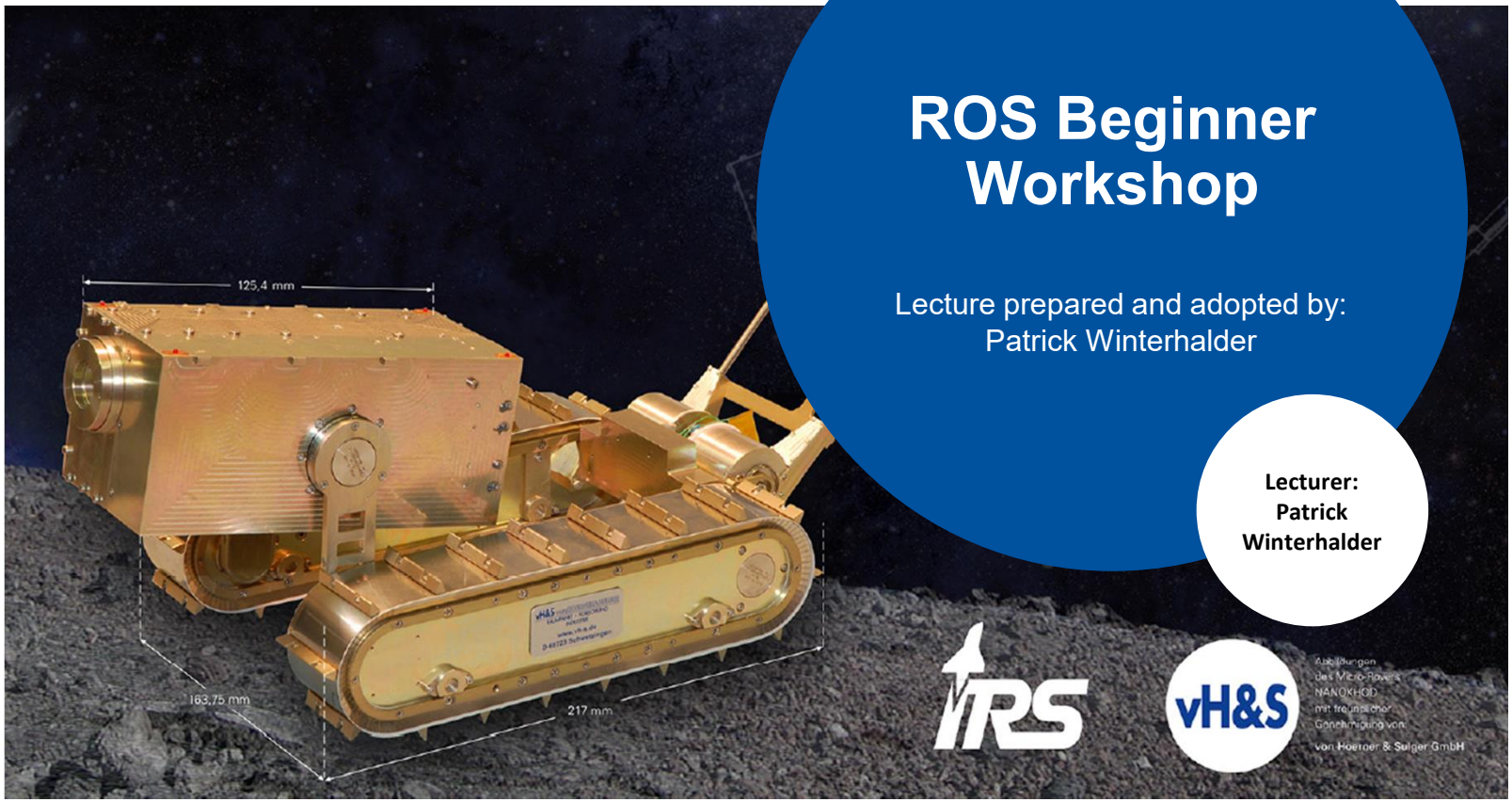
Universität Stuttgart
Institute of Space Systems

ROS Beginner Workshop

Lecture prepared and adopted by:
Patrick Winterhalder



Lecturer:
Patrick Winterhalder

Lesson Outline

ROS Workshop

What is the goal of this workshop?

- Give you a brief overview of the Robot Operating System, especially:
  - File Structure
  - Basic console commands
  - Internodal communication (Topics & Messages)
  - Tools to check the integrity of your system (graph, plot, Rviz)

- Setup easier handling of ROS (→ bashrc)

- Give you Python code template:
  - Create nodes
  - Create publisher / subscriber
  - Use custom messages

ROS Workshop

What you should have done up to now:

- Work through ROS lecture (#4)
- Get comfortable with the Linux console, Python (TutorialsPoint, w3schools)
- Setup Ubuntu,
- Install ROS2 Foxy Fitzroy
- Install Git
- Install Visual Studio Code
- Work through “Beginner: CLI Tools”

ROS Distributions

ROS 1:

Distro	Release date	Poster	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 2:

Distro	Release date	Logo	EOL date
Foxy Fitzroy	June 5th, 2020		May 2023
Eloquent Elusor	Nov 22nd, 2019		Nov 2020
Dashing Diademata	May 31st, 2019		May 2021
Crystal Clemmys	December 14th, 2018		Dec 2019

More Information: https://index.ros.org/doc/ros2/

ROS Workshop

Lesson Outline

ROS Workspace Environment

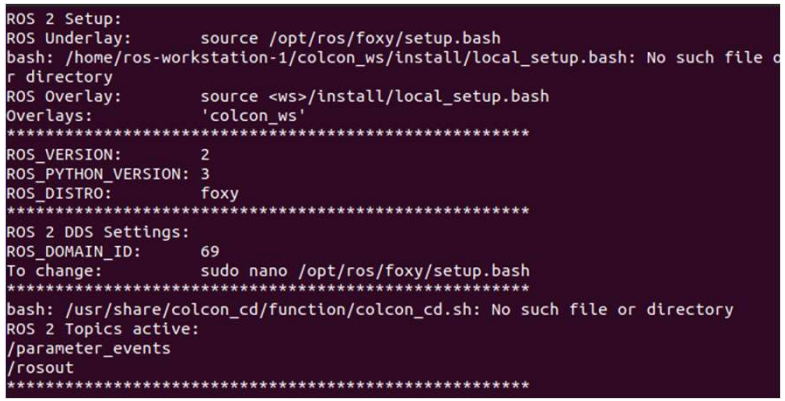
Let‘s code

Improve handling & check correct installation

- Open console (CTRL+ALT+T)
- Open .bashrc:

```
$ sudo nano ~/.bashrc
```

- Add the content of “bashrc_addons.bash” to .bashrc (console paste: CTRL+Lshift+V) (→ https://github.com/patrickw135/pubsub)
- Close and Re-open console, now this should be the output:



- Problem: No workspace exists yet → “`No such file or directory`”
  We will solve this next…

