Introduction to ROS

Laurent Lequièvre
Juan Antonio Corrales Ramon
Institut Pascal – Clermont-Ferrand - France

Index

- Introduction
  - Definition
  - History
  - Robots
  - Advantages
  - Applications
- Basic components
  - Nodes
  - Topics
  - Services

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

INTRODUCTION

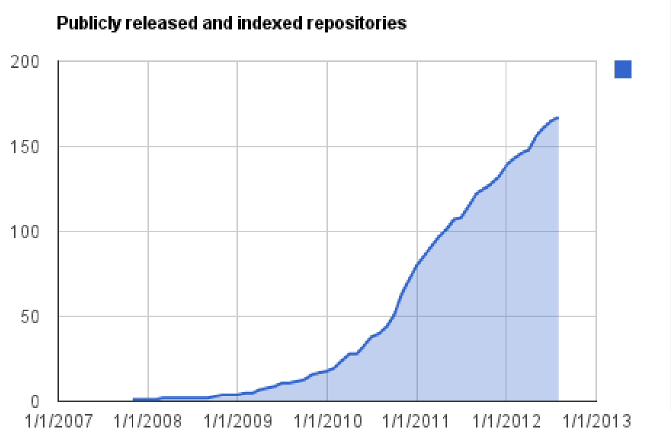
Definition of ROS

- ROS (Robot Operating System) is a **software platform** that is able to build and execute code between several computers and several robots.
- It provides services similar to an **operating system** for working with robots:
  - Hardware abstraction
  - Device low level control
  - Message-based communication
  - Commands and utilities
  - Package management

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

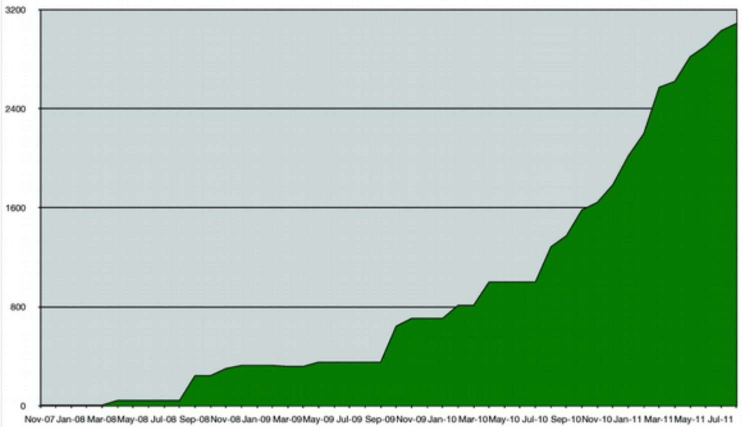
History of ROS

- Initially developed by **Standford University** in the "STAIR" project in 2007.
- From 2008 to 2013, its development continued mainly due to the contributions of **Willow Garage** for their humanoid robot PR2 and the scientific community.
- From 2013, ROS depends on the **Open Source Robotics Foundation** and its development continues.
- Exponential increase in the number of repositories (more than 170 in 2013).



Robots in ROS

- Exponential increase in the number of packages (more than 3500 in 2012).



