ROS 2 – Robot Operating System

Stefano Carpin

Department of Computer Science and Engineering
School of Engineering
University of California, Merced
`https://sites.ucmerced.edu/scarpin`
`https//robotics.ucmerced.edu`

What is ROS?

ROS – Robot Operating System is *not* an operating system but but a framework to develop robot control software.

1. a software middleware enabling secure communications between components using different patterns (e.g., asynchronous, synchronous, and more);
2. a collection of tools to simplify the development and debugging of complex robotic applications;
3. implementations of numerous algorithms solving basic robotic problems that can be composed together to develop more complex functionalities (e.g., localization, navigation, planning, etc.);
4. definitions of various data types (messages) to process and exchange data commonly needed to implement robotic applications (e.g., quaternions, transformation matrices, sensor data, etc.).

ROS Distributions

Distribution	Supported Platforms
Humble	Ubuntu 22.04, Windows 10
Iron	Ubuntu 22.04, Windows 10
Jazzy	Ubuntu 24.04, Windows 10
Kilted	Ubuntu 24.04, Windows 10

Table: Recent ROS 2 distributions.

In this course we will use Jazzy

How Can I practice with ROS?

- install Ubuntu on a dedicated partition [OR]
- install Ubuntu through Docker (works for MacOs, Windows, or Linux distributions different from Ubuntu 22)
- follow instructions on `https://github.com/stefanocarpin/MRTP/`
- One more alternative: create a free account on `https://www.theconstructsim.com/` and use a fully online solution
- ROS2 can also run on Windows via WSL

Some Important ROS concepts

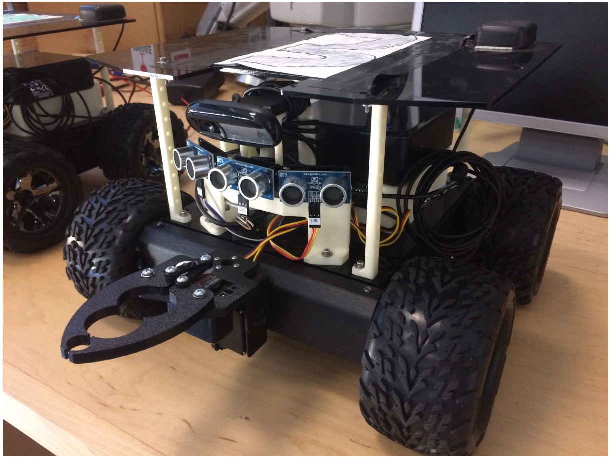
- node
- topic
- message
- service and action
- package and workspace

Nodes

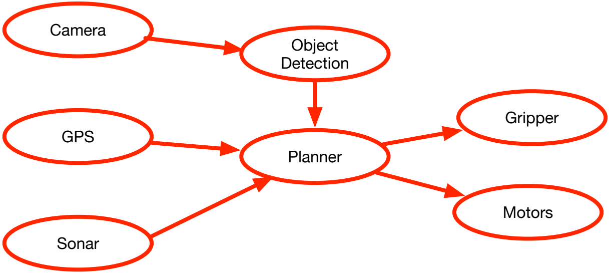
A *ROS program* consists of various interacting computational threads called nodes:

- each node is independently executed
- nodes are executed *concurrently*
- nodes can be written using various ROS client libraries available in different languages (C++, Python, Java, Lisp, ...)
- nodes interact with each other through messages, services, actions
- node decomposition facilitates code reuse and debugging, and increases robustness
- nodes are *loosely coupled*
- typical ROS applications mix ROS provided nodes with user-developed nodes

Multiple Interacting Subsystems



Multiple Interacting Subsystems – Possible ROS decomposition



ROS Nodes

- a ROS application typically runs both nodes written by the programmer to solve a specific task, as well as nodes part of the ROS standard distribution
- for many basic/common tasks, ROS provides implementations through nodes ready to use
- learning how to navigate the ROS documentation to take advantage of available code is essential to develop robust applications and reduce development time

