THE BIOROBOTICS INSTITUTE

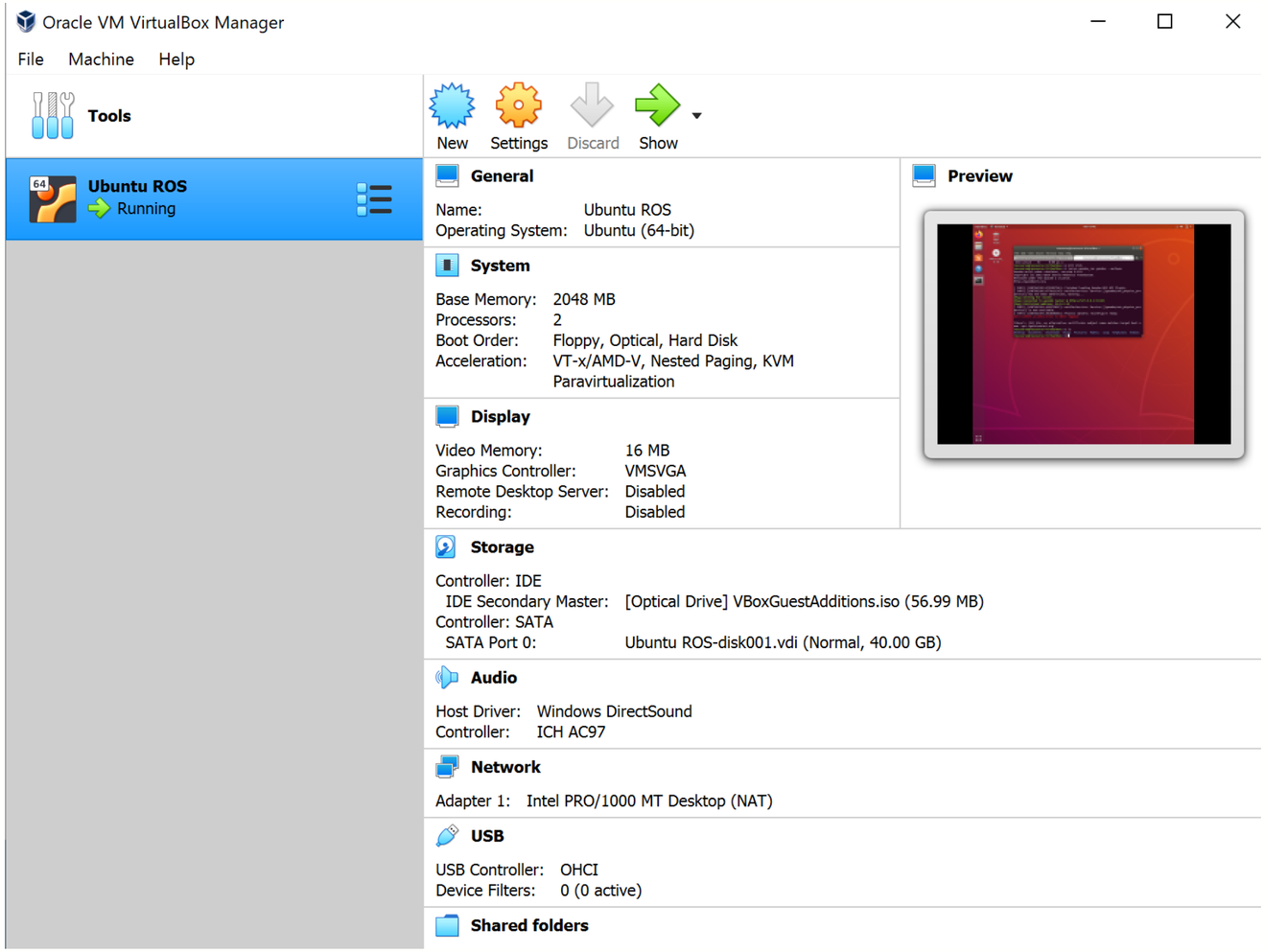
Sant'Anna
School of Advanced Studies – Pisa