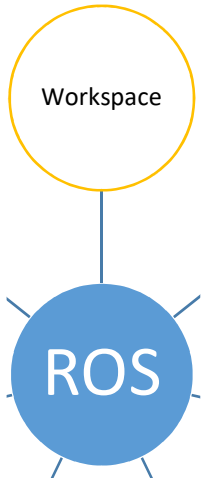
ROS Workspace Environment

Improve handling & check correct installation

- What does .bashrc do?
- .bashrc is run everytime you start a new console (eg. `$ source ~/.bashrc`)
- ROS requires the sourcing of workspaces in every console
  → Use .bashrc to source workspaces automatically, every time you start a new console, eg. `$ source ~/colcon_ws/install/local_setup.bash`
- This improves the handling of ROS2
- .bashrc also shows the currently published topics:

```
ROS 2 Topics active:
/parameter_events
/rosout
**************************************************
```

ROS Workspace Environment



ROS Workspace Environment

Colcon Build System

- colcon is the ROS build tool to generate executables, libraries, interfaces and packages

```
cd ~/colcon_ws                          →Navigate to your colcon workspace
colcon build                            →Build your workspace
source install/local_setup.bash         →Source your overlay after every new build
```

- The colcon workspace usually contains the following folders:

```
ls -l ~/colcon_ws
|
|── build        The build space keeps cache information and other intermediate files.
|
|── install      The install space is where built elements are placed.
|
|
└── src          The source space contains the source code. This is where you work.
                 Here you can clone, create, and edit source code for the packages you want to build.
```

- Running colcon build is necessary after every change to source directory, then source **local_setup.bash** after every colcon build (re-open console → .bashrc start script)

ROS Workspace Environment

Colcon Build System

- If you ever have a problem (e.g. build error, missing libraries, topic errors) delete **`build`** and **`install`** and re-build (`colcon build`)
- **Do no delete `src` !!!** This directory contains your work!

```
ls -l ~/colcon_ws
|
|── build
|
|── install
|
|
└── src
```

Note

Other useful arguments for `colcon build`:

- `--packages-up-to` builds the package you want, plus all its dependencies, but not the whole workspace (saves time)
- `--symlink-install` saves you from having to rebuild every time you tweak python scripts
- `--event-handlers console_direct+` shows console output while building (can otherwise be found in the `log` directory)

- **Let's build our first workspace…**

ROS Workspace Environment

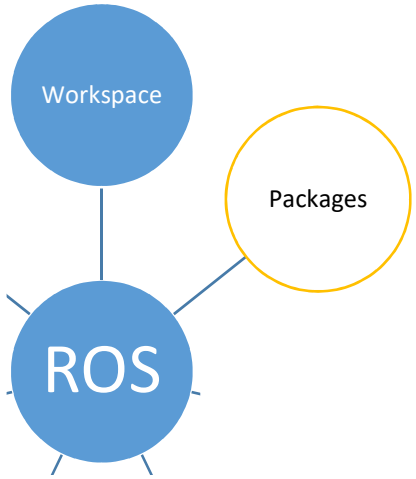
Let's code

Let's create a colcon workspace

1. If not yet created, create new directory:
   ```
   $ mkdir –p ~/colcon_ws/src
   ```
2. Move to src directory:
   ```
   $ cd ~/colcon_ws/src
   ```
3. Clone sample repo from Github:
   ```
   $ git clone https://github.com/ros/ros_tutorials.git -b foxy-devel
   ```
4. Move back to root:
   ```
   $ cd ..
   ```
   OR
   ```
   $ cd ~/colcon_ws
   ```
5. Before building, resolve package dependencies:
   (When cloning repositories, always check dependencies before building)
   ```
   $ rosdep install -i --from-path src --rosdistro foxy –y
   ```
6. Build workspace:
   ```
   $ colcon build
   ```
7. Source underlay:
   ```
   $ source /opt/ros/foxy/setup.bash
   ```
8. Source overlay:
   ```
   $ source ~/colcon_ws/install/local_setup.bash
   ```

   .bashrc
9. Check if pkg available:
   ```
   $ ros2 pkg list | grep "pkg_name"
   ```


Source: Beginner: Client Libraries: Creating a workspace




```
workspace_folder/
    src/
      package_1/
          CMakeLists.txt
          package.xml

      package_2/
          setup.py
          package.xml
          resource/package_2
      ...
      package_n/
          CMakeLists.txt
          package.xml
```

Workspace

ROS Workspace Environment

ROS Packages

- Packages are the software organization unit of ROS code
- Packages are stored in the source folder of the workspace `/colcon_ws/src`
- Each package can contain libraries, executables, scripts, etc.
- Depending on programming language (C++, Python) each package has a different structure and standard files:

CMake | Python

```
my_package/
      CMakeLists.txt
      package.xml
```

CMake | **Python**

```
my_package/
      setup.py
      package.xml
      resource/my_package
```

(Standard package structure after automatic creation using **`$ ros2 pkg create`**)

```
workspace_folder/
    src/
      package_1/
          CMakeLists.txt
          package.xml

      package_2/
          setup.py
          package.xml
          resource/package_2
      ...
      package_n/
          CMakeLists.txt
          package.xml
```

ROS Workspace Environment

ROS Packages

- C++ Package (ament_cmake):

CMake | Python

```
my_package/
      CMakeLists.txt
      package.xml
```

Put interfaces here (eg custom msg, srv, actions)

Put dependencies here (what does this pkg depend on, eg other pkgs)

- Python Package (ament_python):

CMake | Python

```
my_package/
      setup.py
      package.xml
      resource/my_package
```

Define nodes here, eg talker = talker_script.py

Put dependencies here (what does this pkg depend on, eg other pkgs)

Leave this be, required by build system

- How to create a ROS package:

```
$ ros2 pkg create --build-type ament_python
<package_name> {dependencies: --node-name
<my_node_name> }
```

Create a new python package (dependencies, e.g. add a node to package, instead of doing this later by hand)

ROS Workspace Environment

Let‘s code

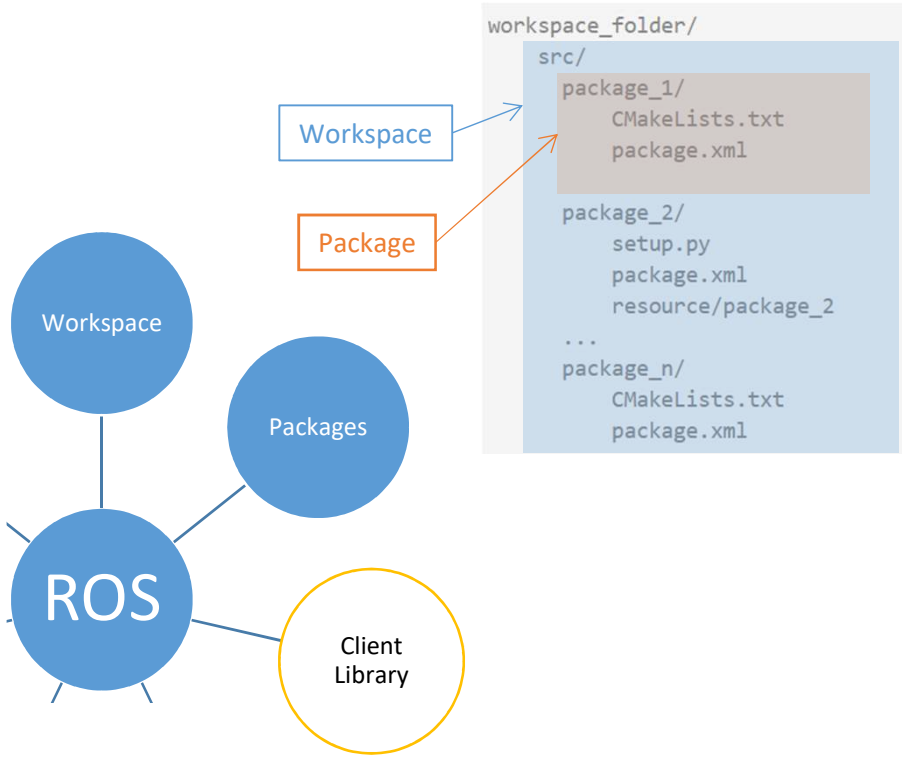
ROS Packages

- Let’s try it: Move to workshop’s source dir: `$ cd ~/colcon_ws/src`
- Create packages for imaginary temperature sensors:
  - C++:
    `$ ros2 pkg create --build-type ament_cmake --node-name temp_publisher temperature_s1`
  - Python:
    `$ ros2 pkg create --build-type ament_python --node-name temp_publisher temperature_s2`