Topics

A topic is an asynchronous, unidirectional stream of messages of a defined type.

asynchronous : publisher and subscriber do not rely on a shared time line

unidirectional : messages always flow from the publisher(s) to the subscriber(s)

stream : messages are sent sequentially and the relative order is not swapped

- nodes *subscribe* (receive from) or *advertise/publish* (send to) topics

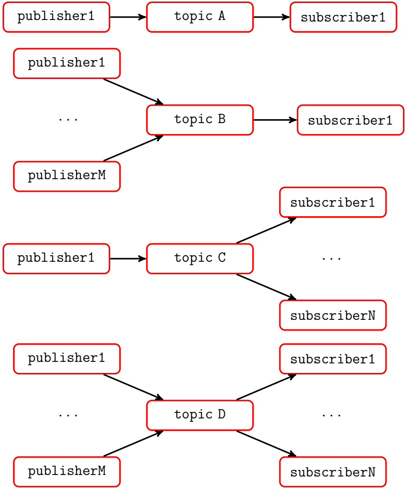
Important features:

- Communication through topics is *many-to-many*
- All messages exchanged through a topic have the same type
    - also (improperly) called the type of the topic

Topics

- handshake happens via the topic name (a string)
- nodes can subscribe/publish to many topics at once
- multiple nodes can use the same topic
- messages through a topic are queued using a finite length buffer (FIFO)

Topics are many-to-many



Packages and workspaces

package : a container of ROS entities (nodes, launch files, datasets, ...)

- (alomst) everything in ROS is located inside a package.
- the fully qualified name of an entity is obtained combining the name of the entity with the name of the package (similar to namespaces in C++)
- packages are stored in folders and each package includes various subfolders

workspace : a container of packages

To automate and expedite the recompilation process, packages and workspaces must be structured following a well defined layout (see colcon commands later on)

- Why packages? Keep things organized and avoid name clashes

Overlay and Underlay

underlay is the workspace including all packages shipped with the ROS 2 distribution

overlay is any additional workspace including the package you develop or get from third party

- underlay and overlay must be made *visible* before they can be used (sourced)
- *sourcing the setup* makes the underlay visible

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

  This command should be added to the startup script
- overlays are made visible with a different command

The command line tool `ros2`

- `ros2`: entry point for numerous fundamental operations nodes, topics, packages and more
- syntax:

  ```
  ros2 <command> <positional parameters>
  ```
- example: list all packages available

  ```
  ros2 pkg list
  ```
- to list all commands:

  ```
  ros2 -h
  ```
- to get help abut a specific command

  ```
  ros2 <command> -h
  ```

`ros2 run`

- `ros2 run` is used to start ROS executables (user never directly starts nodes)
- needs (at least) name of the package and of the executable

  ```
  ros2 run <packagename> <executablename>
  ```

- accepts additional parameters; will be discussed later

Running nodes

- simplest possible example: run two nodes that exchange information through a topic (sender/receiver)
- nodes are part of the `demo_nodes_cpp` package
- `ros2 run demo_nodes_cpp talker`
- `ros2 run demo_nodes_cpp listener`

Try this example and try restarting the nodes at different times or starting multiple instances of the nodes.

Running Turtlesim

Turtlesim is a simple application designed to familiarize and experiment with various ROS concepts

```
ros2 pkg executables turtlesim
```

List of executables

```
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

Let's teleoperate the turtle running the nodes `turtlesim_node` and `turtle_telop_key`:

```
ros2 run turtlesim turtlesim_node
ros2 run turtlesim turtle_teleop_key
```

Command Line Debugging Tools

- `ros2 node`
- `ros2 topic`
- `rqt_graph`
- `ros2 interface`

`ros2 node`

- `ros2 node list`: show all the running nodes
- `ros2 node info [nodename]`: shows detailed information about a running node
    - additional options available; see documentation

ros2 topic

- command line tool providing information about currently active topics
- typically used together with `ros2 interface` (see next slides)
- To get list of all topics
  `ros2 topic list`
- typical output (e.g., when running `turtle_sim`)

