UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria
e Scienza dell'Informazione

Fundamentals of Robotics
ROS programming

Luigi Palopoli & Nicu Sebe & Michele Focchi

Introducing me...

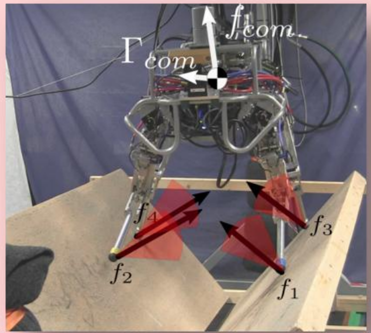
Introducing Michele Focchi

Research:

- Quadrupeds
- Control
- Locomotion
- Planning

ISTITUTO ITALIANO DI TECNOLOGIA

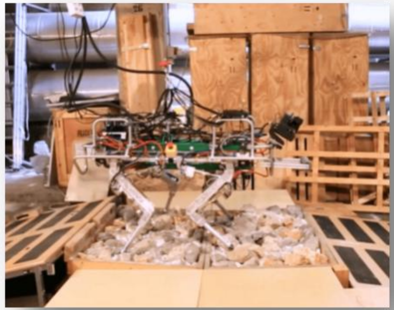
PhD



Stability control

ISTITUTO ITALIANO DI TECNOLOGIA

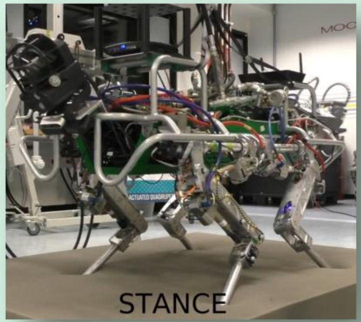
Postdoc



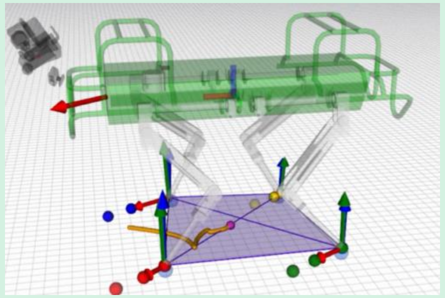
Heuristic Planning

ISTITUTO ITALIANO DI TECNOLOGIA

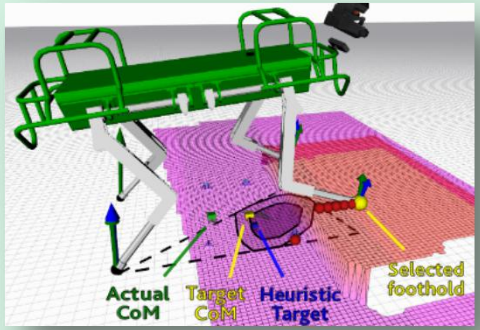
Researcher



Locomotion on soft-terrain



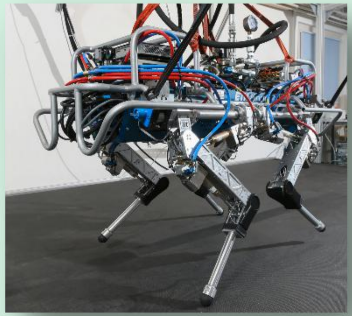
MPC re-planning



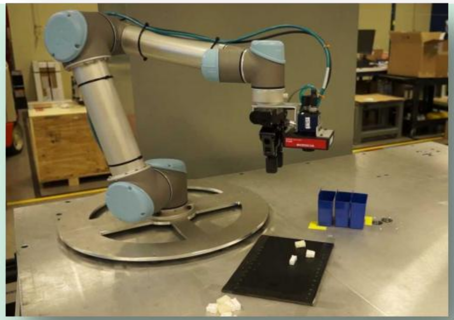
Stability-based planning

UNITRENTO

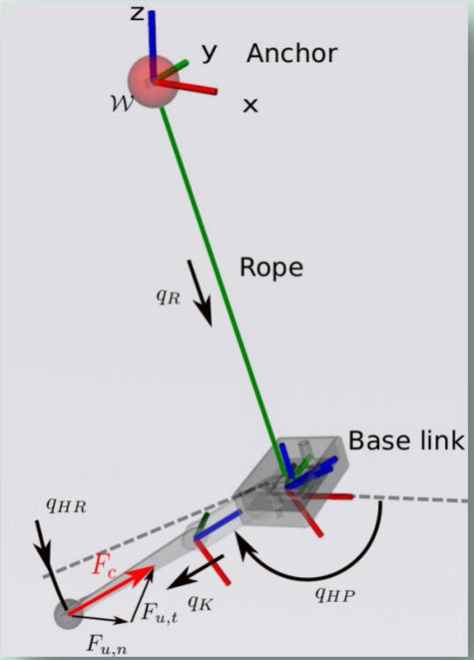
Assistant Professor



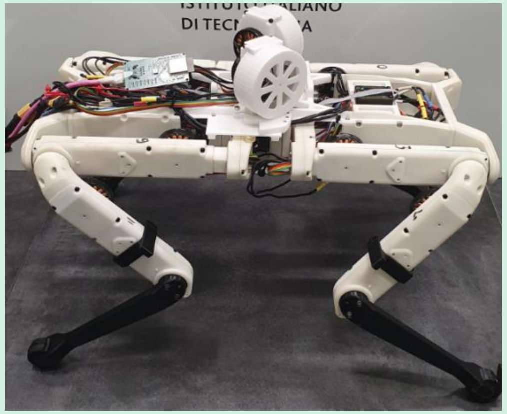
Setting-up a new legged robotics lab at DII



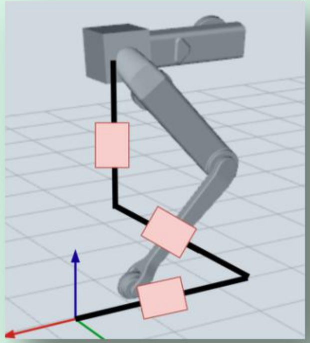
Setting-up a new IoT robotics lab at DISI



Rope-aided locomotion



Aerial maneuvers



RL applied to Jump

Starbot
Jumpleg
All-terrain truck
…
…

Design of new robots

What is a robot?

- Go back to our Longman definition of a robot

A machine that can move and do some of the work of a person and is usually controlled by a computer

- Robots are computer controlled machine that operate in the physical world
- How are they programmed?

The world of industrial robots

- Classic industrial robots are quite “closed” environment that execute repetitive tasks
- Robot programming can be done in two ways
  - Guiding: the robot arm is guided (manually or by a remote controller) through a set of points
  - Off-line: the robot follows a program written in a script language
- The script languages used for programming are often proprietary
  - ….. But extremely simple and similar with each other

Example - Offline programming

- Consider the following example (human readable pseudo code)

```
Move to P1 (a general safe position)
Move to P2 (an approach to P3)
Move to P3 (a position to pick the object)
Close gripper
Move to P4 (an approach to P5)
Move to P5 (a position to place the object)
Open gripper
Move to P1 and finish
```

Example - Offline programming

- Translation into VAL (Unimate)

```
PROGRAM PICKPLACE
         1. MOVE P1
         2. MOVE P2
         3. MOVE P3
         4. CLOSEI 0.00
         5. MOVE P4
         6. MOVE P5
         7. OPENI 0.00
         8. MOVE P1
.END
```

- RC+ (EPSON) with vacuum gripper

```
Function PickPlace
         Jump P1
         Jump P2
         Jump P3
         On vacuum
         Wait .1
         Jump P4
         Jump P5
         Off vacuum
         Wait .1
         Jump P1
Fend
```

Offline languages

- Each vendor has its own scripting language

Robot brand	Language name
ABB	RAPID
Comau	PDL2
Fanuc	Karel
Kawasaki	AS
Kuka	KRL
Stäubli	VAL3
Yaskawa	Inform

- The scripts are interpreted and translated on the fly into real—time actions

Modern Robots

- Modern robots are much more complex
  - They operate in open environment
  - They need sophisticated perception abilities
  - They have to re-plan in real-time in order to reach to unexpected conditions
  - They interact with humans
  - They collaborate with other robots
- All these requirements transform robot programming into a multidisciplinary activity
- Each discipline has its own programming framework and languages.
- So integration can be a titanic effort

The babel tower

- Each community has its own mindset and language

Bruegel, 1563

Software for Robotics: a quick (and incomplete) survey

Activity	Methodologies	Framwork → Languages
Sensing and actuation	Micro-controller programming	FreeRTOS, Proprietary IDE →C
Rigid body kinematics and dynamics libraries	Model based control design	Pinocchio, RobCoGen, RBDL, MATLAB-Simulink → Python,C/C++
Perception (Detection/classification)	Machine Learning	Yolo, OpenCV → Python, C/C++
SLAM, Data fusion	Statistical learning	Matlab-Simulink → C/C++
Motion Planning	Optimisation techniques	Crocoddyl, OCS2→ C++
Task Planning	Discrete optimisation, formal methods	PDDL,… → Python, C++
Simulation	Recreate the physical behavior of robot	Gazebo, Pybullet, NVIDIA Omniverse→ C++

Additional problems

- Many applications require robot-to-robot communication
- Sometimes we have to use external services in the cloud (e.g., for strategic decisions)
- We operate with multiple types of hardware
  - Microcontrollers
  - GPU
  - Industrial PC
- The different computations have timing constraints
  - Particularly true for unstable systems like drones or legged robots
- Finally, most of the times the developers of SW components for robotics are not necessarily computer experts
  - …not your case ☺

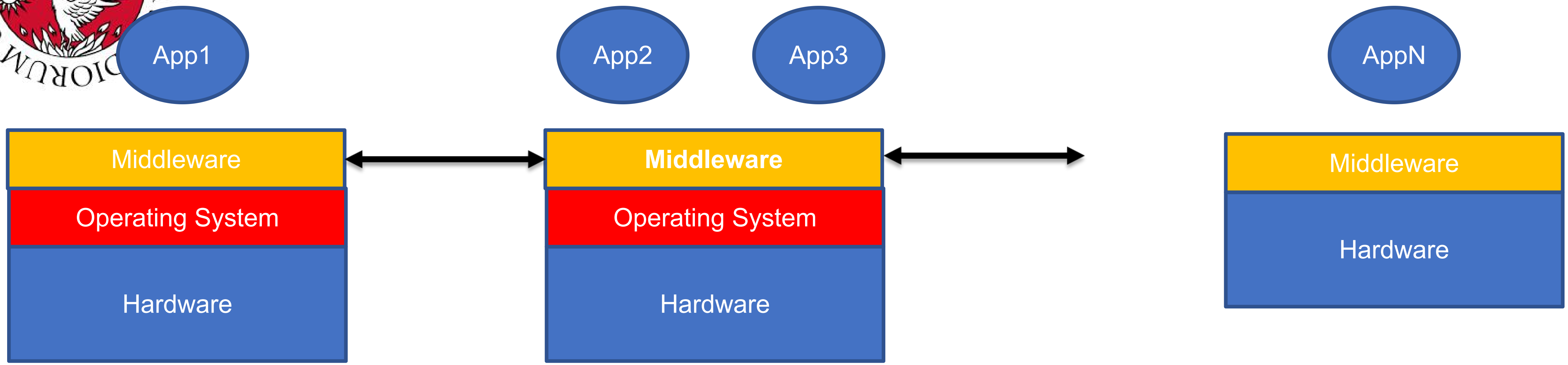
The solution

- The solution is the adoption of a middleware

Middleware: computer software that enables communication between multiple software applications, possibly running on more than one machine.