- Take a look inside pkg directories

- Move up to root: `$ cd ..`
- Build workspace: `$ colcon build`
- Source .bashrc: `$ . .bashrc` (identical to starting new console)
- Check list of pkgs: `$ ros2 pkg list | grep “temp”`

- Delete pkgs by deleting the directories **temperature_s1** and **temperature_s2**
- Note:
  Do not copy empty pkg structure to create new pkg!!!
  Setup.py, Package.xml, etc. are created depending on pkg name


workspace_folder/
src/
package_1/
CMakeLists.txt
package.xml
package_2/
setup.py
package.xml
resource/package_2
...
package_n/
CMakeLists.txt
package.xml
Workspace
Package
Workspace
Packages
ROS
Client
Library

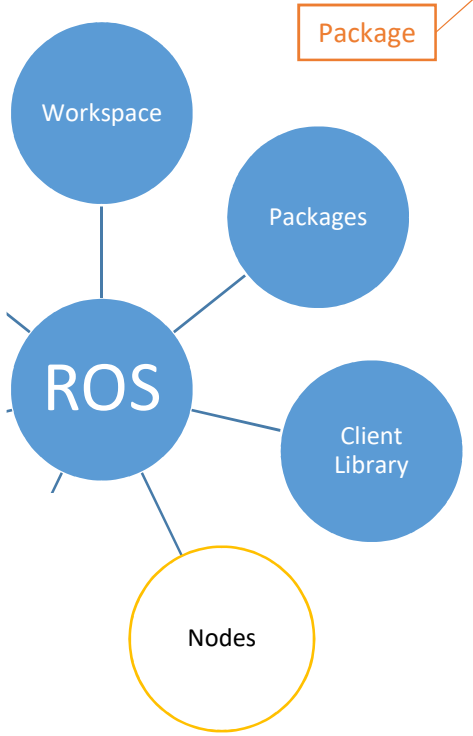
ROS Client Libraries

Python node coding example

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String


def talker():
   pub = rospy.Publisher('chatter', String,
             queue_size=10)
   rospy.init_node('talker', anonymous=True)
   rate = rospy.Rate(10) # 10hz




   while not rospy.is_shutdown():
      hello_str = "hello world %s" % ...
            rospy.get_time()
      rospy.loginfo(hello_str)
      pub.publish(hello_str)
      rate.sleep()


if __name__ == '__main__':
   try:
      talker()
   except rospy.ROSInterruptException:
   pass
```

C++ node coding example (ROS1)

```
#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;
}
```




```
workspace_folder/
    src/
        package_1/
            CMakeLists.txt
            package.xml
        package_2/
            setup.py
            package.xml
            resource/package_2
        ...
        package_n/
            CMakeLists.txt
            package.xml
```

Workspace

Package

Python	**ROS1: rospy** **ROS2: rclpy**
C++	**ROS1: ros.h** **ROS2: rclcpp**

ROS Nodes

Nodes are

- single purposed programs (e.g. sensor/actuator driver)
- create the interface between your custom code and ROS infrastructure
- started by scripts from inside a package (→ always belong to package, that's how you call them: `$ ros2 run <pkg_name> <node_name>`)
- written using ROS client library (C++: rclcpp, Python: rclpy)

Creating a node is required in order to use ROS infrastructure (Topics/Services/Actions), only then nodes can

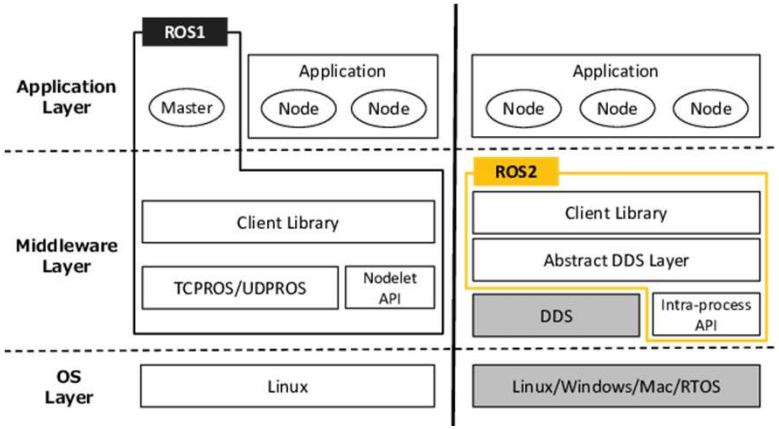
- publish or subscribe to Topics
- provide or use Service or Action
- If no node is created code is only a python script but has no ROS functions

- Nodes are not scripts / files inside package, Nodes are created by a ROS function when called inside a script and destroyed when ending the script

ROS Nodes

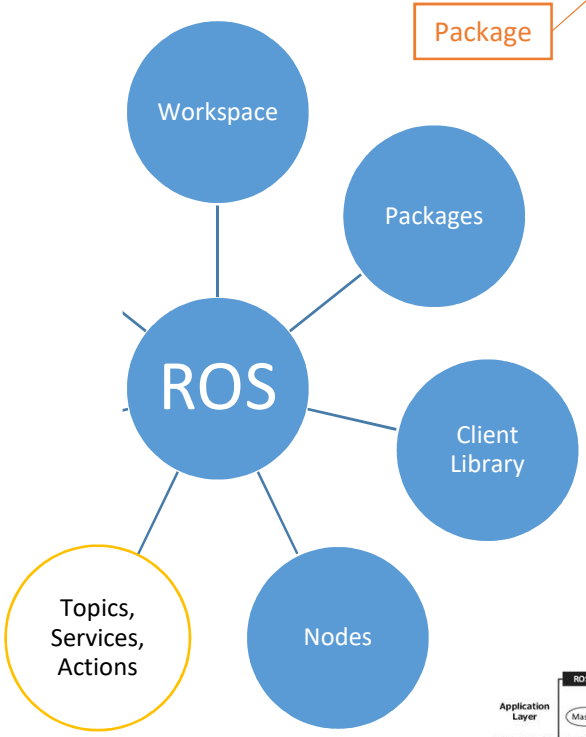
Nodes in ROS2 no longer need a Master Node

- Every Node works for itself
- No central computer necessary → subsystem failure



Console commands:

$ ros2 node list	Prints list of all running nodes
$ ros2 node info {node_name}	Gives information about specific node

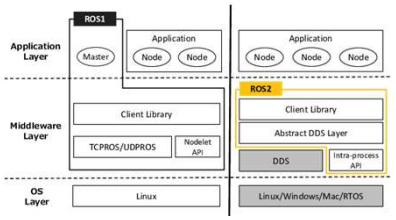



