main_loop() {
    ros::Rate loop_rate(10);
    while (ros::ok()) {
      ROS_INFO("Current sum: %d", this->sum);
      ros::spinOnce();
      loop_rate.sleep();
    }
  }
```

Main loop function

It's just a regular function, which we will call after initialization, when we want the main loop to start executing

CALLBACKS IN ROS: CLASSES

```
public: ←——————  (No need to repeated it in our code)
  void countCallback(const std_msgs::Int32::ConstPtr& msg) {
    ROS_INFO("Received: %d", msg->data);
    this->sum = this->sum + msg->data;


    std_msgs::Int32 sum_msg;
    sum_msg.data = this->sum;
    this->pub.publish(sum_msg);
  }
};  ←——————  End of class definition
```

Callback function
No need for extra arguments as it can already see every member variables inside the class

CALLBACKS IN ROS: CLASSES

We can add our main function, where the execution will start:

```
int main(int argc, char **argv) {
  ros::init(argc, argv, "callbacks_sub");
  Subscriber my_subscriber;
  my_subscriber.main_loop();
  return 0;
}
```

Call to `ros::init` at beginning of execution

Create an instance (object) of `Subscriber` class. The constructor is internally called.

Start the main loop

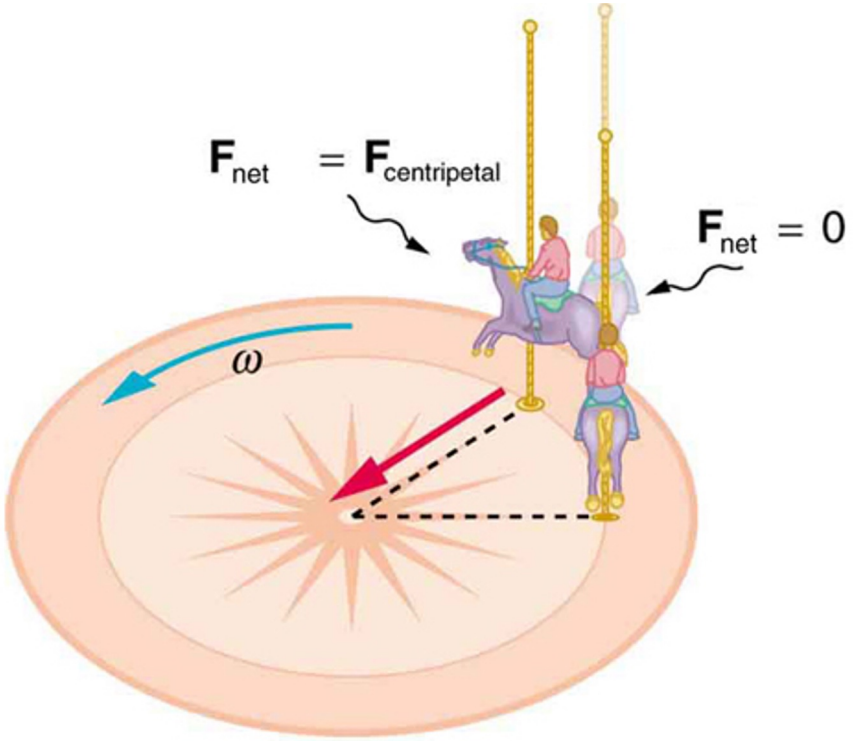
TF

ROBOTICS

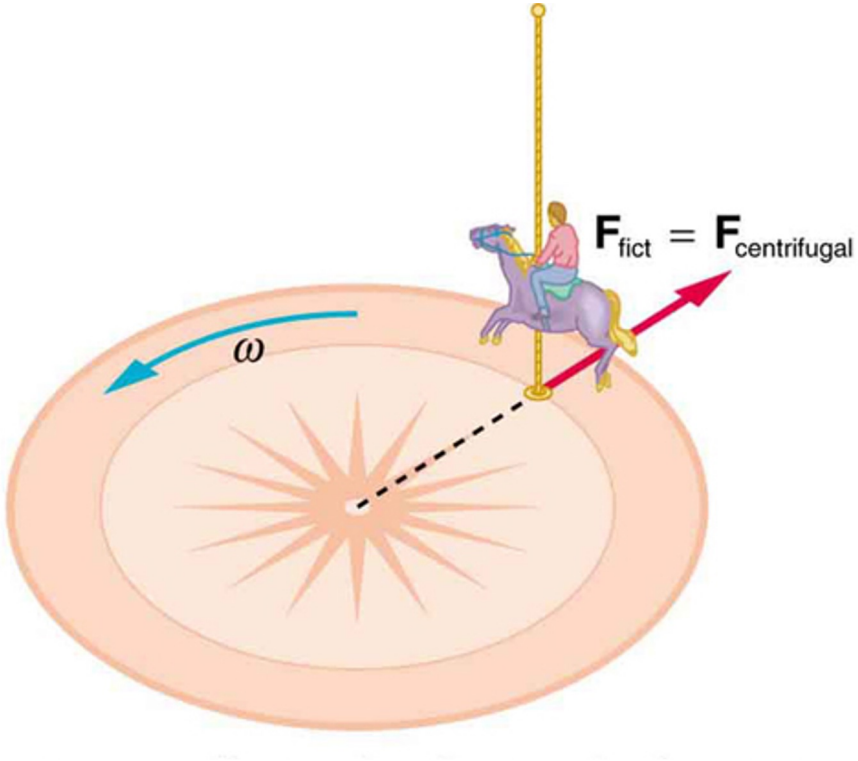
POLITECNICO
MILANO 1863

REFERENCE SYSTEMS: IN PHYSICS

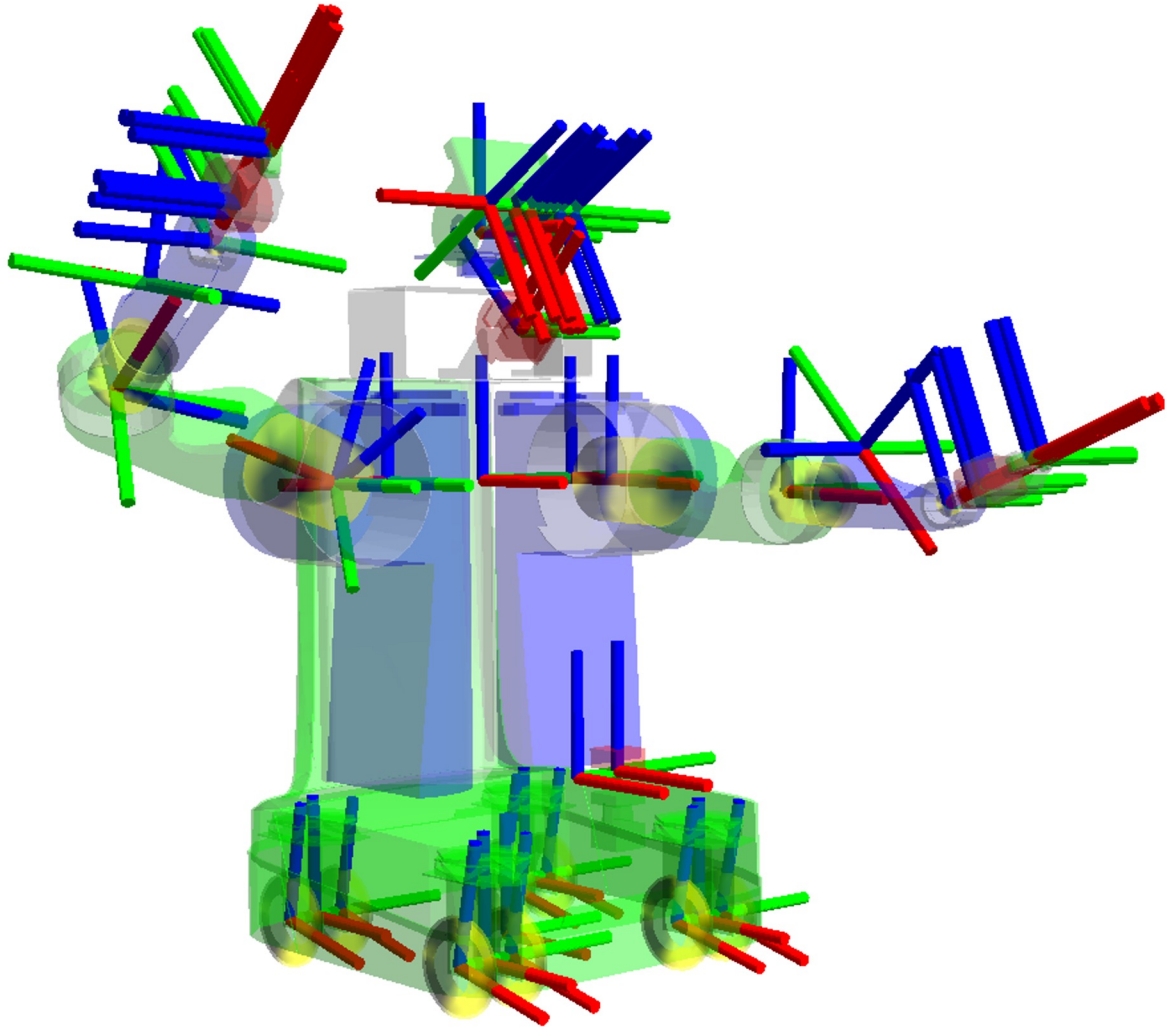
Reference System is everything



VS



REFERENCE SYSTEMS: IN ROBOTICS



IN ROBOTICS

For manipulators:

- A moving reference frame for each joint
- A base reference frame
- A world reference frame

For autonomous vehicles:

- A fixed reference frame for each sensor
- A base reference frame
- A world reference frame
- A map reference frame

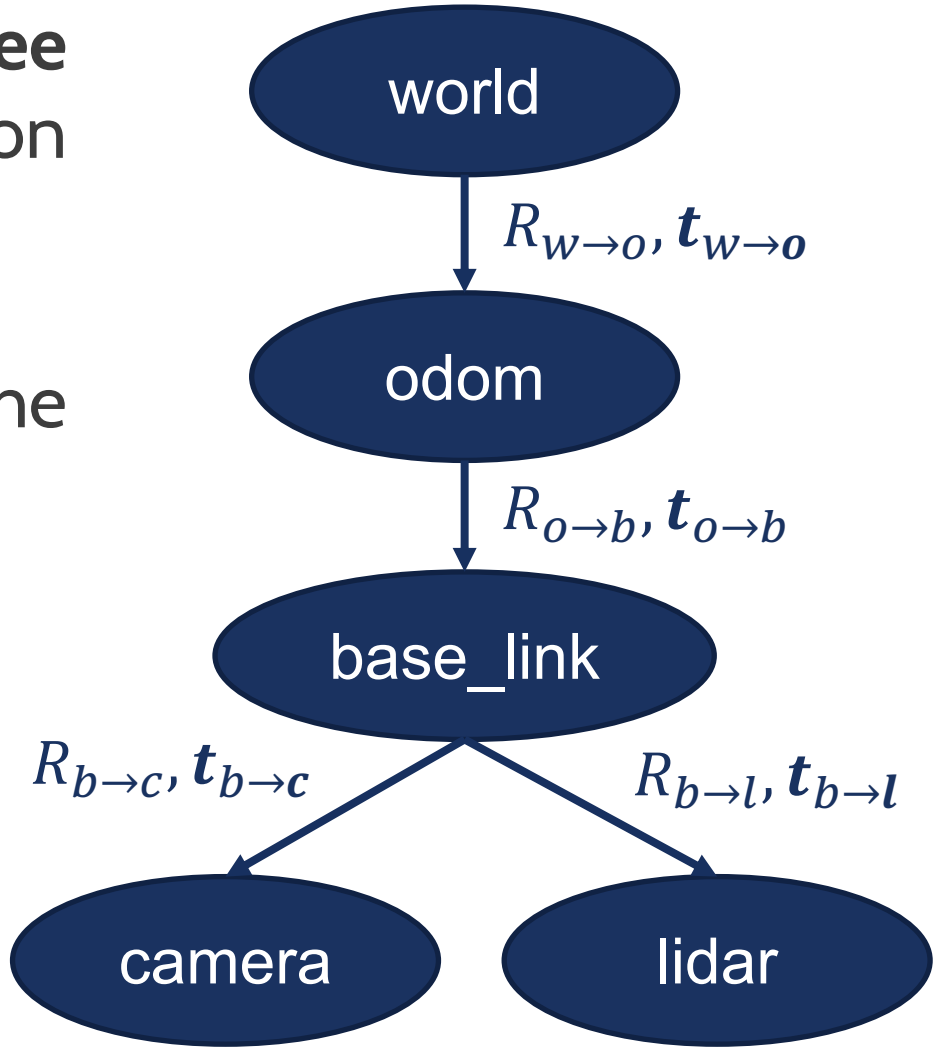
How is it possible to convert form one frame to another?
Math, lot of it.

IN ROBOTICS

To simplify things, frames are described in a **tree**
Each frame comes with a transformation between itself and its father

The **world frame** is the most important, but the others are used for simplicity

In a tree of reference frames, we define a roto-translation between parent and child
To compute the transformation between two frames, we can simply combine the roto-translation found trasversing the tree



TF: TRANSFORMATION FRAMES