- Use in more than 50 robots: http://wiki.ros.org/Robots

https://www.youtube.com/watch?v=PGaXiLZD2KQ

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

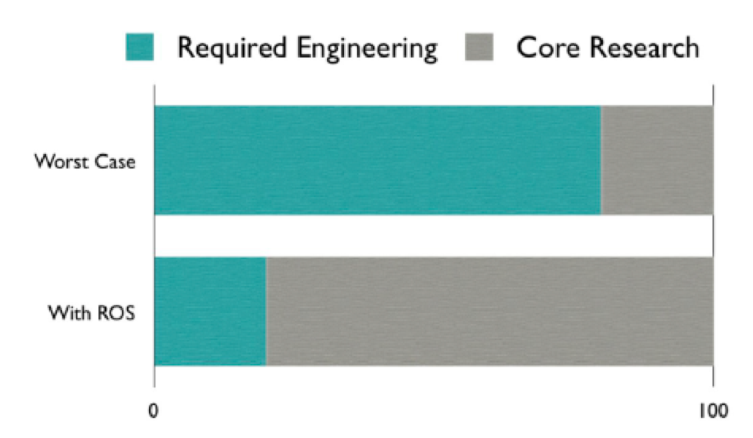
Advantages

- The main features of ROS are :
  - **Peer-to-peer:** It consists of a group of processes, that can be in different systems, are connected between them in a peer-to-peer topology.
  - **Multi-language:** ROS packages can be implemented in different programming languages (C++, Python, Octave y LISP). ROS messages are defined in a neutral lenguage (IDL) that enables the automatic generation of code in different languages.
  - **Tools-Based**: Micro-kernel design with a high number of small tools. Bigger stability and better complexity management are possible.
  - **Light:** The ROS system is built in a modular way. All complexity is moved towards external libraries: easy importation and exportation.
  - **Open-source:** All source is available and most packages have BSD licence (commercial and non-commercial projects).

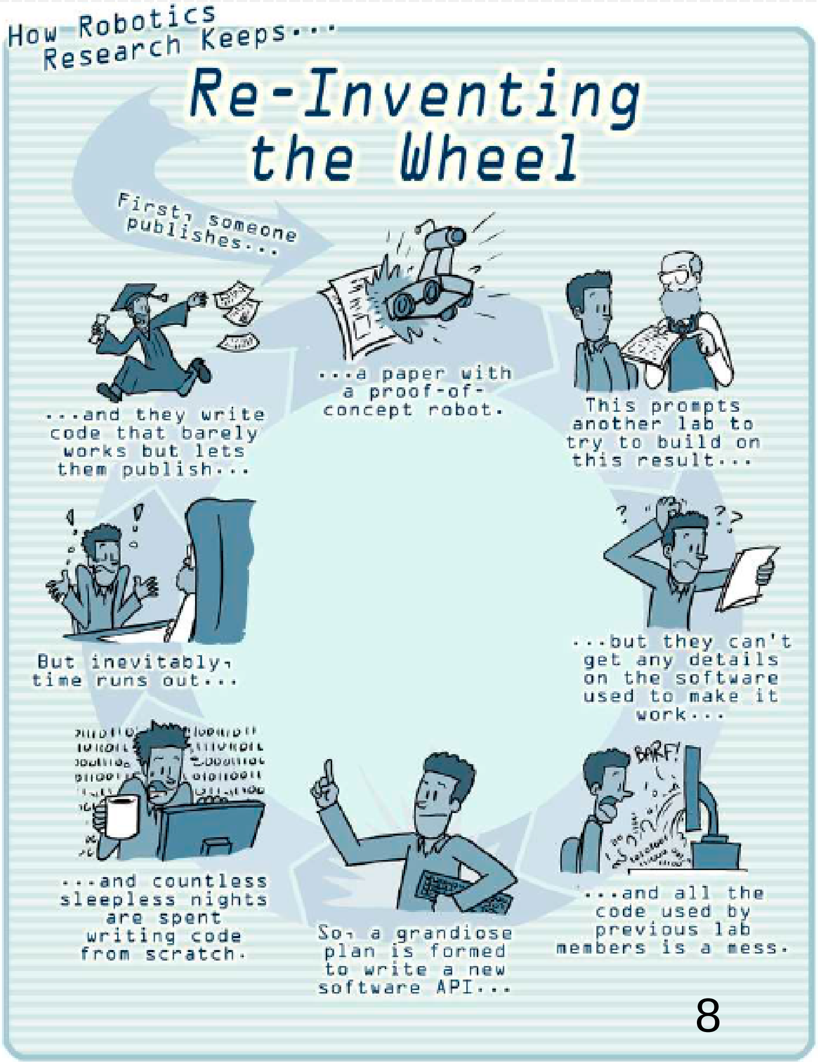
INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Advantages

- Main goal of ROS:
  - Reusing code and comparing results
  - Having more time for research
  - Less time for re-implementation
  - Current Platform: > 150 People-Year.