```
    /parameter_events
    /rosout
    /turtle1/cmd_vel
    /turtle1/color_sensor
    /turtle1/pose
```

ros2 topic

- To get information about the type of message exchanged through a topic:

```
ros2 topic type /turtle1/cmd_vel
```

- typical output

```
    geometry_msgs/mgs/Twist
```

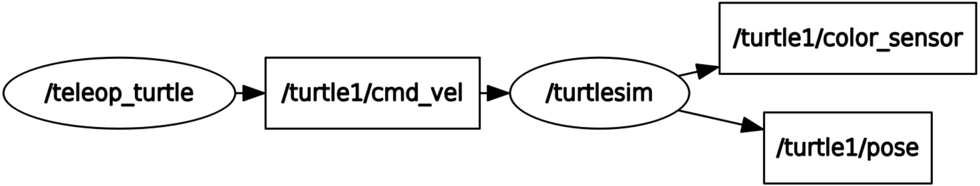
- to get more info about a topic:

```
ros2 topic info /turtle1/cmd_vel
    Type: geometry_msgs/msg/Twist
    Publisher count: 1
    Subscription count: 1
```

- to see the messages transmitted through a topic: `ro2 topic echo [topicname]`
- type `ros2 topic -h` to see for full documentation

The ROS graph

- Useful debugging tool to make sure nodes are properly communicating
- run it with `rqt_graph`
- GUI enables/disables various components to be visualized



Messages

- a message is an instance of a data structure exchanged between nodes
- messages are typed
- messages can contain basic data types or other messages; very similar to a C++ `struct`
- a topic is the communication channel through which messages are exchanged

ros2 interface

- command line utility to get information about the structure of messages.
- typical use: `ros2 interface show [messagename]`
- e.g. `ros2 interface show geometry_msgs/msg/Twist` produces the output:

```
Vector3  linear
 float64 x
 float64 y
 float64 z
Vector3  angular
 float64 x
 float64 y
 float64 z
```

- we can also use `ros2 interface show geometry_msgs/msg/Vector3` to determine the structure of `Vector3`, and so on
- type `ros2 interface -h` to see for full documentation

using ros2 topic to publish messages

- useful to test subscribers
- uses YAML language to map strings into messages (give the command on just one line)

```
ros2 topic pub /turtle1/cmd_vel geometry_msgs/msg/Twist
'{linear:  {x: 0.1, y: 0.0, z: 0.0},
 angular: {x: 0.0,y: 0.0,z: 0.0}}'
```

- see online documentation for full details

List ros2 commands (partial list)

Command	Scope
action	Command related to actions
bag	Command related to recording and replaying data
component	Command related to components
doctor	Diagnostic functions
interface	Information about interfaces (aka messages)
launch	Runs a launch file
lifecycle	Information about nodes lifecycle
node	Retrieves information about running nodes
param	Set/get parameters
pkg	Information about packages and package creation
run	Runs an executable from a package
service	Command related to services
topic	Information and interaction with topics

Services and Actions

service : remote procedure call between nodes.

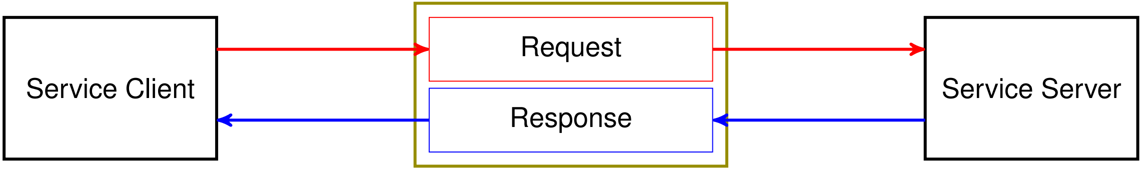
- suited for quick computations, e.g., turn on or reconfigure a sensor
- returns a result

action : *non-blocking* remote procedure call between nodes

- suited for complex tasks, e.g., navigate to a far away location
- returns a result at the end, and updates while executing

Services and actions use messages to pass information between involved nodes

Services



Interacting with services

- `ros2 service list` shows all available services
- each service has a type, i.e., a is implemented as an exchange of a message of a given type. To determine the type of a service:
  `ros2 service list -t`
- the message associated with a service has one section for the request and one section for the response
- `ros2 interface` can be used to determine the structure of a service request/response

Interacting with services

- `ros2 interface show turtlesim/srv/Spawn`