- Middleware enable the development of distributed, multilingual applications without requiring the direct use of Operating System and networking primitives

The concept of middleware

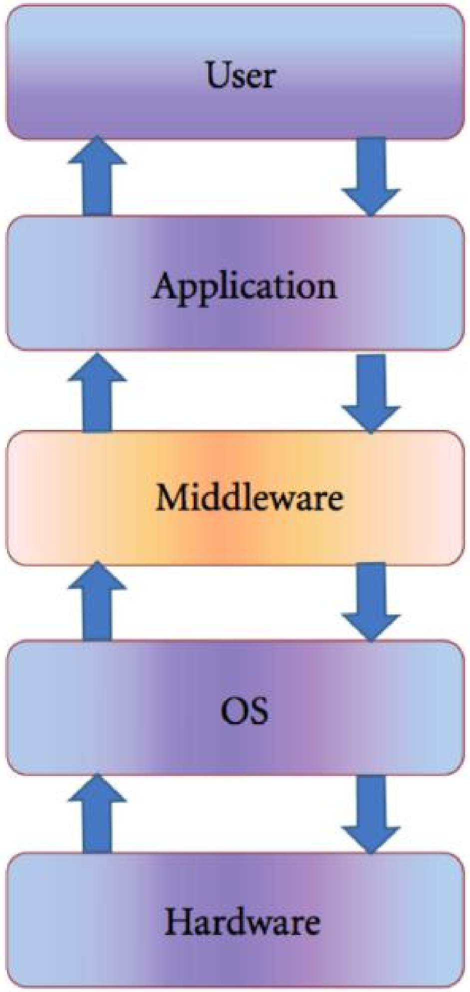


- Applications communicate through standard messages
- They are unaware of where the partner application is located (it can be on the same node or anywhere)
- The middleware sees to the correct delivery of the messages
  - E.g., by implementing a publish-subscribe mechanism
- Applications can be written in any language as long as they get connected through a client library
- Applications do not rely directly on any OS (which in some case is not there)

The concept of middleware

- In a word a Middleware is an abstraction layer that significantly simplifies the development and the integration of distributed applications
- Middleware provides the low-level implementation; you can focus on the higer-level logic.
- Well-designed applications separate higer-level logic from communication logic.
- There are many types of middleware

Type	Services	Examples
Message Oriented Middleware	Receiving and sending of messages over distributed applications	Amazon Simple Notification System (SNS), IBM MQ, Amazon AWS IoT Core
Remote Procedure Call (RPC)	Calling procedures on remote systems and performing synchronous or asynchronous interactions	Oracle: Open Network Computing RPC, SOAP (xmlRPC),…
Database middleware	Allowing for direct access to databases	ODBC, JDBC,EDA/SQL …
Embedded Middleware	Supporting embedded applications	zMQ, **ROS**, IoT middlewares….



Middleware for robotics

- **Robot Operating System (ROS)**
  - De-facto standard for hundreds of available components
  - Some issues with complexity and latency… (addressed by ROS 2.0)
- zMQ
  - General message oriented middleware for lightweight embedded applications
  - Usable (and used) in robotics
  - Very low and controlled latencies
  - Integrated into ROS 2.0

In this course, we will use ROS for the huge availability of software and services

Middleware components

Every middleware **must** provide:

- Abstraction from sensors/actuators hardware;
- Communication protocol for data transport.

Every middleware **should** have:

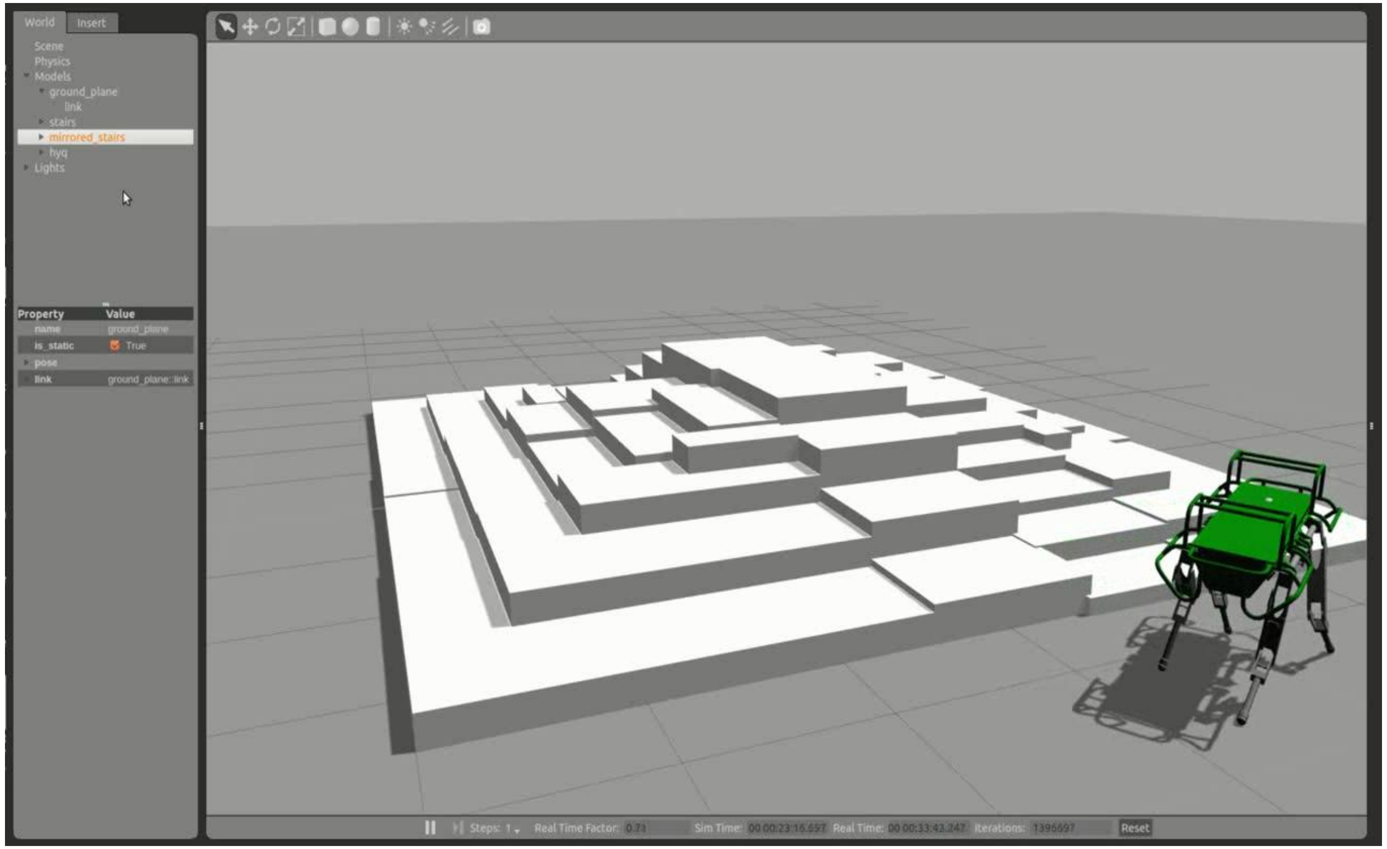
- A tool for takinglogs;
- A tool for playing back logs;
- Tools for timing analysis (latency/throughput).

Simulation

- Modelling and simulation is a standard process in system development

- A simulator needs a mathematical model to predict the robot's behaviour, based on the laws that govern the motion of the mechanical structure
- To support design tasks or to validate new algorithms or controllers

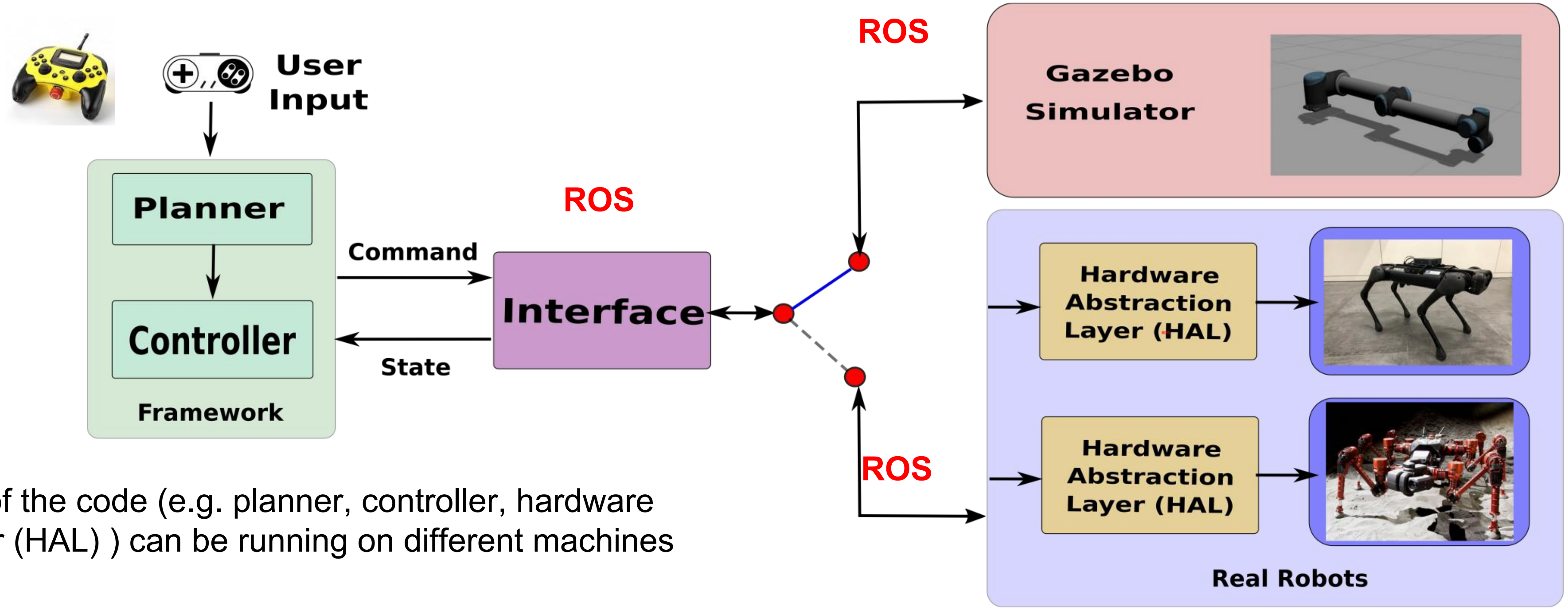
- Limits the number of failures when tests are performed on the real system



Work by V. Barasuol (IIT)

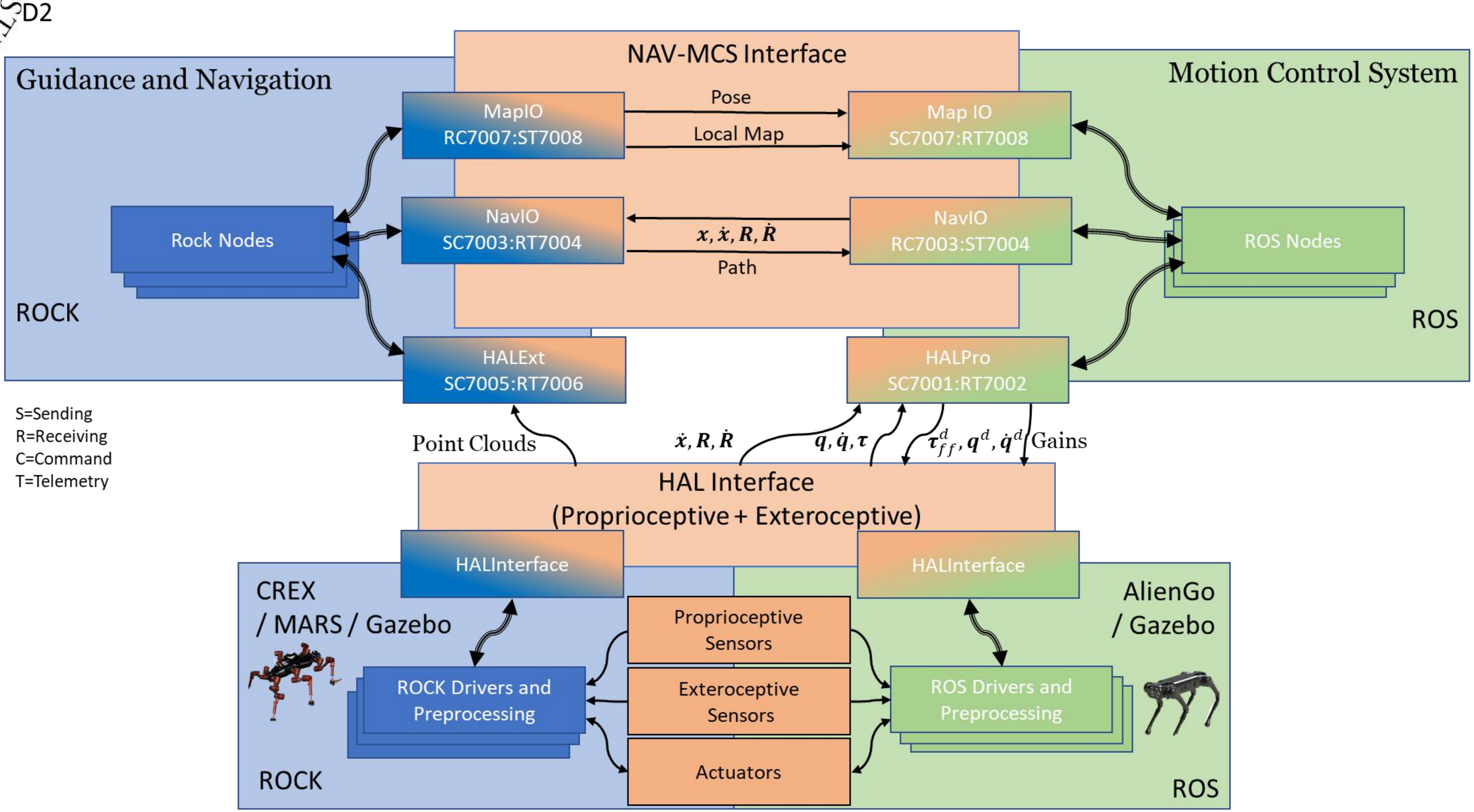
Example of middleware application: the idea of digital twin

- Simulation models that accurately simulate a physical system in real-time are known as digital twin
- the same code can interact with the simulator and the real robot (or different robots) because they share the same **interface**



- different parts of the code (e.g. planner, controller, hardware abstaction layer (HAL)) can be running on different machines
- the implementation of interfaces is simplified by the usage of ROS middleware

Example of interfaces for ESA project



…into ROS….