```
workspace_folder/
    src/
        package_1/
            CMakeLists.txt
            package.xml
        package_2/
            setup.py
            package.xml
            resource/package_2
        ...
        package_n/
            CMakeLists.txt
            package.xml
```

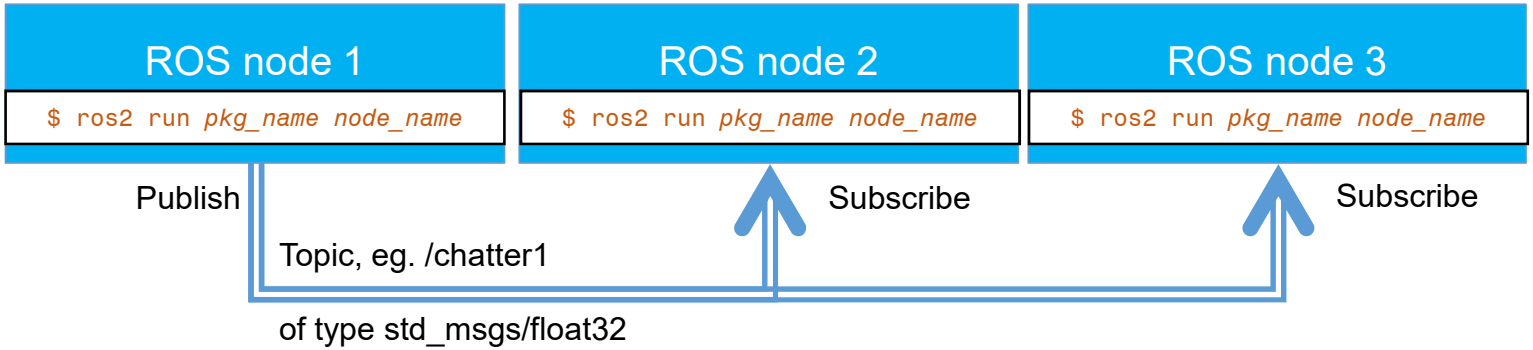
Workspace

Package

Python	**ROS1: rospy** **ROS2: rclpy**
C++	**ROS1: ros.h** **ROS2: rclcpp**



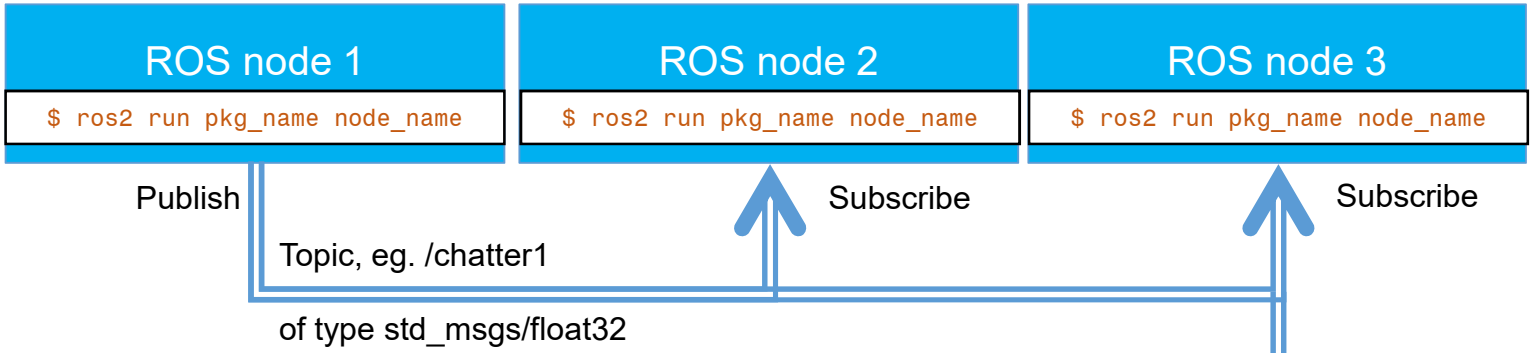
ROS Topics and Messages



- ROS topics
  - Nodes can communicate via topics (streams of messages)
  - Nodes can either publish or subscribe to a topic “X”
    (usually 1 publisher and n subscribers per topic “X”)
  - Topic ≈ communication channel (eg. /chatter1)
  - Content of Topic/Channel = Message (eg. std_msgs/float32)
  - Most important ROS2 topic commands:

`$ ros2 topic list`	Print list of all active topics (already included in .bashrc addons)
`$ ros2 topic list –t`	Print list of active topic including the message type
`$ ros2 topic echo topic_name`	Creates a subscriber node and prints the topic‘s contents (use this to check if your published data is correct)

ROS Topics and Messages



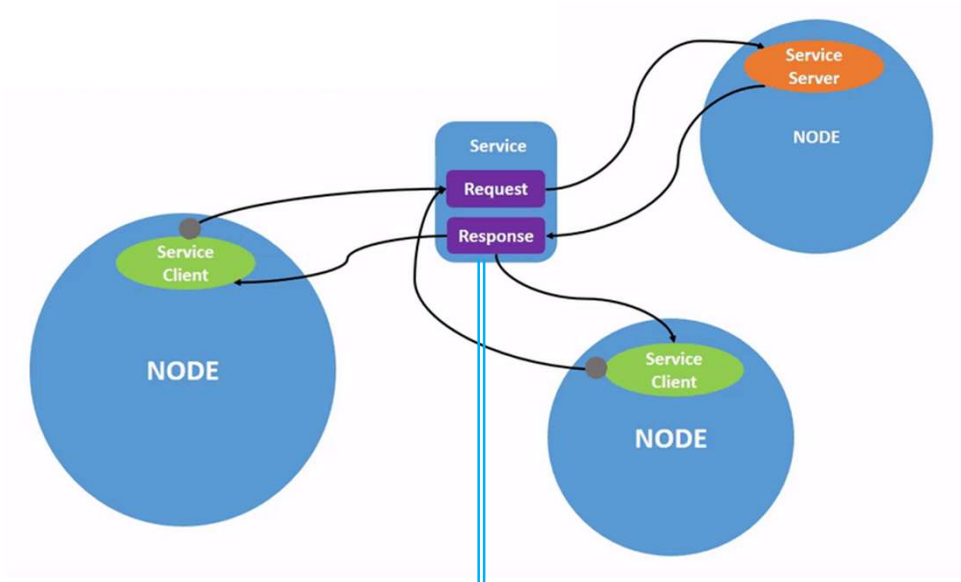
- ROS messages
  - Message define content of Topic (*.msg file determines data type, eg. integer)
  - ROS has pre-built standard message types (std_msgs) and more complex sensor_msgs
  - ROS allows for custom message types (*.msg files); Can be data structures of multiple integers, floats, strings, bools, etc. all transmitted simultaneously in one topic
  - More topic commands:

`$ ros2 topic type topic_name`	Display the type of a topic
`$ ros2 interface show message_name`	Display the message type
`$ ros2 topic pub topic_name message_type {YAML_dict} args`	Publish a topic of a specific message type, the content is given in YAML dictionary syntax
`$ ros2 topic hz topic_name`	Display the publishing rate of a topic

