Introduction to ROS

Robot Programming and Control

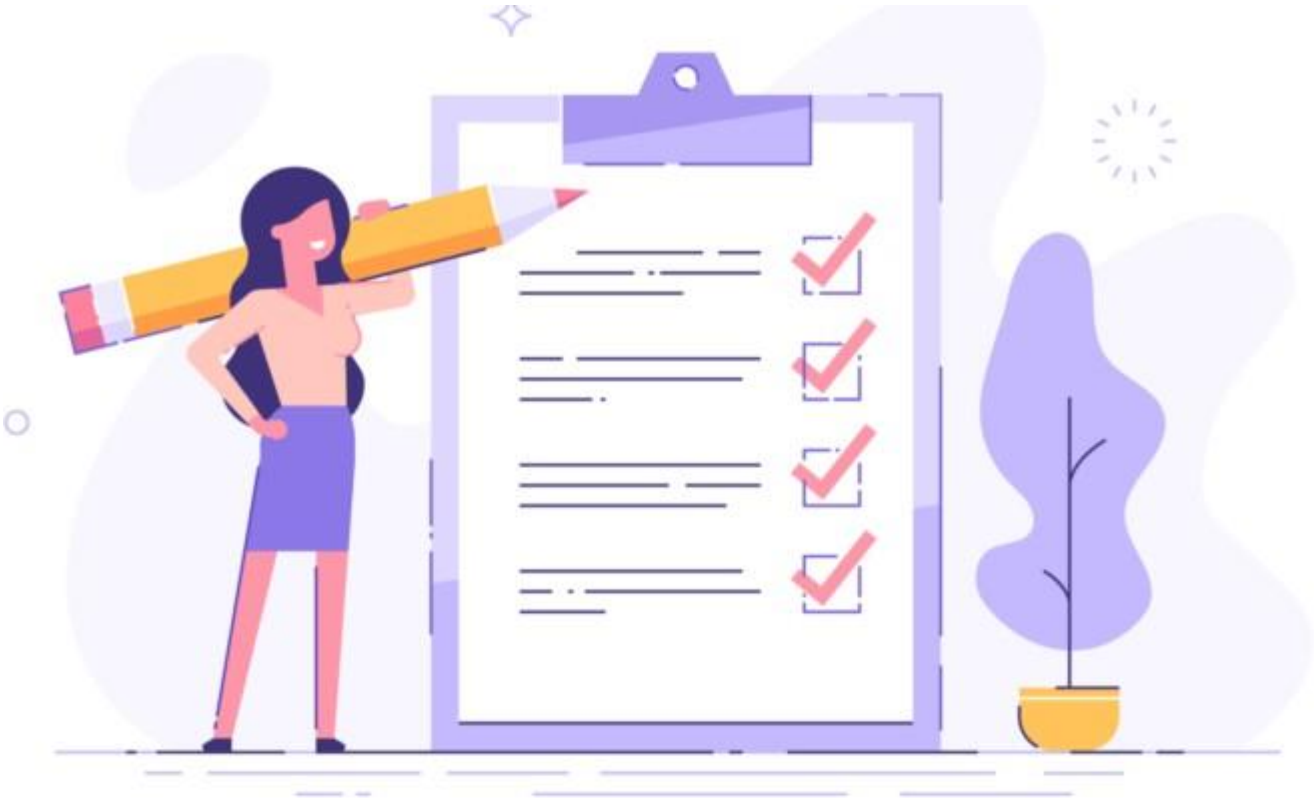
Accademic Year 2021-2022

Diego Dall’Alba diego.dallalba@univr.it

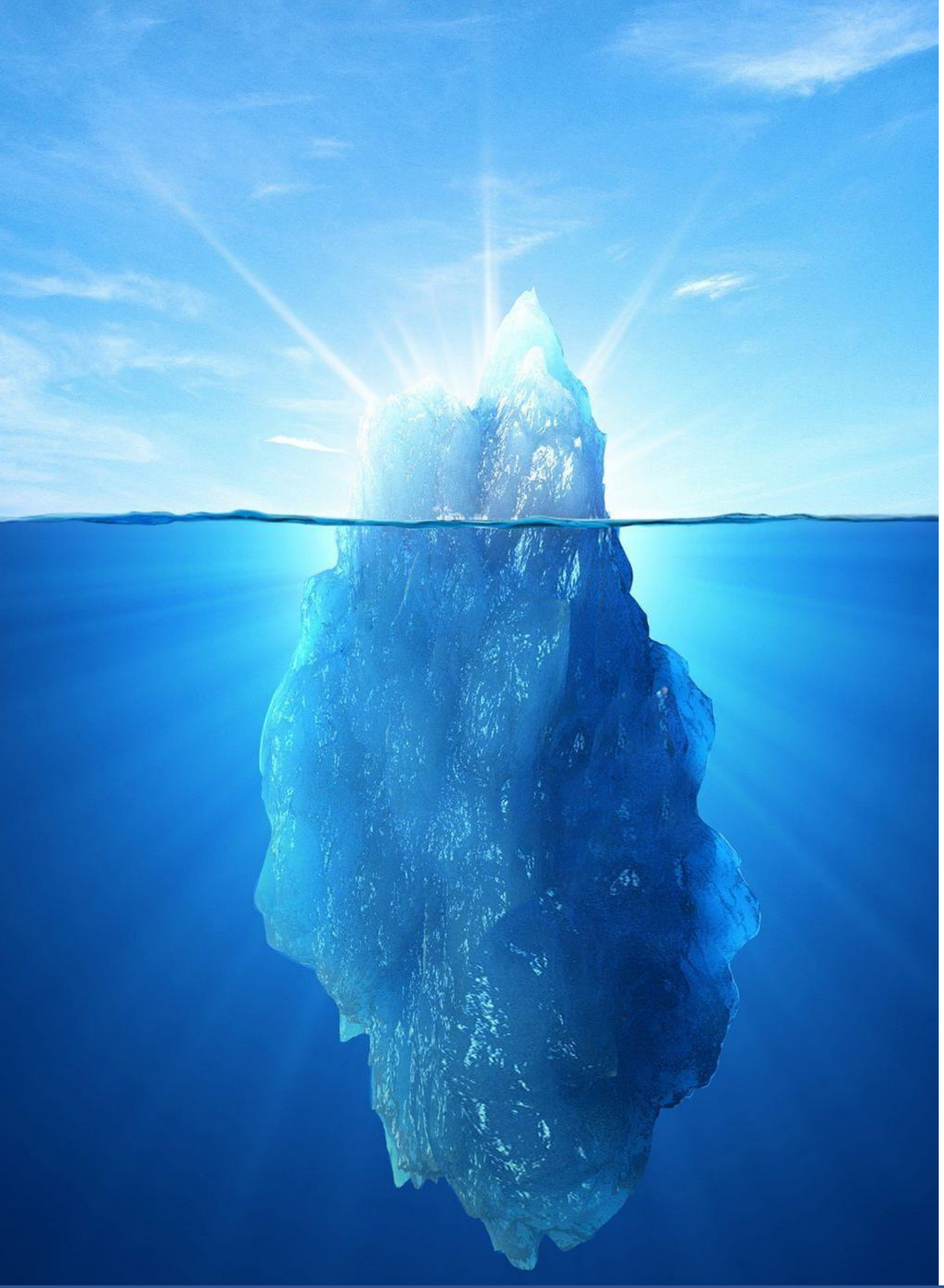
Department of Computer Science – University of Verona

Altair Robotics Lab

Please fill the following survey (15 minutes)



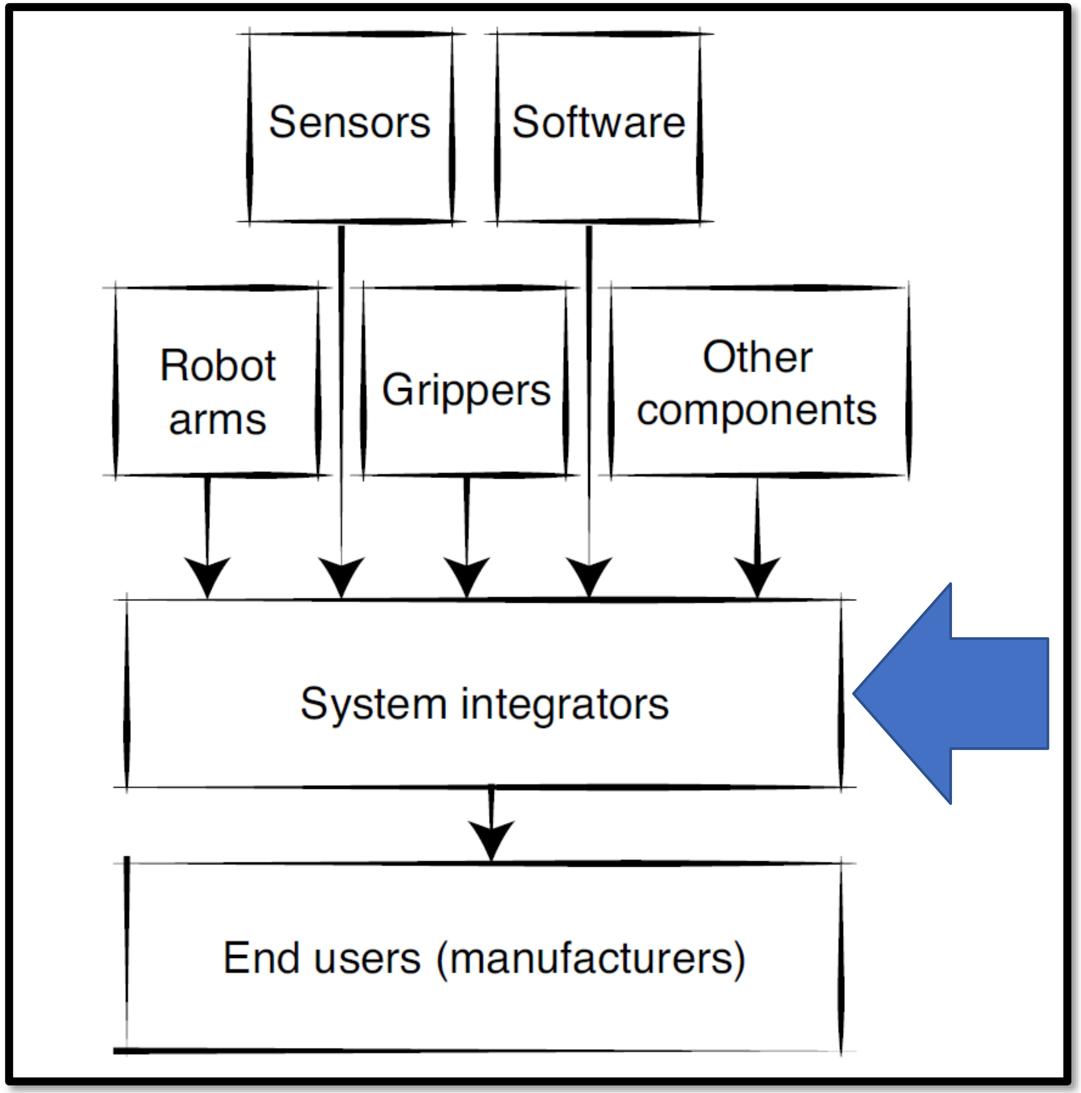
https://forms.gle/cz5XcMBhJ2MUPa3a7



Overview

- ROS architecture & philosophy
- ROS master, nodes, and topics
- Catkin workspace and build system
- ROS package structure
- Console commands
- Launch-files
- ROS C++ client library (roscpp)
- ROS subscribers and publishers
- ROS parameter server
- ROS services
- ROS actions (actionlib)