Historical notes

- The ROS project was started in 2007, the main development of ROS happened at Willow Garage to create software tools for the PR2 robot to develop research projects
- Since 2013 the project was taken over by OSRF (Open source Robotics Foundations)
- Most of the high-end robotics companies are now porting their software to ROS.
- In a few years the knowledge in ROS will be an essential requirement for a robotics engineer.

Main features

- **Peer-to-peer**: applications use a standardized API to exchange messages
- **Distributed:** the framework fully supports applications running on multiple computers
- **Multilingual:** the RoS components can be developed in any language as long as a client library exists (C++, Python, Matlab, Java, Ruby…)
- **Light-weight (especially 2.0):** applications are connected through a very simple and thin layer
- **Free and open-source:** most of RoS applications are open-source and free to use. But, its permissive licensing policy allows for the development of closed and commercial applications

ROS Master

- The ROS master is the manager of the middleware
  - Provides naming and registration services
  - It connects publishers and subscriber to *topics and services*
  - It enables nodes to see each other and establish peer-to-peer communication
  - It provides the Parameter Server (access to a dictionary containing the parameters of all objects)
- It provides an XMLRPC-based API, to which client libraries like *roscpp* and *rospy* connect in order to retrieve information

- The ROS Master activated by

```
> roscore
```

which loads ROS master and other essential components

Why ROS for robotics?

- **Modularity:** issue in most of standalone robotic applications: if any of the threads of main code crashes, the entire robot application crash. In ROS we write different nodes for each process and if one node crashes, the system can still work. Also, ROS provides robust methods to resume operation even if any sensors or motors are dead.

- **Concurrent resource handling:** Handling a hardware resource by more than two processes requires careful implementations for concurrency (e.g. semaphores, shared memory, etc). With topics, any number of ROS nodes can subscribe to the same message (e.g. an image from the ROS driver).

- **Inter-platform operability:** nodes can be programmed in any language that has ROS client libraries. We can write high performance nodes in C++ or C and other nodes in Python or Java. This kind of flexibility is not available in other frameworks.

- **Availability of many packages and tools:** ROS is packed with many tools for debugging, visualizing, and performing simulation, and many packages for existing algorithms (e.g. SLAM) that are highly reconfigurable. This avoids reinventing wheels.

- Supports **high-end sensors and actuators**: ROS is packed with device **drivers** for various sensors (e.g. LIDAR, kinect, Realsense) and actuators in robotics.

ROS Nodes

- A node is a single process delivering a service
- It is an executable program that is compiled and executed
- Nodes can be combined together to form graphs
- Example:
  - One node manages the laser range-finder
  - One node implements localisation
  - One node implements motion planning
  - One node implements wheel control
- The use of nodes allows the developers to **decouple** their work and improve maintainability and robustness of their code

ROS Nodes

- All nodes have a **graph resource name** that uniquely identifies them in the system, e.g.

  `/hokuyo_laser`

- Node also have a **type** (package name + name of the executable file)

- When requested to activate a node, ROS scans the package looking for all nodes with the executable name and chooses the first one

ROS Nodes

- Nodes have to be registered with the master
- They are organised in packages

Execution of a node

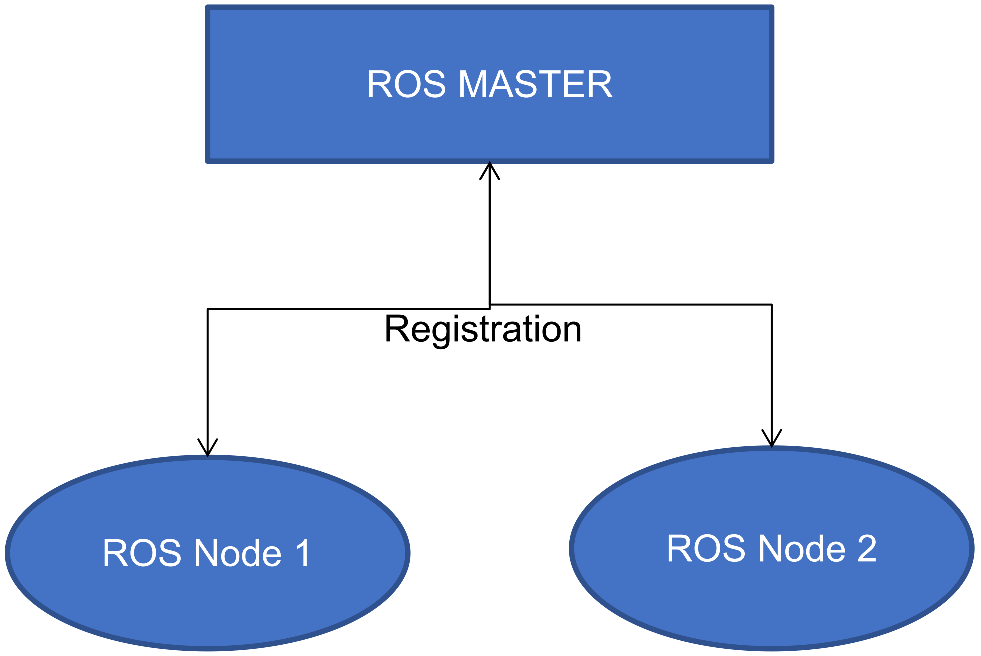
```
rosrun package_name node_name
```

List of active nodes

```
rosnode list
```

Information retrieval on a node

```
rosnode info node_name
```

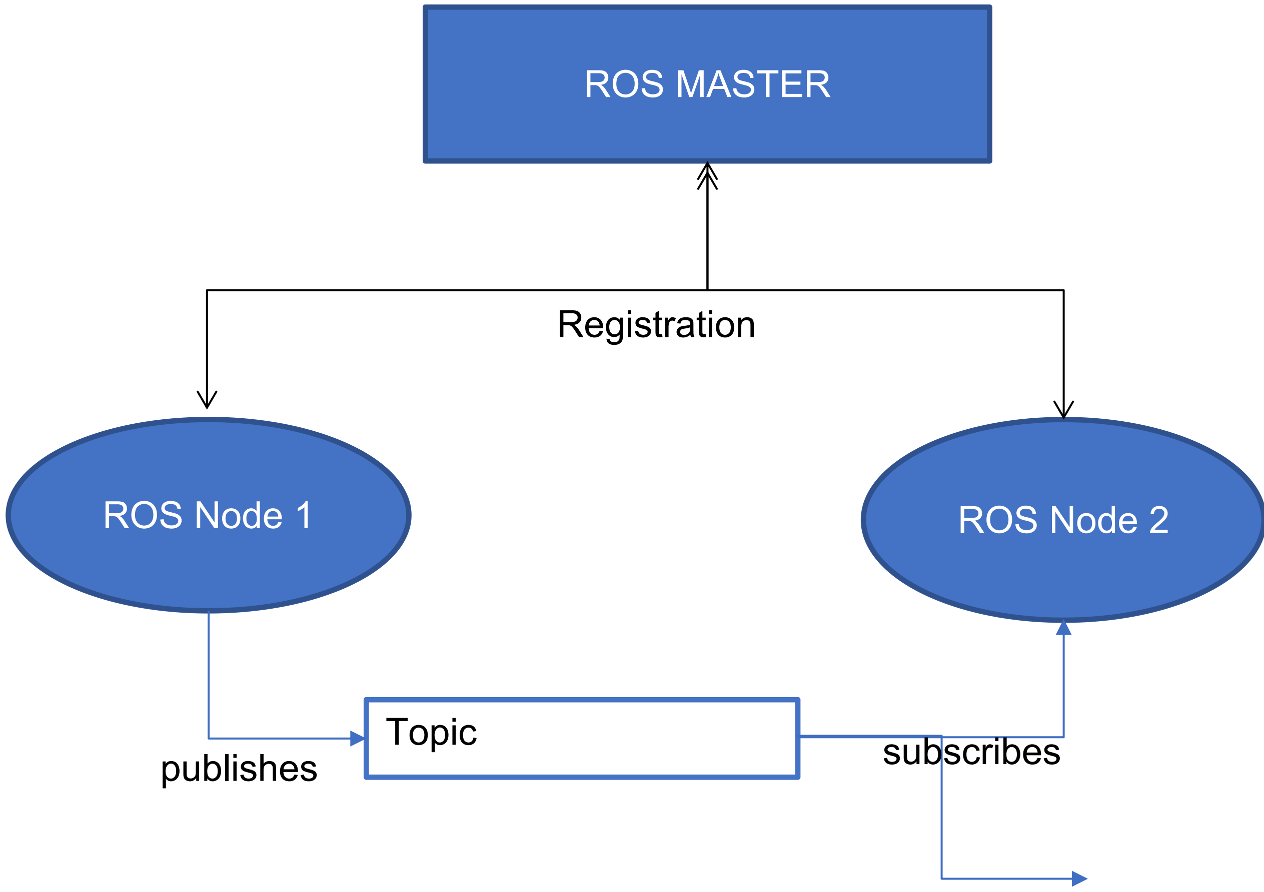


Communication between nodes

- There are three ways to communicate between nodes
  - Streaming topics
  - RPC service
  - Parameter server
- Let us focus on topics and messages