```
sensor_msgs/servo_pos
# Header
time stamp

# Servo Values
float64 angle
uint32 motor_number
```

ROS Services

- ROS services
  - Services are another way that nodes can communicate
  - Nodes can provide or use a service (Server/Client model)
  - Services allow the client node to send a request and receive a response
  - Made up of a **pair of messages (request and response)**, defined by *.srv file, request and response separated by " - - - "



Service type (*.srv)

`$ ros2 service list {-t}`	List active services (with type)
`$ ros2 servise type srv_name`	Display the type of a service
`$ ros2 interface show srv_name`	Display the definition of the service
`$ ros2 service call srv_name srv_type args`	Call a service of a specific type with the provided arguments
`$ ros2 service find srv_type`	Find services by their service type

nav_srvs/GetPlan.srv

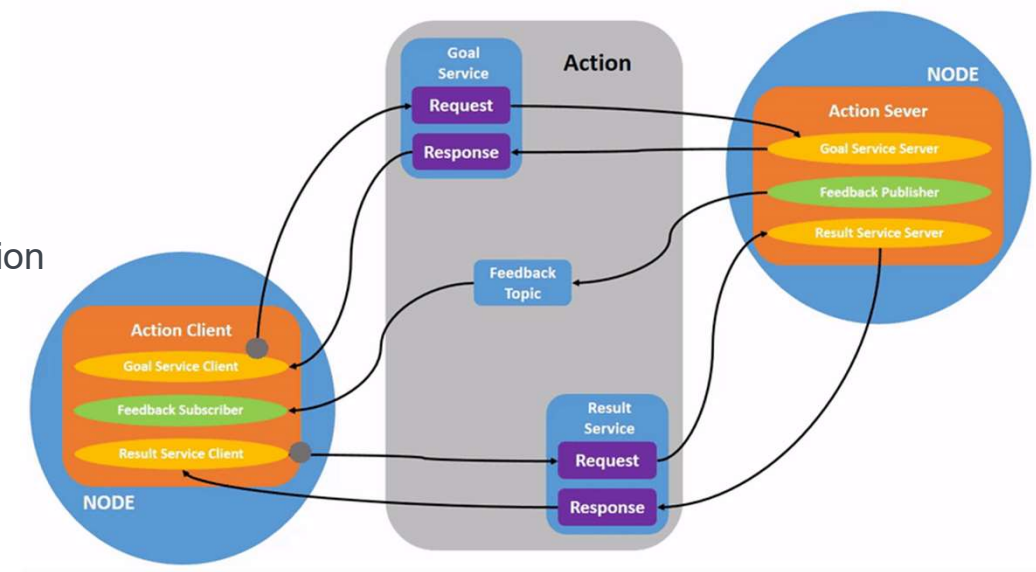
(request)

geometry_msgs/PoseStamped start
geometry_msgs/PoseStamped goal
float32 tolerance
---
nav_msgs/Path plan *(response)*

ROS Actions

- ROS actions
  - Third type of communication between nodes
  - Defined in *.action file by 3 messages, again separated by “ - - - ”
  - Similar to service calls, but Actions provide possibility to:
    - Cancel the task (preempt)
    - Receive feedback on the progress
  - Basic commands:

`$ ros2 action list {-t}`	List active actions (and types)
`$ ros2 action info act_name`	Display info about an actions
`$ ros2 interface show act_type`	Display the type of an actions



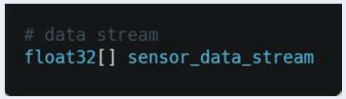
Action Type

FollowPlan.action

navigation_msgs/Path path *(Goal)*
---
bool success *(Result)*
---
float32 remaining_distance *(Feedback)*
float32 initial_distance

ROS Topics, Services & Actions

Comparison of Topics, Services & Actions

	Topics	Services	Actions
Description	Continuous data streams	Blocking of script during the processing of a request	Non-blocking, Interruptible, Goal oriented tasks
Application	One-way continuous data flow	Remote procedure calls that terminate quickly	Discrete behavior that moves a robot or that runs for a longer time and provides feedback
Examples	Sensor data, robot state	Trigger change, request states or specific data, compute quantity	Navigation, perception, grasping, motion execution
			

Lesson Outline

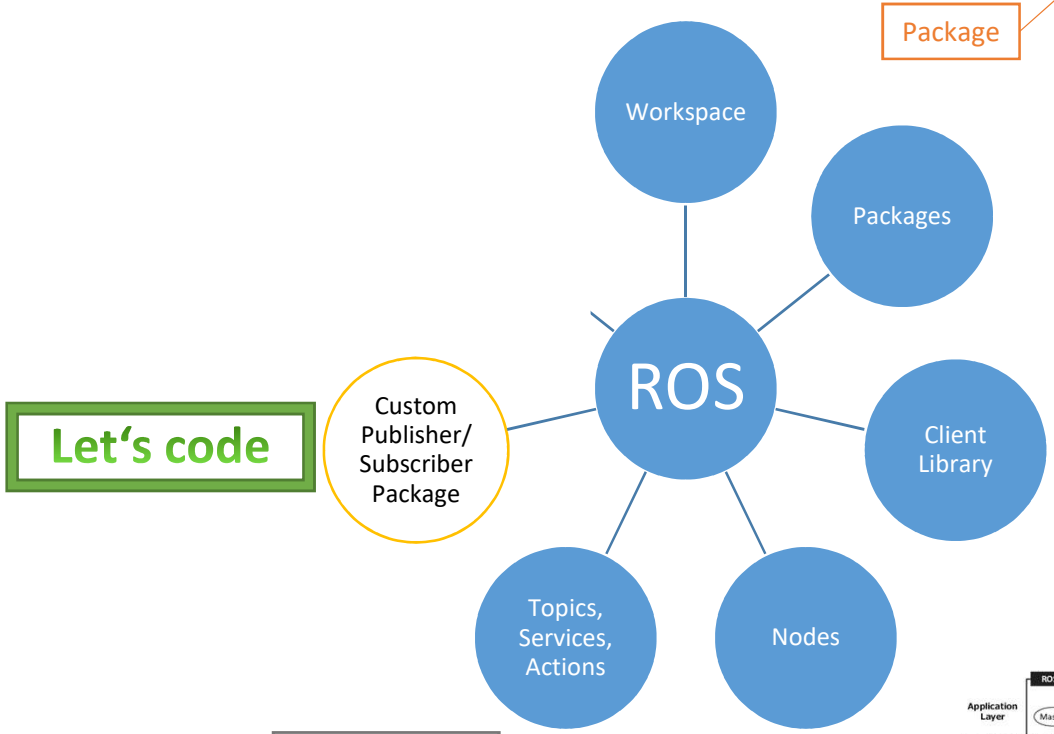
1. Introduction 1
2. Recap of “CLI Tools” Tutorial 2
3. Publisher and Subscriber (“Beginner: Client Library” Tutorial) 3
4. Useful ROS Tools and Features 4
5. System Requirements 5

ROS Publisher & Subscriber

```
workspace_folder/
    src/
        package_1/
            CMakeLists.txt
            package.xml
        package_2/
            setup.py
            package.xml
            resource/package_2
        ...
        package_n/
            CMakeLists.txt
            package.xml
```

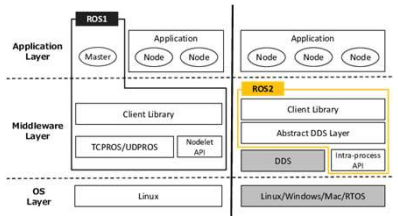
Workspace

Package



Python	**ROS1: rospy** **ROS2: rclpy**
C++	**ROS1: ros.h** **ROS2: rclcpp**

```
# Header
time stamp

# Servo Values
float64 angle
uint32
motor_number
```



ROS Publisher & Subscriber

Let's code

Creating a Package and Node

1. Create workspace directory (if not yet created)
2. Create package:

```
$ ros2 pkg create --build-type ament_python --node-name publisher pubsub
```

ros2→package→create | Choose Python as build system parameter | Node Name | Package Name

3. Work on package (create msg files, write Python script, define nodes) (Instructions) **or clone** github repo:

```
$ cd ~/<your_workspace_path>
```
(root of workspace)

```
$ git init
$ git remote add origin https://github.com/patrickw135/pubsub.git
$ git fetch --all
$ git reset –hard origin/master
```

4. Build package → `$ colcon build`

Note: The workspace directory becomes workspace when `colcon build` is executed and sourced, before that it's just like any other directory.

ROS Publisher & Subscriber

Let's code

Creating a Package and Node

5. Close all consoles
6. Open two new consoles → .bashrc will sources ROS setup files
7. Check availability of pkgs: `$ ros2 pkg list | grep "pubsub"`
8. Run talker: `$ ros2 run pubsub talker`

```
ros-workstation-1@ros-ws-1:~$ ros2 run pubsub talker
        - /chatter1

        - /chatter2
```

9. Run listener: `$ ros2 run pubsub listener` (→ receives /chatter1 & /chatter2)

```
[SUBSCRIBER] pubsub_listener_2 received topic:  //chatter2
Received:       0.00,   0.22,
[SUBSCRIBER] pubsub_listener_1 received topic:  //chatter1
Received:       24.50,  25.50,  26.50,  1011.50,        1012.55,        1010.11,        0.00,   0.00,   0.00,
[SUBSCRIBER] pubsub_listener_2 received topic:  //chatter2
Received:       0.00,   0.22,
[SUBSCRIBER] pubsub_listener_1 received topic:  //chatter1
Received:       24.50,  25.50,  26.50,  1011.50,        1012.55,        1010.11,        0.00,   0.00,   0.00,
[SUBSCRIBER] pubsub_listener_2 received topic:  //chatter2
Received:       0.00,   0.22,
[SUBSCRIBER] pubsub_listener_1 received topic:  //chatter1
Received:       24.50,  25.50,  26.50,  1011.50,        1012.55,        1010.11,        0.00,   0.00,   0.00,
[SUBSCRIBER] pubsub_listener_2 received topic:  //chatter2
```

10. Open another console → active topics are printed here

```
ROS 2 Topics active:
/chatter1
/chatter2
/parameter_events
/rosout
*******************************************************
```

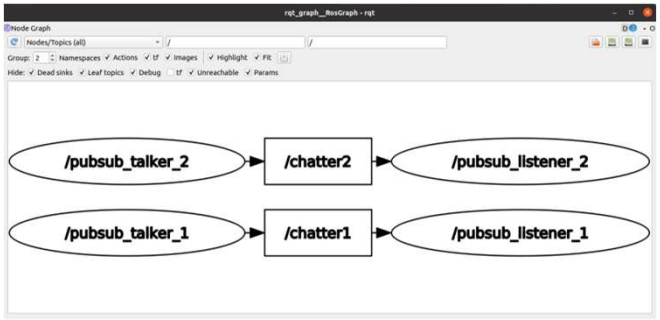
ROS Publisher & Subscriber

Let‘s code

Creating a Package and Node

10. Look at how the nodes are linked

    New console, open graph: `$ rqt_graph`



11. Listen to topic over ros2 commands

    New console: `$ ros2 topic echo /chatter1`

    New console: `$ ros2 topic echo /chatter2`

When building your own package you can use these packages as “templates” but DO NOT copy! This will not work, instead use the ROS API to create new packages (`$ ros2 pkg create`) and look at how the files in **/pubsub** and **/pubsub_msg** are configured.

ROS Publisher & Subscriber

Creating a Package and Node

When to choose custom messages:

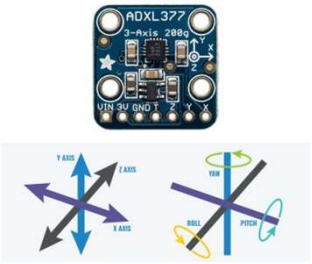
- Better organization of information
- Packaging of data flow, keeping all associated data together

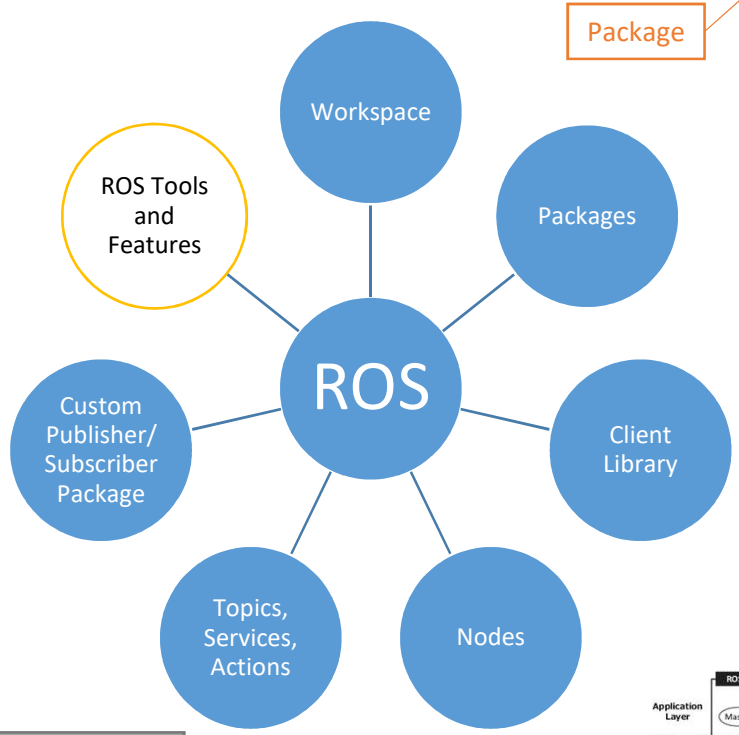
Eg.:

Sensor	Data transmitted	Best message type
Inertial measurement unit	6+ values: vectors, matrices	Custom message useful, or use premade message (other premade sensor messages)
Optical distance sensor	1 value: floating point number	No sense in using custom message, use std_msgs/float32

Lesson Outline

ROS Tools and Features



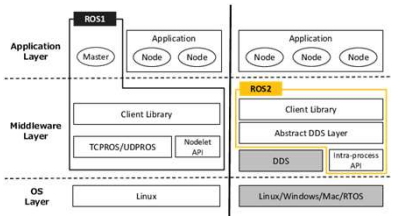