If we specify the transformation tree, ROS does all the hard work for us, thanks to **TF!**

We can do interpolation, transformation, tracking, ...

TF is a ROS tool that:

- keeps track of all the dynamic transformation for a limited period of time
- is decentralized
- provides the position of a point in each possible reference frame

COMMON FRAMES

For each transformation we specify in our transformation tree, we have to specify

- the name of the father frame
- the name of the child frame
- the rototranslation between the two

Frame names can be arbitrary, but some particular frames are usually named following a convention:

- world → the root frame, fixed
- map → used in navigation, mostly fixed, but can be realigned over time
- odom → frame in which we express the odometry of the vehicle
- base_link → the main frame of the robot, tipically in its center of gravity

TF TREE TOOLS

ROS offers different tools to analyze the transformation tree:

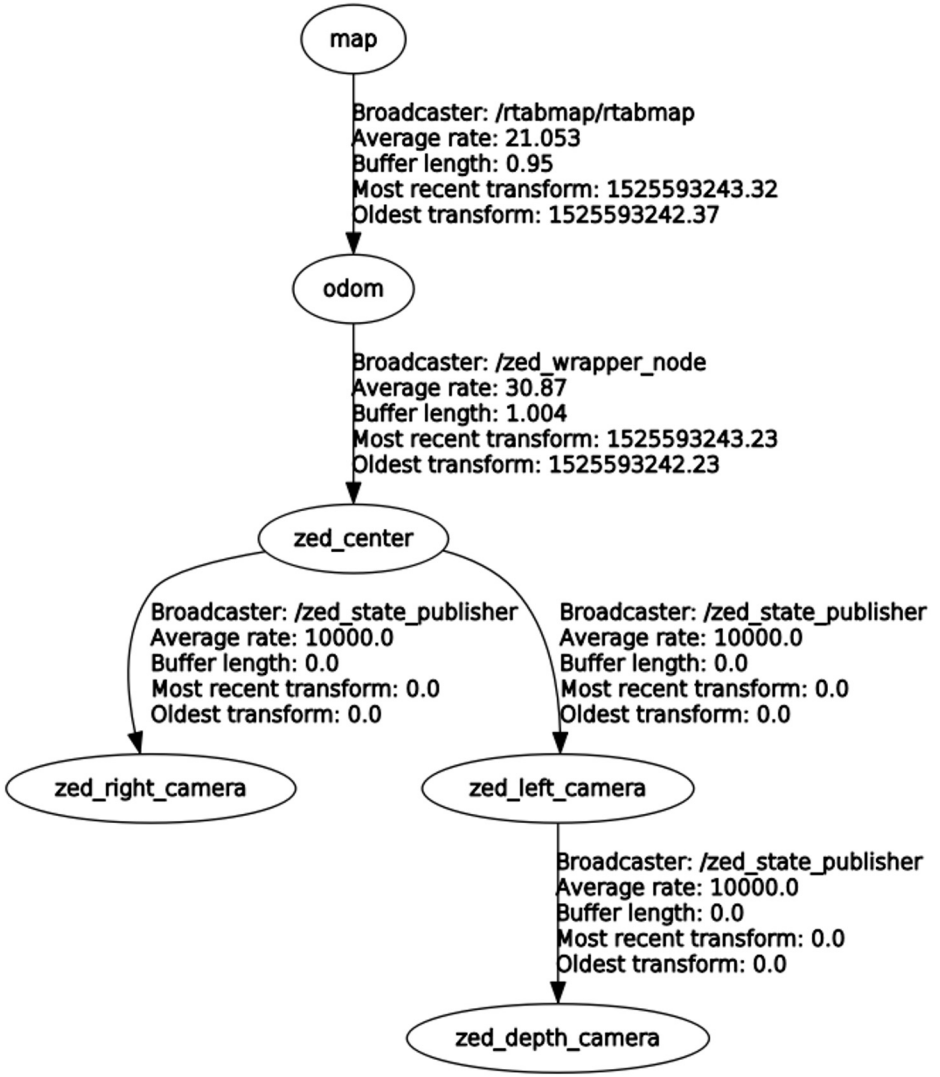
- `rosrun rqt_tf_tree rqt_tf_tree`

  visualizes the tf tree at the current time

- `rosrun tf view_frames`

  listens for 5 seconds to the `/tf` topic and creates a pdf file with the tf tree

HOW TF_TREE LOOKS LIKE IN PRACTICE



TF AND TF2

ROS originally shipped with TF

Then several bug fixed, to the point that a new version of TF was created: TF2

Some legacy packages still require TF, otherwise always use TF2

In class, we will only see TF2.

At home, you can find TF example in our package as well.

WRITING A TF BROADCASTER

Now that we got an idea of how tf works and why it's useful, we can take a look at how to specify our TF tree

We do so using a **tf broadcaster**

Since we do not have a physical robot here,
we will use turtlesim for our example

WRITING A TF BROADCASTER

Turtlesim provides us with a topic indicating its pose /turtlesim/pose

(we can see it as a very precise odometry)

We will use this topic to broadcast a **(dynamic) transformation**

`world` → `turtle`, which will change as the turtle moves around

We can also add a **static transformations** from turtle to each of its 4 legs:

`turtle` → `FRleg`, `turtle` → `FLleg`, `turtle` → `BRleg`, `turtle` → `BRleg`

WRITING A TF BROADCASTER

We create a package called `tf_examples` inside the `src` folder of our catkin environment, adding dependencies `roscpp std_msgs tf2 tf2_ros turtlesim`