Topics and messages

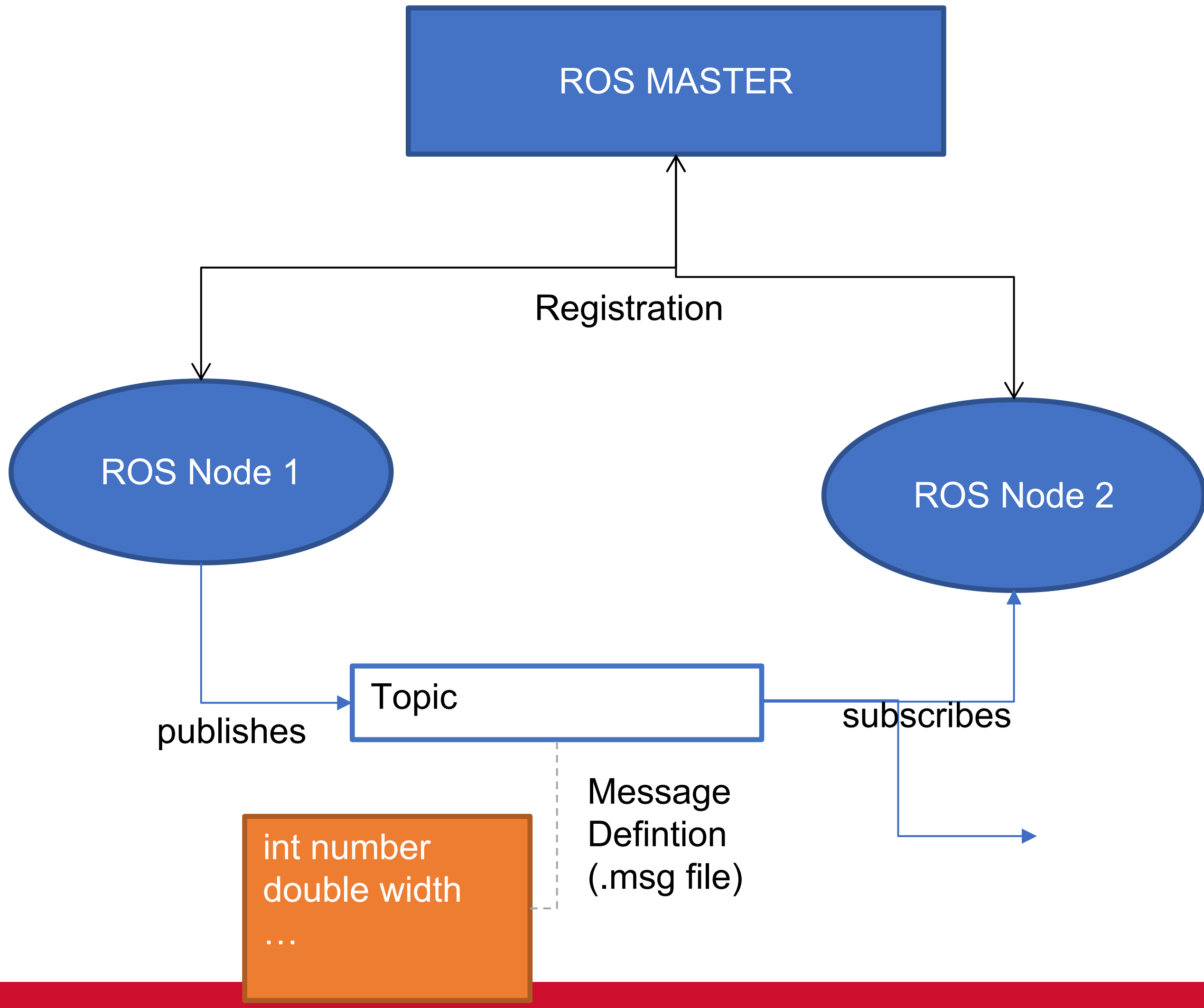
- A **topic** is a name for a stream of messages
- Topics are the primary way for establishing a communication
- Nodes can *publish* or *subscribe* to a topic
- The typical situation is one publisher and multiple subscribers
- This scheme is intended for unidirectional streaming
- The receiver does not need to know the publisher, but only the topic



Messages

- Nodes communicate through topics exchanging ***messages***
- A message defines the type of a topic
- It is defined in a *.msg* file
- A message is a data structure comprising typed fields
- Fields
  - Integers
  - Booleans
  - Strings
  - struct (c-like)

```
>rostopic info /chatter
```



Messages

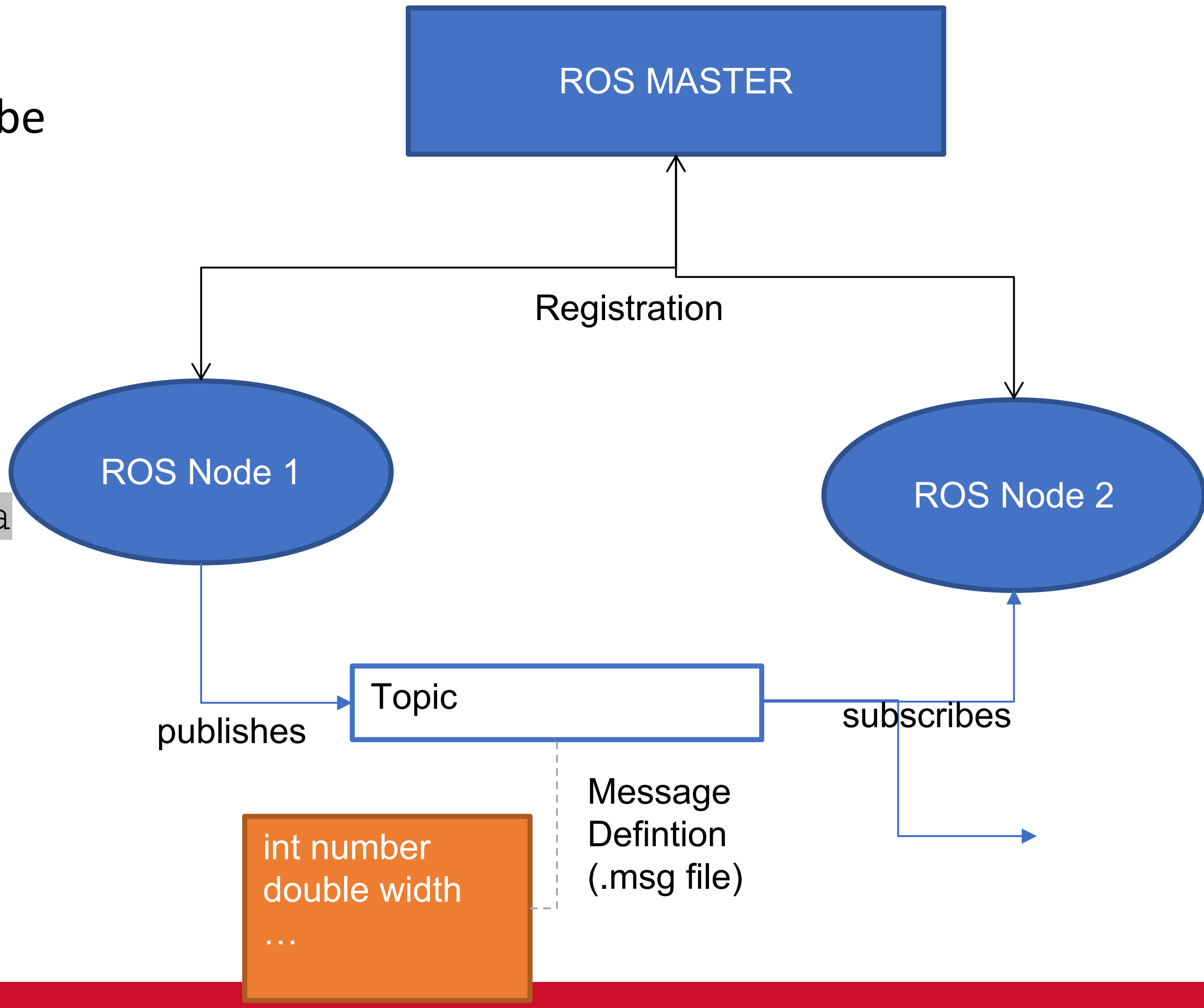
- They type of a topic (i.e., the message structure) can be seen by

  `>rostopic type /topic`

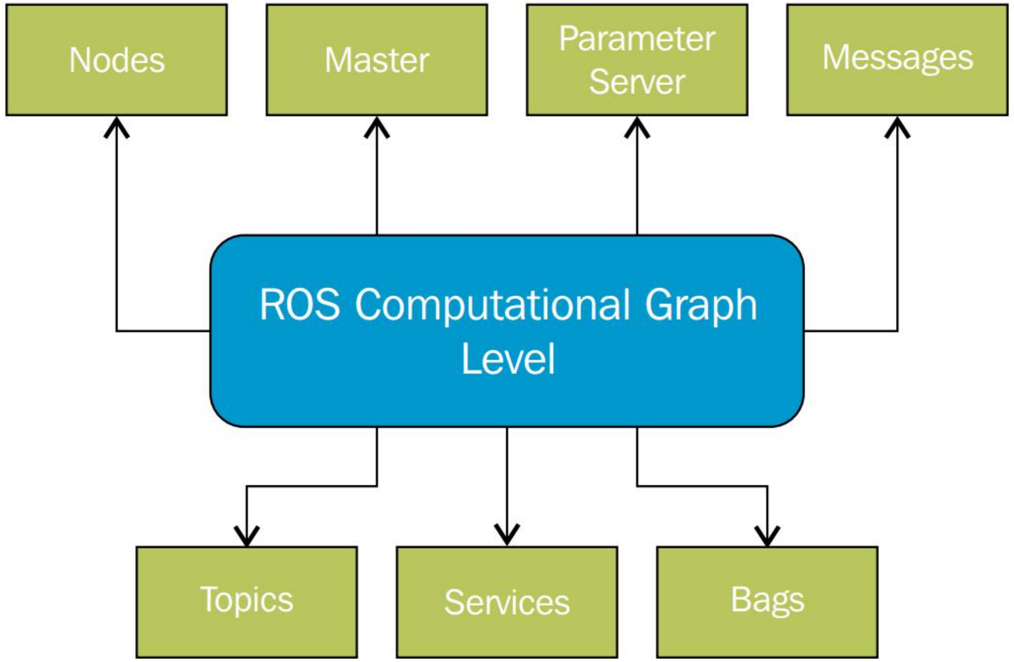
- A message can be published by

  `> rostopic pub topic_name topic_type data`

- Naming convention: package+name of the .msg file. Es.
  `std_msgs/msg/String.msg`



Summary: structure of the ROS Graph layer



- **Nodes:** the process that perform computation.
- **Messages:** Nodes communicate with each other using message (data structures)
- **Topics:** Each message in ROS is transported using named buses called topics.

- The ROS message-passing middleware allows communicating between different nodes.
- When a node sends a message through a topic, we say the node is **publishing** a topic, when a node receives a message through a topic, is **subscribing** to a topic
- **Services:** implement asynchronous request/response interaction.

Message examples

Pose stamped examples

geometry_msgs/Point.msg

```
float64 x
float64 y
float64 z
```

sensor_msgs/Image.msg

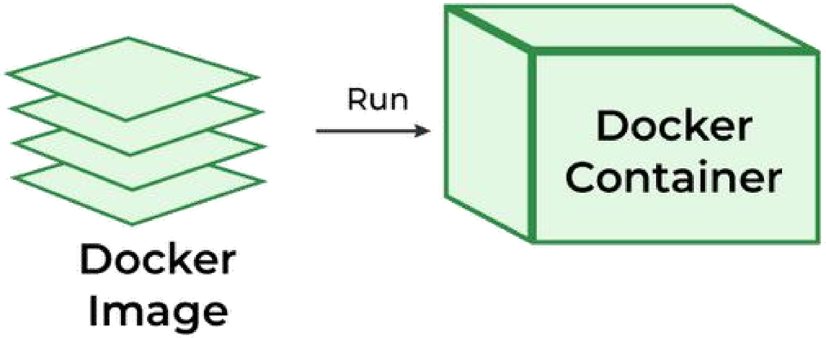
```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

geometry_msgs/PoseStamped.msg

