ETH zürich (ROS Course)

① Lecture 1

ROS = Robot Operating System

→ It is not an Operating System

It is middle where that sits between operating system and actual program that you write.

Plumbing
- → Process management
- → Inter-Process communication
- → Device drivers

Tools
- → Simulation
- → Visulization
- → Graphical user Interface.
- → Data logging

Capbilities
- → Control
- → Planning
- → Perception
- → Mapping
- → Manipulation

Ecosystem
- → Package Organization
- → Software distribution
- → Documentation
- → Tutorials

★ Ros Philosphy

Peer to Peer

⇒ Individual programs communicates over defined API (ROS messages, Services etc..)

Distributed

→ Programs can be run on multiple computers and communicate over the network.

Multi-lingual

→ ROS modules can be written in any language for which a client library exists (C++, Python, Java etc).

Light-Weight

Free and Open-Source

★ ROS Masters

→ Manages the communication between nodes.
→ Every node registers at startup with the master.

`roscore` ⇒ To start ROS Master

★ ROS Node

→ Single-purpose
→ executable program
→ Individually compiled, executed and managed.
→ Organized in package.

`rosrun` package_name node_name
↳ To run the node

`rosnode list` ⟶ To list all the active nodes

`rosnode info` node_name → Retrieve information about a node.

* ROS Topic

⇒ 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.

`ros topic list` ⇒ List active topics

`ros topic echo` /topic ⇒ Subscribe & print the Contents of a topic.

`ros topic info` /topic ⇒ Shows information about the topic

* ROS Messages

⇒ Data structure defining the type of a topic.

⇒ Defined in *.msg file

`ros topic type` /topic
→ To See the type of a topic.

`ros topic pub` /topic type args
→ Publish message to a topic

⇒ Ros Messages can be nested

`ros topic hz` /topic
→ frequency of Publish in Hz

Catkin workspace

- src — The source space contains the source code.
- build — where CMake is invoked to build the packages in the source space. Cache information and other intermediate files are kept here.
- devel — The development (devel) space is where built targets are placed.

catkin clean ⇒ Cleans the entire build and devel space without affecting the src folder

Catkin config → Catkin workspace setup can be checked here.

ROS Launch

⇒ launch is a tool for launching multiple nodes (as well as setting parameters)

⇒ Are written in XML as *.launch file.

⇒ If not yet running, launch automatically starts a roscore.

roslaunch file_name.launch

→ {If you are already in the folder}

roslaunch Package_name file_name.launch

→ {Start launch file from a package}

```
<launch>
    <node name="Given name" Pkg="Package name" type="original node name"
          Output='screen"/>
<launch>
```

⇒ launch file can be nested.

* <arg> tag

<arg name="arg_name" default="default_Value"/>

$(arg arg_name) → Can be used in launch file

⇒ Launching launch file with argument

roslaunch Launch_file.launch arg_name:=Value

⇒ Including other launch file

<include file="$(find Package_name)/relative path.launch"/>

* Gazebo Simulator

→ Simulate 3d rigid-body dynamics
→ Simulate a variety of sensors including noise.
→ 3d visualization and user interaction.
→ Include a database of many robots and environments
→ Provides a ROS interface
→ Extensible with plugins

⇒ To run Gazebo with ROS

rosrun gazebo_ros gazebo

① Lecture 2

★ ROS Package

⇒ ROS software is organized into packages, which can contain source code, launch files, configuration files, messages, definations, data and documentation.

⇒ A package that builds up on/requires other packages, declares these as dependencies.

⇒ To create new package

```
catkin_create_pkg packagename [dependencies]
```

→ In src folder

⇒ Separate the message definition Package from other package.

★ Package.xml

⇒ The package.xml file defines the properties of the package.

- Package name
- Version number
- Authors
- Dependencies on other pakage
- ...

* CMakeList.txt

↳ Input to CMake build System.

1. Requires CMake version

```
cmake_minimum_required
```

2. Package Name

```
Project()
```

3. Find other CMake/Catkin packages needed for build.

```
find_package()
```

4. Message/Service/Action Generators

```
add_message_files()
add_service_files()
add_action_files()
```

5. Invoke message/service/action generation

```
generate_message()
```

6. Specify package build info export.

```
catkin_package()
```

7. Libraries/Executables to build

```
add_library()
add_executable()
target_link_libraries()
```

8. Tests to build

```
catkin_add_gtest()
```

9. Install rules

```
install()
```

★ ROS C++ Client Library (roscpp)

```
#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;
}
```

★ Node Handle

⇒ There are four main types of Node handles :-

1. Default (Public) node handle:

```
nh_ = ros:: NodeHandle();
```

2. Private node handle:

```
nh_private_ = ros::NodeHandle("~");
```

3. Namespaced node handle:

```
nh_eth_ = ros::NodeHandle("eth");
```

* Logging

⇒ Mechanism for logging human readable text from nodes in the console and to log file.

⇒ Automatic logging to console, log file, and /rosout topic.

⇒ Different Severity levels:

Debug → Info → Warn → Error → Fatal.

⇒ 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..

:-

* ROS Parameter Server

⇒ Nodes use the Parameter server to store and retrive parameters at runtime.

⇒ Best used for static data such as configuration Parameter

⇒ Parameters can be defined in launch files or Separate YAML file.

{Example

Config.yaml

```
Camera:
  left:
    name: left_Camera
    exposure: 1
  right:
    name: right_Camera
    exposure: 1.1
```

{launching it through launch file}

```
<node name="Name" PKg="Package" type="node_type">
    <rosparam command="load"
    file="$(find package)/config/config.yaml"/>
</node>
```

C++

⇒ To

No

★ RV

⇒ 3D

⇒ Su

m

⇒ Di

,

⇒ In

⇒ Se

⇒ Ex

⇒ Ru

C++ API for Parameter Server

⇒ To get a parameter in C++:

NodeHandle.getParam(Parameter_name, Variable)

★ RViz

⇒ 3D Visualization tool for ROS

⇒ Subscribes to topics and Visualizes the message contents.

⇒ Different camera Views (orthographic, top, down etc)

⇒ Interactive tools to publish user information

⇒ Save and load setup as RViz configuration

⇒ Extensible with Plugins.

⇒ Run RViz with

rosrun rviz rviz

③ Lecture 3

★ TF Transformation System

⇒ Tool for Keeping track of coordinate frames over time.

⇒ Maintains relationship between coordinate frames in a tree structure buffered in time.

⇒ Lets the user transform point, vectors, etc. between coordinate frames at desired time.

⇒ Implemented as publisher/subscriber model on the topic /tf and /tf_static

Tools

(A) Command line

`rosrun tf tf_monitor`

↳ Prints information about the current transform tree.

`rosrun tf tf_echo` source_frame target_frame

↳ Prints information about the transformation between two frame.

(B) View Frames

`rosrun tf view_frames`

↳ Creates a visual graph (PDF) of the transform tree.

(C) RViz

3D Visulization of the transformation

* RQT User Interface

- → Custom interface can be setup
- → Lots of existing plugins exists
- → Simple to write own plugins

⇒ Run RQT with

`rosrun rqt_gui rqt_gui`

```
# rqt_image_view
# rqt_multiplot
# rqt_graph
# rqt_Consol
# rqt_logger_level
```

* Robot Models

(Unified Robot Description Format)
(URDF)

⇒ Defines an XML format for representing a robot model.

⇒ Kinematic & dynamic description
⇒ Visual representation
⇒ Collision model

⇒ URDF generation can be scripted with XACRO.

link Joint

Example : ~~robot.urdf~~ link

```
<link name="link_name">
  <Visual>
    <Geometry>
      <mesh filename="mesh.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>
```

Example: Joint

```
<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>
```

⇒ The robot description (URDF) is stored on the parameter server (typically) under /robot description.

⇒ You can visualize the robot model in Rviz with the RobotModel plugin.

* Simulation Description (Simulation Description Format) (SDF)

→ Standard format for Gazebo

⇒ Defines an XML format to describe

- # Environment (lighting, gravity etc)
- # Objects (static & dynamic)
- # Sensor
- # Robot.

⇒ Gazebo Converts a URDF to SDF automatically.

---

ros::Duration

```
ros::Duration duration(0.5);
```

ros::Rate

```
ros::Rate rate(10);
```

* ROS Bags (*.bag)

⇒ Bag is a format used to store message data

⇒ Suited for logging and recording dataset for later visualization and analysis.

④ Lecture 4

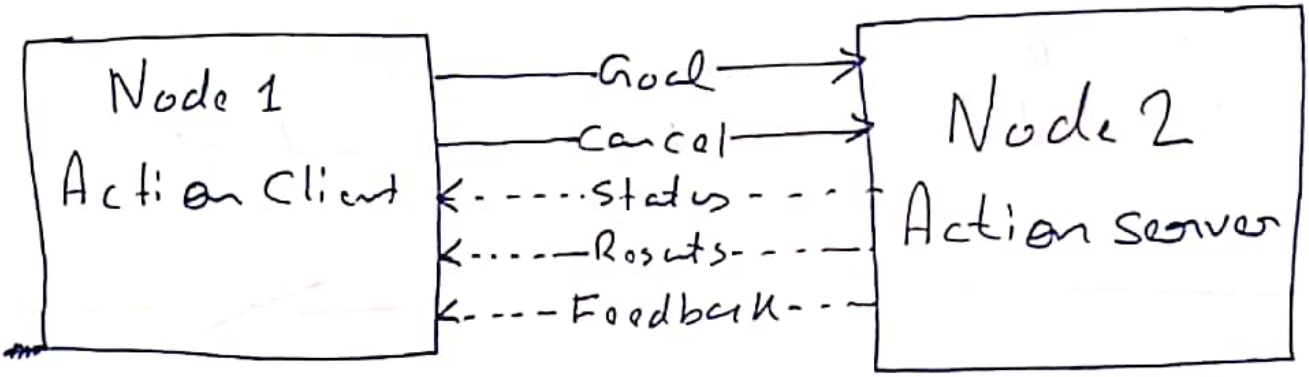
★ ROS Services

⇒ Request/response communication between nodes is realized with services.

⇒ Services are defined in *.srv file.

★ ROS Actions (actionlib)

↳ Very similar to services but meant for request that take a longer time.



*.action file

★ ROS time

↳ Normally, ROS uses the PC's system clock as time source (wall time)

⇒ To take advantage of the simulated time, you should always use the ROS Time APIs:

⊕ ros::Time

```
ros::Time begin = ros::Time::now()
double secs = begin.toSec()
```

* Debugging Strategies

⇒ Compile and run code often to catch bugs early.

⇒ Understand compilation and runtime error messages.

⇒ Use analysis tools to check data flow.

⇒ Visualize and plot data

⇒ Divide program into smaller steps and check intermediate results.

⇒ Make your code robust with argument and return value checks and catch exceptions.