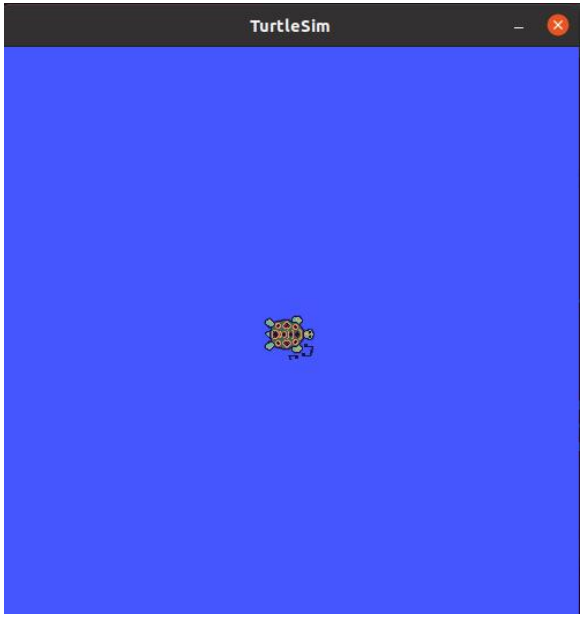
ROS



About ROS

- What is ROS?
- Why ROS?
- Installing ROS
- TurtleSim
- Packages & Nodes
- Topics & Messages

What is ROS?

- ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including *hardware abstraction*, *low-level device control*, *implementation of commonly-used functionality*, *message-passing between processes*, and *package management*. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

Why ROS?

- Many modern robot systems rely on software that spans many different processes and runs across several different computers. For example:
  - Some robots carry multiple computers, each of which controls a subset of the robot's sensors or actuators.
  - Even within a single computer, it's often a good idea to divide the robot's software into small, stand-alone parts that cooperate to achieve the overall goal. This approach is sometimes called "complexity via composition."

Why ROS?

- When multiple robots attempt to cooperate on a shared task, they often need to communicate with one another to coordinate their efforts.
- Human users often send commands to a robot from a laptop, a desktop computer, or mobile device. We can think of this human interface as an extension of the robot's software.

ROS provides two relatively simple, seamless mechanisms for this kind of communication.

Why ROS?

- The rapid progress of robotics research has resulted in a growing collection of good algorithms for common tasks such as navigation, motion planning, mapping, and many others.
- ROS's standard packages provide stable, debugged implementations of many important robotics algorithms.

Why ROS?

- One of the reasons that software development for robots is often more challenging than other kinds of development is that testing can be time consuming and error-prone.
- Well-designed ROS systems separate the low-level direct control of the hardware and high-level processing and decision making into separate programs. Because of this separation, we can temporarily replace those low-level programs (and their corresponding hardware) with a simulator, to test the behavior of the high-level part of the system.

Why ROS?

- Of course, ROS is not the only platform that offers these capabilities. What is unique about ROS,is the level of widespread support for ROS across the robotics community. This “critical mass” of support makes it reasonable to predict that ROS will continue to evolve, expand, and improve in the future.

Installing ROS

- 官网安装说明:

http://wiki.ros.org/cn/noetic/Installation/Ubuntu

- Matlab ros虚拟机安装说明:

https://ww2.mathworks.cn/support/product/robotics/ros2-vm-installation-instructions-v6.html

- Windows虚拟机镜像地址:

https://ssd.mathworks.com/supportfiles/ros/virtual_machines/v3/ros_noetic_foxy_gazebov11_linux_win_v1.zip

Setting environment variables

- ROS relies on a few environment variables to locate the files it needs. To set these environment variables, you'll need to execute the setup.bash script that ROS provides, using this command:

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

- You can then confirm that the environment variables are set correctly using a command like this:

```
export | grep ROS
```

turtlesim

- Starting turtlesim In three separate terminals, execute these three commands:

roscore

rosrun turtlesim turtlesim_node

rosrun turtlesim turtle_teleop_key

Packages

- All ROS software is organized into packages. A ROS package is a coherent collection of files, generally including both executables and supporting files, that serves a specific purpose.
- In the previous example, we used two executables called *turtlesim_node* and *turtle_teleop_key*, both of which are members of the *turtlesim* package.

Packages

- ROS provides several commands for interacting with installed packages.
- Listing and locating packages You can obtain a list of all of the installed ROS packages using this command:

rospack list

Packages

- Each package is defined by a manifest, which is a file called *package.xml*. This file defines some details about the package, including its name, version, maintainer, and dependencies.
- The directory containing *package.xml* is called the package directory. (In fact, this is the definition of a ROS package: Any directory that ROS can find that contains a file named *package.xml* is a package directory.) This directory stores most of the package's files.