```
std_msgs/Header header
 uint32 seq
 time stamp
 string frame_id
geometry_msgs/Pose pose
  geometry_msgs/Point position
    float64 x
    float64 y
    float64 z
  geometry_msgs/Quaternion orientation
    float64 x
    float64 y
    float64 z
    float64 w
```

How do I get the code? Docker

- Docker is a software development tool and a virtualization technology that makes it easy to develop applications by using containers.

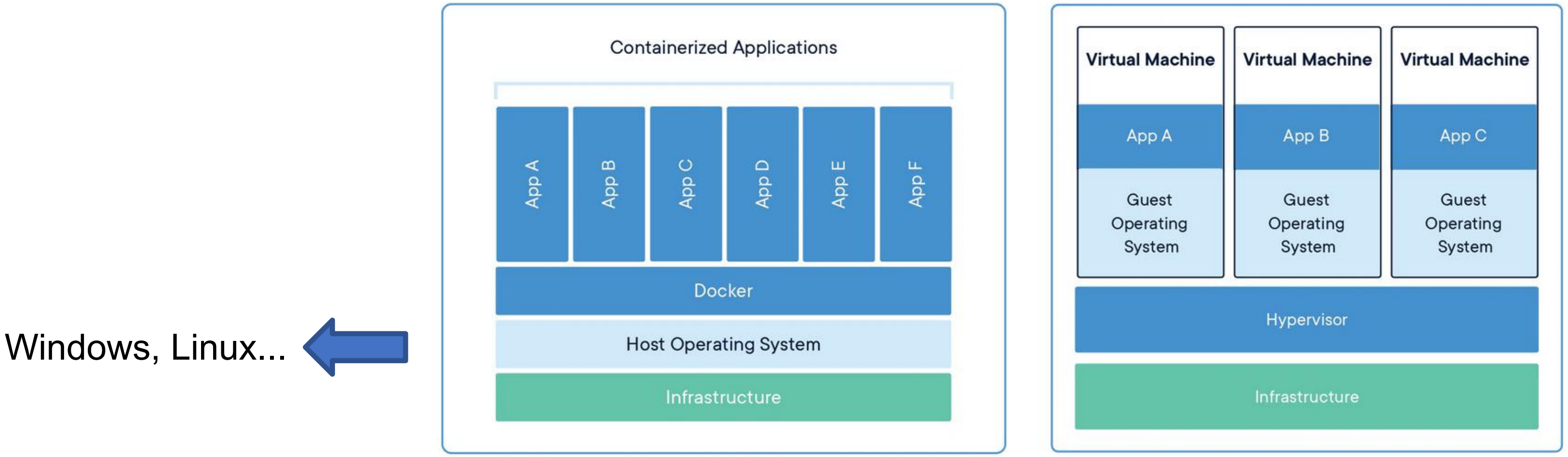


- A container refers to a lightweight, stand-alone, executable package of a piece of software that contains all the libraries, configuration files, and dependencies for your code

- Allow you to have a **platform independent** clean environment with all the necessary dependencies for your code already pre-installed

Docker VS Virtual machine

- A virtual machine is isolated from the rest of the system; the software inside the virtual machine cannot communicate with the host computer.
- Docker containers run directly within the host's machine kernel without the need of an hypervisor (e.g. VirtualBox) like in the case of virtual machines



- A VM includes a full copy of an operating system, the application, necessary binaries and libraries – taking up tens of GBs. VMs are slow to boot and have poor performances (e.g. difficult to support GPU).
- Code running in docker containers is as fast as with the native OS and can be run with the real robot

Installation

- For this course, we provide a docker image with ROS installed together with all the needed dependencies

- you just need to install the **Docker Engine** on your computer, following this tutorial (windows, mac, linux users):

github.com/mfocchi/lab-docker

- Alternatively you can install everything natively (only Linux users):

github.com/mfocchi/locosim

- or download a virtual machine (windows, mac, linux users):

www.dropbox.com/sh/5trh0s5y1xzdjds/AACchznJb7606MbQKb6-fUiUa

Docker usage

- After the docker installation you need to pull the docker image (only once):

```
docker pull mfocchi/trento_lab_framework:introrob
```

- Now that you have an image locally, you can open a new container running in a new terminal (lab alias):

```
lab-docker.py --api run    -f -nv   mfocchi/trento_lab_framework:introrob
```

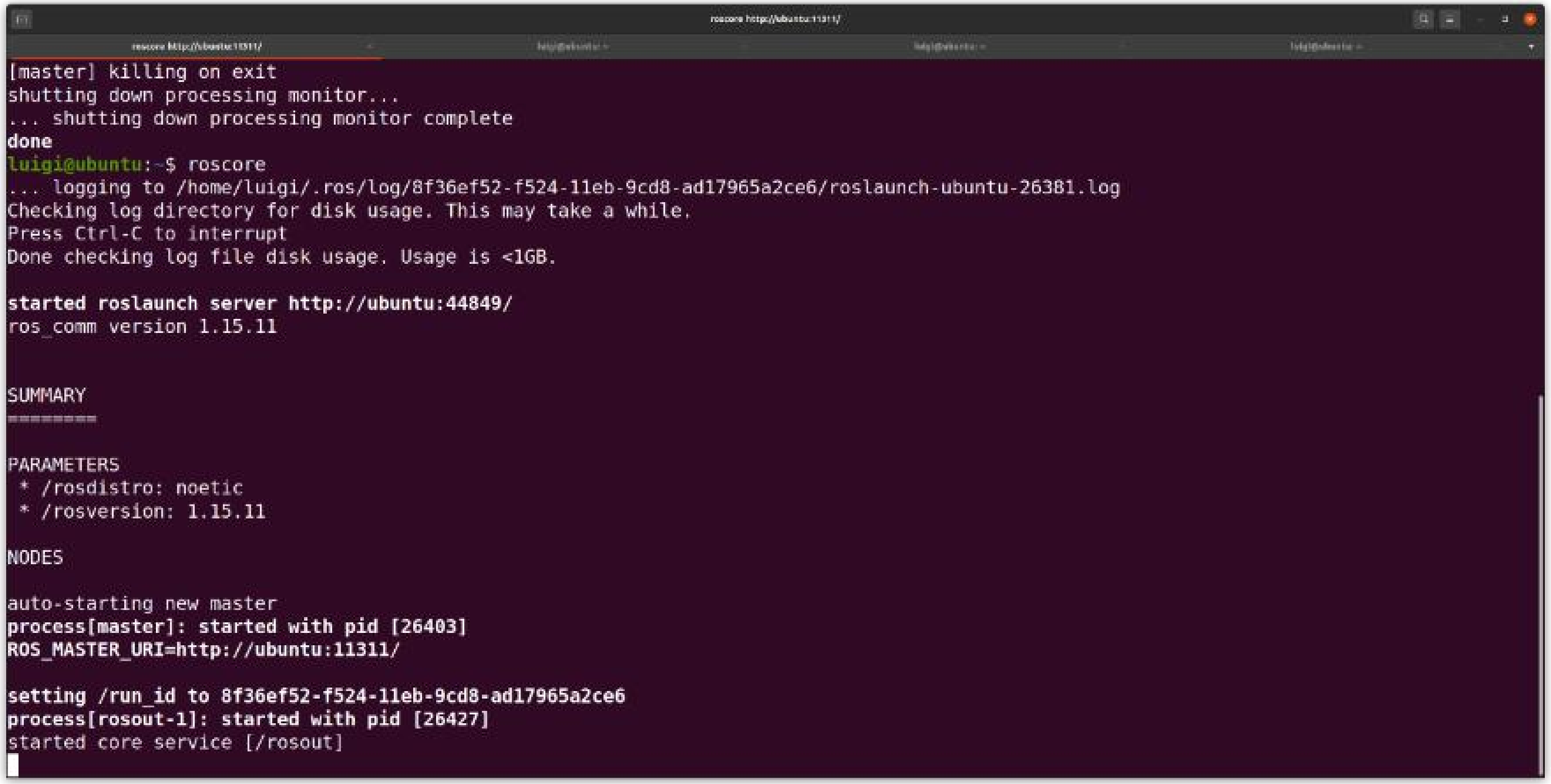
- The **lab-docker.py** script will create the folder **~/trento_lab_home** on your host computer. This folder is mapped to **$HOME** inside the docker container.
- This means that any files you place in your **~/trento_lab_home** folder will survive the stop/starting of a new docker container. All other files and installed programs will **disappear** on the next run. We will implement our new code there.
- To link other terminals to the same image you should a command that will "attach" to the image previously opened without killing it (dock-other alias):

```
lab-docker.py attach
```

Example

- Let us look at a first example going through the ROS tutorial
- Step 1: open a console and activate the master

```
> roscore
```



Example

- Step 2: start a talker demo

```
> rosrun roscpp_tutorials talker
```

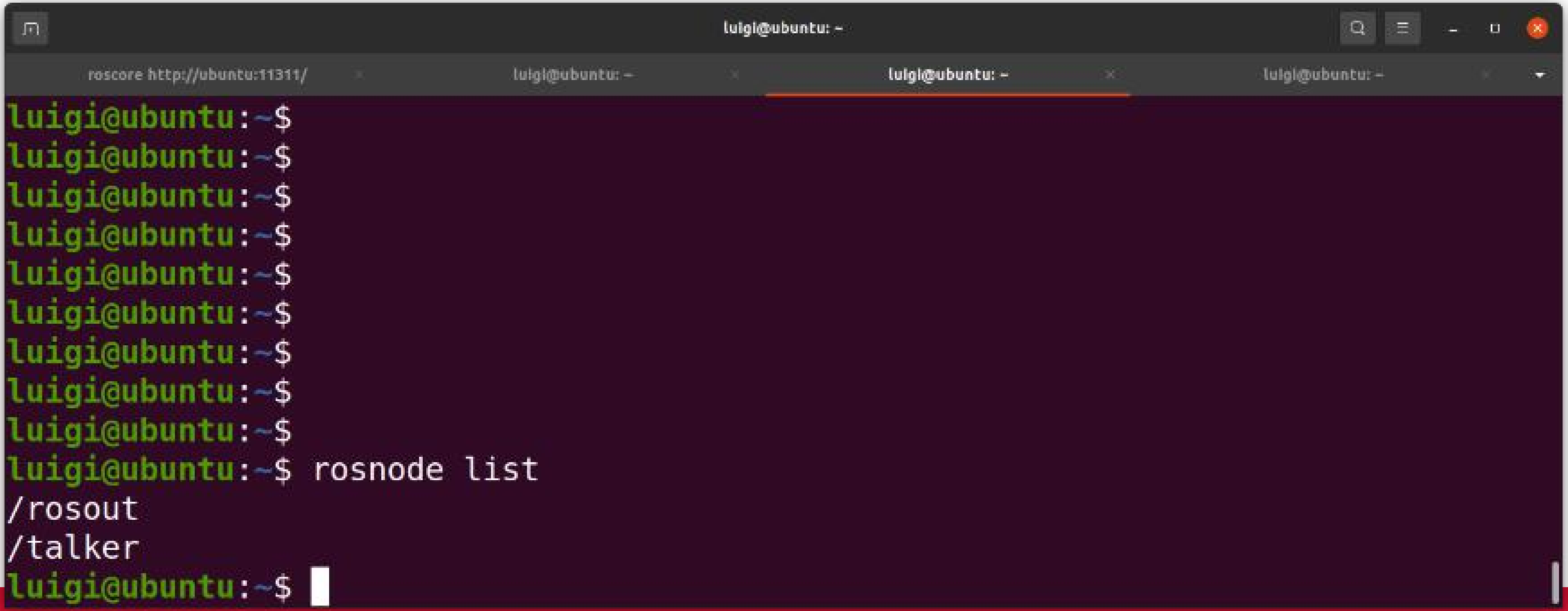
```
[ INFO] [1628082567.813881812]: hello world 403
[ INFO] [1628082567.913858930]: hello world 404
[ INFO] [1628082568.013885830]: hello world 405
[ INFO] [1628082568.114072895]: hello world 406
[ INFO] [1628082568.213900278]: hello world 407
[ INFO] [1628082568.313915462]: hello world 408
[ INFO] [1628082568.413882588]: hello world 409
[ INFO] [1628082568.513895009]: hello world 410
[ INFO] [1628082568.613892970]: hello world 411
[ INFO] [1628082568.713920795]: hello world 412
[ INFO] [1628082568.813914516]: hello world 413
[ INFO] [1628082568.913869450]: hello world 414
[ INFO] [1628082569.013920703]: hello world 415
[ INFO] [1628082569.113861715]: hello world 416
[ INFO] [1628082569.213892758]: hello world 417
[ INFO] [1628082569.313840988]: hello world 418
[ INFO] [1628082569.413904962]: hello world 419
[ INFO] [1628082569.513914834]: hello world 420
[ INFO] [1628082569.613911691]: hello world 421
[ INFO] [1628082569.713928499]: hello world 422
[ INFO] [1628082569.813884481]: hello world 423
[ INFO] [1628082569.914040760]: hello world 424
[ INFO] [1628082570.013885774]: hello world 425
[ INFO] [1628082570.113908228]: hello world 426
[ INFO] [1628082570.213895286]: hello world 427
[ INFO] [1628082570.313882078]: hello world 428
[ INFO] [1628082570.413871490]: hello world 429
[ INFO] [1628082570.513843144]: hello world 430
[ INFO] [1628082570.613848896]: hello world 431
[ INFO] [1628082570.713889876]: hello world 432
```

Example

- Step 3.1: analyse the newly activated node

  List of all active nodes