```
float32 x
float32 y
float32 theta
string name #Optional. A unique name will be created and returned
                                              if this is empty
---
string name
```

Interacting with services

- `ros2 service` can be used to initiate service calls and receive responses

```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 1, y: 1,
 theta: 0, name: ’T1’}"
```

- output

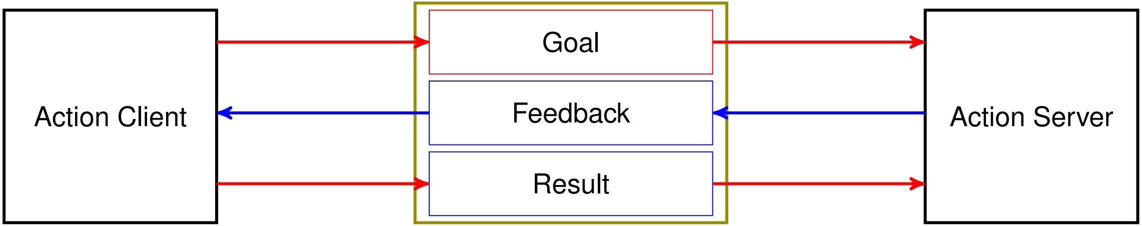
```
requester: making request: turtlesim.srv.Spawn_Request(x=1.0,
y=1.0, theta=0.0, name=’T1’)

response:
turtlesim.srv.Spawn_Response(name=’T1’)
```

Try `ros2 topic list` for the list of topics available now...

Actions

- suited for tasks taking longer to execute (e.g., navigate to a point)
- provide back to caller both *response* at the end, and *feedback* during execution
- can be interrupted by caller before they finish
- differently from services, actions are non-blocking

Actions



Interacting with actions

- `ros2 action list`: lists all available actions
- `ros2 action list -t`: lists all available actions and their type

  ```
  /turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
  ```
- `ros2 interface show turtlesim/action/RotateAbsolute`
- output:

  ```
  # The desired heading in radians
  float32 theta
  ---
  # The angular displacement in radians to the starting position
  float32 delta
  ---
  # The remaining rotation in radians
  float32 remaining
  ```

Interacting with actions

- `ros2 action send_goal`: can be used to initiate an action

  ```
  ros2 action send_goal  /turtle1/rotate_absolute
  turtlesim/action/RotateAbsolute ’{theta: 0.5}’
  ```

ROS Launch Files

- XML files to specify nodes to be run and their parameters (can also be written in other languages)
- reside in a package (in a folder called `launch`), and have extension `.launch.xml`
- can be run in two ways:
  1. `ros2 launch <packagename> <launchfilename>`
  2. `ros2 launch <absolutefilepath>`
- use the second form for debugging only; everything should go into a package

ROS Launch File Example

```
<launch>
  <node pkg="demo_nodes_cpp" exec="talker" name="talkerA" />
  <node pkg="demo_nodes_cpp" exec="talker" name="talkerB" />
  <node pkg="demo_nodes_cpp" exec="listener" />
</launch>
```

On GitHub: `MRTP/MRTP/unsorted/topics.launch.xml`

ROS Launch File Example with Separate Terminals

```
<launch>
  <node pkg="turtlesim" exec="turtlesim_node" name="turtlesim" />
  <node pkg="turtlesim" exec="turtle_teleop_key" name="teleop_key"
    launch-prefix="gnome-terminal␣--" />
</launch>
```

On GitHub: `MRTP/MRTP/unsorted/turtle.launch.xml`