Packages

- To find the directory of a single package, use the rospack find command:

rospack find package-name

- Of course, there may be times when you don't know (or can't remember) the complete name of the package that you're interested in. In these cases, it's quite convenient that rospack supports tab completion for package names. For example, you could type

rospack find turtle

- before pressing *Enter*, press the *Tab* key twice to see a list of all of the installed ROS packages whose names start with turtle.

Packages

- **Inspecting a package**
- To view the files in a package directory, use a command like this:

rosls package-name

- If you'd like to "go to" a package directory, you can change the current directory to a particular package, using a command like this:

roscd package-name

Master

- **The master**
- One of the basic goals of ROS is to enable roboticists to design software as a collection of small, mostly independent programs called *nodes* that all run at the same time. For this to work, those nodes must be able to communicate with one another. The part of ROS that facilitates this communication is called the *ROS master*. To start the master, use this command:

```
roscore
```

Nodes

- Once you've started *roscore*, you can run programs that use ROS. A running instance of a ROS program is called a **node**.
- In the turtlesim example, we created two nodes. One node is an instance of an executable called *turtlesim_node*. This node is responsible for creating the turtlesim window and simulating the motion of the turtle.
- The second node is an instance of an executable called *turtle_teleop_key*. The abbreviation teleop is a shortened form of the word *teleoperation*, which refers to situations in which a human controls a robotremotely by giving direct movement commands. This node waits for an arrow key to be pressed, converts that key press to a movement command, and sends that command to the *turtlesim_node* node.

Nodes

- Starting nodes The basic command to create a node (also known as “running a ROS program”) is *rosrun*:

rosrun package-name executable-name

- There are two required parameters to rosrun. The first parameter is a package name. We discussed package names previously. The second parameter is simply the name of an executable file within that package.

Nodes

- Listing nodes ROS provides a few ways to get information about the nodes that are running at any particular time. To get a list of running nodes, try this command:

rosnode list

- If you do this after executing the previous commands, you'll see a list of three nodes:

/rosout

/teleop_turtle

/turtlesim

Nodes

- The /rosout node is a special node that is started automatically by *roscore*. Its purpose is somewhat similar to the standard output (i.e. std::cout) that you might use in a console program.
- The other two nodes should be fairly clear: They're the simulator (*turtlesim*) and the teleoperation program (*teleop_turtle*) we started previously.

Nodes

- Inspecting a node You can get some information about a particular node using this command:

rosnode info node-name

- The output includes a list of topics for which that node is a publisher or subscriber, the services offered by that node, its Linux process identifier (PID), and a summary of the connections it has made to other nodes.

Nodes

- Killing a node To kill a node you can use this command:

rosnode kill node-name

Topics and messages

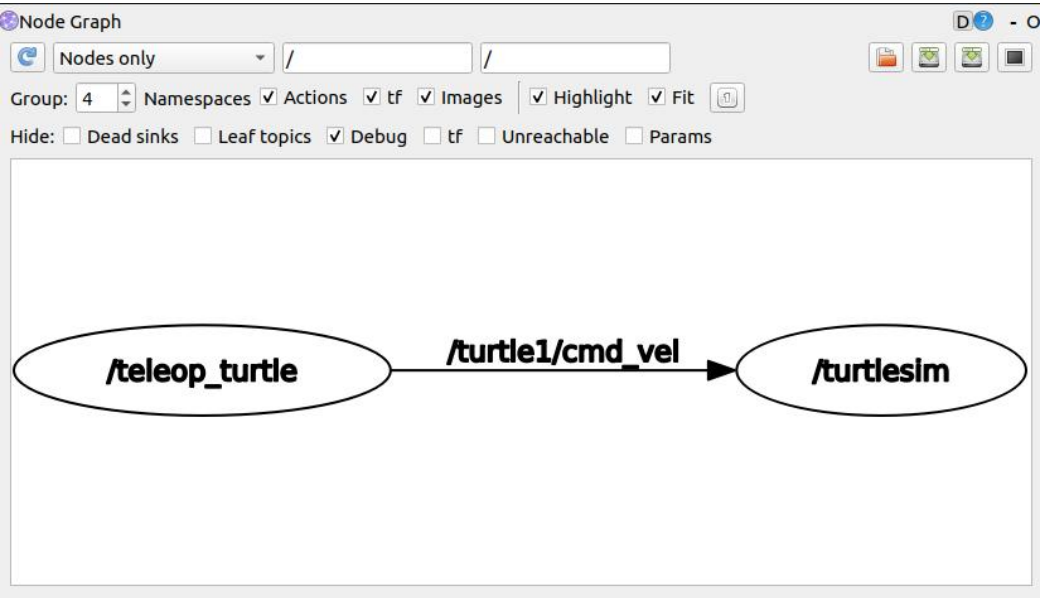
- The primary mechanism that ROS nodes use to communicate is to send messages. Messages in ROS are organized into named **topics**.
- The idea is that a node that wants to share information will *publish messages* on the appropriate topic or topics; a node that wants to receive information will *subscribe to the topic* or topics that it's interested in. The ROS master takes care of ensuring that publishers and subscribers can find each other; the messages themselves are sent directly from publisher to subscriber.