Robotic middlewares and ROS(1) - Introduction

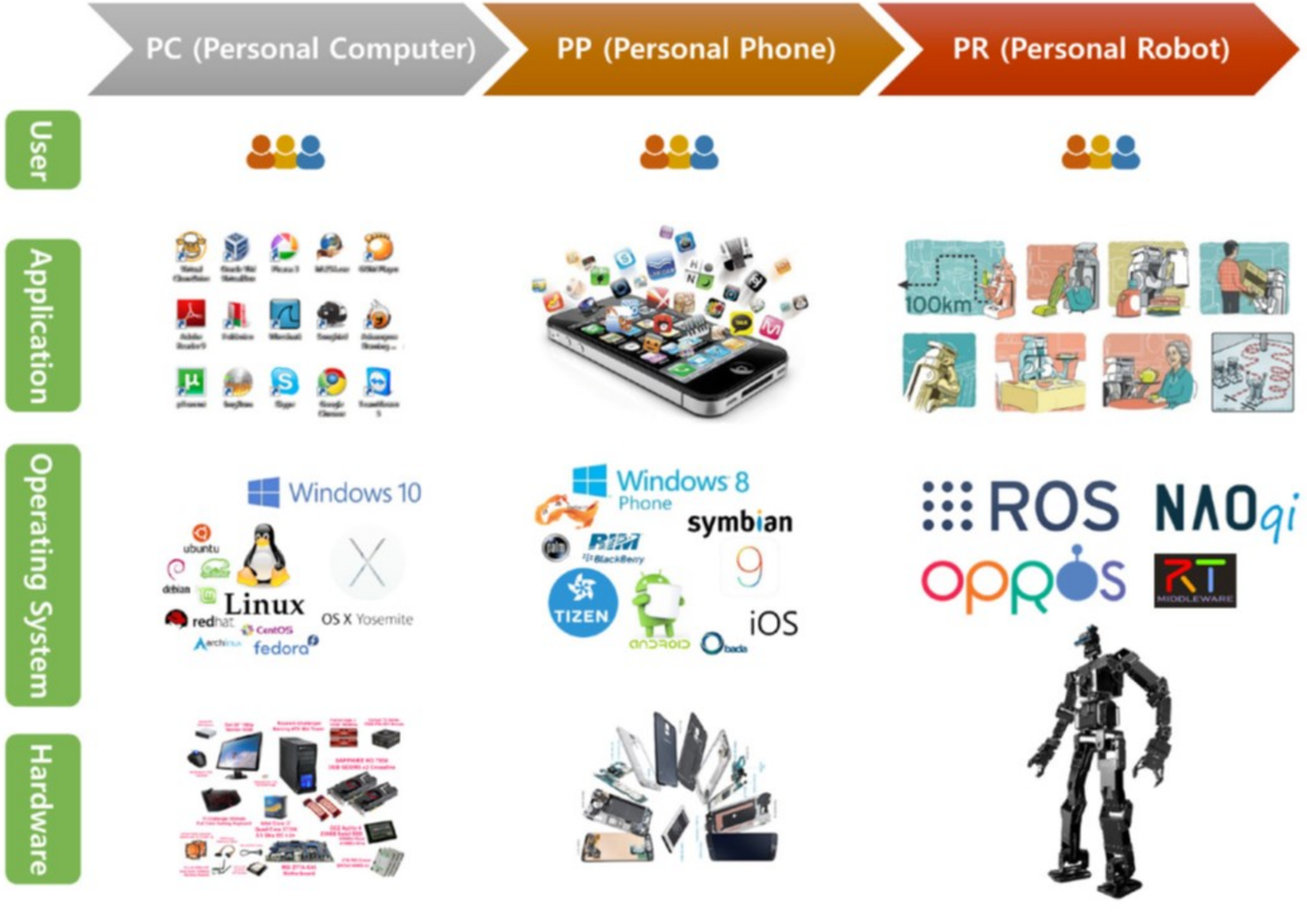
Egidio Falotico

TOOLS


ORACLE
VirtualBox


Oracle VM VirtualBox Manager
File
Machine
Help
Tools
New
Settings
Discard
Show
Ubuntu ROS
Running
General
Name: Ubuntu ROS
Operating System: Ubuntu (64-bit)
System
Base Memory: 2048 MB
Processors: 2
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, KVM
Paravirtualization
Display
Video Memory: 16 MB
Graphics Controller: VMSVGA
Remote Desktop Server: Disabled
Recording: Disabled
Preview
Storage
Controller: IDE
IDE Secondary Master: [Optical Drive] VBoxGuestAdditions.iso (56.99 MB)
Controller: SATA
SATA Port 0: Ubuntu ROS-disk001.vdi (Normal, 40.00 GB)
Audio
Host Driver: Windows DirectSound
Controller: ICH AC97
Network
Adapter 1: Intel PRO/1000 MT Desktop (NAT)
USB
USB Controller: OHCI
Device Filters: 0 (0 active)
Shared folders

Platforms Components



Robot Software Platform

A platform is divided into software platform and hardware platform.

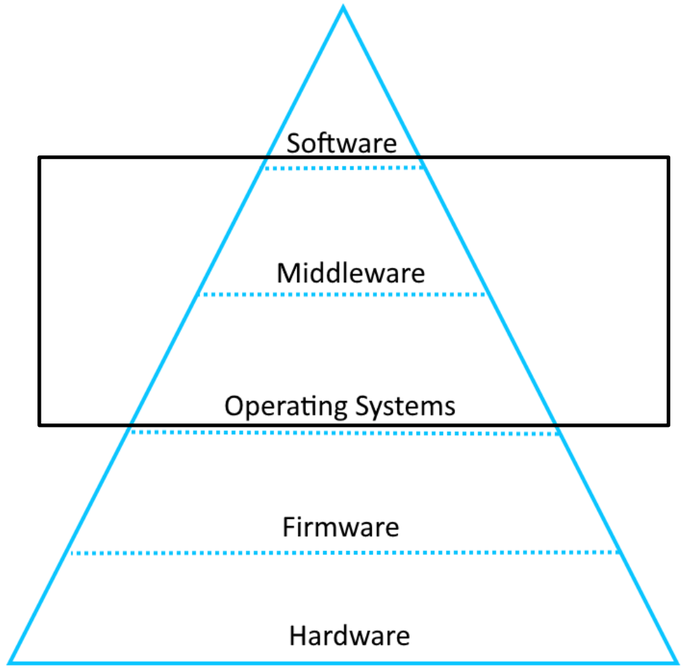
A robot software platform includes tools that are used to develop robot application programs such as:

- hardware abstraction
- low-level device
- control
- sensing, recognition
- SLAM(Simultaneous Localization And Mapping)
- navigation
- manipulation and package management
- libraries
- debugging and development tools

Why a Robot Software Platform

- Hardware abstraction is occurring in conjunction with the software platforms, making it possible to develop application programs using a software platform even without having expertise in hardware.

- This is the same with how we can develop mobile apps without knowing the hardware composition or specifications of the latest smartphone.



Why should we use a Robot Software Platform

- **Reusability of the program**
  - Focus only on features of interest
- **Communication-based program**
  - Allows modularization
- **Availability of support of development tools**
  - Debugging, visualization
- **Active community**
  - Sharing and collaborating to improve and speed up the development

ROS – Robot Operating System (v1)

ROS is a meta operating system

- A system that performs processes such as scheduling, loading, monitoring and error handling by utilizing virtualization layer between applications and distributed computing resources
- ROS runs on the existing operating system acting as a **middleware**

Objectives of ROS

- ROS is focused on **maximizing code reuse** in the robotics research and development.

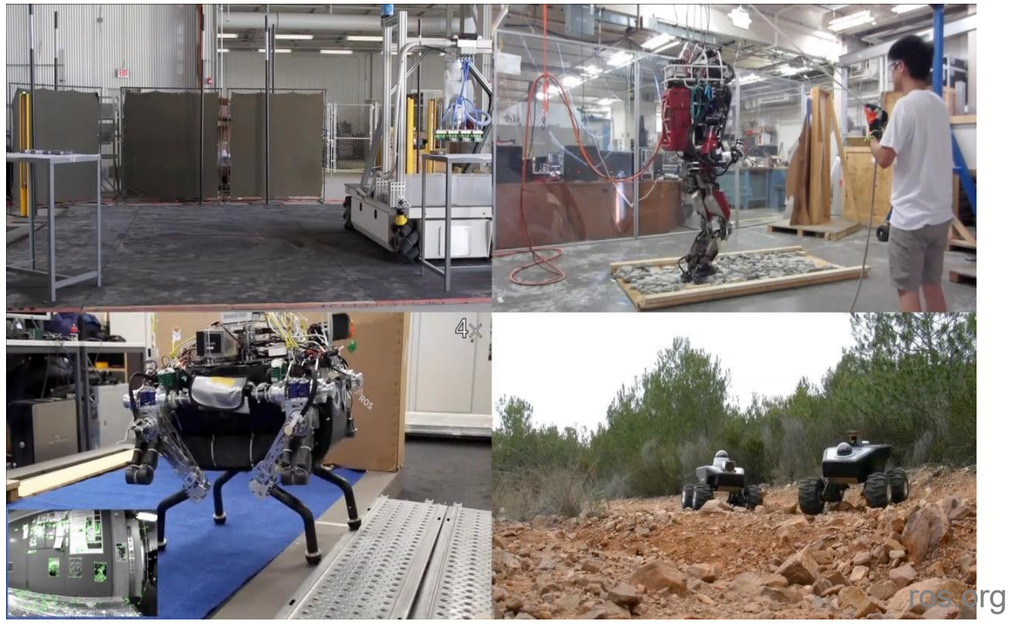
- To support this, ROS has the following characteristics:
  - Distributed process
  - Package management
  - Public repository + API
  - Supporting different programming languages: Python, C++, Java, Matlab

History of ROS

- Originally developed in 2007 at the Stanford Artificial Intelligence Laboratory
- Since 2013 managed by OSRF that became Open Robotics in May 2017
- Today used by many robots, universities and companies
- It can be considered a standard for robot programming

Open Source Robotics Foundation

open robotics



ROS(1) Master

- Acts as a name server for node-to-node connection
  - Manages the communication between nodes (processes)
  - Every node registers at startup with the master
- The communication is impossible without the Master