Small recap from last lesson



Sensors

Software

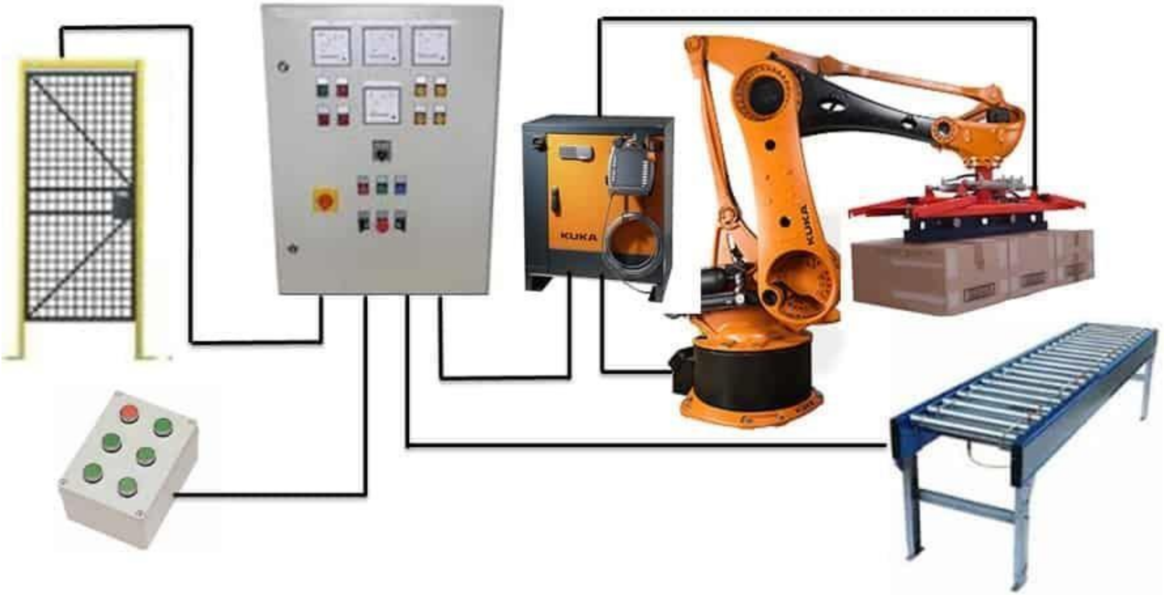
Robot arms

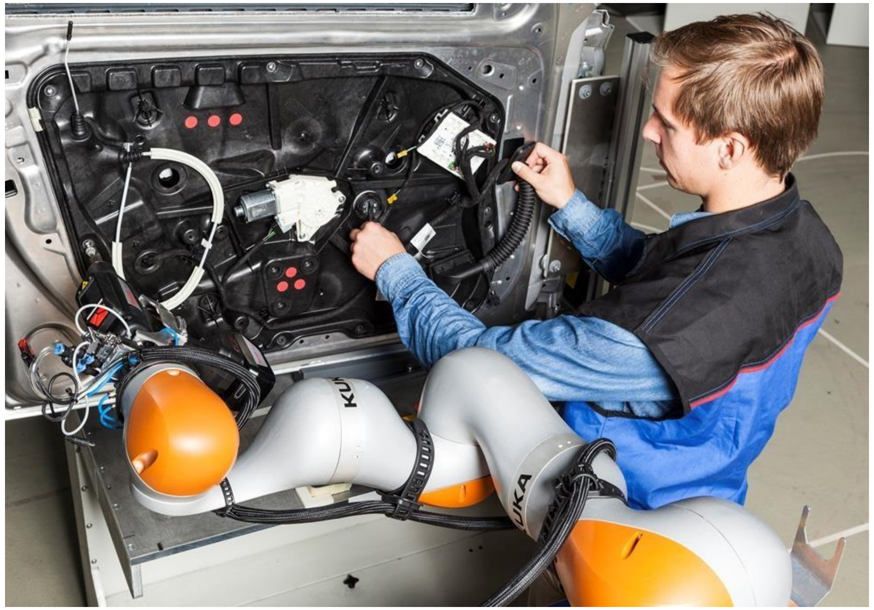
Grippers

Other components

System integrators

End users (manufacturers)

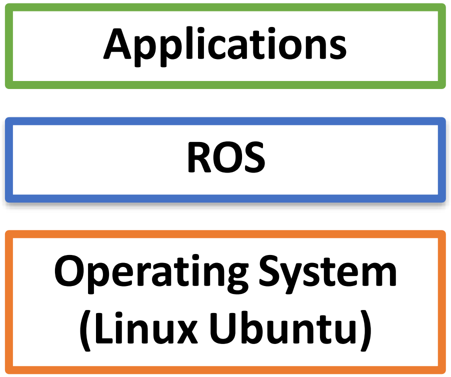




What is ROS (Robotic Operating System)? - VIDEO

What is ROS (Robotic Operating System)?

- It is not a Operating System (OS)
- It is not an Application Programming Interface (API)
- It is not a «simple» framework



Applications

ROS

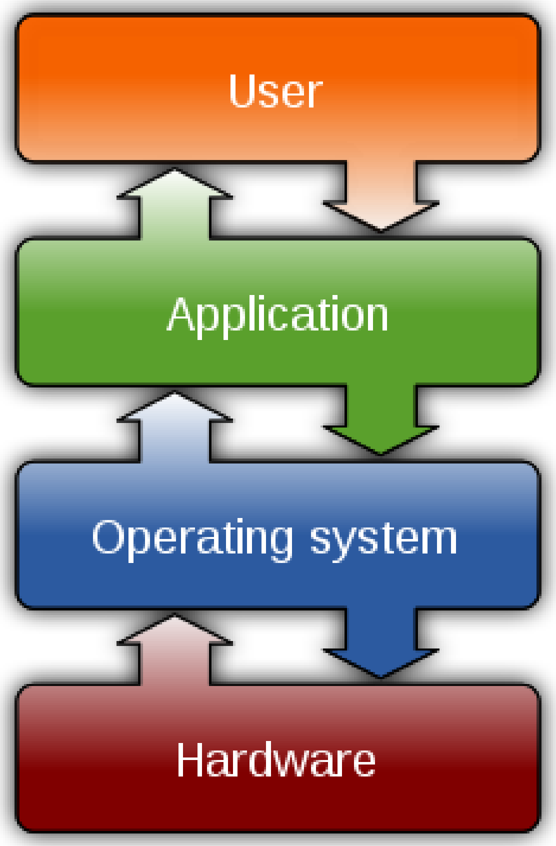
Operating System (Linux Ubuntu)

ROS is a middleware for robotic programming, specifically designed for complex applications

BTW, What are OS, API, Framework and Middleware? Which are the differences?

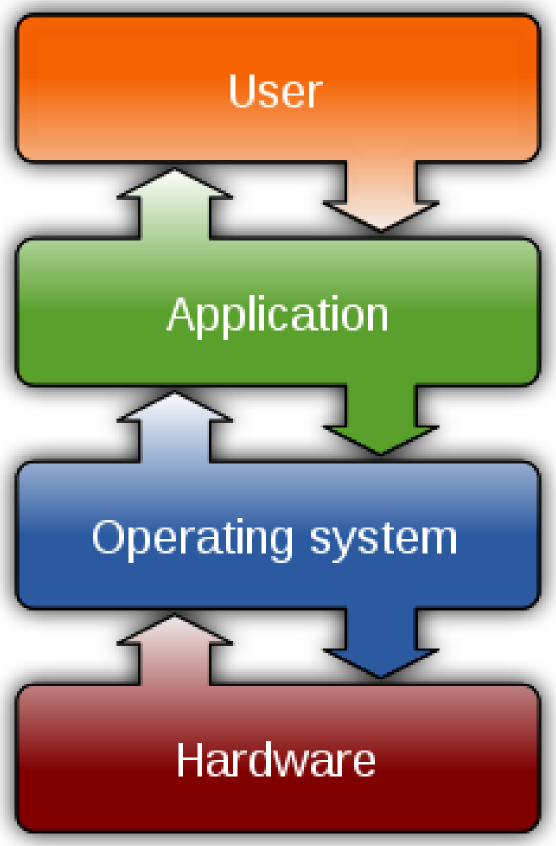
What are OS, API, Framework and Middleware?

- An application programming interface (API) is an interface (e.g. set of functions and methods, data types)intended to simplify the implementation and maintenance of software.
- An operating system (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs.



What are OS, API, Framework and Middleware?

- Framework provide an infrastructure and a methodology for quickly developing and distributing complex software applications. Do not try to do things not supported by the framework!
- Middleware is a set of software tools (including APIs and Frameworks) that provides services to applications to enable easy communication and integration of different modules/functionalities. It can be described as "software glue".



Why a middleware for robotic programming?

- Simplify development process
- Provide simple and transparent inter-processes communication
- Provide software functionalities that are frequently needed in robotic applications
- Abstract high complexity and heterogeneity of different hardware and software components
- Provide an automatic and efficient process for configuring and managing different resources and components
- Supporting embedded system and “low-resources devices”

Quick background about robotic middleware

Many robotic middleware have been proposed, for example:

- **Player/Stage**: based on client-server architecture
- **Miro - Middleware for Robots**: distributed inter-process communication(based on CORBA)
- **OROCOS**: designed for real-time applications
- **URBI:** focusing on component architecture and management
- **YARP**: Yet another robotic platform ☺

You could find a PARTIAL list of robotic middleware at:

https://en.wikipedia.org/wiki/Robotics_middleware

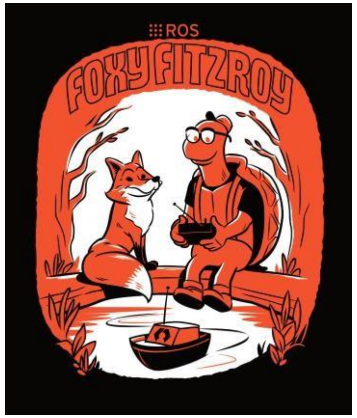
NOTE: The European Union has fundend at least 2 big research projects (RoSta 1M and BRICS 10M). In the USA also DARPA invested a huge amount of resources in the development robotic middleware

Quick background about ROS

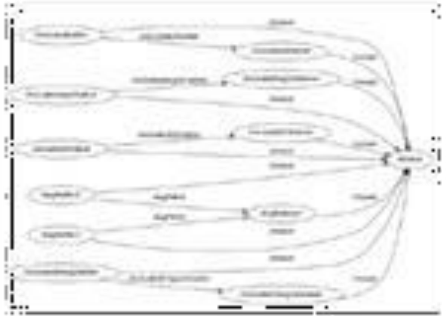
- Complete timeline/History: http://www.ros.org/history

- Originally developed, around 2007, from Stanford University, Artificial Intelligence Lab
- Then developed with the collaboration of other research groups, in particular Willow Garage
- Since 2013 developed and maintained by Open Source Robotic Foundation (OSRF)
- **It is de-facto standard for high level robotic programming in research environment**

- Recently the development of ROS2 has started but it is progressing fast. There is also a consortium called ROS Industrial focused in transferring ROS modules in industrial applications



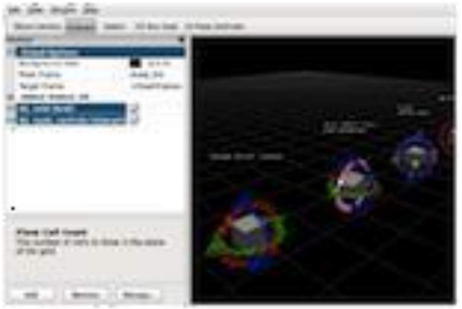
ROS Characteristics



+

Plumbing

- Process management
- Inter-process communication
- Device drivers



+

Tools

- Simulation
- Visualization
- Graphical user interface
- Data logging

+

Capabilities

- Control
- Planning
- Perception
- Mapping
- Manipulation

Ecosystem

- Package organization
- Software distribution
- Documentation
- Tutorials

ROS Philosophy

- **Peer to peer :** Individual programs communicate over defined API (ROS *messages*, *services*, etc.).
- **Distributed:** Programs can be run on multiple computers and communicate over the network.
- **Multi-language support:** ROS modules can be written in any programming language for which a client library exists (C++, Python, MATLAB, Java, etc.).
- **Light-weight:** Stand-alone libraries are wrapped around with a thin ROS layer.
- **Free and open-source:** Most ROS software is open-source and free to use.

ROS Distributions

- A ROS distribution is a versioned set of ROS packages.
- These are similar to Linux distributions (e.g. Ubuntu).
- The purpose of the ROS distributions is to let developers work against a relatively stable codebase

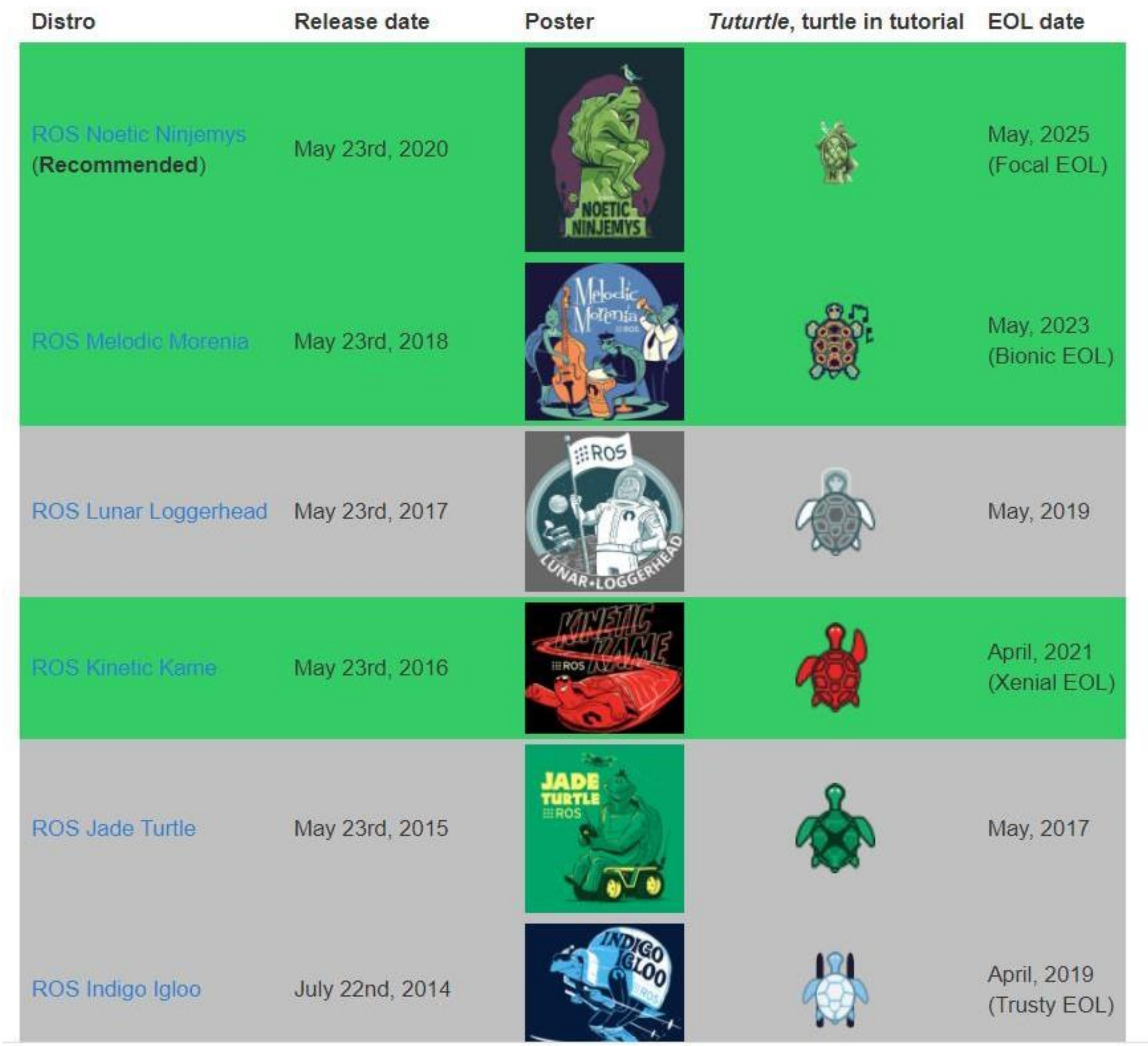
Release rules

- ROS release timing is based on need and available resources
- All future ROS 1 releases are LTS, supported for five years
- ROS releases will drop support for EOL Ubuntu distributions, even if the ROS release is still supported.