```
workspace_folder/
    src/
        package_1/
            CMakeLists.txt
            package.xml
        package_2/
            setup.py
            package.xml
            resource/package_2
        ...
        package_n/
            CMakeLists.txt
            package.xml
```

Workspace

Package

Python	**ROS1: rospy** **ROS2: rclpy**
C++	**ROS1: ros.h** **ROS2: rclcpp**

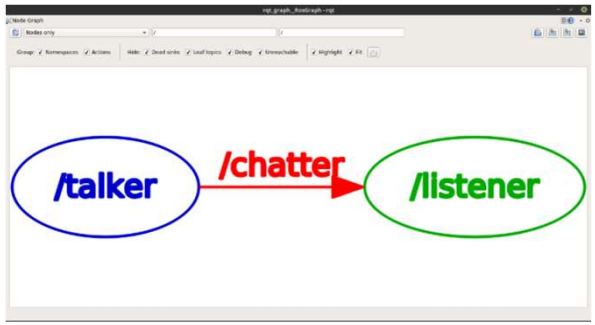



```
# Header
time stamp

# Servo Values
float64 angle
uint32
motor_number
```

Robot Operating System

ROS Standard Tools (Graph, Plot, Rviz)

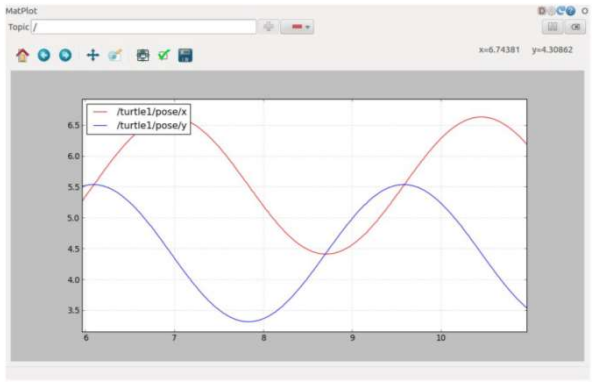
- Graph:



Visual representation of node and message structure currently running on the network, e.g. data paths between nodes.

Start Graph: `$ rqt_graph`

- Plot



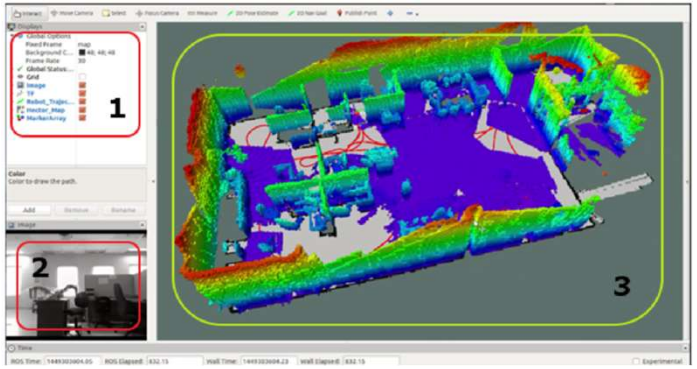
Visual representation of the data transferred using Topic, Services or Actions, e.g. sensor data exchanged in a topic.

Start Plot: `$ rqt`
Plugins → Visualization → Plot

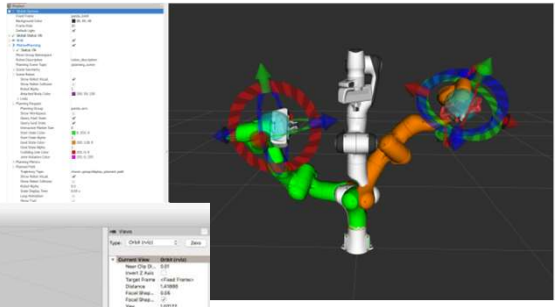
Robot Operating System – Core Elements

ROS Standard Tools (Graph, Plot, Rviz)

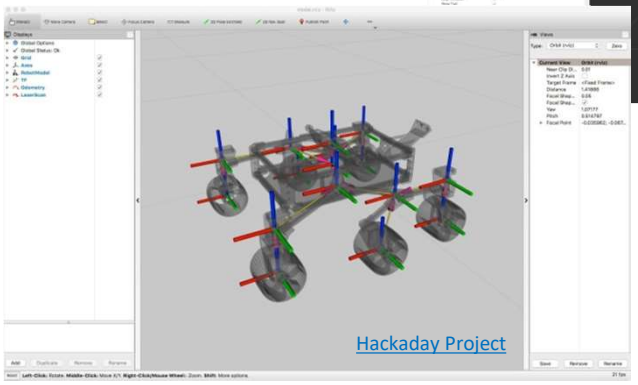
- Rviz:
  - 3D visualization tool for ROS
  - Subscribes to topics and visualizes the transferred data
  - Different camera views (orthographic, topdown, etc.)
  - Interactive tools to publish user information
  - Save and load current setup as RViz configuration
  - Extensible with plugins
  - Start Rviz: `$ rviz2`



A ROS-based human-robot interaction for indoor exploration and mapping



MoveIt! Instruction Material



Hackaday Project

ROS Tools and Features

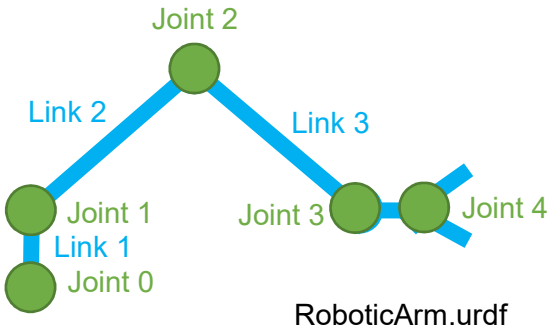
ROS Network Distribution

- ROS is designed with distributed computing in mind
- Allows you to spread computational power across multiple machines
- Prerequisite:
  - ROS1:
    - All nodes must use the same master (→ Variable: ROS_MASTER_URI)
    - Bi-directional connectivity between all machines, on all ports (→ ROS Network setup)
  - ROS2:
    - Connect all machines on the same network
    - Choose the same DDS Domain ID (.bashrc: `export ROS_DOMAIN_ID=666`)
    - Every node acts like a ROS1 Master, therefore no further configuration required

ROS Tools and Features

Unified Robot Description Format (URDF)

- XML format for representing a robot model
- Description consists of a set of link elements and a set of joint elements
- Joints connect the links together



RoboticArm.urdf

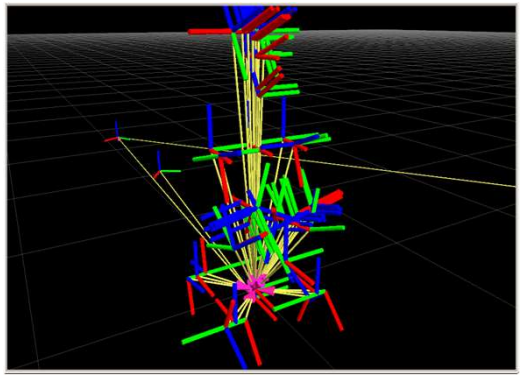
```
<robot name="robotic arm">
  <link> ... </link>
  <link> ... </link>
  <link> ... </link>

  <joint> .... </joint>
  <joint> .... </joint>
  <joint> .... </joint>
</robot>
```