INSTITUT PASCAL
sciences de l'ingénierie et des systèmes



Applications

- Main applications of ROS in robotics:
  - Visualization and simulation: **rviz**, stage (2D), **gazebo** (3D).
  - Drivers: camera_drivers, laser_drivers, imu_drivers.
  - 3D processing: perception_pcl (**PCL**), laser_pipeline.
  - Image processing (2D): vision_opencv (**OpenCV**), visp.
  - Transformations: tf, tf_conversions.
  - Navigation (odometry, ego-motion, SLAM): navigation.
  - Controllers (position, force, speed, transmissions): ros_control.
  - Robot modelling: urdf (XML description of robots).
  - Motion planning: **MoveIt!** (library OMPL).
  - Grasping and manipulation: GraspIt!, OpenRAVE.

Index

- Introduction
  - Definition
  - History
  - Robots
  - Advantages
  - Applications
- Basic components
  - Nodes
  - Topics
  - Services

BASIC COMPONENTS

ROS Levels

- The basic concepts of ROS can be analyzed from 3 different levels:
  - **File-system level:** Elements that are in the hard-drive
    - Packages, manifests, meta-packages (stacks), message types (msg) and service types (srv).
  - **Execution level:** ROS process that treat data in a peer-to-peer architecture
    - Nodes, master, parameter server, messages, topics, services and bags.
  - **Community level:** Resources for ROS (software and documentation) that are shared between different groups of users.
    - Distributions, repositories, ROS Wiki, mailing lists, ROS Answers and ROS Blog.

Packages

- Packages are the **basic unit** that organize software in ROS.
- They can contain: processes (nodes), libraries, data, configuration files, etc.
- It is a folder inside ROS_ROOT (installation) or **ROS_PACKAGE_PATH** (workspace) that contains a file **package.xml**
- Packages usually have the same structure:

  include/: headers for libraries C++.
  msg/: Types of messages (msg).
  src/: Source code.
  srv/: Types of services (srv).
  scripts/: executable scripts (normally in Python).
  CMakeLists.txt: Cmake file, necessary for compilation of the package.
  package.xml: File with XML specification of package meta-data (Two versions → New: **<package format="2">**)
  - <name>, <version>, <description>, <maintainer><license>: General information of the package.
  - <build_depend>/<build_export_depend>(v.2): Dependencies for building the package or others based on it.
  - <exec_depend>(v.2) /<run_depend>(v.1): Dependencies on other packages for execution of this package.
  - <depend>(v.2): All types of dependencies (build, execution, export) on other packages.
  - <export>: Definition of meta-packages or external tags (e.g.: inclusion of Gazebo plugins)
  - rosdep command (*rosdep install [package]*) will use it for installing dependencies of the package.

- Command **rospack** (options 'find, depends, list') and command **roscd**.

INSTITUT PASCAL sciences de l'ingénierie et des systèmes

Workspace

- It is a folder for working with packages: modify, compile and install them.
- It is composed by 4 spaces:
  - **src:** source code files (cpp/py) of packages
  - **build:** intermediate files of CMake
  - **devel:** output files from building process (executables, libraries, msgs/service code...)
  - **install:** installation of generated files
- Automatic generation of code: **catkin**
- Creation of catkin workspace:
  1. mkdir -p ~/catkin_ws/src
  2. cd ~/catkin_ws/src
  3. catkin_init_workspace 4. cd ~/catkin_ws/
  5. catkin_make 6. source devel/setup.bash

```
workspace_folder/        -- CATKIN WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- The 'toplevel' CMake file
    package_1/
      CmakeLists.txt     – CMake file for package_1
      package.xml
      ...
    package_n/
      CMakeLists.txt
      package.xml
      ...
  build/                 -- BUILD SPACE
  devel/                 -- DEVELOPMENT SPACE
    bin/
    etc/
    include/
    lib/
    share/
    setup.bash
    ...
  install/               -- INSTALL SPACE
```

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Types of Messages

- ROS uses a text file in the IDL language for describing **data types** (messages) that are publish by ROS nodes.

- This description is stored in **.msg files** inside the **msg/ subfolder** of a ROS package.