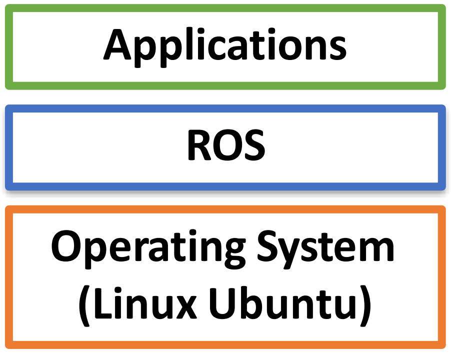
ROS Melodic Morenia
Released May, 2018
Latest LTS, supported until May, 2023

ROS Noetic Ninjemys
Released May, 2020
Latest LTS, supported until May, 2025
Recommended for Ubuntu 20.04

Partial List of ROS and Ubuntu Distributions



Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Noetic Ninjemys **(Recommended)**	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)

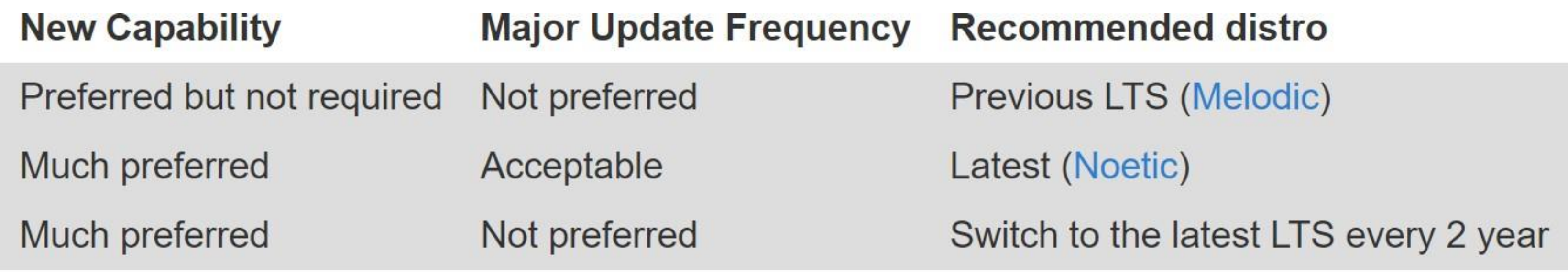


Applications

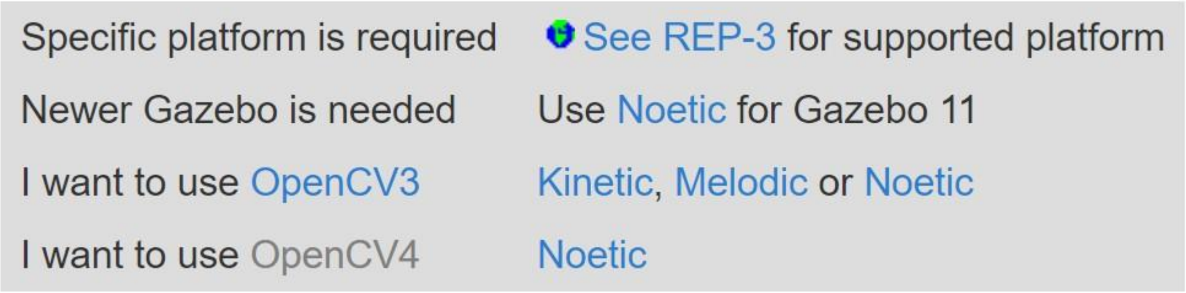
ROS

Operating System (Linux Ubuntu)

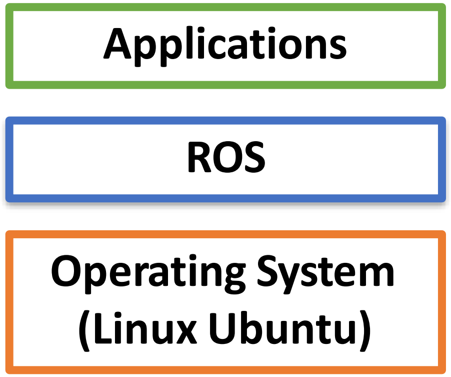
Choosing the right ROS distribution



New Capability	Major Update Frequency	Recommended distro
Preferred but not required	Not preferred	Previous LTS (Melodic)
Much preferred	Acceptable	Latest (Noetic)
Much preferred	Not preferred	Switch to the latest LTS every 2 year



Specific platform is required	See REP-3 for supported platform
Newer Gazebo is needed	Use Noetic for Gazebo 11
I want to use OpenCV3	Kinetic, Melodic or Noetic
I want to use OpenCV4	Noetic

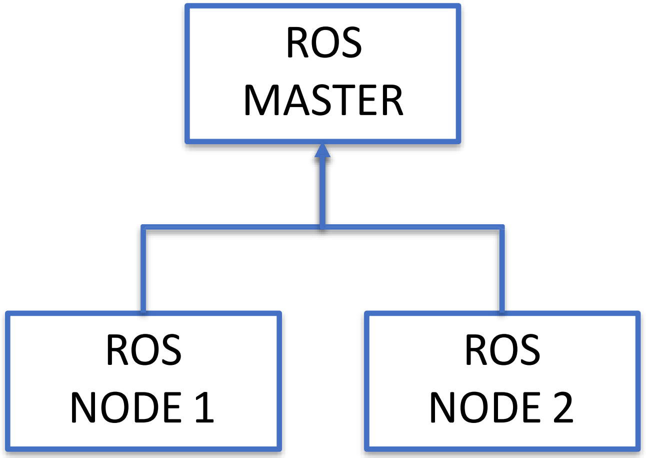


- Changing ROS Distribution is usually quite complex, it depends on the specific application and development cycle
- Try to keep the same distribution in the same project
- Separate different distribution in different machine
- We will use TBD

ROS Architecture: Basics

ROS MASTER

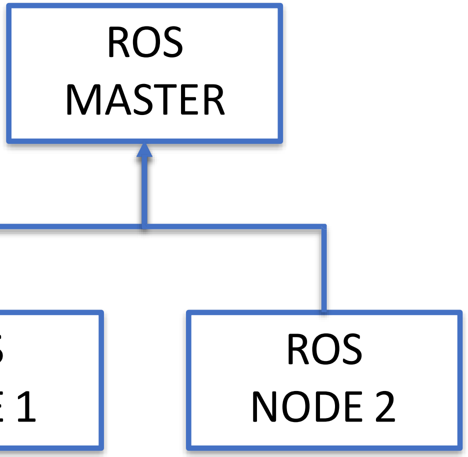
- Manages the communication between nodes (XML-RPC server + naming and communication services)
- Every node registers at start-up with the master
- Nodes can run on different workstation and communicate through network (transparent to user)



ROS NODE

- Single-purpose, executable program
- Individually compiled, executed, and managed
- Organized in *packages*

Configuring the ROS environment



ROS MASTER

I am assuming that you have installed ROS following the official guide available at:

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

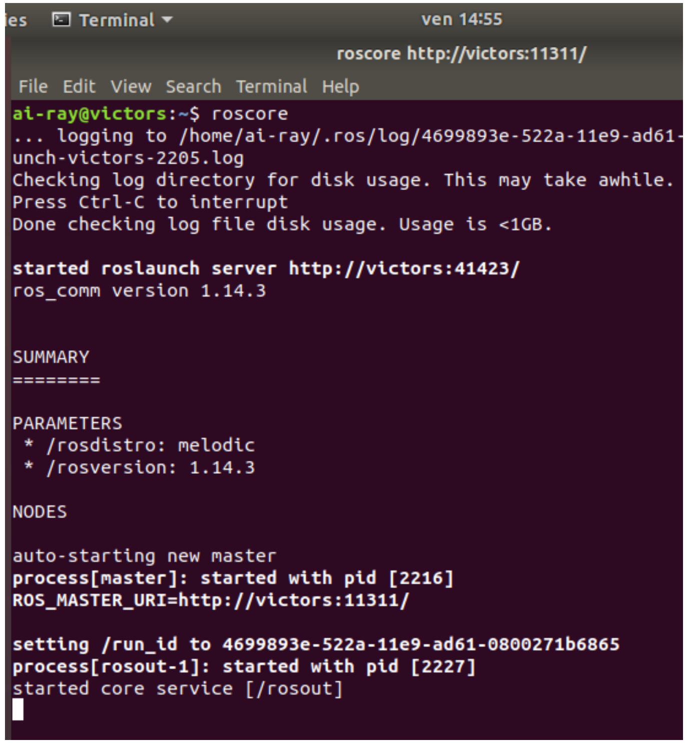
The first step is always configuring the Linux environment:

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

Then you will be able to run

```
roscore
```

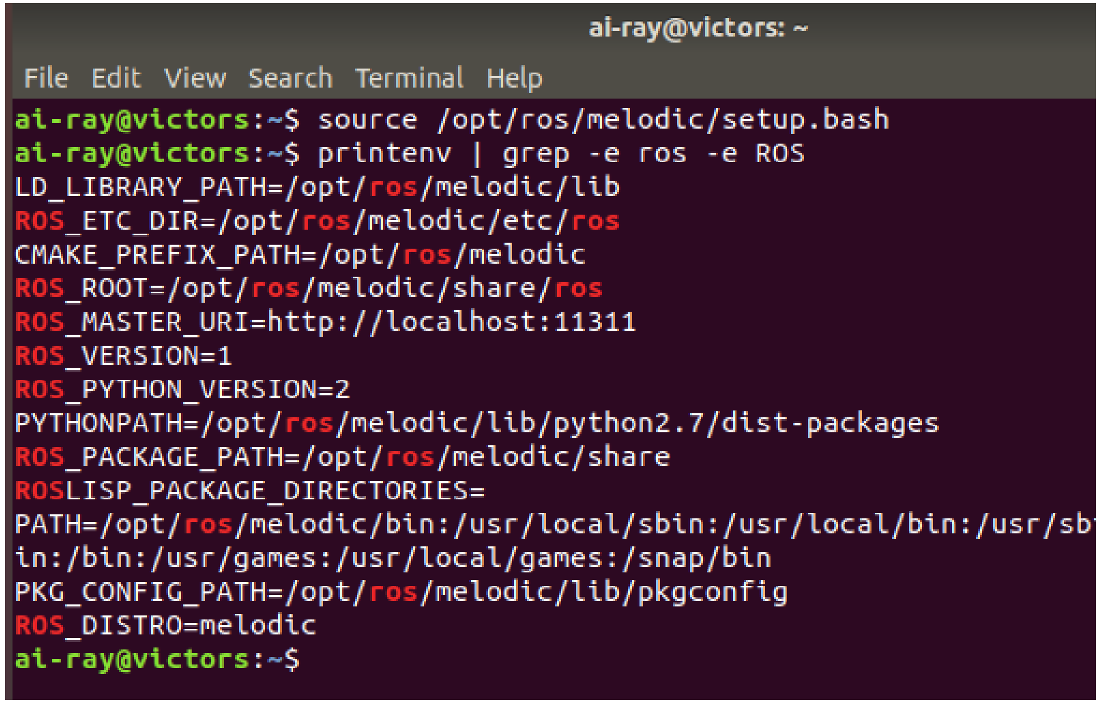
It will run ROS master + other important services (logging and parameters server)


ies
Terminal
ven 14:55
roscore http://victors:11311/
File Edit View Search Terminal Help

```
ai-ray@victors:~$ roscore
... logging to /home/ai-ray/.ros/log/4699893e-522a-11e9-ad61-
unch-victors-2205.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://victors:41423/
ros_comm version 1.14.3


SUMMARY
========

PARAMETERS
 * /rosdistro: melodic
 * /rosversion: 1.14.3

NODES

auto-starting new master
process[master]: started with pid [2216]
ROS_MASTER_URI=http://victors:11311/

setting /run_id to 4699893e-522a-11e9-ad61-0800271b6865
process[rosout-1]: started with pid [2227]
started core service [/rosout]
```