```
<link name="link_name">
  <visual>
    <geometry>
      <mesh filename="ellbow.dae"/>
    </geometry>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="10"/>
    <inertia ixx="0.4" ixy="0.0" …/>
  </inertial>
</link>
```

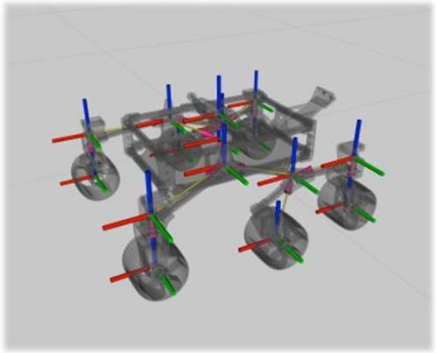
```
<joint name="joint_name" type="revolute">
  <axis xyz="0 0 1"/>
  <limit effort="1000.0" upper="0.548" … />
  <origin rpy="0 0 0" xyz="0.2 0.01 0"/>
  <parent link="parent_link_name"/>
  <child link="child_link_name"/>
</joint>
```

ROS Tools and Features

- ROS Launch
  - a tool for launching multiple scripts and setting parameters:
    `$ ros2 launch <pkg_name> <launch_file_name>`
  - Launch files are written in Python (*.py) (ROS1: XML, *.launch)
- ROS Bag
  - File format for storing timestamped message data → e.g. sensor feedback data
  - Record and log topics for visualization, analysis and replay (eg. use to test the function of new packages)
- ROS TF Transformation System
  - Tool for keeping track of coordinate frames over time
  - Lets the user transform points, vectors, etc. between coordinate frames at desired time
  - Can be visualized in RViz



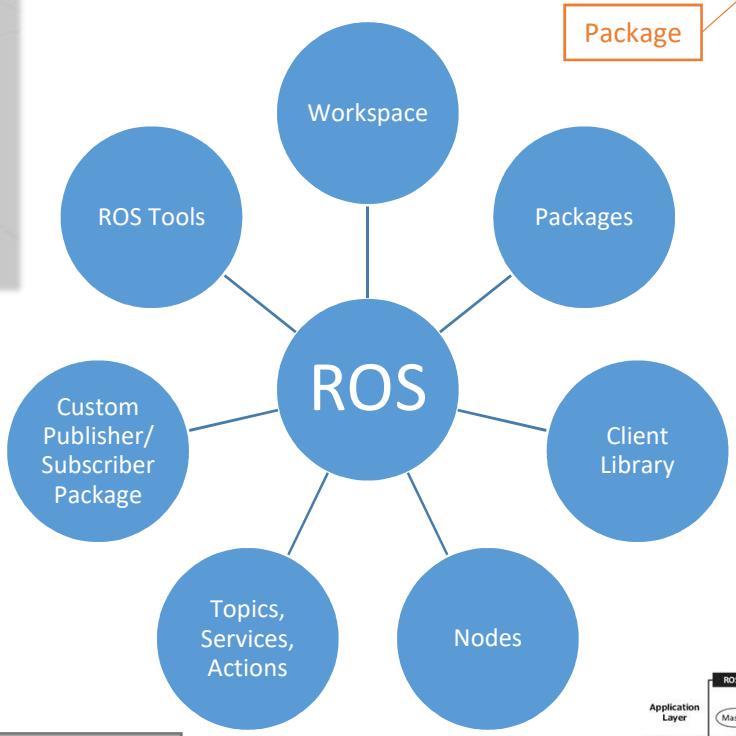
ROS Workshop

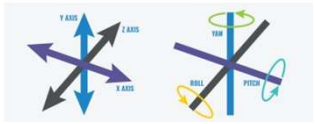


```
workspace_folder/
    src/
        package_1/
            CMakeLists.txt
            package.xml
        package_2/
            setup.py
            package.xml
            resource/package_2
        ...
        package_n/
            CMakeLists.txt
            package.xml
```

Workspace

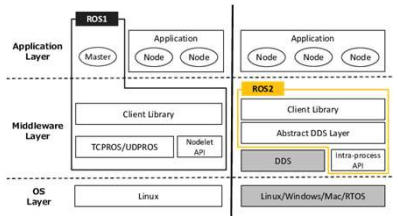
Package





Python	**ROS1: rospy** **ROS2: rclpy**
C++	**ROS1: ros.h** **ROS2: rclcpp**

```
# Header
time stamp

# Servo Values
float64 angle
uint32
motor_number
```



Lesson Outline

1. Introduction 1
2. Recap of “CLI Tools” Tutorial 2
3. Publisher and Subscriber (“Beginner: Client Library” Tutorial) 3
4. Useful ROS Tools and Features 4
5. System Requirements 5

System Requirements

Hard Requirements:

- Control of your system from outside world must happen through ROS interfaces (Topics, Services, Actions)
- ROS system must run independently and must not depend on other hardware
- All system states must be published over ROS Topics
- Besides Service and Action Feedback all sensor and actuator data must be readable over Topics in order to be able to log everything (Services & Actions cannot be logged)
- All Data must be logged in order to be replayed and evaluated afterwards (Logging → Bag Files)

Nice-To-Haves:

- Visualization of your sensor data using Rviz (depending on the type of measured parameters, e.g. IMU)
- Control and visualization of payload using a GUI (e.g. rqt, ros_gui)
- Code should be organized and easily readable, e.g. creating library files and importing functions is very simple in Python
- Keep it minimalistic but reliable, simplicity is key

Lesson Outline

Robot Operating System

Group programming tips:

- Code on Desktop/Laptop PC (GUI)

- Use Git:
  - Exchange Code
  - Local and Cloud Storage
  - Version Control

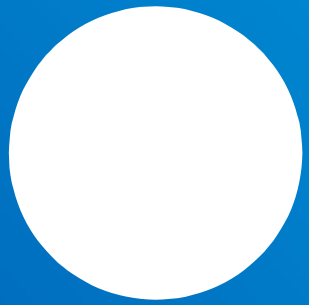
- Git can also be used to transfer ROS packages from your PC to the Raspberry Pi:
  - Ubuntu Server only offers CLI (Console Line Interface) & moving files is complicated
  - However downloading from Github repository is simple
  - Steps:
    - “Commit” & “Push” from PC to Github
    - Use CLI on Rpi to “Fetch”/”Pull”/”Clone” from Github to Rpi
    - Use CLI to build workspace on Rpi over CLI

References

- Github Pubsub Example: https://github.com/patrickw135/pubsub
- ROS2 CLI Cheatsheets: https://github.com/ubuntu-robotics/ros2_cheats_sheet
- ROS1 Wiki (still useful, mindset and package API still the same): http://wiki.ros.org/
- Install Foxy Fitzroy: https://index.ros.org/doc/ros2/Installation/Foxy/Linux-Install-Debians/
- Tutorials: https://index.ros.org/doc/ros2/Tutorials/

University of Stuttgart
Institute of Space Systems

Thank you!



Patrick Winterhalder

e-mail winterhalderp@irs.uni-stuttgart.de
Phone +49 (0) 711 685 69655
www. irs.uni-stuttgart.de

University of Stuttgart
Institute of Space Systems
Pfaffenwaldring 29, 70569 Stuttgart

Universität Stuttgart
Institute of Space Systems

Development of an exploration rover system

12.11.2020