- There are two parts in a .msg file:

  Fields (Data types that are sent inside the message):
  - Field: data type + name. Example: int32 x
  - Data types:
    - Basic: bool, int32, float32, float 64, string, time, duration, etc.
    - Arrays.
    - Other messages. Example: geometry_msgs/PoseStamped.
    - Header: ID, timestamp, frame ID.

  Constants (Values for interpreting the fields):
  - Constant: Type_constant name_constant= constant_value. Example: int32 X=123.

- Automatic generation of message through CmakeLists.txt and package.xml

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Types of Services

- ROS uses one text file in a descriptive language for indicating data types of the **request/response** of a service.

- This description is stored in **ficheros .srv files** inside the **srv/ subfolder** of a ROS package.
- There are two parts in a .srv file, separated by a line containing "---":
  - Request
  - Response

```
#request
int8 foobar
another_pkg/AnotherMessage
---
#response
another_pkg/YetAnotherMessage
val uint32 an_integer
```

- Automatic generation of services through CmakeLists.txt and package.xml

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Nodes

- A node is a **process** that executes a computation task.
- Nodes **communicate between them** through: services, topics and parameters.
- ROS is composed in the execution level by a set of nodes that communicate between them and that can be distributed in different machines.
- Nodes are programmed through client libraries: roscpp, rospy y roslisp.
- Command **rosnode** (options 'list, info, kill, ping') and command **rosrun** (ex: turtlesim).

Master

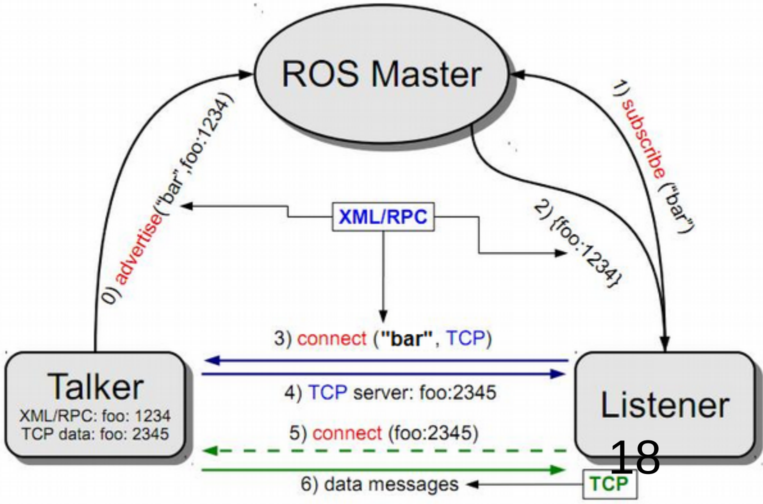
- Program that enables localization between ROS nodes (similar to DNS).
- Once nodes are localized, the master does not participate and there is peer-to-peer communication between them.
- **It registers topics and services** of every node.
- It is initialized by the command **roscore**.

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Topics

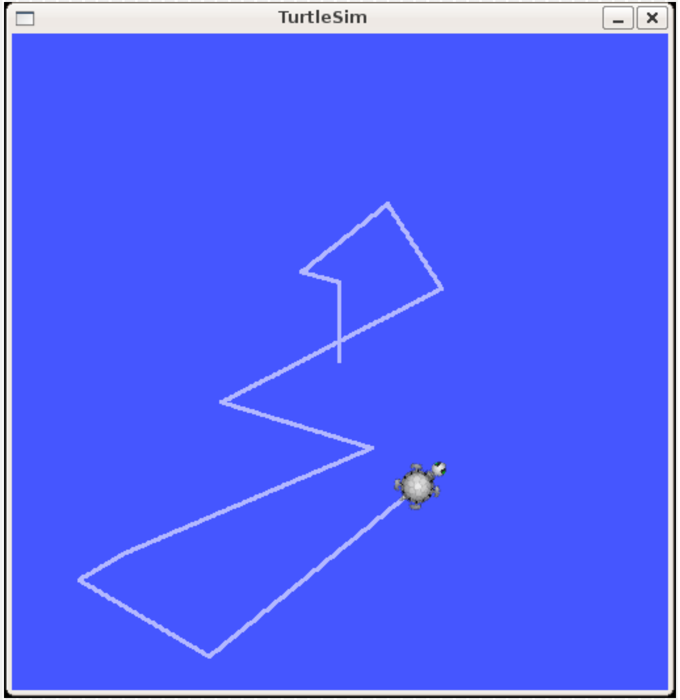
- A topic is a **bus with a name** through which nodes interchange messages.
- It is a method of **asynchronous communication** between the nodes:
  - **Publishers**: Nodes that publish messages in a topic.
  - **Subscribers**: Nodes that receive messages through a topic.
  - There can be several publishers and subscribers for a same topic.