```
> rosnode list
```

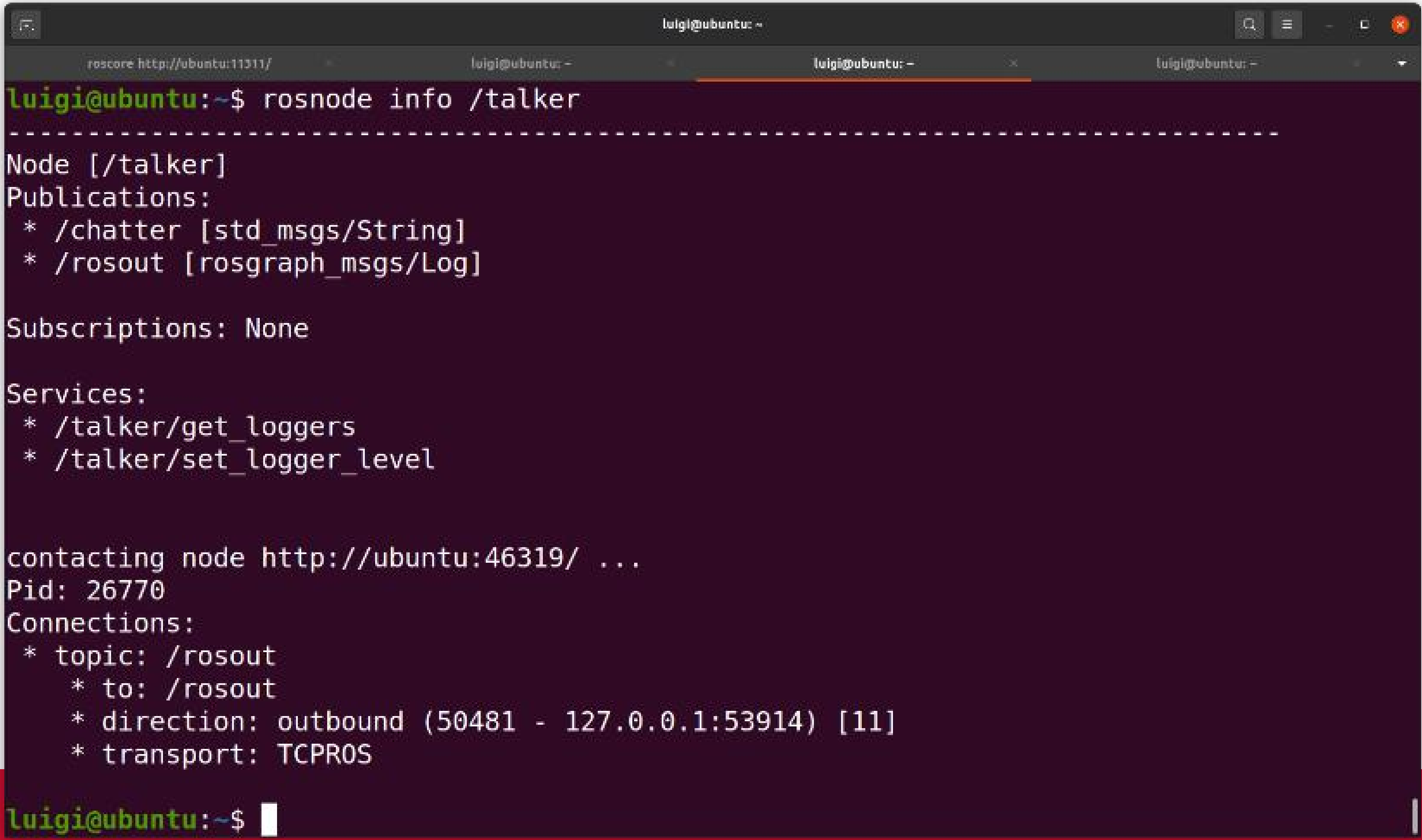


Example

- Step 3.2: analyse the newly activated node

Info about talker

```
> rosnode info /talker
```

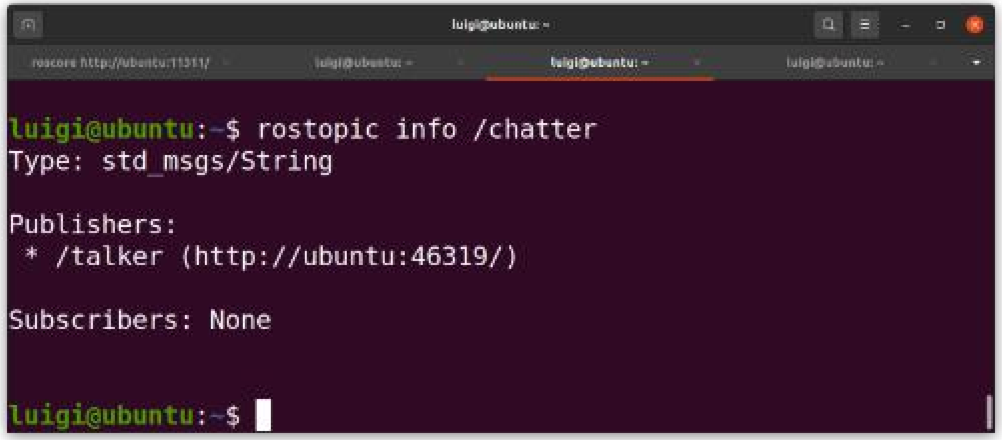


Example

- Step 3.3: analyse the newly activated node

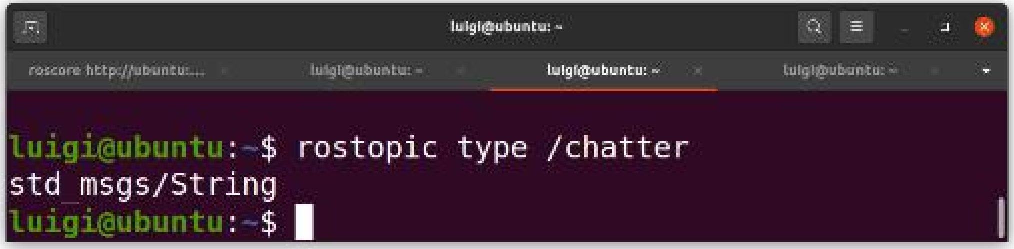
Info about the *chatter* topic

```
> rostopic info /chatter
```



Type check

```
> rostopic type /chatter
```



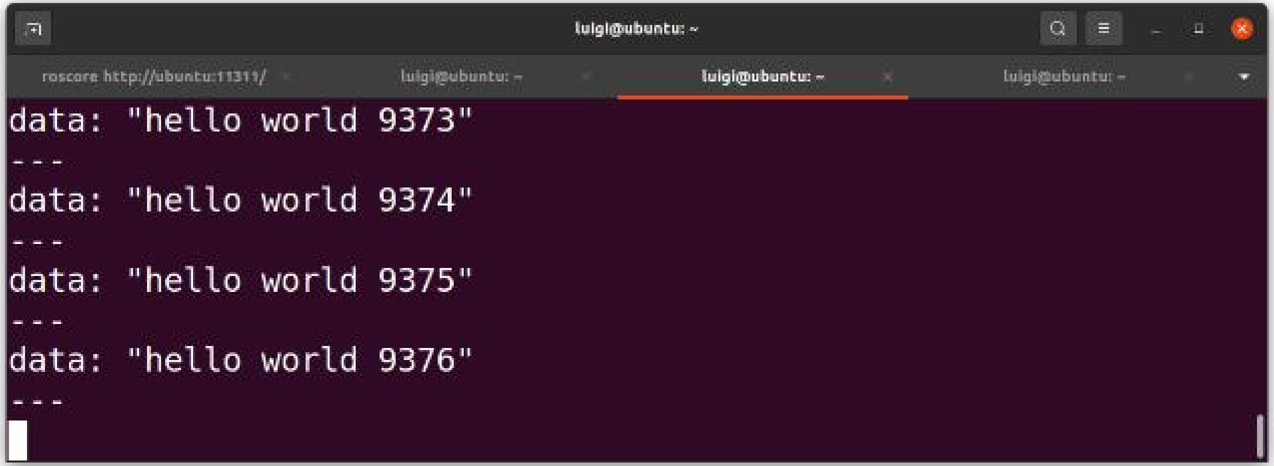
Example

- Step 3.3: analyse the newly activated node

Some more info about the *chatter* topic

Message content

`rostopic echo /chatter`



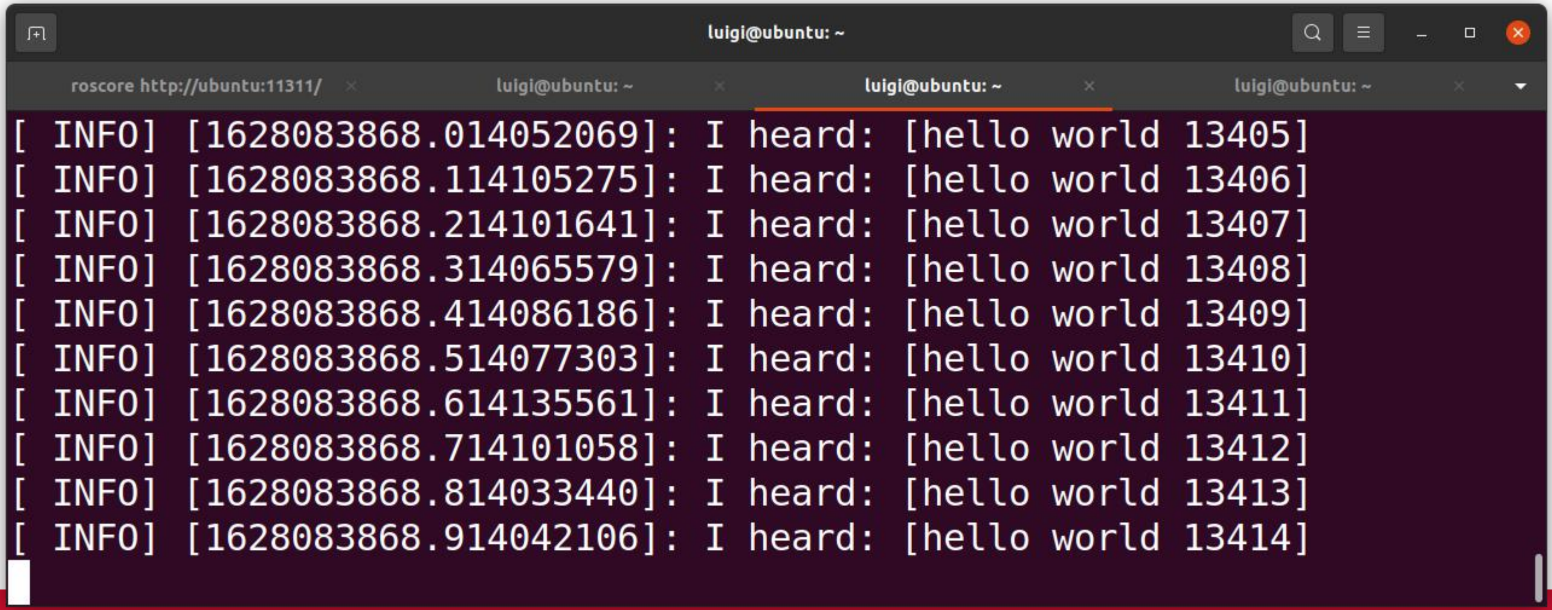
Frequency check

`rostopic hz /chatter`

Example

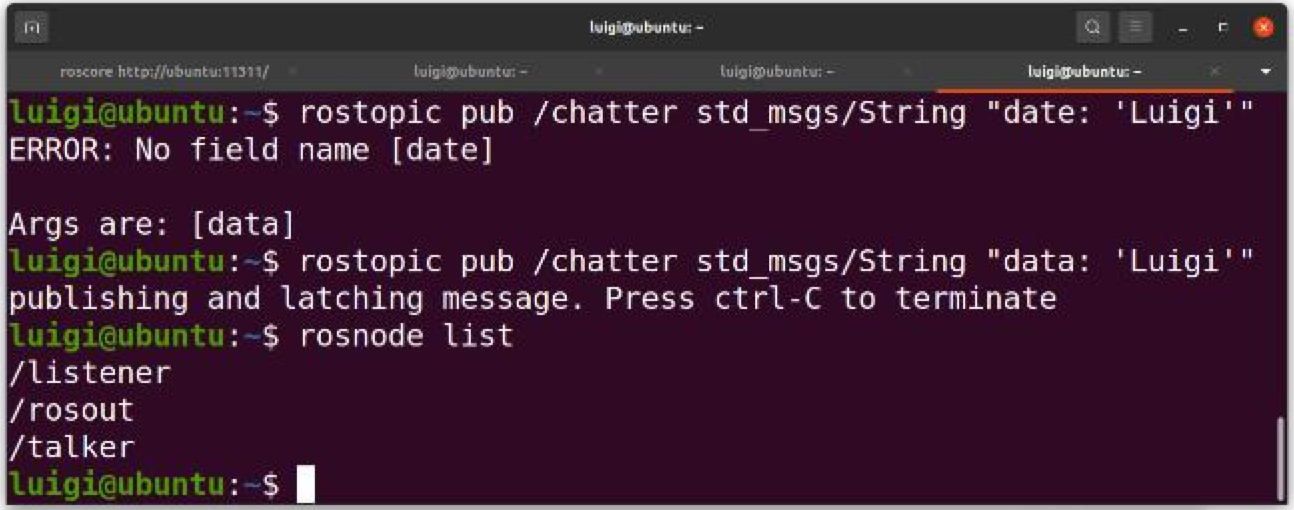
- Step 4: start a listener node
  Move to a different console and type

```
>rosrun roscpp_tutorials listener
```



Example

- Step 5: Analyse the new graph
  Move to a different console and type



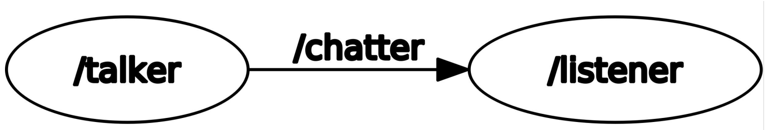

```
>rosnode list
```

Connection between the nodes through chatter

```
>rosnode info /chatter
```

Check GUI plugin for visualizing the ROS computation graph.

```
>rosrun rqt_graph rqt_graph
```



Example

- Step 5: Publish a message from console

Move to the *talker* console and close the node by ctrl+c

Publish message through

```
>rostopic pub /chatter std_msgs/String “data: ‘my message’”
```

```
luigi@ubuntu:~$ rostopic pub /chatter std_msgs/String "data: 'my messa
ge'"
publishing and latching message. Press ctrl-C to terminate
```

Check the output on the listener console

```
[ INFO] [1628084786.014510482]: I heard: [hello world 22585]
[ INFO] [1628084786.114515159]: I heard: [hello world 22586]
[ INFO] [1628084786.214654978]: I heard: [hello world 22587]
[ INFO] [1628084786.314529753]: I heard: [hello world 22588]
[ INFO] [1628084792.822829555]: I heard: [my message]
```

Before moving on

- For the examples in the next slides, we will assume to have rqt and turtlesim installed, the second for illustration purposes
- If you have not done it already, take your time and type in your linux console

```
>sudo apt-get install ros-noetic-rqt ros-noetic-rqt-common-plugins ros-noetic-turtlesim
```

ROS Services

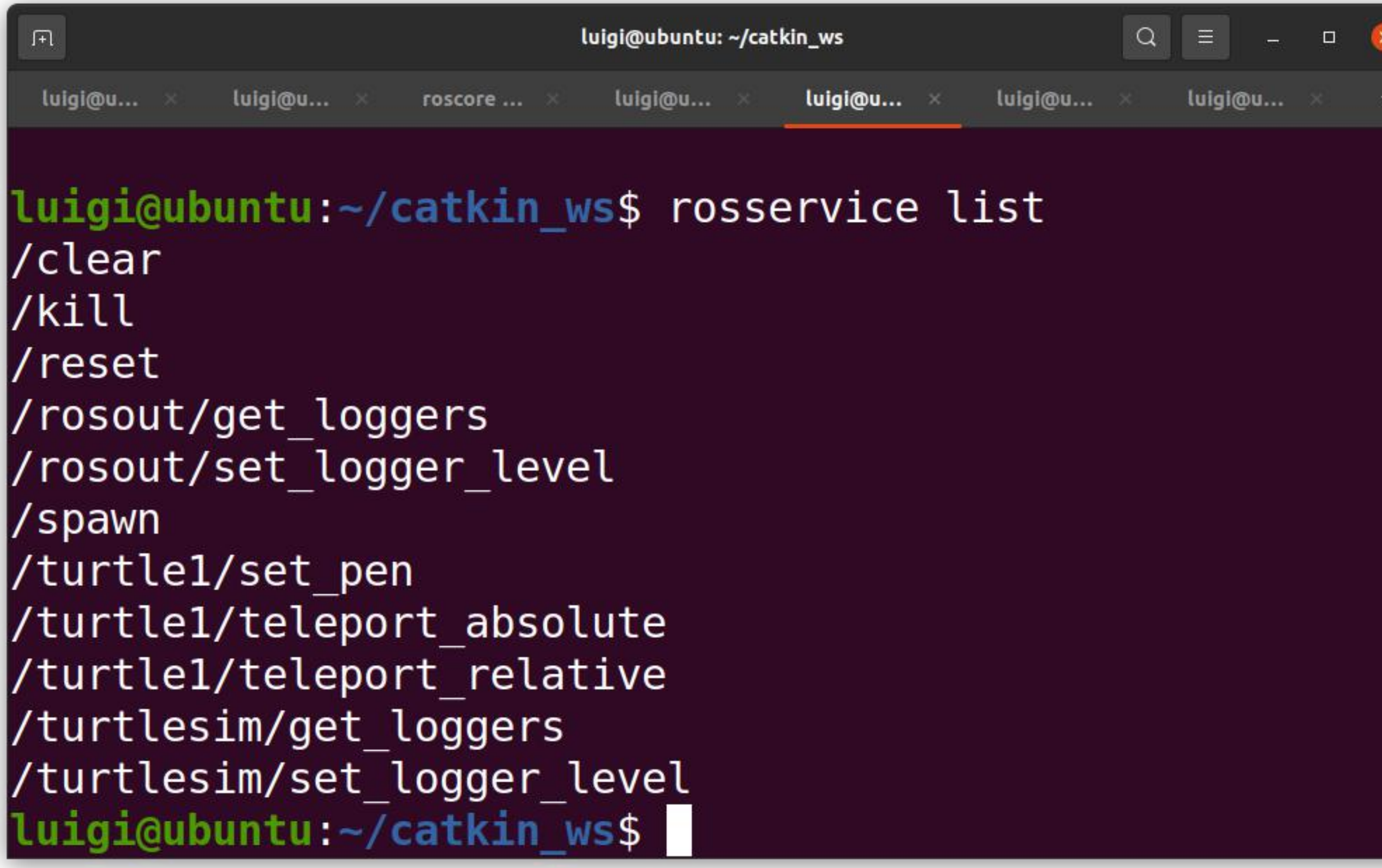
- Services are a different way for two nodes to communicate
- Services allow a node to send a request and receive a response
- The commands related to services are

```
rosservice list         print information about active services
rosservice call         call the service with the provided args
rosservice type         print service type
rosservice find         find services by service type
rosservice uri          print service ROSRPC uri
```

ROS Services

- If we try:

```
>rosrun turtlesim turtlesim_node
>rosservice list
```



- We can see that the turtlesim node provides nine services: reset, clear, spawn, kill.

ROS Services

- Let us check the type of a services

```
>rosservice type /clear
```

- We can see that the service is empty (meaning that it takes no argument)
- The service can be invoked by

```
>rosservice call /clear
```

which clear the screen of the turtle simulator

ROS Services

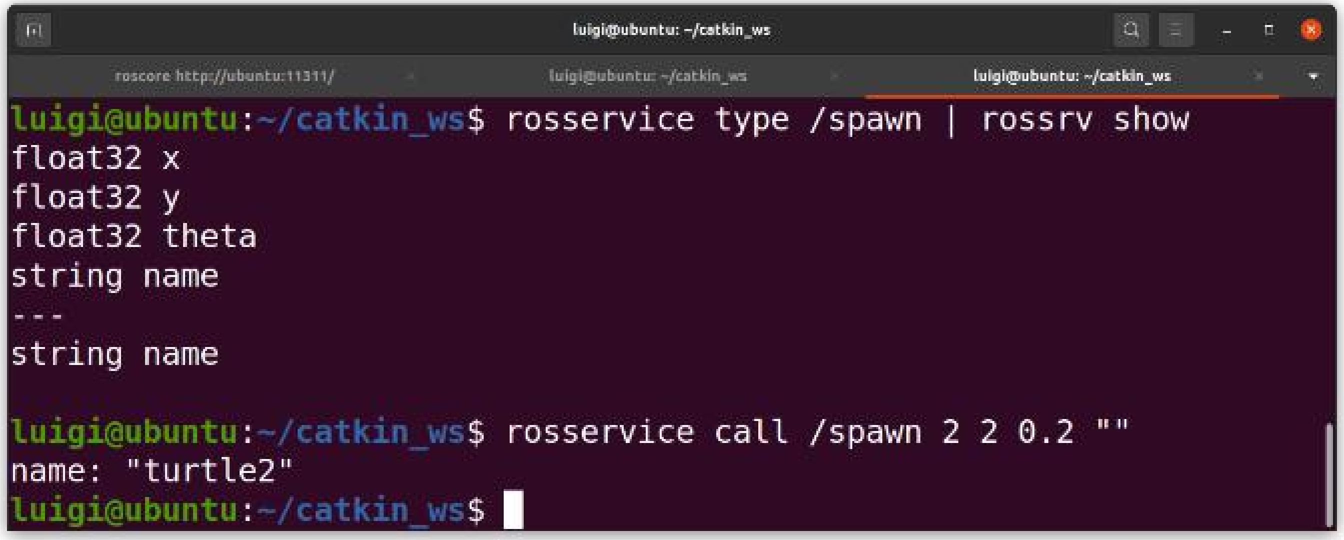
- Let us now try a service with arguments. Type

`>rosservice type /spawn | rossrv show`

- Followed by

`>rosservice call /spawn 2 2 0.2 ""`

- The outcome is to create a new turtle in a different position



ROS Parameters

- The program `rosparam` allows us to create and manipulate data into the ROS parameter server
- The possible commands to manipulate parameters are

```
rosparam set            set parameter
rosparam get            get parameter
rosparam load           load parameters from file
rosparam dump           dump parameters to file
rosparam delete         delete parameter
rosparam list           list parameter names
```

ROS Parameters

- Example (with roscore open in another console)

`>rosparam list`

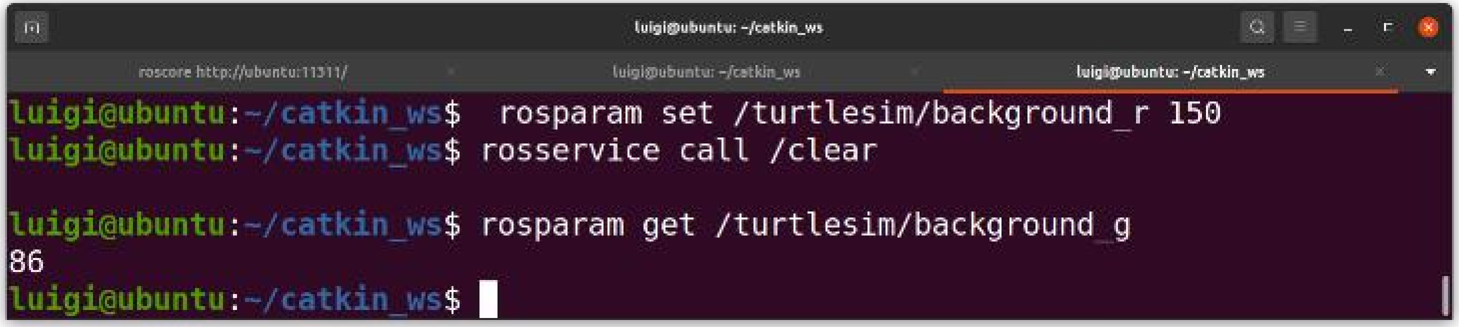
```
luigi@ubuntu:~/catkin_ws$ rosparam list
/rosdistro
/roslaunch/uris/host_ubuntu__38083
/rosversion
/run_id
/turtlesim/background_b
/turtlesim/background_g
/turtlesim/background_r
luigi@ubuntu:~/catkin_ws$
```

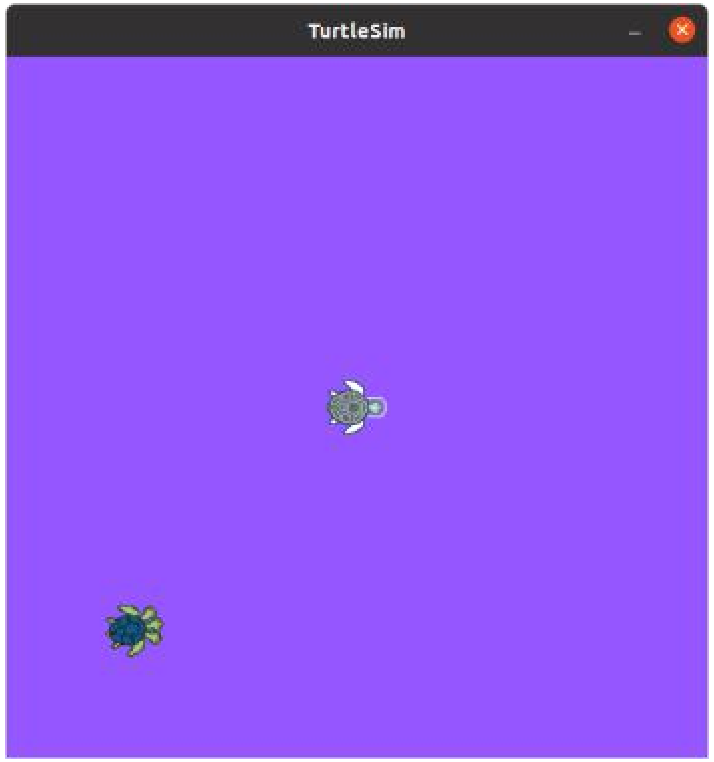
- We can see there is a param for background colour

ROS Parameters

- Let us try

```
>rosparam set /turtlesim/background_r 150
>rosservice call /clear
>rosparam get /turtlesim/background_r
```



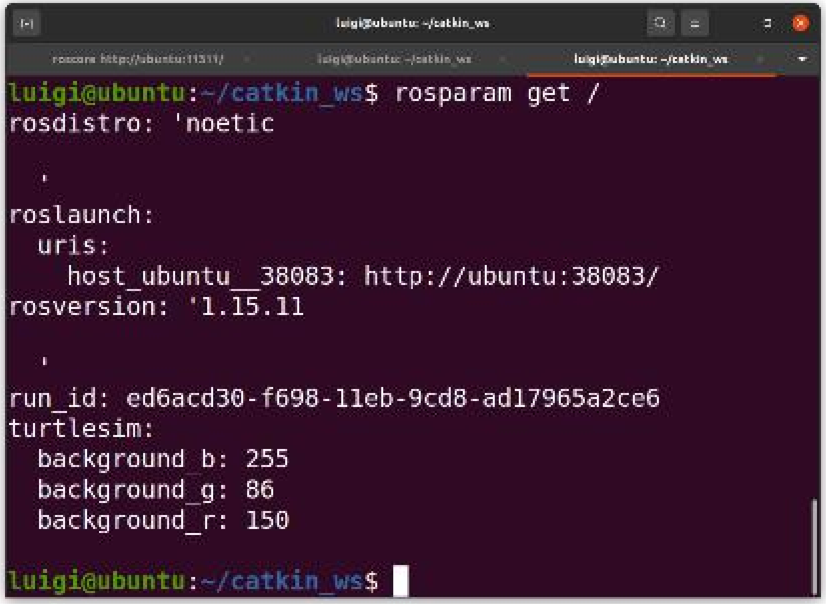


- We will change the colour of the simulation background and look up the red component

ROS Parameters

- You can print out the entire Parameter set by

```
>rosparam get /
```



- You can dump into file an retrieve by:

```
>rosparam dump [file_name]
>resparam load [file_name]
```

Debugging and launching nodes

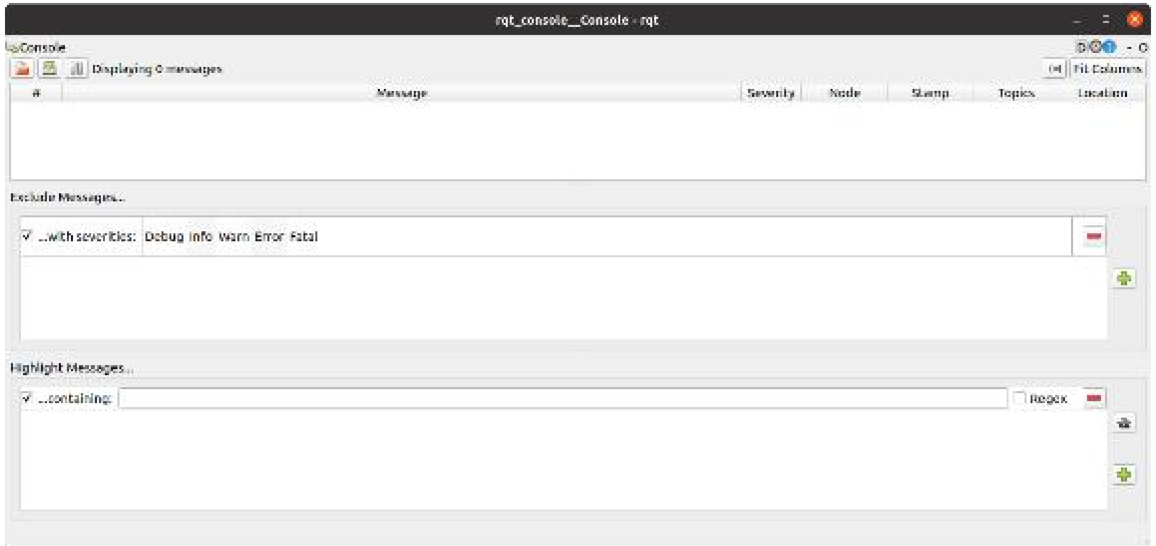
- Also for this part, we will assume to have rqt and turtlesim packages installed
- `rqt_console` attaches to the ROS's logging framework to display output from loggers
- The rqt_logger_level allows us to change the verbosity level (DEBUG, INFO, WARN, ERROR, FATAL)
- The four levels are in increasing level of criticality (DEBUG passes all, FATAL only the critical messages)

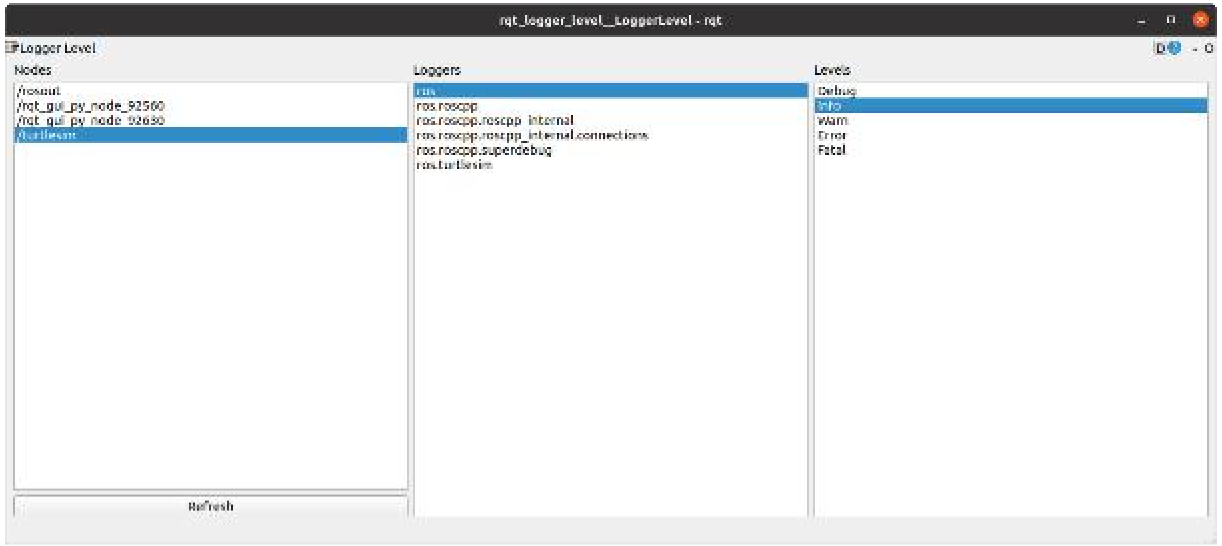
Using rqt_console

- Just type *in two different terminals* the following:

`>rosrun rqt_console rqt_console`

`>rosrun rqt_logger_level rqt_logger_level`





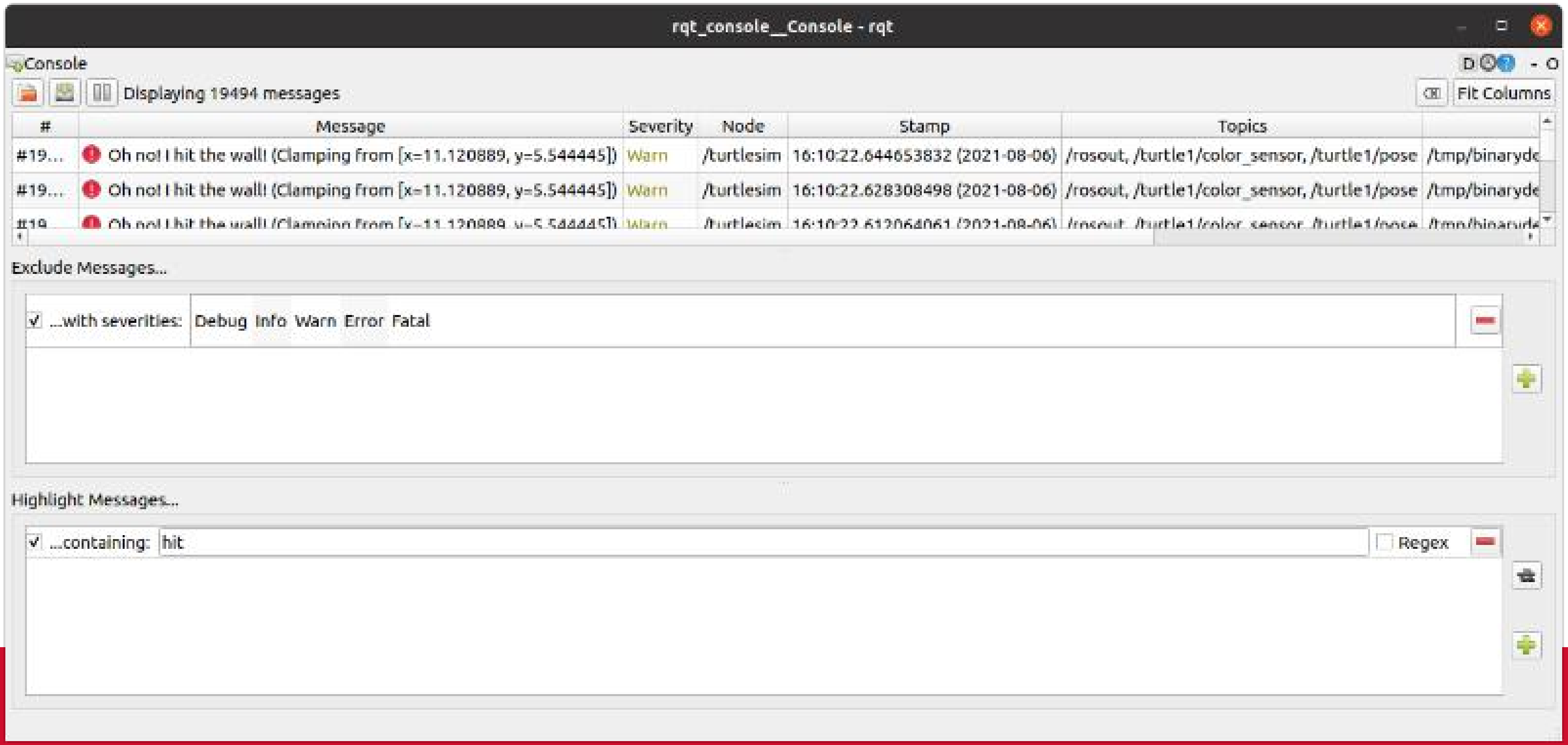
Using rqt_console

- Now let us activate turtlesim and let us make the turtle walk until it crashes against the wall (use two different terminals)

```
>rosrun turtlesim turtlesim_node
```

```
>rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 --
'[1., 0.0, 0.0]' '[0.0, 0.0, 0.0]'
```

- On the Console we will start seeing



Using rqt_console

- If we now set the turtlesim logger level to Error, the message stream in the console will stop