It uses IP address of local machine and as default port the 11311

Start a master with

```
> roscore
```

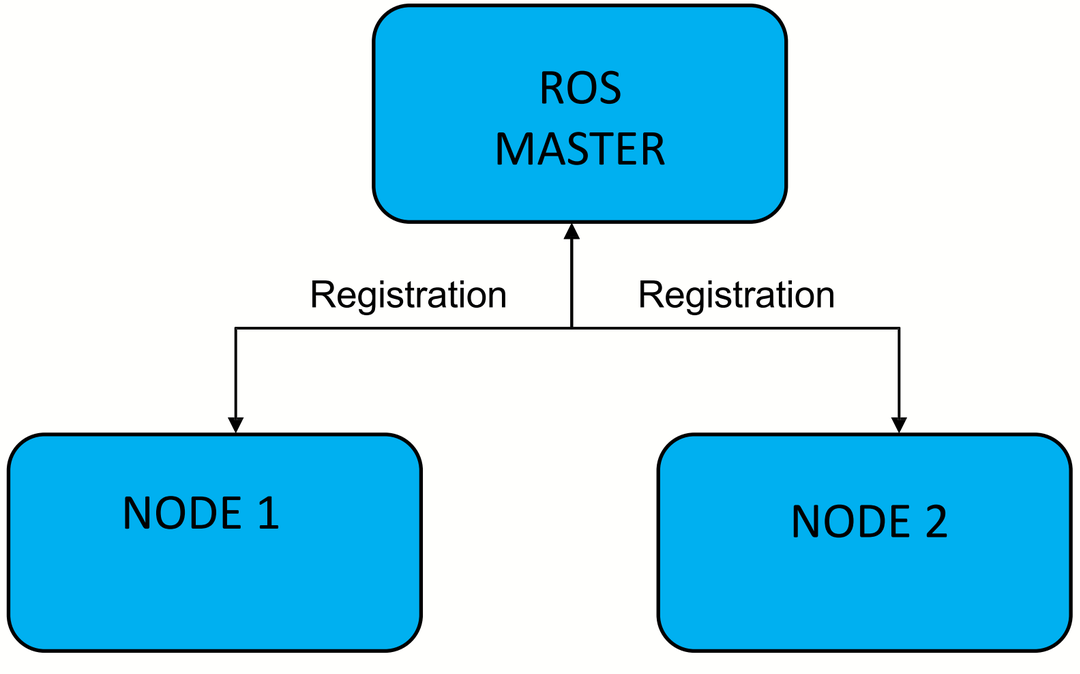
ROS Node

- A node refers to the smallest unit of processor running in ROS.
- ROS recommends creating one single node for each purpose, and it is recommended to develop for easy reusability.

- The node uses XMLRPC for communicating with the master and uses XMLRPC or TCPROS of the TCP/IP protocols when communicating between nodes

ROS Nodes

- Smallest unit of ROS process
- Individually compiled, executed, and managed
- Organized in *packages*
- They use **XMLRPC** for communicating with the Master and **TCP/IP** protocols when communicating between nodes



ROS Nodes

Execution of a node with

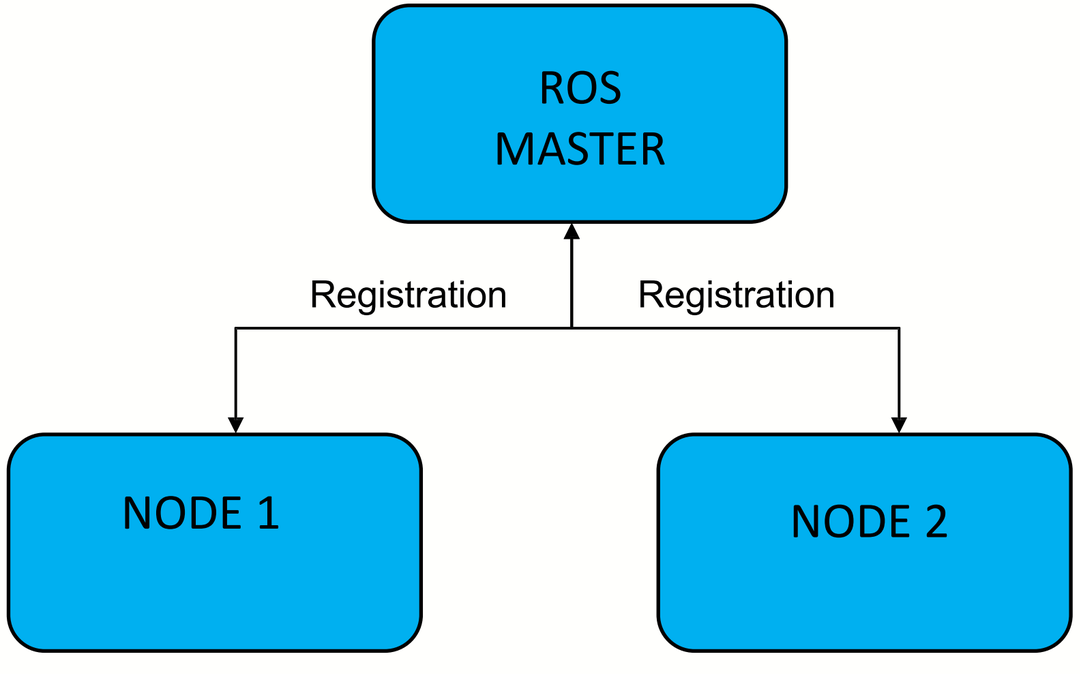
```
> rosrun package_name node_name
```

Check active nodes

```
> rosnode list
```

Get information about a node with

```
> rosnode info node_name
```



Package

- A package is the basic unit of ROS.
- The ROS application is developed on a package basis, and the package contains either a configuration file to launch other packages or nodes.
- The package also contains all the files necessary for running the package, including ROS dependency libraries for running various processes, datasets, and configuration file.

Message

- A node sends or receives data between nodes via a message. Messages are variables such as integer, floating point, and boolean.
- Nested message structure that contains another messages or an array of messages can be used in the message.
- TCPROS and UDPROS communication protocol is used for message delivery.
- **Topic** is used in unidirectional message delivery while service is used in bidirectional message delivery that request and response are involved

Publish and Publisher

The term ‘publish’ stands for the action of transmitting relative messages corresponding to the topic. The publisher node registers its own information and topic with the master, and sends a message to connected subscriber nodes that are interested in the same topic.

The publisher is declared in the node and can be declared multiple times in one node.

Subscribe and Subscriber I

- The term ‘subscribe’ stands for the action of **receiving relative messages** corresponding to the topic.
- The subscriber node registers its own information and topic with the master, and receives publisher information that publishes relative topic from the master.
- Based on received publisher information, the subscriber node directly requests connection to the publisher node and receives messages from the connected publisher node. A subscriber is declared in the node and can be declared multiple times in one node.

Subscribe and Subscriber II

The topic communication is an **asynchronous** communication which is based on publisher and subscriber, and it is useful to transfer certain data. Since the topic continuously transmits and receives stream of messages once connected, it is often used for sensors that must periodically transmit data.

ROS Topics

- Nodes may communicate through *topics*
  - Nodes can *publish* or *subscribe* to a topic
  - Usually for one topic : 1 publisher and *n* subscribers (but publishers can be more than one)
- Topic is a name for a *stream of messages*

List active topics with

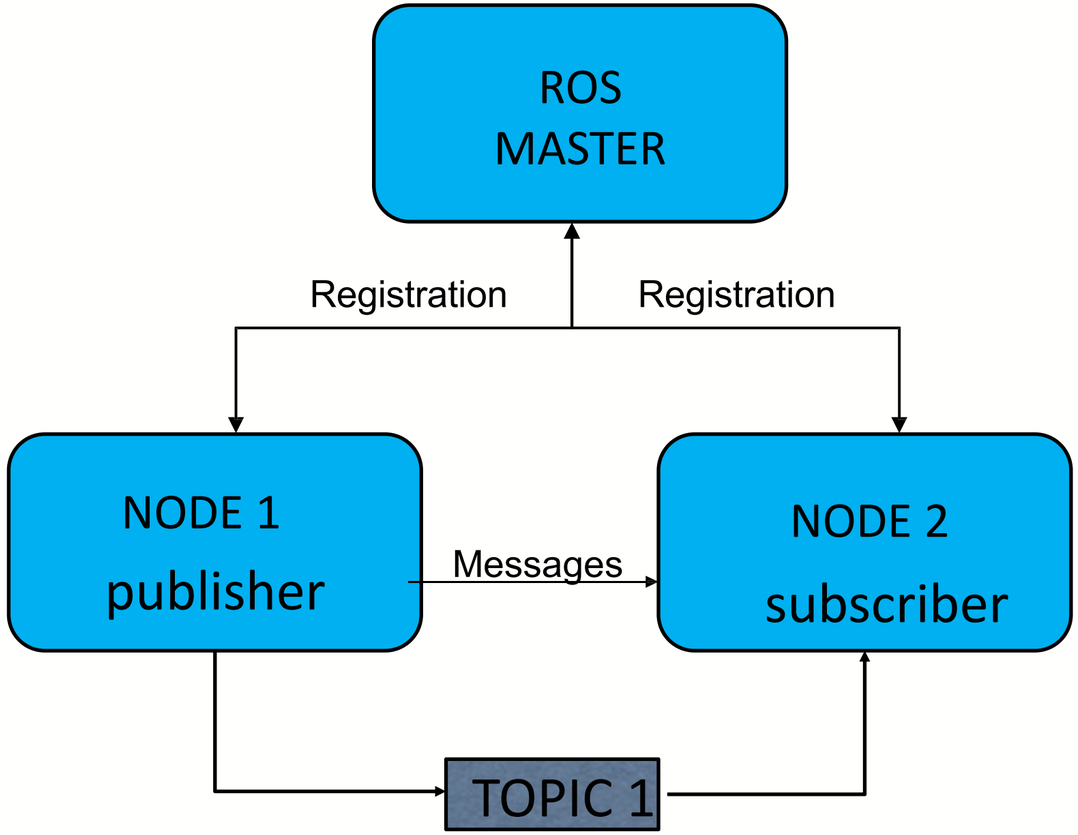
```
> rostopic list
```

Subscribe and print the content of the topic with:

```
> rostopic echo /topic
```

Retrieve information about a topic with

```
> rostopic info /topic
```



ROS Messages

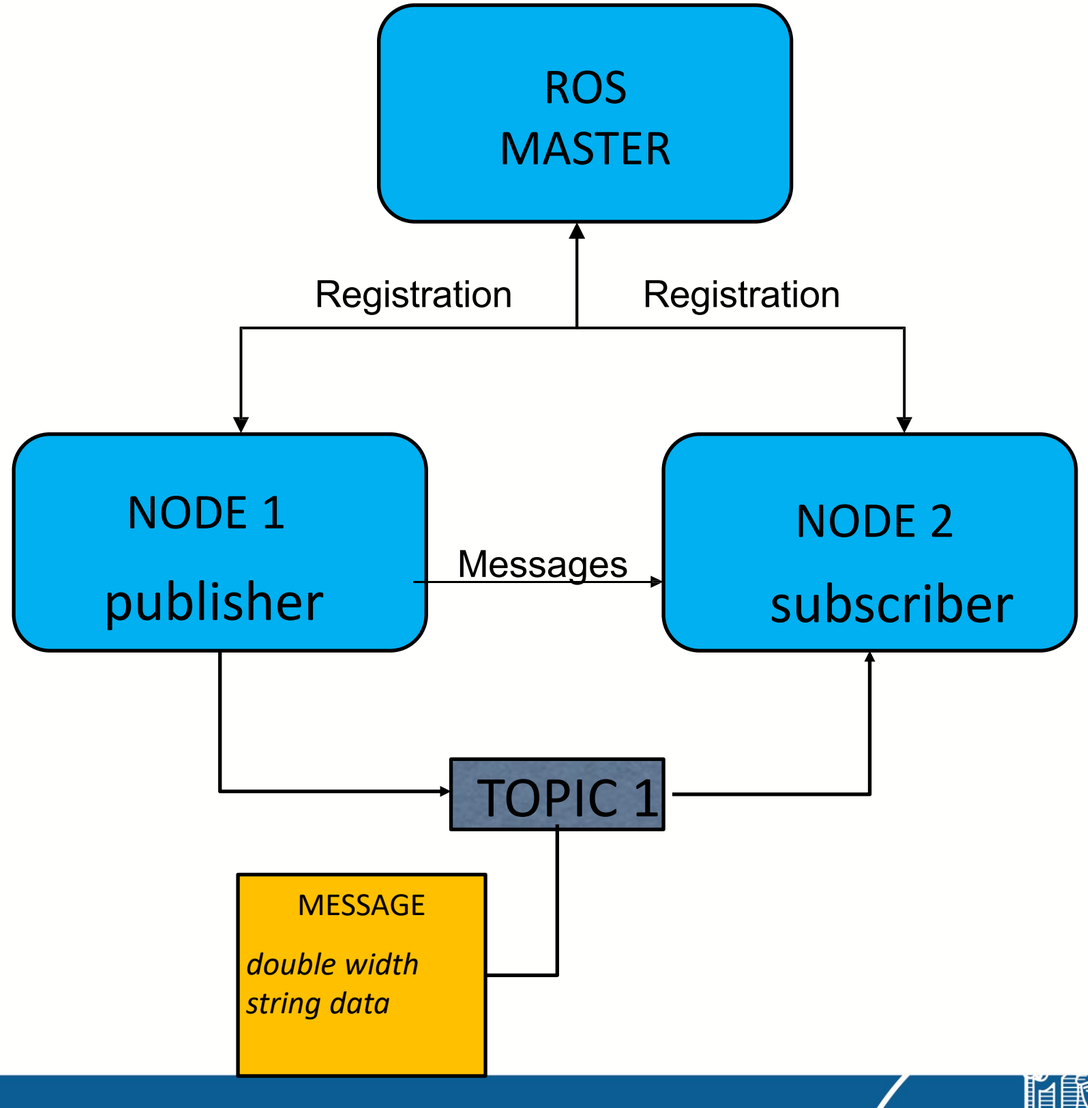
- Data structure defining the *type* of a topic
- Comprised of a nested structure of integers, floats, Booleans, double… and arrays of objects
- Defined in **.msg* files

To see the type of a topic

```
> rostopic type /topic
```

Publish a message to a topic

```
> rostopic pub /topic type data
```



ROS Messages
Pose Stamped Example

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
```

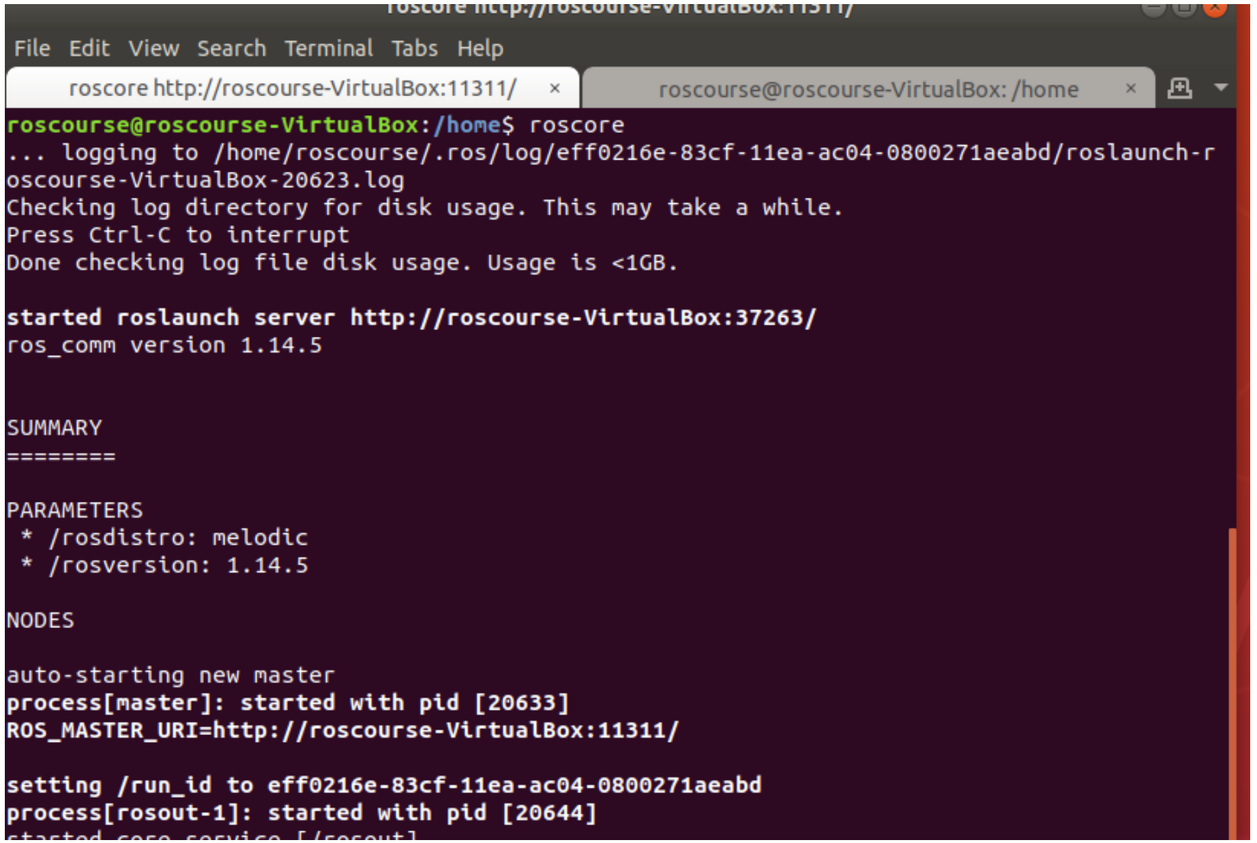
geometry_msgs/PoseStamped.msg

```
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
```

Example - Console 1 – Starting a *roscore*

Start a roscore with

```
> roscore
```

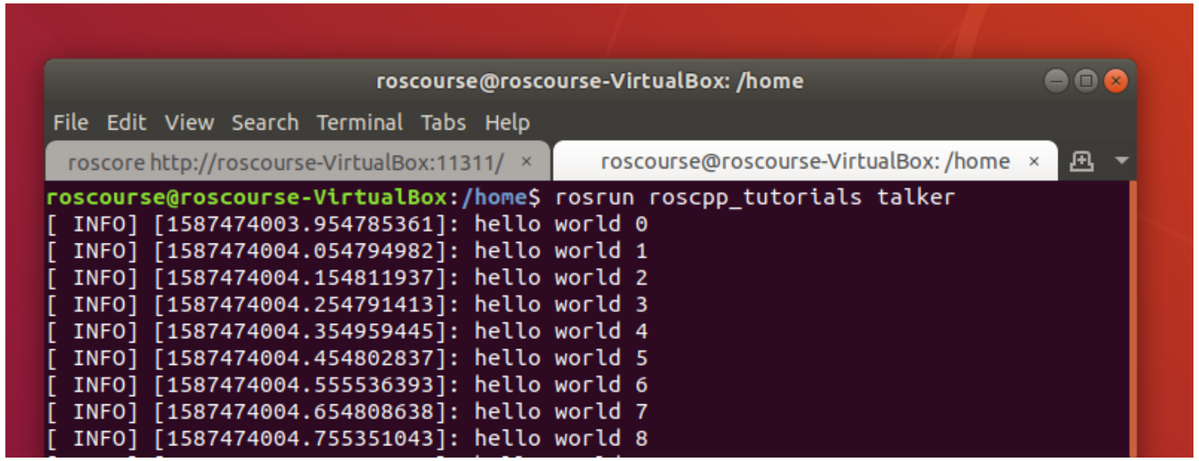


Example - Console 2 – Starting a *talker* node

Run a talker demo node

```
> rosrun roscpp_tutorials talker
```

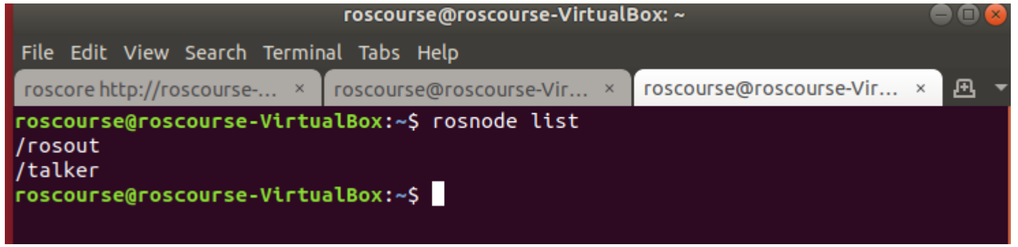
PACKAGE NAME



Example - Console Tab Nr. 3 – T*alker* node

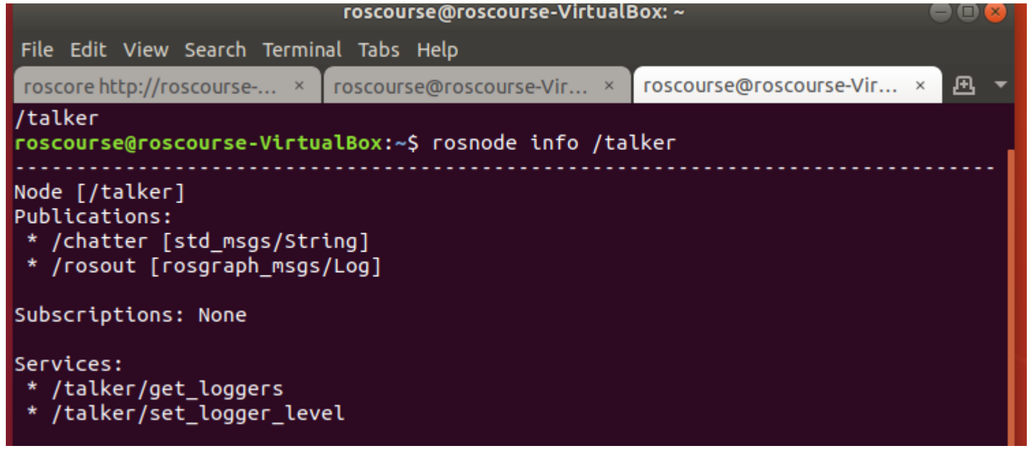
List of active nodes

```
> rosnode list
```



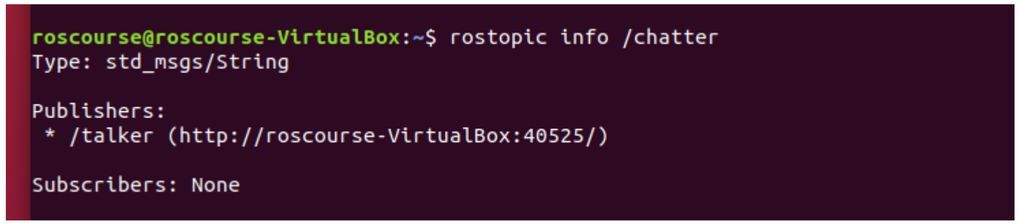
Information about the *talker* node

```
> rosnode info /talker
```



Information about the *chatter* topic

```
> rostopic info /chatter
```



Example - Console Tab Nr. 3 – *Chatter* topic

Check the type of the *chatter* topic

```
> rostopic type /chatter
```

```
roscourse@roscourse-VirtualBox:~$ rostopic type /chatter
std_msgs/String
roscourse@roscourse-VirtualBox:~$
```

Show the message contents of the topic

```
> rostopic echo /chatter
```

```
roscourse@roscourse-VirtualBox:~$ rostopic echo /chatter
data: "hello world 3889"
---
data: "hello world 3890"
---
data: "hello world 3891"
---
data: "hello world 3892"
---
```

Analyze the frequency

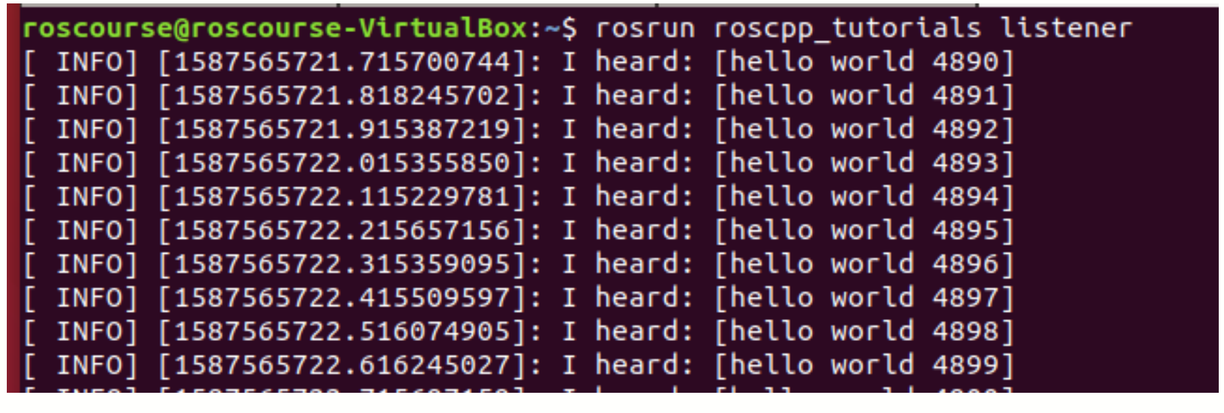
```
> rostopic hz /chatter
```

```
^Croscourse@roscourse-VirtualBox:~$ rostopic hz /chatter
subscribed to [/chatter]
average rate: 9.994
        min: 0.100s max: 0.101s std dev: 0.00046s window: 9
average rate: 9.973
        min: 0.081s max: 0.119s std dev: 0.00645s window: 19
average rate: 9.928
        min: 0.081s max: 0.119s std dev: 0.00646s window: 29
average rate: 10.000
```

Example Console Tab Nr. 4 – *Listener* node

listener demo node

```
> rosrun roscpp_tutorials listener
```



Example - Console Tab Nr. 3 – Analyze nodes and topic

See the new *listener* node with

```
> rosnode list
```

```
roscourse@roscourse-VirtualBox:~$ rosnode list
/listener
/rosout
/talker
```

Show the connection of the nodes over the chatter topic with

```
> rostopic info /chatter
```

```
roscourse@roscourse-VirtualBox:~$ rostopic info /chatter
Type: std_msgs/String

Publishers:
 * /talker (http://roscourse-VirtualBox:40525/)

Subscribers:
 * /listener (http://roscourse-VirtualBox:34775/)
```

ROS Launch

- While rosrun is a command to execute a single node, roslaunch in contrast executes multiple nodes.
- It is a ROS command specialized in node execution with additional functions such as changing package parameters or node names

ROS Launch

- *launch* is a tool for launching multiple nodes (as well as setting parameters)
- Written in XML as **.launch* files
- If not yet running, launch **automatically** starts a roscore

Browse to the folder and start a launch file with

```
> roslaunch file_name.launch
```

Start a launch file from a package with

```
> roslaunch package_name file_name.launch
```

More info

http://wiki.ros.org/roslaunch

Example of console output for
`roslaunch roscpp_tutorials talker_listener.launch`

```
^Croscourse@roscourse-VirtualBox:~$ roslaunch roscpp_tutorials talker_listener.
lnch
... logging to /home/roscourse/.ros/log/6b9fb8fc-84a6-11ea-9286-0800271aeabd/ro
slaunch-roscourse-VirtualBox-5030.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://roscourse-VirtualBox:41459/

SUMMARY
========

PARAMETERS
 * /rosdistro: melodic
 * /rosversion: 1.14.5

NODES
  /
    listener (roscpp_tutorials/listener)
    talker (roscpp_tutorials/talker)

ROS_MASTER_URI=http://localhost:11311

process[listener-1]: started with pid [5045]
process[talker-2]: started with pid [5046]
[ INFO] [1587566098.731803347]: hello world 0
[ INFO] [1587566098.832244355]: hello world 1
[ INFO] [1587566098.932558426]: hello world 2
```

ROS Launch File Structure

talker_listener.launch

```
<launch>
    <node name="listener" pkg="roscpp_tutorials" type="listener" output="screen"/>
      <node name="talker" pkg="roscpp_tutorials" type="talker" output="screen"/>
</launch>
```

- **launch**: Root element of the launch file
- **node**: Each *<node>* tag is a node to be launched
  - **name**: Node name
  - **pkg**: Package containing the node
  - **type**: Type of the node, there must be a corresponding executable (with the same name)
  - **output**: Specifies where to output log messages (`screen`: console, `log`: log file)

More info

http://wiki.ros.org/roslaunch/XML
http://wiki.ros.org/roslaunch/Tutorials/Roslaunch%20tips%20for%20larger%20projects

ROS Launch Arguments

- Create re-usable launch files with `<arg>` tag, which works like a parameter (default optional)

```
<arg name="arg_name" default="default_value"/>
```

- Use arguments in launch file with

```
$(arg arg_name)
```

  - Arguments can be set with

```
> roslaunch launch_file.launch arg_name:=value
```

range_world.launch (simplified)

```
<?xml version="1.0"?>
<launch>
  <arg name="use_sim_time" default="true"/>
  <arg name="world" default="gazebo_ros_range"/>
  <arg name="debug" default="false"/>
  <arg name="physics" default="ode"/>

  <group if="$(arg use_sim_time)">
    <param name="/use_sim_time" value="true" />
  </group>

  <include file="$(find gazebo_ros)
                 /launch/empty_world.launch">
    <arg name="world_name" value="$(find gazebo_plugins)/
                 test/test_worlds/$(arg world).world"/>
    <arg name="debug" value="$(arg debug)"/>
    <arg name="physics" value="$(arg physics)"/>
  </include>
</launch>
```

More info

http://wiki.ros.org/roslaunch/XML/arg

Service

On the other hands, there is a need for synchronous communication with which request and response are used. Therefore, ROS provides a message synchronization method called '**service**'.

The service is **synchronous** bidirectional communication between the service client that requests a service regarding a particular task and the service server that is responsible for responding to requests.

Service Server

The ‘service server’ is a server in the service message communication that receives a request as an input and sends a response as an output.

Both request and response are in the form of messages. Upon the service request, the server performs the designated service and delivers the result to the service client as a response. The service server is implemented in the **node that receives and executes a given request**.

Service Client

The ‘service client’ is a client in the service message communication that **requests service** to the server and receives a response as an input.

Both request and response are in the form of message.

The client sends a request to the service server and receives the response. The service client is implemented in the node which requests specified command and receives results

ROS Services - Example

std_srvs/Trigger.srv

```
---
bool success
string message
```

Request

Response

nav_msgs/GetPlan.srv

```
geometry_msgs/PoseStamped start
geometry_msgs/PoseStamped goal
float32 tolerance
---
nav_msgs/Path plan
```

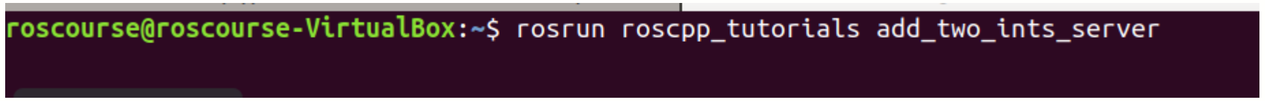
ROS Service Example
Starting a *roscore* and *add_two_ints_server* node

In console nr. 1:
Run roscore with

```
> roscore
```

In console nr. 2:
Run a service demo node with

```
> rosrun roscpp_tutorials add_two_ints_server
```



ROS Service Example - Analyze and call service

Available services with

```
> rosservice list
```

```
roscourse@roscourse-VirtualBox:~$ rosservice list
/add_two_ints
/add_two_ints_server/get_loggers
/add_two_ints_server/set_logger_level
/rosout/get_loggers
/rosout/set_logger_level
roscourse@roscourse-VirtualBox:~$
```

See the type of the service

```
> rosservice type /add_two_ints
```

```
roscourse@roscourse-VirtualBox:~$ rosservice type /add_two_ints
roscpp_tutorials/TwoInts
```

Show the service definition with

```
> rossrv show roscpp_tutorials/TwoInts
```

```
roscourse@roscourse-VirtualBox:~$ rossrv show roscpp_tutorials/TwoInts
int64 a
int64 b
---
int64 sum
```

Call the service

a *b*

```
> rosservice call /add_two_ints 100 200
```

```
roscourse@roscourse-VirtualBox:~$ rosservice call /add_two_ints 100 200
sum: 300
```

Action I

The action is another message communication method used for an **asynchronous bidirectional communication**.

Action is used where it takes longer time to respond after receiving a request and intermediate responses are required until the result is returned.

However, feedback data section for **intermediate response** is added along with goal and result data section which are represented as request and response in service respectively.

Action II

There are action client that sets the goal of the action and action server that performs the action specified by the goal and returns feedback and result to the action client.

Action Server

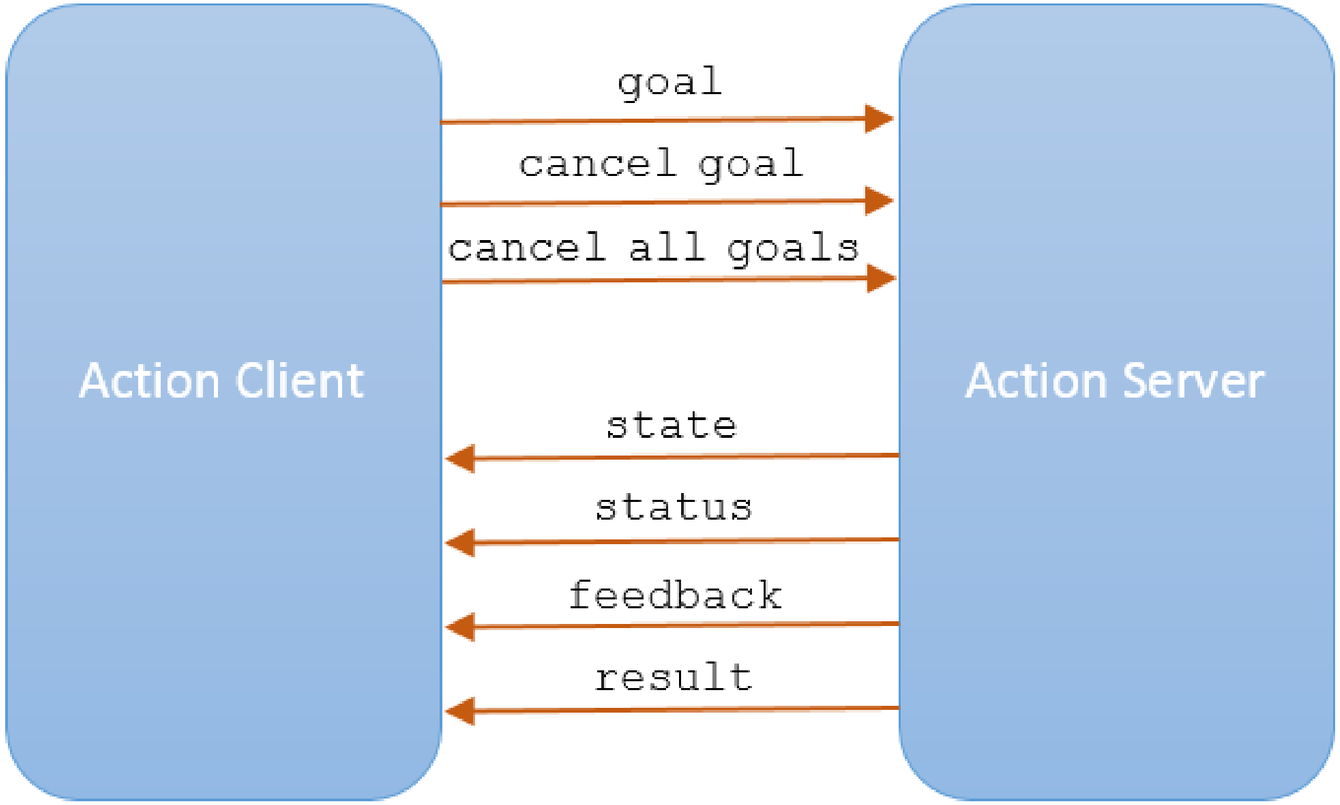
The ‘*action server*’ is in charge of accepting goal from the client and responding with feedback and result.
Once the server receives goal from the client, it performs predefined process.

Action Client

The ‘action client’ is in charge of transmitting the goal to the server and receives result or feedback data as inputs from the action server. The client delivers the goal to the action server, then receives corresponding result or feedback, and transmits follow up instructions or cancel instruction

ROS Actions (actionlib)

- They provide the possibility to:
  - Cancel the task (preempt)
  - Receive feedback on intermediate results
- It is a way to implement interfaces to time-extended tasks
- Similar structure of services, action are defined in **.action* files
- Internally, actions are implemented with topics



**.action* Action definition

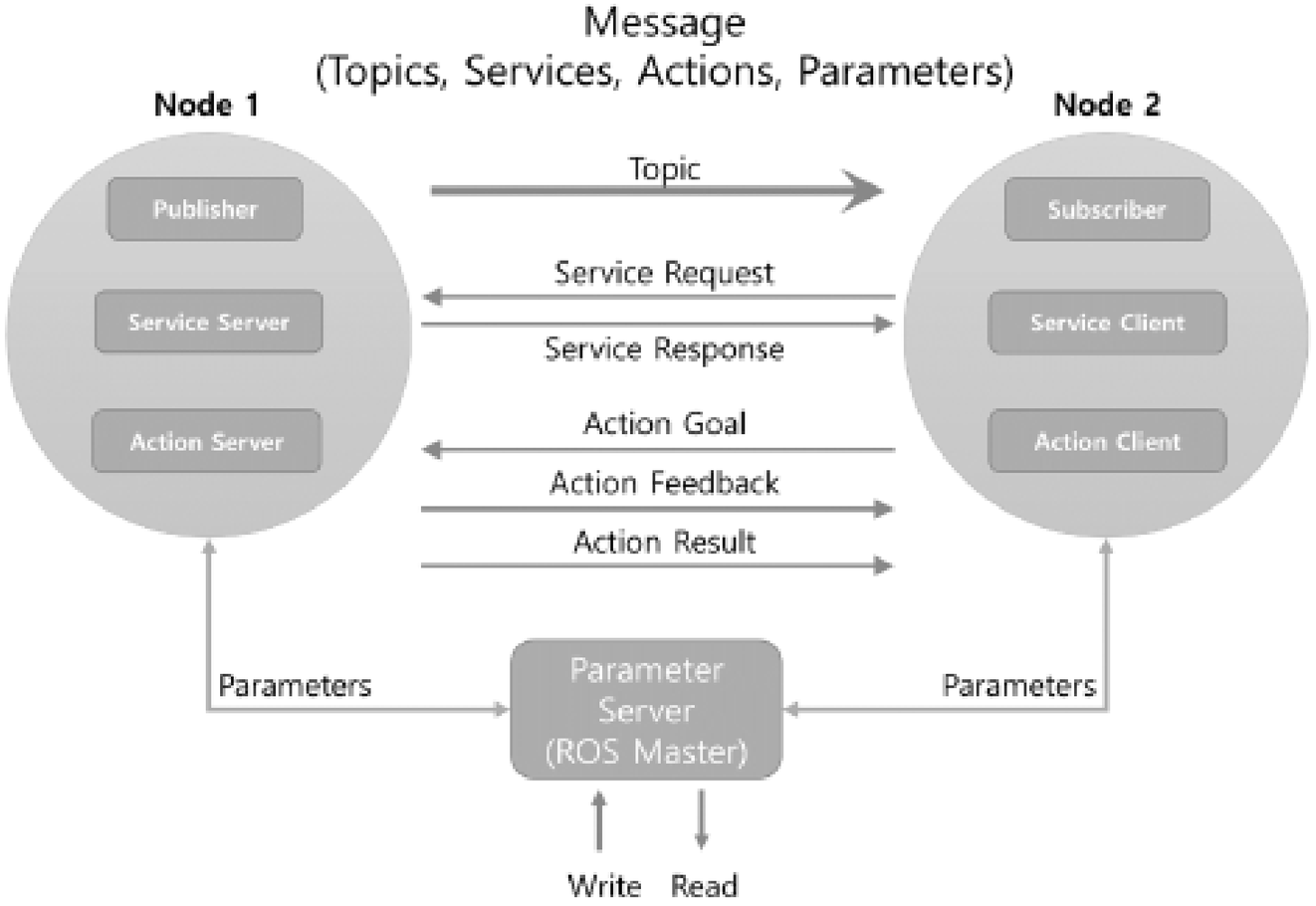
```
Goal
---
Intermediate Result
---
Feedback
```

More info

http://wiki.ros.org/actionlib

http://wiki.ros.org/actionlib/DetailedDescription

Message Communication



Message Communication

Type	Features		Description
Topic	Asynchronous	Unidirectional	Used when exchanging data continuously
Service	Synchronous	Bi-directional	Used when request processing requests and responds current states
Action	Asynchronous	Bi-directional	Used when it is difficult to use the service due to long response times after the request or when an intermediate feedback value is needed

Bag I

The data from the ROS messages can be recorded. The file format used is called bag, and ‘*.bag’ is used as the file extension.

In ROS, bag can be used to record messages and play them back when necessary to reproduce the environment when messages are recorded. For example, when performing a robot experiment using a sensor, sensor values are stored in the message form using the bag.

Bag II

This recorded message can be repeatedly loaded without performing the same test by playing the saved bag file. Record and play functions of rosbag are especially useful when developing an algorithm with frequent program modifications.

ROS Bags

- A *bag* is usedstoring message data
- Binary format with file extension *.bag
- Suited for logging and recording datasets for visualization and analysis

Record all topics in the bag

```
> rosbag record --all
```

Record from given topics

```
> rosbag record topic_1 topic_2 topic_3
```

Stop bag recording with `Ctrl + C`
Bags are saved with start date and time as file name in the current folder (e.g. `2020-04-23-10-27-13.bag`)

Information about a bag

```
> rosbag info bag_name.bag
```

Playback options can be defined as:

```
> rosbag play --rate=0.5 bag_name.bag
```

↑
You can define the rate

https://www.pishrobot.com/wp-content/uploads/2018/02/ROS-robot-programming-book-by-turtlebo3-developers-EN.pdf