- The initial connection between subscribers and publishers is done through the Master. Later, their communication is peer-to-peer through TCP:
  1. The publisher registers the topic in the Master.
  2. The subscriber asks the topic to the Master.
  3. The master gives the URI of the publisher to the subscriber.
  4. The subscriber asks the publisher for a topic connection.
  5. The publisher informs about the TCP setup.
  6. The subscriber connects to the TCP data port.
  7. Bidirectional communication of data through TCP is done.



Topics

- Example turtlesim:
  - Execute publisher: rosrun turtlesim turtle_teleop_key
  - Execute subscriber: rosrun turtlesim turtlesim_node
  - Communication through topic /turtle1/command_velocity
  - Command **rqt_graph** to see execution graph (nodes+topics):



  - Command **rostopic** (options: list, echo, type, pub):

    Examples: **rostopic echo /turtle1/cmd_vel ; rostopic type /turtle1/cmd_vel**

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

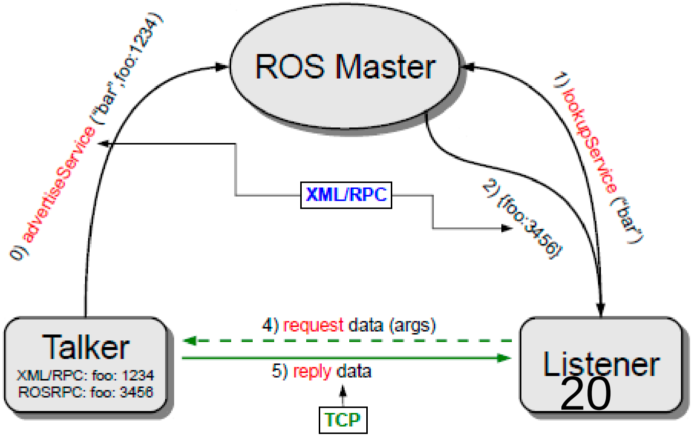
    rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'

Services

- A service enables a node to send a request (message **request**) and receive a response (message **response**) from other node.

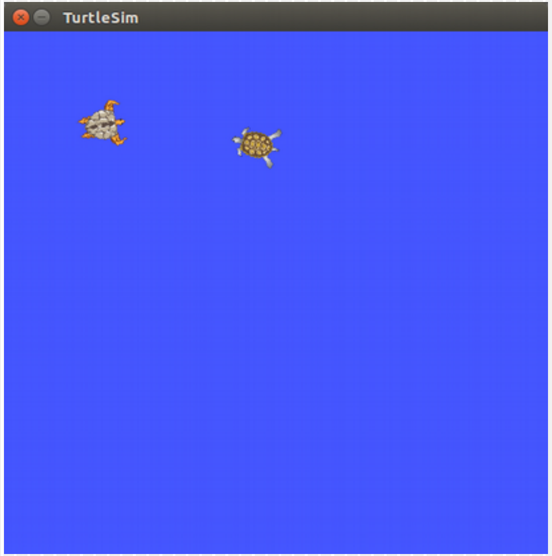
- It s a method of **synchronous communication** between the nodes (similar to RPC):
  - **Provider**: Node that provides the service (it receives request and sends response).
  - **Client**: Node that asks for the service (it sends request and receives response).

- The initial connection between providers and clients is done through the Master. Later, the communication is peer-to-peer through TCP:
  1. The provider registers the service in the Master.
  2. The client asks the Master for the service.
  3. The master informs the client about the TCP setup.
  4. The client sends the request message to the provider.
  5. The provider sends the response message to the client.