Configuring the ROS environment

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



```
ai-ray@victors:~$ source /opt/ros/melodic/setup.bash
ai-ray@victors:~$ printenv | grep -e ros -e ROS
LD_LIBRARY_PATH=/opt/ros/melodic/lib
ROS_ETC_DIR=/opt/ros/melodic/etc/ros
CMAKE_PREFIX_PATH=/opt/ros/melodic
ROS_ROOT=/opt/ros/melodic/share/ros
ROS_MASTER_URI=http://localhost:11311
ROS_VERSION=1
ROS_PYTHON_VERSION=2
PYTHONPATH=/opt/ros/melodic/lib/python2.7/dist-packages
ROS_PACKAGE_PATH=/opt/ros/melodic/share
ROSLISP_PACKAGE_DIRECTORIES=
PATH=/opt/ros/melodic/bin:/usr/local/sbin:/usr/local/bin:/usr/sb
in:/bin:/usr/games:/usr/local/games:/snap/bin
PKG_CONFIG_PATH=/opt/ros/melodic/lib/pkgconfig
ROS_DISTRO=melodic
ai-ray@victors:~$
```

This command is fundamental for correctly configuring all environment variables required for:

- Finding packages
- Effecting a Node runtime
- Modifying the build system

Essential variables are:

- ROS_ROOT sets the location where the ROS core packages are installed.
- ROS_MASTER_URI is a required setting that tells nodes where they can locate the master.
- ROS requires that your PYTHONPATH be updated, even if you don't program in Python! Many ROS infrastructure tools rely on Python

ROS Build System (1)

CATKIN BUILD SYSTEM = CMake + python

catkin is the official build system of ROS starting from ROS Groovy and the successor to the original ROS build system, rosbuild.

catkin combines CMake macros and Python scripts to provide some functionality on top of CMake's normal workflow (improved automatic dependencies management and compilation of large project)

The name catkin comes from the tail-shaped flower cluster found on willow trees -- a reference to Willow Garage where catkin was created.

It is essential to know catkin build process for proficiently use ROS build system, having a good knowledge of CMake is also helping a lot in solving many problem when working in ROS

ROS Build System (2)

catkin build system is organized in a workspace containing different spaces and packages, this feature is very useful for having a common files/directory structure and for building multiple packages with complex dependencies.

A typical catkin workspace contains 4 (5) spaces:

- Source Space
- Build Space
- Devel space
- Install space
- (Log Space)

Result Space

Please keep separate catkin workspace when you use catkin_make and where you use catkin command line tools (e.g. catkin init ; catkin build).

Many tutorial available online use catkin_make, even if I strongly suggest using catkin build

NEVER MIX THE TWO COMMANDS IN THE SAME WS

ROS Build System (3)

CATKIN
BUILD SYSTEM

Work Here

The *source space* contains the source code. This is where
you can clone, create, and edit source code for the packages you want to build.

Don’t Touch

The build space is where CMake is invoked to build the packages in the source space. Cache information and other intermediate files are kept here.

Don’t Touch

The *development (devel) space* is where built targets are placed (prior to being installed).

Example of creating of a new catkin workspace using command line tools

```
source /opt/ros/kinetic/setup.bash
mkdir -p /tmp/quickstart_ws/src           # Make a new workspace
cd /tmp/quickstart_ws                     # Navigate to the workspace root
catkin init                               # Initialize it
cd /tmp/quickstart_ws/src                 # Navigate to the source space
catkin create pkg pkg_a                   # Populate the source space
catkin create pkg pkg_b
catkin create pkg pkg_c --catkin-deps pkg_a
catkin create pkg pkg_d --catkin-deps pkg_a pkg_b
catkin list                               # List the packages in the workspace
catkin build                              # Build all packages in the workspace
source /tmp/quickstart_ws/devel/setup.bash
```

Typical structure of Catkin Source Space

The *source space* contains the source code.

Organized in different *packages*

```
workspace_folder/            --  CATKIN WORKSPACE
 └─src/                         -- SOURCE SPACE
     ├──package_1/
     │       CMakeLists.txt   -- CMakeLists.txt file for package_1
     │       package.xml      -- Package manifest for package_1
     │  …
     └──package_n/
             CMakeLists.txt   -- CMakeLists.txt file for package_n
             package.xml      -- Package manifest for package_n
```

CMakeLists.txt is the configuration file for CMake → see Cmake docs for more details

Package.xml is a supporting file providing additiona package info and dependencies for catkin build system.

Typical structure of a package.xml

```
<package>
  <name>foo_core</name>
  <version>1.2.4</version>
  <description>
    This package provides foo capability.
  </description>
  <maintainer email="ivana@willowgarage.com">Ivana
Bildbotz</maintainer>
  <license>BSD</license>

 <buildtool_depend>catkin</buildtool_depend>
</package>
```

Typical structure of a package.xml

```
<package>
  <name>foo_core</name>
  <version>1.2.4</version>
  <description> This package provides foo capability. </description>
  <maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
  <license>BSD</license>

  <url>http://ros.org/wiki/foo_core</url>
  <author>Ivana Bildbotz</author>
  <buildtool_depend>catkin</buildtool_depend>
```

See previous slide

```
  <build_depend>message_generation</build_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>

  <run_depend>message_runtime</run_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>

  <test_depend>python-mock</test_depend>
</package>
```

For more details please check:

http://wiki.ros.org/catkin/conceptual_overview#Dependency_Management

<build_depend> Build Dependencies

<run_depend> Run Dependencies

<test_depend>Test Dependencies

<buildtool_depend> Build Tool Dependencies

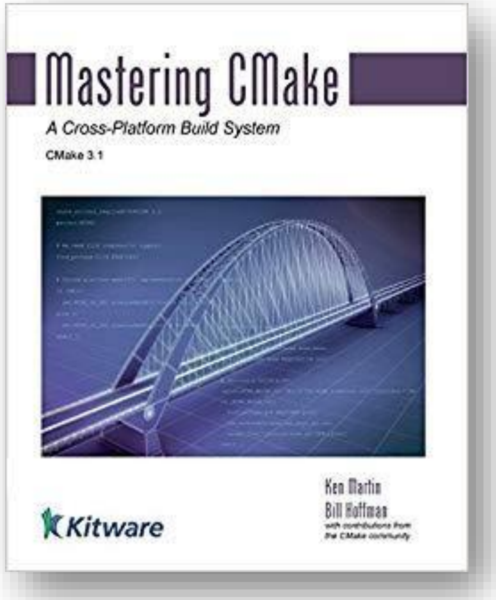
Typical structure of a CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
project(app_project)
add_executable(myapp main.c)
install(TARGETS myapp DESTINATION bin)
```

```
cmake_minimum_required(VERSION 2.8)
project(libtest_project)
add_library(test STATIC test.c)
install(TARGETS test DESTINATION lib)
install(FILES test.h DESTINATION include)
```

```
cmake_minimum_required(VERSION 2.8)
project(myapp)
add_subdirectory(libtest_project)
add_executable(myapp main.c)
target_link_libraries(myapp test)
install(TARGETS myapp DESTINATION bin)
```

More than 300 pages!



Mastering CMake

A Cross-Platform Build System

CMake 3.1

Kitware

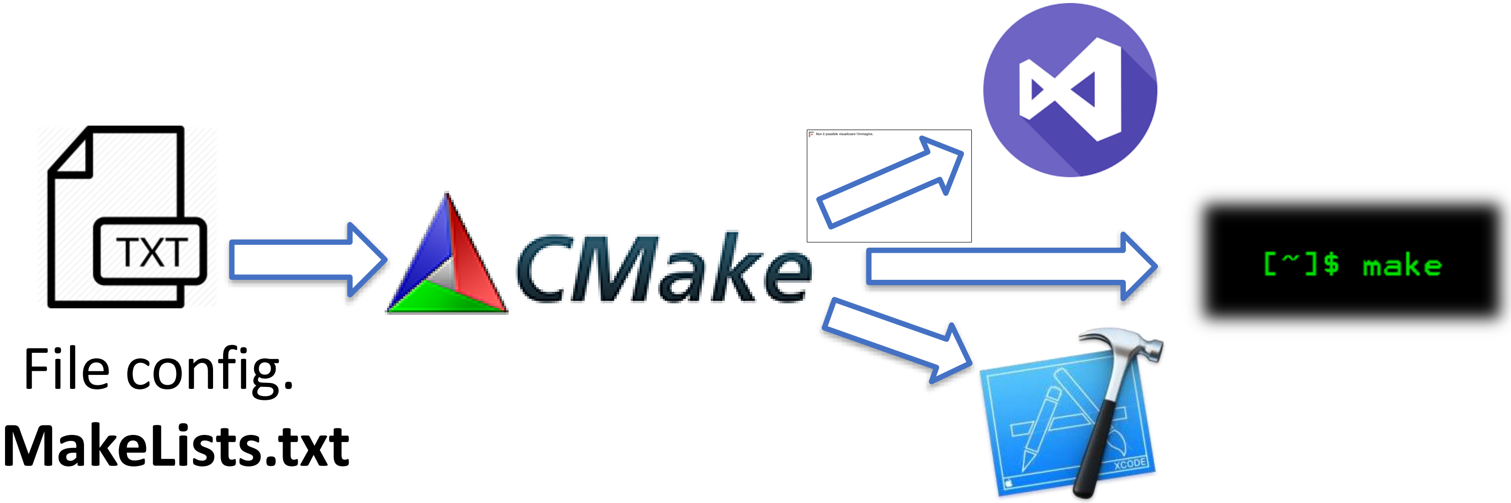
Ken Martin

Bill Hoffman

with contributions from the CMake community

CMake could be considered as a "meta build system"

CMake support a specific scripting language for the creation of its configuration files



```
[~]$ make
```

File config.

CMakeLists.txt

A more realistic CMakeLists.txt

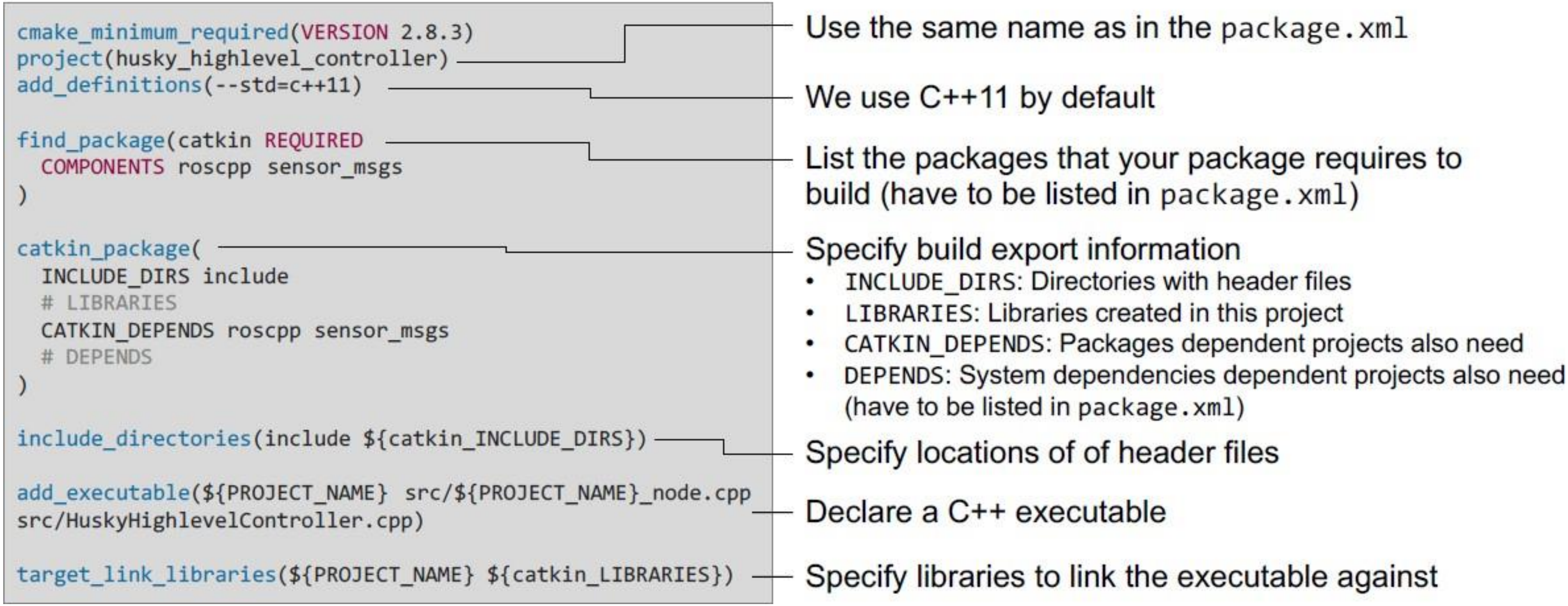
```
ExternalProject_Add(project_luajit
  URL http://luajit.org/download/LuaJIT-2.0.1.tar.gz
  PREFIX ${CMAKE_CURRENT_BINARY_DIR}/luajit-2.0.1
  CONFIGURE_COMMAND ""
  BUILD_COMMAND make
  INSTALL_COMMAND make install
  PREFIX=${CMAKE_CURRENT_BINARY_DIR}/luajit-2.0.1
)
ExternalProject_Get_Property(project_luajit install_dir)
add_library(luajit STATIC IMPORTED)
set_property(TARGET luajit PROPERTY IMPORTED_LOCATION
${install_dir}/lib/libluajit-5.1.a)
add_dependencies(luajit project_luajit)
add_executable(myapp main.c)
include_directories(${install_dir}/include/luajit-2.0)
target_link_libraries(myapp luajit)
```

When working in ROS (using C++ API) you need to modify CMakeLists.txt file prepared by catkin.

If you correctly use catkin the modification of the CMakeLists.txt are (almost ☺) straightforward

Many problems (i.e., errors) when working with ROS are related to wrong configuration of CMake build process → useful for searching the right solution ☺

Example of ROS Cmakelists.txt



```
cmake_minimum_required(VERSION 2.8.3)
project(husky_highlevel_controller)
add_definitions(--std=c++11)

find_package(catkin REQUIRED
  COMPONENTS roscpp sensor_msgs
)

catkin_package(
  INCLUDE_DIRS include
  # LIBRARIES
  CATKIN_DEPENDS roscpp sensor_msgs
  # DEPENDS
)

include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(${PROJECT_NAME} src/${PROJECT_NAME}_node.cpp
src/HuskyHighlevelController.cpp)

target_link_libraries(${PROJECT_NAME} ${catkin_LIBRARIES})
```

Use the same name as in the `package.xml`

We use C++11 by default

List the packages that your package requires to build (have to be listed in `package.xml`)

Specify build export information

- `INCLUDE_DIRS`: Directories with header files
- `LIBRARIES`: Libraries created in this project
- `CATKIN_DEPENDS`: Packages dependent projects also need
- `DEPENDS`: System dependencies dependent projects also need (have to be listed in `package.xml`)

Specify locations of of header files

Declare a C++ executable

Specify libraries to link the executable against

ROS Nodes

Single-purpose, executable program

Individually compiled, executed, and managed

Organized in *packages*

Run a node with

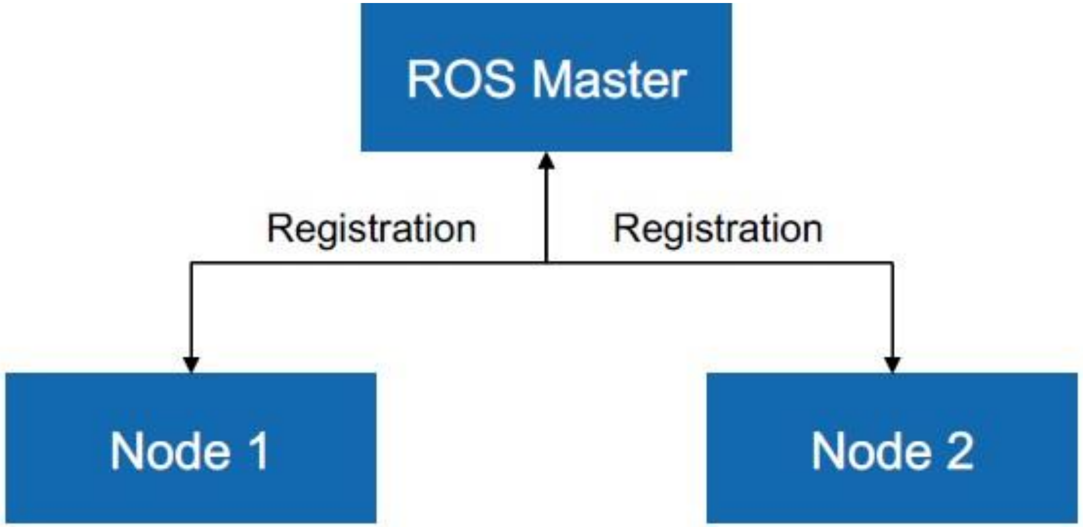
```
> rosrun package_name node_name
```

See active nodes with

```
> rosnode list
```

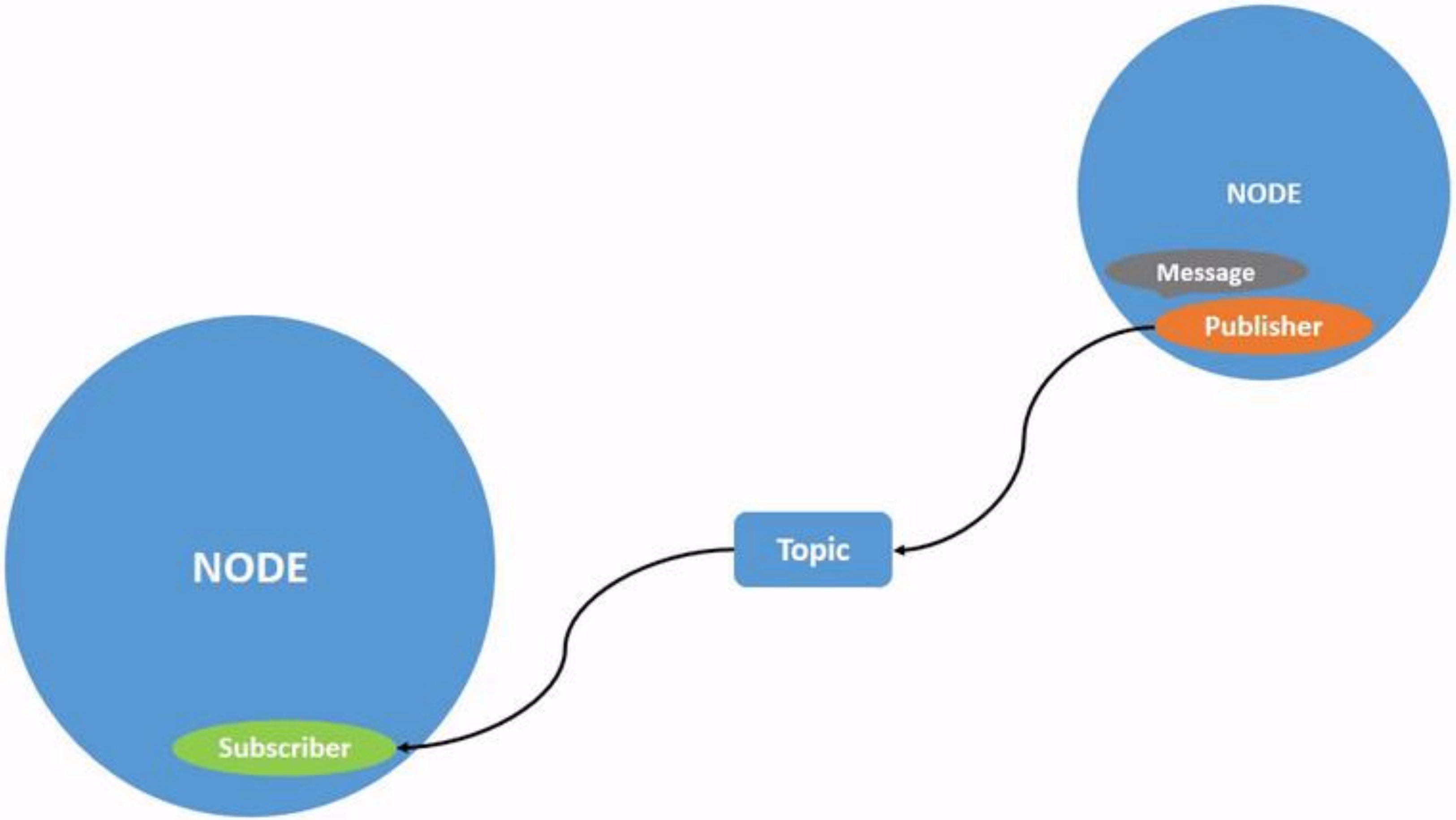
Retrieve information about a node with

```
> rosnode info node_name
```

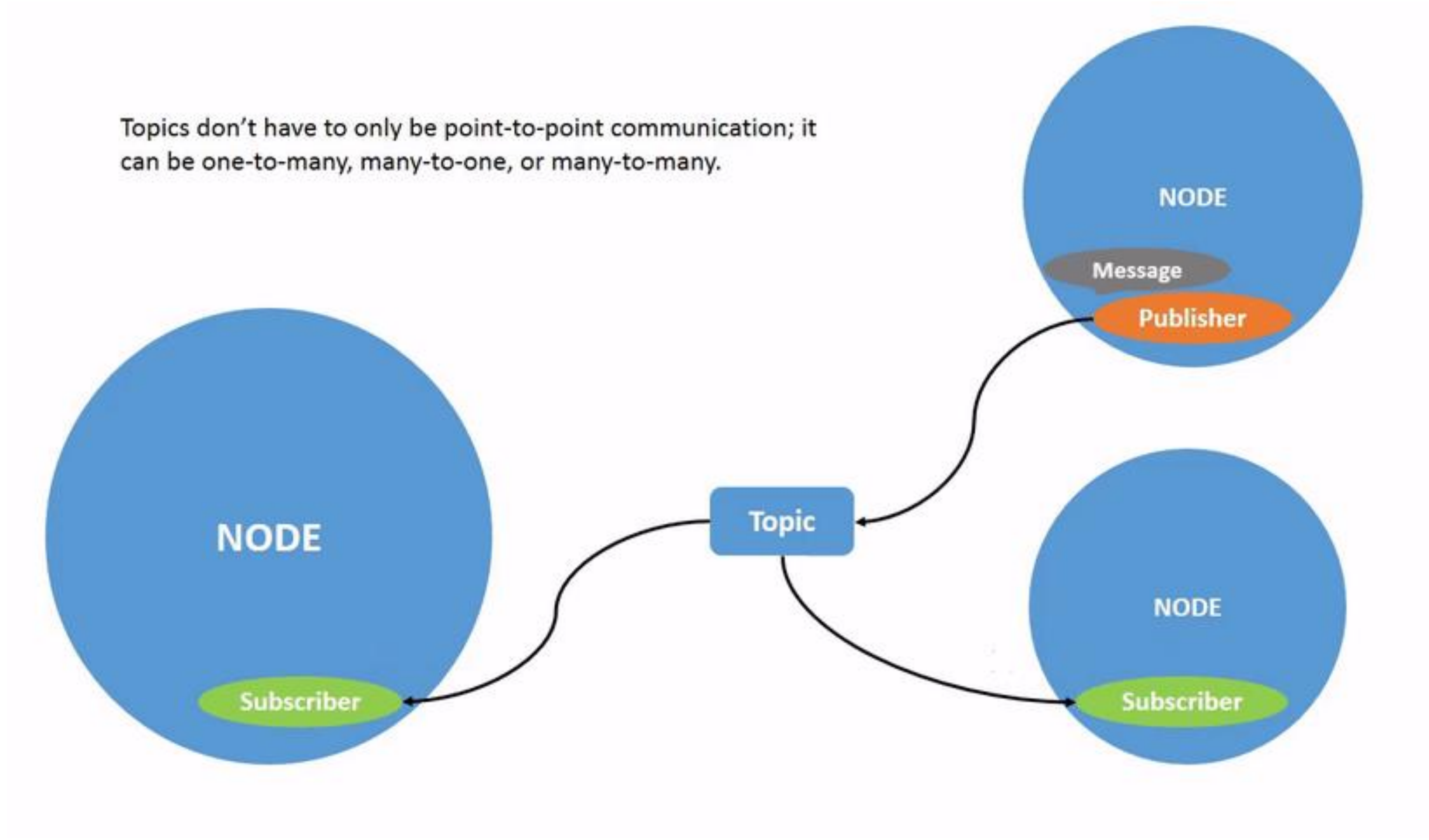


More info
http://wiki.ros.org/rosnode

ROS Nodes simplest communication



ROS Nodes general communication



Topics don’t have to only be point-to-point communication; it can be one-to-many, many-to-one, or many-to-many.

ROS Topics

- Nodes communicate over *topics*
  - Nodes can *publish* or *subscribe* to a topic
  - Typically, 1 publisher and *n* subscribers
- Topic is a name for a stream of *messages*

List active topics with

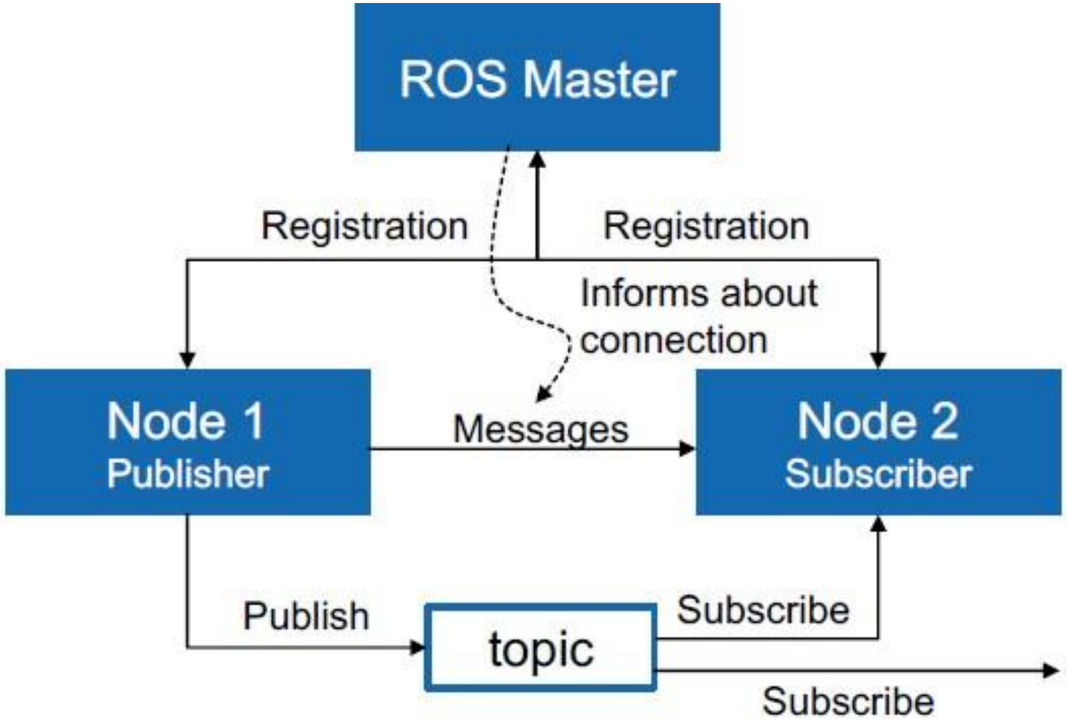
```
> rostopic list
```

Subscribe and print the contents of a topic with

```
> rostopic echo /topic
```

Show information about a topic with

```
> rostopic info /topic
```



More info
http://wiki.ros.org/rostopic

ROS Messages

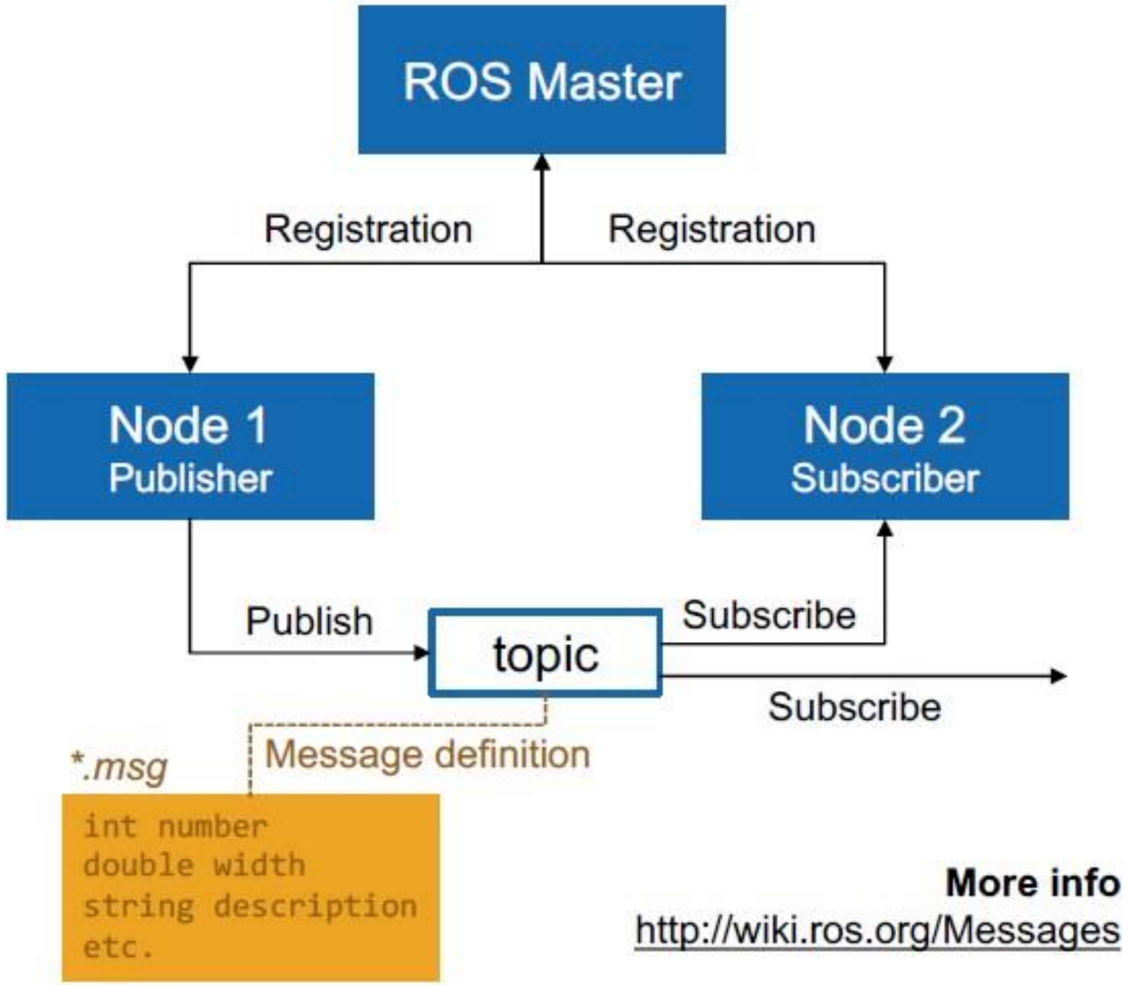
- Data structure defining the *type* of a topic
- Compromised of a nested structure of integers, floats, booleans, strings etc. and arrays of objects
- Defined in **.msg* files

See the type of a topic

```
> rostopic type /topic
```

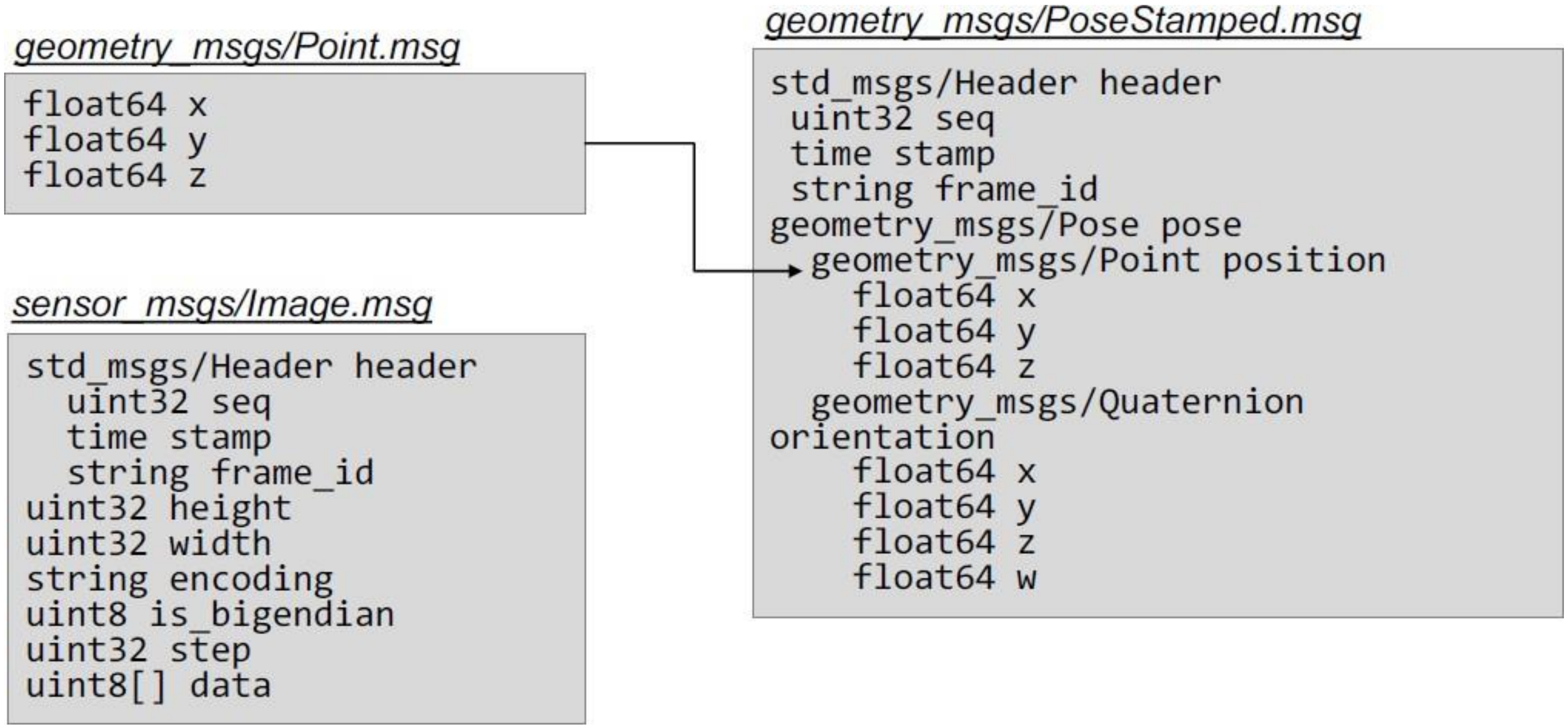
Publish a message to a topic

```
> rostopic pub /topic type args
```



More info
http://wiki.ros.org/Messages

ROS Message Example: PoseStamped



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
```

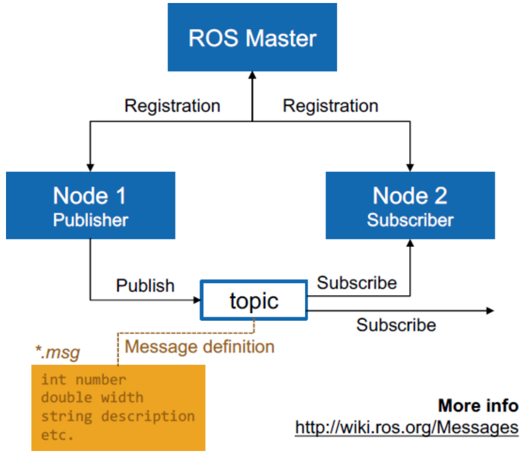
ROS Client Library (1)

A ROS client library is a collection of code that eases the job of the ROS programmer.

It takes many of the ROS concepts and makes them accessible via code.

In general, these libraries let you to:

- write ROS nodes,
- publish and subscribe to topics,
- write and call services,
- use the Parameter Server.



```
int number
double width
string description
etc.
```

More info
http://wiki.ros.org/Messages

Such a library can be implemented in any programming language

Main Client Libraries

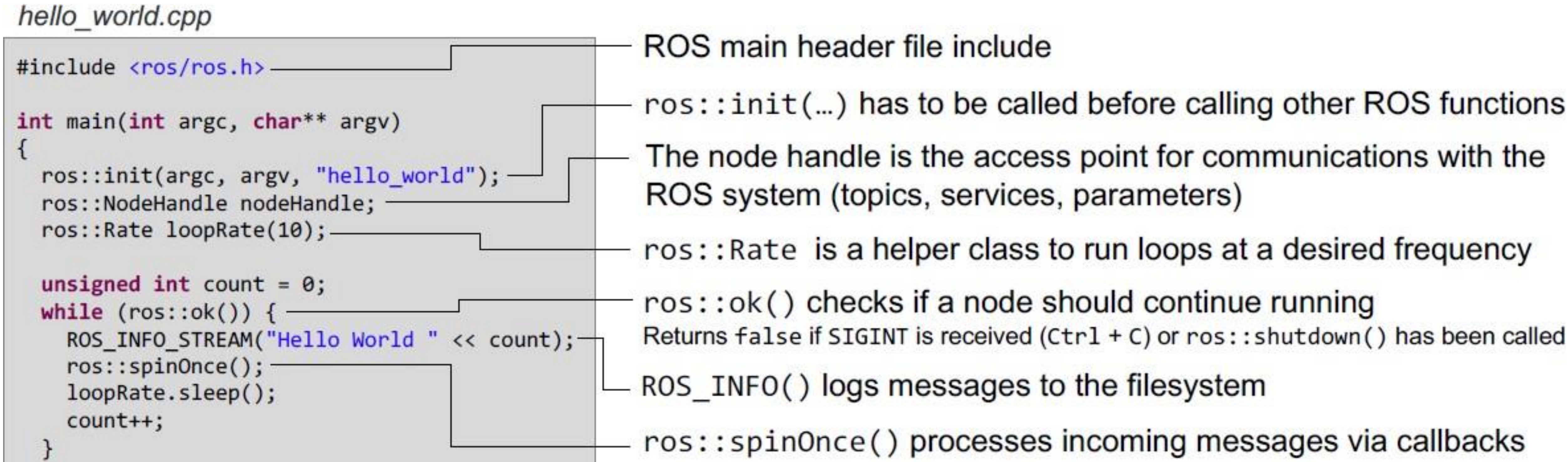
- **roscpp :** roscpp is a C++ client library for ROS. It is the most widely used ROS client library and is designed to be the high performance library for ROS.
- **rospy:** rospy is the pure Python client library for ROS and is designed to provide the advantages of an object-oriented scripting language to ROS. The design of rospy favors implementation speed (i.e. developer time) over runtime performance.