```
$ catkin_create_pkg tf_examples roscpp std_msgs tf2 tf2_ros turtlesim
```

Then we can create a `broadcaster_tf2.cpp`

WRITING A TF BROADCASTER

In `broadcaster_tf2.cpp`:

Includes:

```
#include "ros/ros.h"
#include "turtlesim/Pose.h"
#include <tf2/LinearMath/Quaternion.h>
#include <tf2_ros/transform_broadcaster.h>
#include <geometry_msgs/TransformStamped.h>
```

WRITING A TF BROADCASTER

Now we have to create our class:

```
class TfBroad {
public:
      TFBroad() {…}
      void callback(const turtlesim::Pose::ConstPtr& msg){…}
private:
      ros::NodeHandle n;
      tf2_ros::TransformBroadcaster br;
      geometry_msgs::TransformStamped transformStamped;
      ros::Subscriber sub;
};
```

WRITING A TF BROADCASTER

In class constructor, we subscribe to `/turtle1/pose`

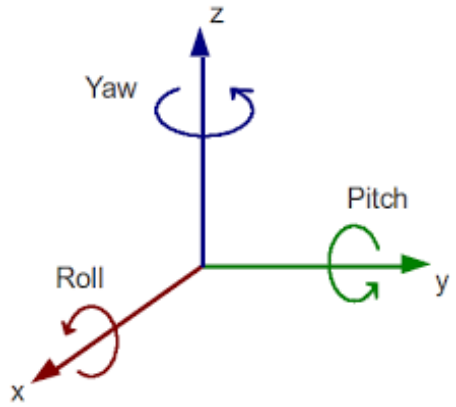
```
sub = n.subscribe("/turtle1/pose", 1000, &TfBroad::callback, this);
```

WRITING A TF BROADCASTER

Inside the callback (every time we get a new pose), we populate a transformation object using the data from the pose message (we are in a 2D environment)

Notice: TF uses quaternions for rotations, but it provides tools to specify them also through roll, pitch and yaw!

Finally, we broadcast the transformation



WRITING A TF BROADCASTER