Services

- Command **rosservice** (Examples with turtlesim):
  - list: List of active services.
  - call: Execute the servie with the indicated parameters.
  - type: Print the type of service (types request/response).
  - find: Find the service.
  - Examples: **rosservice call clear ; rosservice call spawn 2 2 0.2 ""**



- Command **rossrv** (Examples with turtlesim):
  - show: Show the types of the request y response messages of the service.
  - package: List the types of services defined in the srv folder of the package.
  - Examples: **rosservice type spawn | rossrv show**

- Command **rosmsg** (Examples with turtlesim):
  - show: Show the fields of the message.
  - package: List the types of the messages defined in the msg folder of the package.

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Concepts of ROS community

- **Distribution:** Group of packages of a version. It is similar to Linux distributions and makes easy software integration (compatible libraries) and system stability (bugs management).

- **Repository:** Web server with ROS packages generated by the same institution. They are organised in a network where each institution keeps its own repository.

- **ROS Wiki:** Website with documentation and tutorials. It is the main source of ROS information (http://www.ros.org/wiki/).

- **ROS Blog:** News (http://www.ros.org/news/).

- **ROS Answers:** Forum where users ask and answer questions (http://answers.ros.org).

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

PROGRAMMING EXAMPLES

Create and compile a new ROS package

- **catkin_create_pkg [package_name] [depend1] [depend2]**…: Command for generating a new package (indicating name and dependencies). It creates automatically the package structure (manifest, Cmakelists.txt, makefiles, etc.).
- **Create a new package** "tutorials" in the workspace:
  cd ~/catkin_ws/src
  catkin_create_pkg tutorials std_msgs rospy roscpp

- **Compile the workspace with the command catkin_make:**

1. Go to main folder of the workspace: cd ~/catkin_ws
2. Compile: catkin_make

3.a. Add workspace to ROS (only in current terminal): source ./devel/setup.bash
3.b. For all terminals: echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc

- Change to the package folder : roscd tutorials
- If it doesn't work, update list of packages of system with: rospack profile

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Create a message

- **Create a message** that contains an 64bit integer in the folder **/msg**:
  roscd tutorials
  mkdir msg
  Copy file positionAngle.msg into msg/ folder
- Update **package.xml** for generating messages:
  <build_depend>**message_generation**</build_depend>
  <exec_depend>**message_runtime**</exec_depend> *(v2)*
  <build_export_depend>**message_runtime**</build_export_depend> *(v2)*
  <depend>**std_msgs**</depend> *DEPENDENCIES ON OTHER MSGS (v2)*
- Update **CmakeLists.txt** for generating messages:
  find_package(catkin REQUIRED COMPONENTS... **message_generation std_msgs**)
  catkin_package(… CATKIN_DEPENDS **message_runtime std_msgs**)
  add_message_files(FILES **positionAngle.msg** …)
  generate_messages(DEPENDENCIES **std_msgs**)
- If you copy CmakeLists.txt, **edit it** so that Cmake is forced to compile it.

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Create a topic publisher (I)

- Copy the code of publisher (file publisher.cpp) in sub-folder /src:
  roscd tutorials
  cp ~/publisher.cpp ~/catkin_ws/src/tutorials/src/publisher.cpp

- EXPLANATION OF THE CODE OF THE PUBLISHER (publisher.cpp)
- #include “ros/ros.h”
  - It includes the common libraries of the ROS system (including access to roscpp functionalities)
- #include “tutorials/positionAngle.h”
  - It includes the message “tutorials/positionAngle” that we generated.
  - This header was generated automatically from the file “positionAngle.msg”.
- ros::init(argc,argv,”position_publisher”);
  - **It initializes a ROS node**, gives access to command line arguments and names the node (unique name).
- ros::NodeHandle n;
  - **It creates a handle** for accessing the node.
  - The first handle starts the node (ros::start()) and the destruction of the last handle finishes it (ros::shutdown()).