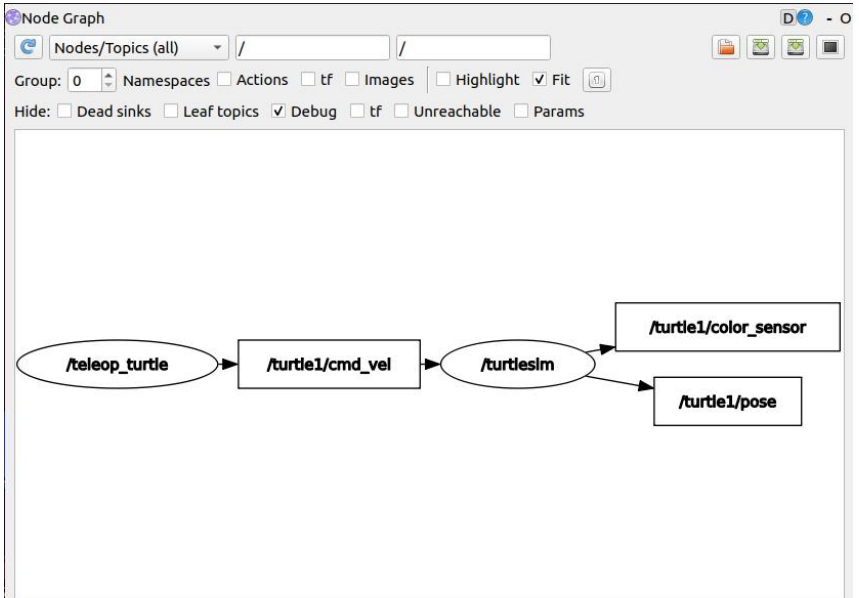
Viewing the graph

- The easiest way to visualize the publishsubscribe relationships between ROS nodes is to use command:

rqt_graph

- You should see a GUI, most of which is devoted to showing the nodes in the current system.





Viewing the graph

- When you press a key, the /teleop_turtle node publishes messages with those movement commands on a topic called /turtle1/cmd_vel. Because it subscribes to that topic, the turtlesim_node receives those messages, and simulates the turtle moving with the requested velocity.
- The simulator doesn't care (or even know) which program publishes those cmd_vel messages. Any program that publishes on that topic can control the turtle.
- The teleoperation program doesn't care (or even know) which program subscribes to the cmd_vel messages it publishes. Any program that subscribes to that topic is free to respond to those commands.

Messages and message types

- **Listing topics**
- To get a list of active topics, use this command:

rostopic list

- In our example, this shows a list of five topics:

/rosout

/rosout_agg

/turtle1/cmd_vel

/turtle1/color_sensor

/turtle1/pose

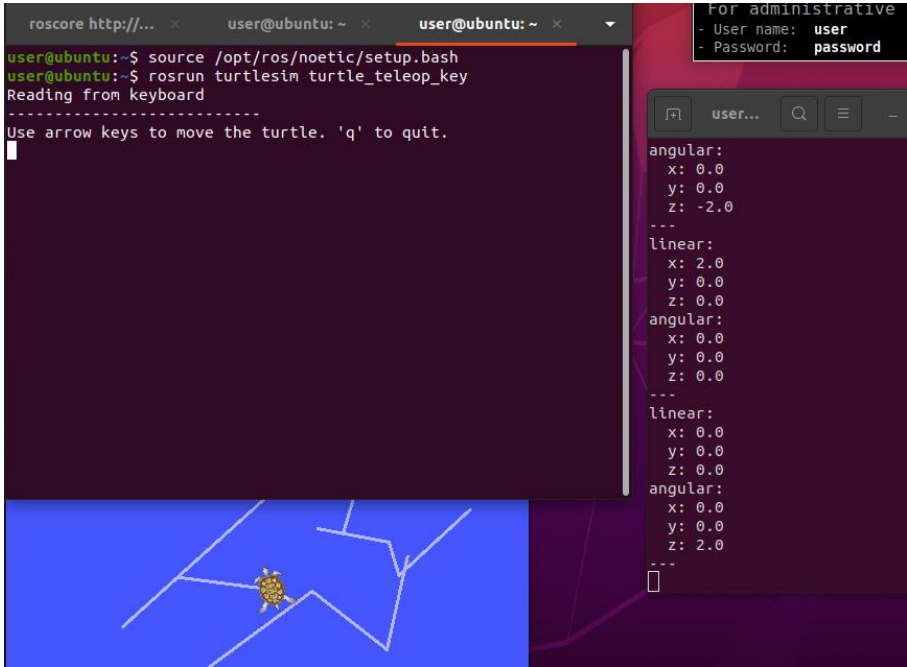
Messages and message types

- **Echoing messages**
- You can see the actual messages that are being published on a single topic using the rostopic command:

rostopic echo topic-name

- This command will dump any messages published on the given topic to the terminal.

rostopic echo /turtle1/cmd_vel



Messages and message types

- **Measuring publication rates**
- There are also two commands for measuring the speed at
- which messages are published and the bandwidth consumed by those messages:

rostopic hz topic-name

rostopic bw topic-name

These commands subscribe to the given topic and output statistics in units of messages per second and bytes per second, respectively

Messages and message types

- **Inspecting a topic**
- You can learn more about a topic using the rostopic info command:

rostopic info topic-name

- For example, from this command:

rostopic info /turtle1/color_sensor

- you should see output similar to this:

Type: turtlesim/Color

Publishers:

* /turtlesim (http://192.168.38.131:41153/)

Subscribers: None

Messages and message types

Type: turtlesim/Color

Publishers:

* /turtlesim (http://192.168.38.131:41153/)

Subscribers: None

- The most important part of this output is the very first line, which shows the message type of that topic. In the case of /turtle1/color_sensor, the message type is **turtlesim/Color**.
- The word “type” in this context is referring to the concept of a data type. It’s important to understand message types because they determine the content of the messages. That is, the message type of a topic tells you what information is included in each message on that topic, and how that information is organized.

Messages and message types

- **Inspecting a message type**
- To see details about a message type, use a command like
- this:

rosmsg show message-type-name

- Let's try using it on the message type for /turtle1/color_sensor that we found above:

rosmsg show turtlesim/Color

- The output is:

uint8 r

uint8 g

uint8 b

Messages and message types

uint8 r

uint8 g

uint8 b

- The format is a list of fields, one per line. Each field is defined by a built-in data type (like int8, bool, or string) and a field name.
- The output above tells us that a **turtlesim/Color** is a thing that contains three unsigned 8-bit integers called r, g, and b. Every message on any topic with message type **turtlesim/Color** is defined by values for these three fields.

Messages and message types

```
rostopic info /turtle1/cmd_vel
Type: geometry_msgs/Twist
Publishers: None
Subscribers:
 * /turtlesim (http://192.168.38.131:41153/)

rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Publishing messages from the command line

- Once you know the message details, you may find it useful at times to publish messages by hand. To do this, use rostopic:

```
rostopic pub -r rate-in-hz topic-name message-type message-content
```

- This command repeatedly publishes the given message on the given topic at the given rate. The final message content parameter should provide values for all of th e fields in the message type, in order. Here's an example:

```
rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist '[2, 0, 0]' '[0, 0, 0]'
```

- Likewise, a command like this will command the robot to rotate in place about its zaxis (which is perpendicular to your computer's screen):

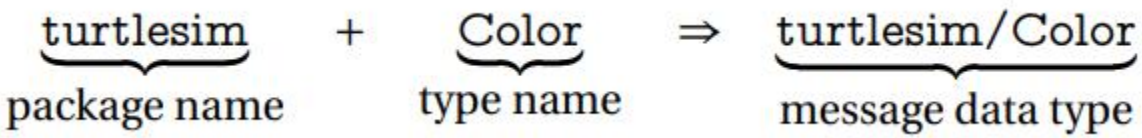
```
rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist '[0, 0, 0]' '[0, 0, 1]'
```

Publishing messages from the command line

- **Understanding message type names**
- Like everything else in ROS, every message type belongs to a specific package. Message type names always contain a slash, and the part before the slash is the name of the containing package:

package-name/type-name

- For example, the turtlesim/Color message type breaks down this way:



Checking for problems

- One final (for now) command line tool, which can be helpful when ROS is not behaving the way you expect, is **roswtf**:

roswtf

- This command performs a broad variety of sanity checks, including examinations of your environment variables, installed files, and running nodes.
- For example, **roswtf** checks whether the rosdep portions of the install process have been completed, whether any nodes appear to have hung or died unexpectedly, and whether the active nodes are correctly connected to each other.