The ROS Master, roslaunch, and other ros tools are developed in rospy, so Python is a core dependency of ROS.

Basic tutorial

- Roscpp tutorial: http://wiki.ros.org/roscpp_tutorials/Tutorials/WritingPublisherSubscriber
- Rospy tutorial: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

ROSCPP Basic Source code



hello_world.cpp

```
#include <ros/ros.h>

int main(int argc, char** argv)
{
  ros::init(argc, argv, "hello_world");
  ros::NodeHandle nodeHandle;
  ros::Rate loopRate(10);

  unsigned int count = 0;
  while (ros::ok()) {
    ROS_INFO_STREAM("Hello World " << count);
    ros::spinOnce();
    loopRate.sleep();
    count++;
  }

  return 0;
}
```

ROS main header file include

`ros::init(...)` has to be called before calling other ROS functions

The node handle is the access point for communications with the ROS system (topics, services, parameters)

`ros::Rate` is a helper class to run loops at a desired frequency

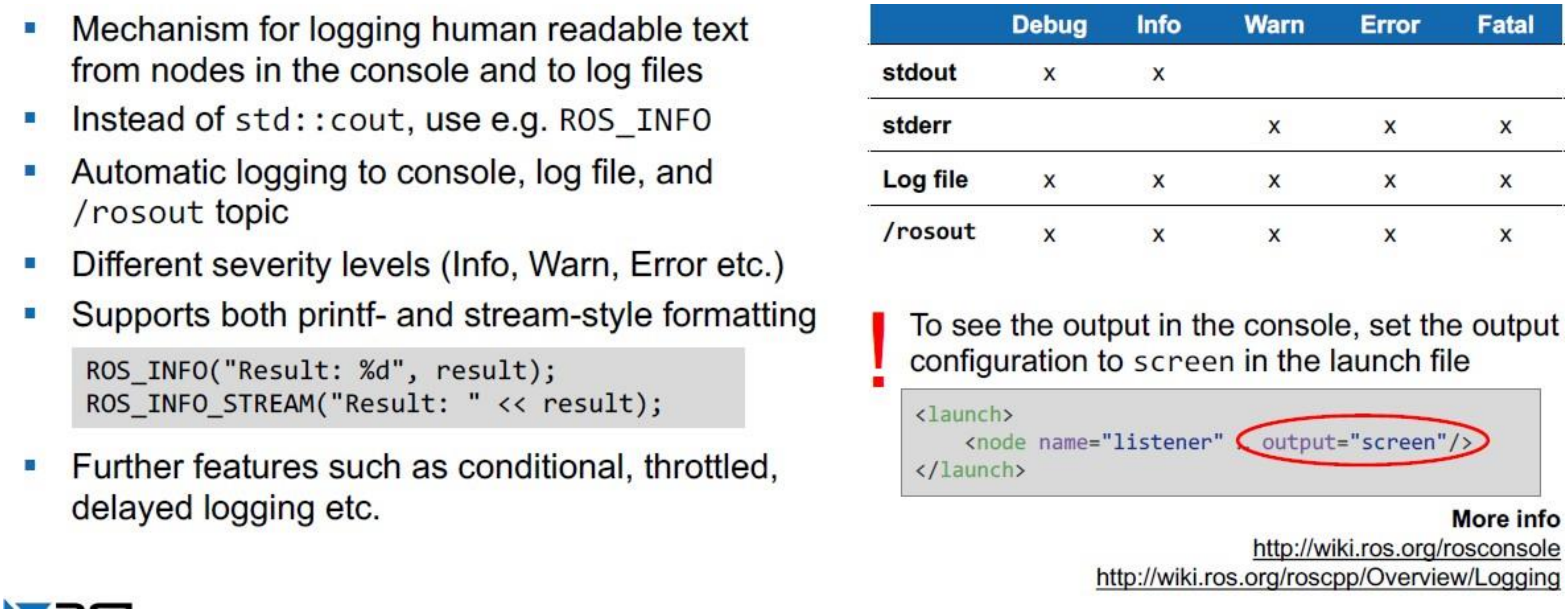
`ros::ok()` checks if a node should continue running
Returns `false` if `SIGINT` is received (`Ctrl` + `C`) or `ros::shutdown()` has been called

`ROS_INFO()` logs messages to the filesystem

`ros::spinOnce()` processes incoming messages via callbacks

More info
http://wiki.ros.org/roscpp
http://wiki.ros.org/roscpp/Overview

ROSCPP Logging



- Mechanism for logging human readable text from nodes in the console and to log files
- Instead of `std::cout`, use e.g. `ROS_INFO`
- Automatic logging to console, log file, and `/rosout` topic
- Different severity levels (Info, Warn, Error etc.)
- Supports both printf- and stream-style formatting

```
ROS_INFO("Result: %d", result);
ROS_INFO_STREAM("Result: " << result);
```

- Further features such as conditional, throttled, delayed logging etc.

	Debug	Info	Warn	Error	Fatal
stdout	x	x			
stderr			x	x	x
Log file	x	x	x	x	x
/rosout	x	x	x	x	x

! To see the output in the console, set the output configuration to `screen` in the launch file

```
<launch>
    <node name="listener" ... output="screen"/>
</launch>
```

More info
http://wiki.ros.org/rosconsole
http://wiki.ros.org/roscpp/Overview/Logging

ROSCPP Subscriber

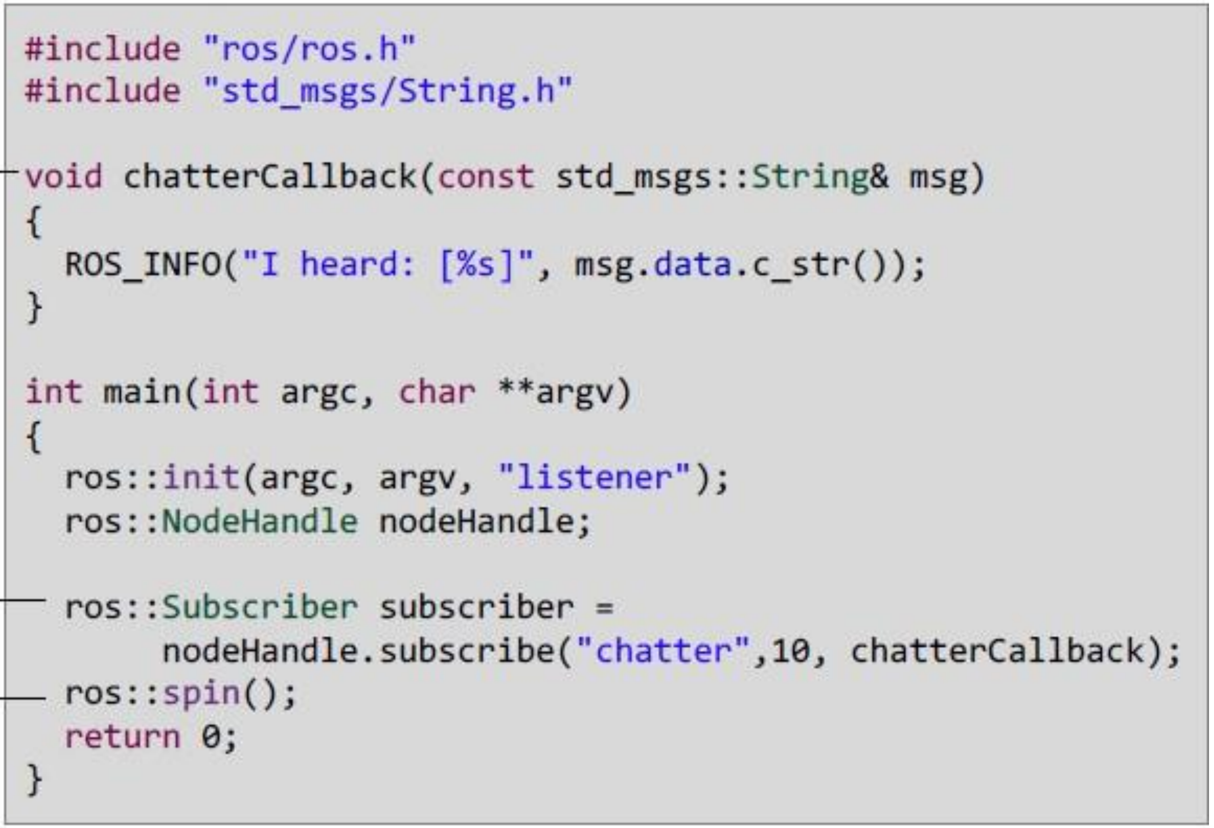
- Start listening to a topic by calling the method `subscribe()` of the node handle

```
ros::Subscriber subscriber =
nodeHandle.subscribe(topic, queue_size,
                     callback_function);
```

- When a message is received, callback function is called with the contents of the message as argument
- Hold on to the subscriber object until you want to unsubscribe

`ros::spin()` processes callbacks and will not return until the node has been shutdown

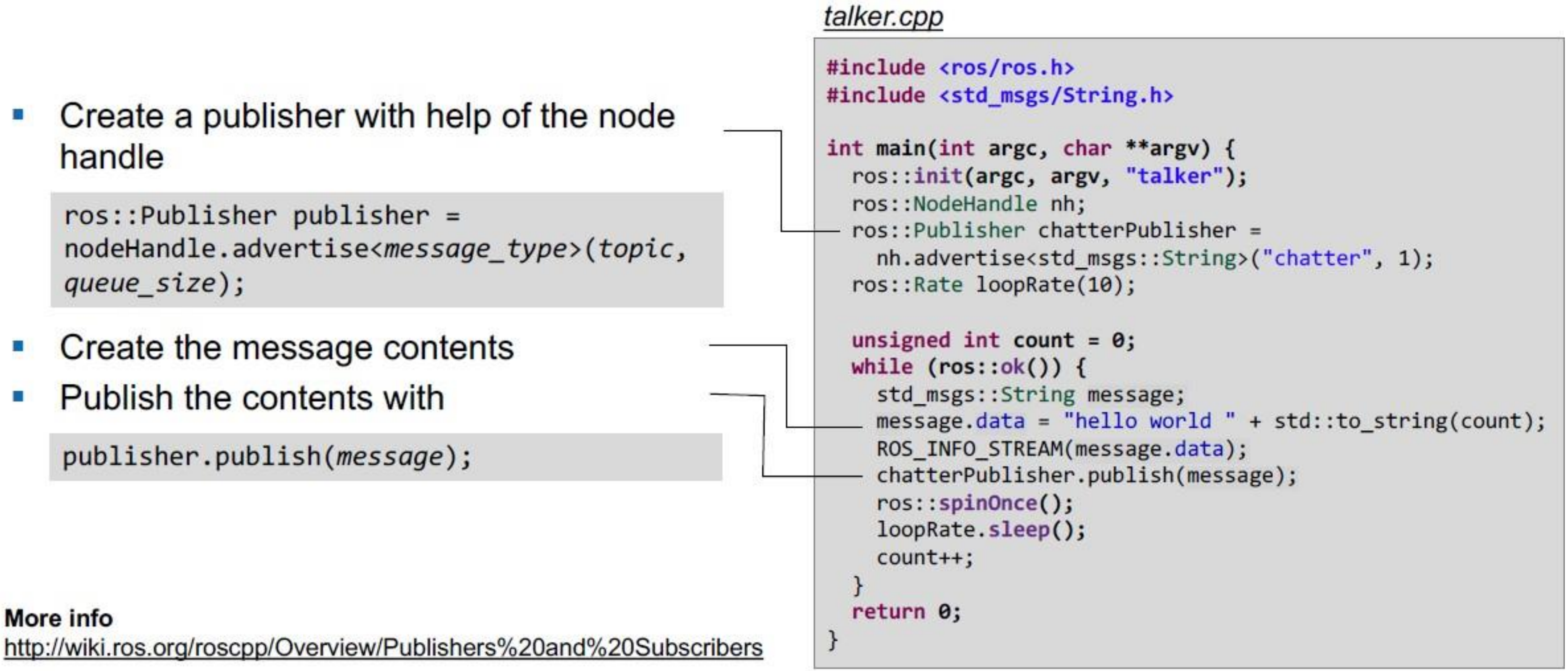
listener.cpp



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

void chatterCallback(const std_msgs::String& msg)
{
  ROS_INFO("I heard: [%s]", msg.data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle nodeHandle;

  ros::Subscriber subscriber =
       nodeHandle.subscribe("chatter",10, chatterCallback);
  ros::spin();
  return 0;
}
```

More info
http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers

ROSCPP Publisher



- Create a publisher with help of the node handle

```
ros::Publisher publisher =
nodeHandle.advertise<message_type>(topic,
queue_size);
```

- Create the message contents
- Publish the contents with

```
publisher.publish(message);
```

More info
http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers

talker.cpp

```
#include <ros/ros.h>
#include <std_msgs/String.h>

int main(int argc, char **argv) {
  ros::init(argc, argv, "talker");
  ros::NodeHandle nh;
  ros::Publisher chatterPublisher =
    nh.advertise<std_msgs::String>("chatter", 1);
  ros::Rate loopRate(10);

  unsigned int count = 0;
  while (ros::ok()) {
    std_msgs::String message;
    message.data = "hello world " + std::to_string(count);
    ROS_INFO_STREAM(message.data);
    chatterPublisher.publish(message);
    ros::spinOnce();
    loopRate.sleep();
    count++;
  }
  return 0;
}
```

ROSCPP Publisher

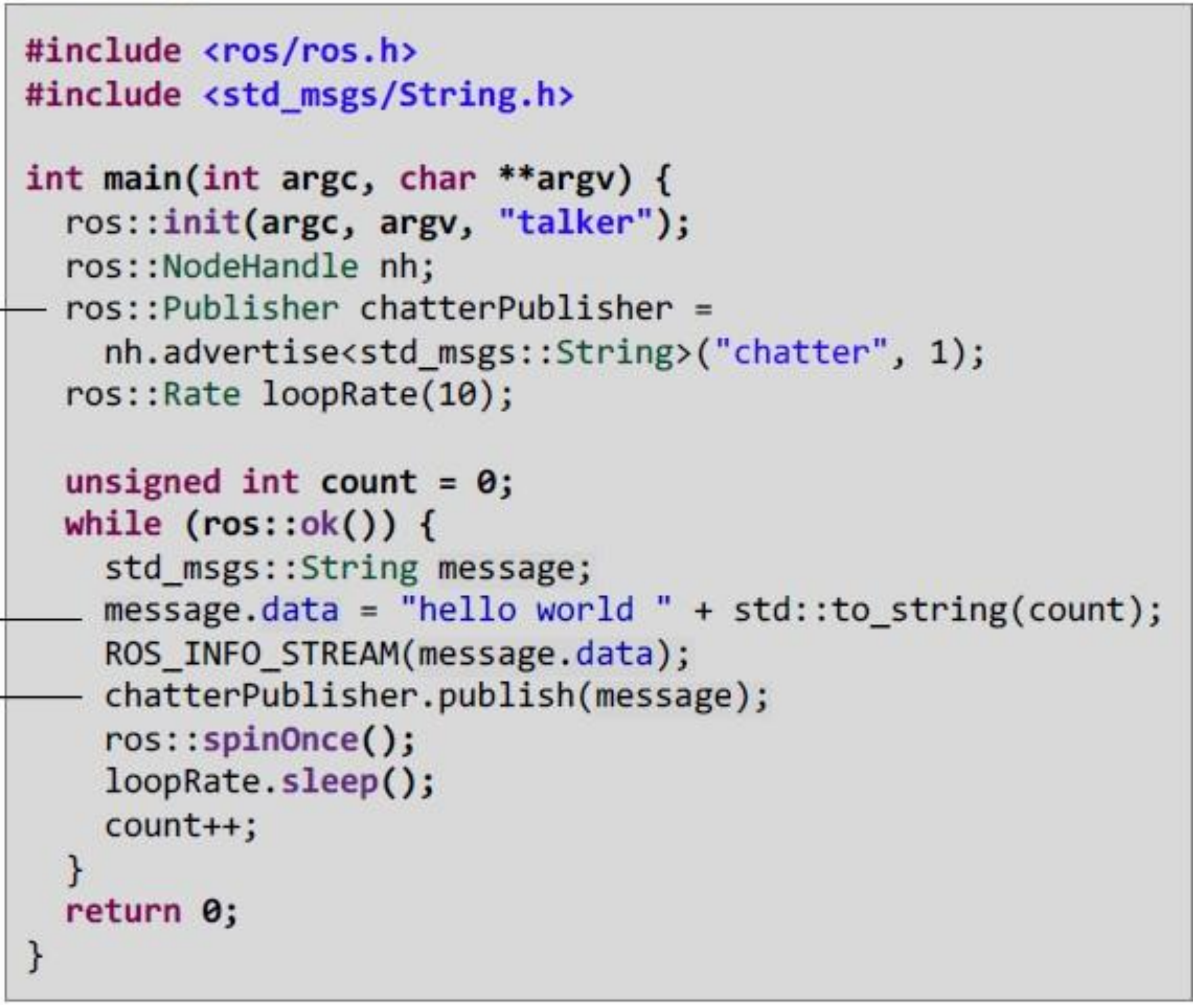
- Create a publisher with help of the node handle

```
ros::Publisher publisher =
nodeHandle.advertise<message_type>(topic,
queue_size);
```

- Create the message contents
- Publish the contents with

```
publisher.publish(message);
```

talker.cpp

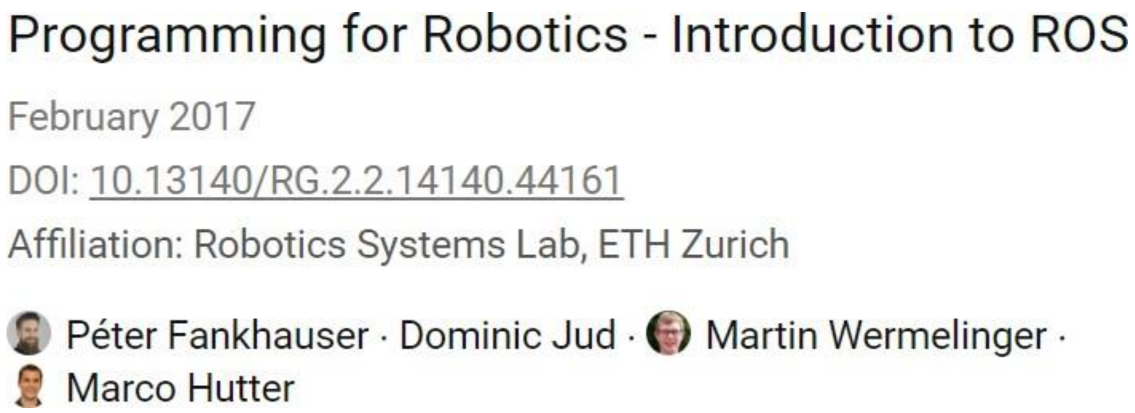


```
#include <ros/ros.h>
#include <std_msgs/String.h>

int main(int argc, char **argv) {
  ros::init(argc, argv, "talker");
  ros::NodeHandle nh;
  ros::Publisher chatterPublisher =
    nh.advertise<std_msgs::String>("chatter", 1);
  ros::Rate loopRate(10);

  unsigned int count = 0;
  while (ros::ok()) {
    std_msgs::String message;
    message.data = "hello world " + std::to_string(count);
    ROS_INFO_STREAM(message.data);
    chatterPublisher.publish(message);
    ros::spinOnce();
    loopRate.sleep();
    count++;
  }
  return 0;
}
```

More info
http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers

Questions?

The contents of these slides are partially based on:



Programming for Robotics - Introduction to ROS

February 2017

DOI: 10.13140/RG.2.2.14140.44161

Affiliation: Robotics Systems Lab, ETH Zurich

Péter Fankhauser · Dominic Jud · Martin Wermelinger · Marco Hutter

Exercise A - FUNDAMENTAL

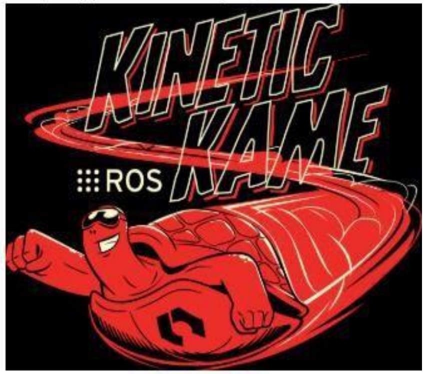
Install Ubuntu and ROS on your machine:

- You can use a virtual machine
- It would be better if we are able to use the same Ubuntu and ROS version

you could follow the tutorial available at:
http://wiki.ros.org/noetic/Installation

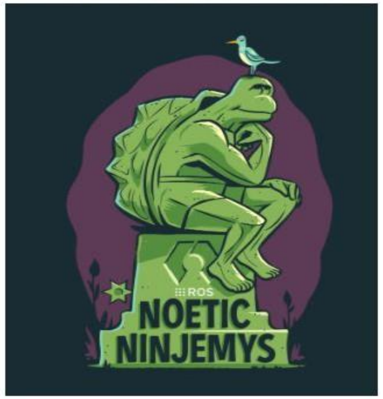
Please contact me if you are not able to complete this step in 1 week , i.e. by <u>Tuesday 15 March 2022</u>

ROS Kinetic Kame
Released May, 2016
LTS, supported until April, 2021



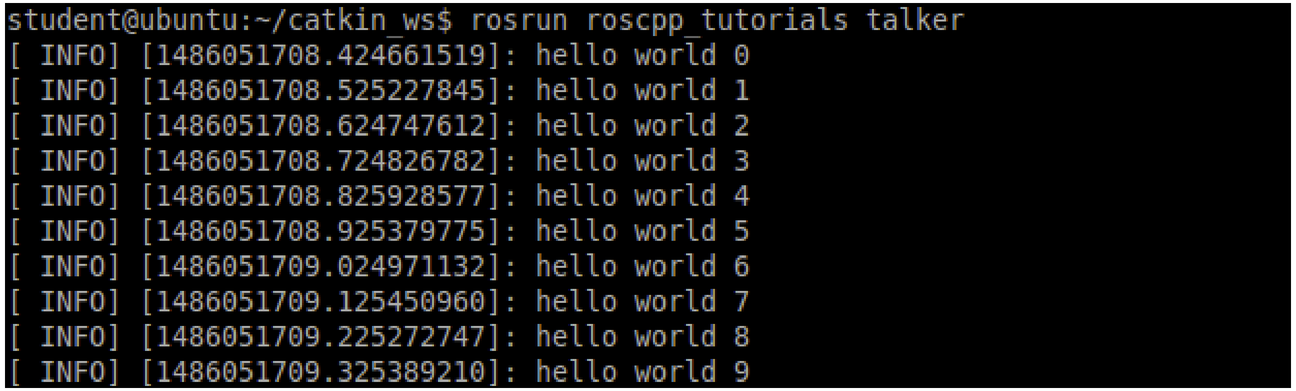
ROS Melodic Morenia
Released May, 2018
Latest LTS, supported until May, 2023

ROS Noetic Ninjemys
Released May, 2020
Latest LTS, supported until May, 2025
Recommended for Ubuntu 20.04



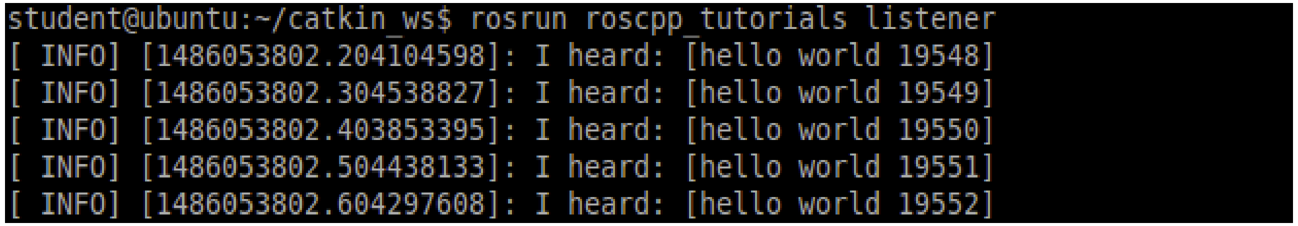
Exercises B

Implement the talker → listener example (following C++ or python tutorial)



```
student@ubuntu:~/catkin_ws$ rosrun roscpp_tutorials talker
[ INFO] [1486051708.424661519]: hello world 0
[ INFO] [1486051708.525227845]: hello world 1
[ INFO] [1486051708.624747612]: hello world 2
[ INFO] [1486051708.724826782]: hello world 3
[ INFO] [1486051708.825928577]: hello world 4
[ INFO] [1486051708.925379775]: hello world 5
[ INFO] [1486051709.024971132]: hello world 6
[ INFO] [1486051709.125450960]: hello world 7
[ INFO] [1486051709.225272747]: hello world 8
[ INFO] [1486051709.325389210]: hello world 9
```

Modify the code for printing the following string «Hello world from YOUR_STUDENT_ID counter»



```
student@ubuntu:~/catkin_ws$ rosrun roscpp_tutorials listener
[ INFO] [1486053802.204104598]: I heard: [hello world 19548]
[ INFO] [1486053802.304538827]: I heard: [hello world 19549]
[ INFO] [1486053802.403853395]: I heard: [hello world 19550]
[ INFO] [1486053802.504438133]: I heard: [hello world 19551]
[ INFO] [1486053802.604297608]: I heard: [hello world 19552]
```