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Create a topic publisher (II)

- EXPLANATION OF THE CODE OF THE PUBLISHER (publisher.cpp)
- ros::Publisher pub= n.advertise<tutorials::positionAngle>("position",1000);
  - The node will publish messages of type "tutorials::positionAngle" in topic "/position".
  - The size of the buffer of messages is 1000.
  - **This method advertises the topic to the master** and returns an object "ros::Publisher" that enables:
    - Publishing messages of type "tutorias::positionAngle" in the topic "/position" by the method "publish()".
    - Removing topic from the master (unadvertise) when this object is destroyed.

- ros::Rate loop_rate(1);
  - **It specifies the execution frequency**(in Hz) of the main loop of the node, as parameter for **Rate::sleep()**.

- while (ros::ok())
  - ros::ok() will return true **while the node is active**. It will return false in the next cases:
    - When the node receives the signal SIGINT (Ctrl+C).
    - If other node with the same name substitutes the current node.
    - When "ros::shutdown()" is executed from other point of the application.

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Create a topic publisher (III)

- EXPLANATION OF THE CODE OF THE PUBLISHER (publisher.cpp)
- tutorials::positionAngle msg;
  msg.header.seq++; msg.header.stamp= ros::Time::now();
  msg.angle= angle; msg.x= posX; msg.y= posY;
  - **Create a msg** of type "tutorials::positionAngle" and set its fields.
- pub.publish(msg);
  - **It publishes the message**. It is copied immediately in the topic queue and later it will be sent to subscribers.
- ros::spinOnce();
  - **It invokes all callbacks** of the node that have received messages after the previous execution of this sentence.
  - **It is not required here** since there are no callback functions to wait for (topic subscribers or service servers).
- ROS_INFO("Published message %d: %f, %f, %f",msg.header.seq, msg.x, msg.y, msg.angle);
  - Similar functionality to printf/std::cout in ROS (roscpp) for showing/storing **log messages**
  - Several level of log messages: ROS_DEBUG, ROS_INFO, ROS_WARN, ROS_ERROR, ROS_FATAL.

Create a topic subscriber (I)

- Copy code of subscriber (file subscriber.cpp) in /src folder
- EXPLANATION OF THE CODE OF THE SUBSCRIBER (subscriber.cpp)

```
void positionCallback(const tutorials::positionAngle::ConstPtr& msg)
{
ROS_INFO("I heard message %d: [%f, %f, %f]", msg-> header.seq, msg->x, msg->y,
msg->angle);
}
```

  - **Callback function that is invoked to treat each message received through the topic** "/position".
  - When ros::spin() is used, the callback is invoked as soon as possible when one message is received.
  - When ros::spinOnce() is used, callbacks of messages accumulated after the previous spinOnce are invoked.
  - The message is passed as a "boost::shared_ptr" so that no additional memory management is required.

- `ros::Subscriber sub= n.subscribe("position",1000, positionCallback);`
  - **Subscription to 'position' topic** with an input buffer of 1000 messages.
  - The callback 'positionCallback' will be called to treat each message previously received and stored in the buffer.
  - This method returns a "ros::Subscriber" object. When destroyed, the subscription to the topic will be cancelled.

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Create a topic subscriber (II)

- EXPLANATION OF THE CODE OF THE SUBSCRIBER (subscriber.cpp):
- ros::spin()
  - **It generates an infinite loop** that automatically invoke callbacks when messages are received.
  - This loop ends in the same conditions when ros::ok() returns false.

- COMPILATION OF BOTH NODES (PUBLISHER AND SUBSCRIBER):
- Add these lines to CMakeLists.txt:

```
# Paths to headers for build dependencies (include_directories) and to export to other packages (catkin_package)
include_directories(include ${catkin_INCLUDE_DIRS})
catkin_package(… INCLUDE_DIRS include …)
# For each executable to be generated, indicate cpp files to compile (add_executable)
add_executable(publisher src/publisher.cpp)
# Dependencies on message generation targets of the current package and of other catkin packages
add_dependencies(publisher ${${PROJECT_NAME}_EXPORTED_TARGETS}  ${catkin_EXPORTED_TARGETS})
target_link_libraries(publisher ${catkin_LIBRARIES}) # link with libraries
# The same 3 lines for subscriber...
```

- Compile the workspace that contains the package: catkin_make
- EXECUTION OF BOTH NODES: rosrun tutorials publisher
  rosrun tutorials subscriber