```
void callback(const turtlesim::Pose::ConstPtr& msg) {
  transformStamped.header.stamp = ros::Time::now();
  transformStamped.header.frame_id = "world";
  transformStamped.child_frame_id = "turtle";

  transformStamped.transform.translation.x = msg->x;
  transformStamped.transform.translation.y = msg->y;
  transformStamped.transform.translation.z = 0.0;

  tf2::Quaternion q;
  q.setRPY(0, 0, msg->theta);
  transformStamped.transform.rotation.x = q.x();
  transformStamped.transform.rotation.y = q.y();
  transformStamped.transform.rotation.z = q.z();
  transformStamped.transform.rotation.w = q.w();

  br.sendTransform(transformStamped);
}
```

Setup the **header**, containing timestamp and parent/child frames

Translation

Rotation
Setup a quaternion using ROS functions to convert from roll pitch yaw

Broadcast the transform

WRITING A TF BROADCASTER

Then we write the main function:

```
int main(int argc, char **argv) {
  ros::init(argc, argv, "tf_broadcast");
  TfBroad my_tf_broadcaster;
  ros::spin();
  return 0;
}
```

WRITING A TF BROADCASTER

As usual, we have to add this new executable to the CMakeLists.

Package dependencies (already specified during the package creation):

```
find_package(catkin REQUIRED COMPONENTS roscpp std_msgs tf2 tf2_ros turtlesim)
```

Add the executable:

```
add_executable(tf2_broad src/broadcaster_tf2.cpp)
add_dependencies(tf2_broad ${catkin_EXPORTED_TARGETS})
target_link_libraries(tf2_broad ${catkin_LIBRARIES})
```

WRITING A TF BROADCASTER

We can setup a launchfile to test our node

We add a `turtlesim_node`, a `turtle_teleop_key`, and our broadcaster node:

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

  <node pkg="tf_examples" type = "tf2_broad" name = "tf_broad"/>
</launch>
```

ADD STATIC TF

Now we can add the transformations for the 4 legs of the turtle

We could use the same mechanism seen so far, adding broadcasters to our node.

However, the transformation between legs and turtle frame will always be fixed (the turtle robot is a rigid object).

For static (fixed) transformations, ROS helps us with the node `static_transform_publisher`, which we can run directly from our launchfile!

ADD STATIC TF

We add the 4 static transform, specifying as args:

- position (x,y,z) and rotation (as a quaternion qx,qy,qz,qw) of the robot
- parent frame
- child frame

```
<node pkg="tf2_ros" type="static_transform_publisher" name="back_right" args="0.3 -0.3 0 0 0 0 1 turtle FRleg " />
<node pkg="tf2_ros" type="static_transform_publisher" name="front_right" args="0.3 0.3 0 0 0 0 1 turtle FLleg " />
<node pkg="tf2_ros" type="static_transform_publisher" name="front_left" args="-0.3 0.3 0 0 0 0 1 turtle BLleg " />
<node pkg="tf2_ros" type="static_transform_publisher" name="back_left" args="-0.3 -0.3 0 0 0 0 1 turtle BRleg " />
```

ADD STATIC TF

We can visualize our tf tree running `rqt_tf_tree`

We can use rviz to visualize the motion of the turtle

We can also see all the published tf using `rostopic echo`

Or we can see specific tf transforms with `tf_echo:`

```
$ rosrun tf tf_echo father child
```

```
$ rosrun tf tf_echo \world \FRleg
```

ROS VISUALIZATION: RVIZ

ROBOTICS

POLITECNICO
MILANO 1863

ODOMETRY

Odometry messages contain estimated information about the position and velocity of the robot in space.

`nav_msgs/Odometry` definition:

```
std_msgs/Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

We can visualize certain messages using rviz, a visualization tool provided by ROS

ODOMETRY ON RVIZ

You can use rviz to display Odometry messages as arrows.
The location of the arrows represents the robot position,
The orientation of the arrow represents the current direction of movement

Type in the terminal:

```
$ rviz
```

In rviz, press on “Add“ in the “Display” panel (bottom left)

- Add an Odometry message and specify the topic name
- You can also select “By Topic” and see directly which topics are available

You can change the visualization properties from the “Display” panel (left)
Change the parameter “Keep” to display or not display past poses

ROS PLOTTING TOOLS

You can plot informations in ROS using `rqt_plot` (basic)

```
rosrun rqt_plot rqt_plot
```

Or using plotjuggler (more advanced)

```
rosrun plotjuggler plotjuggler
```