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Create a service

- **Create a service** that performs the sum of two integers in the folder **/srv**:
  roscd tutorials
  mkdir srv
  Copy file AddTwoInts.srv in srv subfolder
- Update **package.xml** to generate messages:
  <build_depend>**message_generation**</build_depend>
  <exec_depend>**message_runtime**</exec_depend> *(v2)*
  <build_export_depend>message_runtime</build_export_depend> *(v2)*
  <depend>**std_msgs**</depend> *DEPENDENCIES ON OTHER MSGS (v2)*
- Update **CmakeLists.txt** to generate messages:
  find_package(catkin REQUIRED COMPONENTS... **message_generation std_msgs**)
  catkin_package(… CATKIN_DEPENDS **message_runtime std_msgs**…)
  add_service_files(FILES **AddTwoInts.srv** …)
  generate_messages(DEPENDENCIES **std_msgs**)
- Verify that ROS finds the new service: rossrv show tutorials/AddTwoInts

Create a service provider and client (I)

- Copy source file (add_two_ints_server.cpp) to the src/ subfolder of our package:
  roscd tutorials
  cp ~/add_two_ints_server.cpp
  ~/catkin_ws/src/tutorials/src/add_two_ints_server.cpp

- EXPLANATION OF THE SERVICE PROVIDER (add_two_ints_server.cpp):
- #include “tutorials/AddTwoInts.h”
  - Include library automatically generated with file srv.
- bool add(tutorials::AddTwoInts::Request &req, tutorials::AddTwoInts::Response &res)

```
{
        res.sum= req.a + req.b;
        …
        return true;
}
```

Callback function that is invoked each time a request for the service “AddTwoInts” is received by the server

Create a service provider and client (II)

- EXPLANATION OF SERVICE PROVIDER (add_two_ints_server.cpp):
- ros::ServiceServer service= n.advertiseService("add_two_ints", add);
  - It advertises the service "add_two_ints" to the master.
  - The second parameter is the callback of the service that is called for receiving the request from the client and generating and sending back the response.
  - When all the copies of the ServiceServer are destroyed, the service is unadvertised and callback stops.

- Copy the code of the service client (file add_two_ints_client.cpp) in /src :

  roscd tutorials

  cp ~/add_two_ints_client.cpp ~/catkin_ws/src/tutorials/src/add_two_ints_client.cpp

- EXPLANATION OF SERVICE CLIENT (add_two_ints_client.cpp):
- ros::ServiceClient client= n.serviceClient <tutorials::AddTwoInts>("add_two_ints");
  - Create a client of service "add_two_ints"
  - The object "ros::ServiceClient" will be used to invoke the service (sending the request message to the server).

INSTITUT PASCAL

sciences de l'ingénierie et des systèmes

Create a service provider and client (III)

- EXPLANATION OF SERVICE CLIENT (add_two_ints_client.cpp):

- tutorials::AddTwoInts srv;
  srv.request.a= atoll(argv[1]);
  srv.request.b= atoll(argv[2]);
  - We create an object from the class of the service: tutorials::AddTwoInts
  - This object will contain two members (request y response). The request should be completed before calling the service while the response will be completed later by the server after the execution of the service.

- if(client.call(srv))
  - This method invokes the service: it sends the request to the service.
  - It is blocking: the client will only continue its execution when the service call is done (with success or not).
  - If the service is executed correctly:
    - The method "call" returns true.
    - The member "srv.response" will continue the response message from the server (provider).

Create a service provider and client (IV)

- COMPILATION OF SOURCE CODE OF BOTH NODES:
- Add the next lines to the CMakeLists.txt of the package:

```
add_executable(add_two_ints_server src/add_two_ints_server.cpp)
add_dependencies(add_two_ints_server ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
add_executable(add_two_ints_client src/add_two_ints_client.cpp)
add_dependencies(add_two_ints_client ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})
```

- Compile the workspace containing the package: catkin_make

- SERVICE EXECUTION:
- rosrun tutorials add_two_ints_server
- rosrun tutorials add_two_ints_client 1 3

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes