

# Introduction to Cognitive Robotics

David Vernon

[www.cognitiverobotics.net](http://www.cognitiverobotics.net)

# Lecture 4

[www.cognitiverobotics.net/CR04.pdf](http://www.cognitiverobotics.net/CR04.pdf)

## Mobile Robots: ROS

- Introduction to ROS (Robot Operating System)

(Rhymes with “gloss”)

- Introduction to writing ROS software
  - Using the ROS Turtlebot simulator `Turtlesim`
  - Here in C++
  - Later in the course in Lisp with `CRAM`



Based mainly on J. M. O’Kane, *A Gentle Introduction to ROS*, 2014.  
<https://cse.sc.edu/~jokane/agitr/>

# 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

<http://wiki.ros.org/ROS/Introduction>

# Features

- Distributed computation
  - Divide software into small stand-alone parts that, together, achieve the overall goal
  - Communication between multiple concurrent process that may or may not be running on the same computer
  - Based on component-based software engineering
- Software reuse
  - ROS's standard packages provide stable implementations of many important algorithms

# Features

## Rapid testing

- Testing the behaviour of high-level parts of the system is facilitated by simulation of low-level parts, including the robot
- Provides a simple way to record and play back sensor data: bags  
A tool called **rosvbag** is used to record and replay

# Features

## Community support

- Hardware drivers
- Libraries: PCL, OpenCV, TF, ...
- Capabilities: avigation, manipulation, control, .
- Applications: fetching beer, making popcorn, ...

## ROS is **not** ...

- A programming language
  - It supports C++, Lisp, Python, Java, among others
- Just a library
  - also include a central server, command-line tools, graphical tools, build systems.
- An integrated development environment (IDE)

# ROS Distributions

- Major versions of ROS are called distributions
- Distributions are named using adjectives that start with successive letters of the alphabet
  - groovy, hydro, indigo, jade, **kinetic**, ...
- Referred to in the ROS documentation by the term **distro**
- Different distributions use different build systems
  - **kinetic** uses **catkin**



# Packages

- All ROS software is organized into `packages`
- A ROS package is a coherent collection of files
  - Serves a specific purpose.
  - Includes executables and supporting files
- All ROS software is part of one package or another
- `rospack list` provide a list of all installed ROS packages

# Packages

- Each package is defined by a **manifest**
- In a is a file called **package.xml**
- Defines details about the package
  - name
  - version
  - maintainer
  - dependencies
- The directory containing **package.xml** is called **the package directory**

# Packages

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

```
rospack find package-name
```

- Use tab completion if you are not sure of the full name and to save time typing

# Packages

- To view the files in a package directory

```
rosls package-name
```

- To change directory to a package directory

```
roscd package-name
```

# Packages

## Stacks and packages

- You may see references to the concept of a **stack**
  - A stack is a **collection of related packages**
- It has now been phased out and replaced by **meta-package**
  - A meta-package is a package
  - It has a manifest just like any other package
  - but **no other packages are stored inside its directory**
- Whereas a stack is a container for packages stored as subdirectories

# ROS Master

So much for how ROS **files** are organized

Let's now turn our attention to how ROS **software** is organized and **executed**

Aside:

- As noted already, ROS software adheres to the paradigm on **component-based software engineering (CBSE)** in which a software system comprises multiple quasi-independent communicating components (programs or processes)
- As we will see, it also adheres to the **component-port-connector** communication model

# ROS Master

- ROS software comprises a collection of small, independent, **loosely-coupled** programs called **nodes** that **all run at the same time**
- These 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 the **roscore** command

↑  
Coupling is effected by sending  
**messages** on **topics** (see below)

# ROS Master

- Typically, you start `roscore` in one terminal
- then open other terminals for your “real” work
- There are not many reasons to stop `roscore`, except when you’ve finished working with ROS
- When you reach that point, you can stop the master by typing Ctrl-C in its terminal



# ROS Master


- Most ROS nodes connect to the master when they start up
- Typically, they do not attempt to reconnect if that connection fails later on
- Therefore, if you stop `roscore`, any other nodes running at the time will be unable to establish new connections,  
`even if you restart roscore later`
- Bottom line: if you restart `roscore` you will have to restart all the nodes

# ROS Master

ROS provides a convenient way of doing this: `roslaunch`


- `roslaunch` starts many nodes at once
- It will also start a master if none is running
- But will also use an existing master if there is one

# Nodes

- Once you've started `roscore`, you can run programs that use ROS.
- A running **instance** of a ROS program is called a **node**  


It is possible to have multiple instances of the same ROS program running concurrently
- The command to create a node (also known as “running a ROS program”) is **`roslaunch`**:

**`roslaunch package-name executable-name`**

 Doesn't register the executable with the ROS master:  
That's done in the node itself



# Nodes

- Just a moment ago, we said

“A running **instance** of a ROS program is called a **node**”

It is possible to have multiple instances  
of the same ROS program running concurrently

- Each instance must have a different name
- You can explicitly set the name of a node as part of the `roslaunch` command:

```
roslaunch package-name executable-name __name:=node-name
```

double underscore



# Topics and Messages

- ROS nodes communicate by sending **messages**
- Messages are organized into named **topics**
  - A node can **publish** messages on a topic
  - Another node that wants to receive the topic messages can **subscribe** to that topic
- The ROS master takes care of linking publishers and subscribers but the messages are sent directly from publisher to subscriber

# Topics and Messages

## Recall: Exercise

Open a terminal and enter

```
[~]$ roscore
```

Open a second terminal and enter

```
[~]$ rosrunc turtlesim turtlesim_node
```

Open a third terminal and enter

```
[~]$ rosrunc turtlesim turtle_teleop_key
```

The separate terminals are intended to allow all three commands to execute simultaneously

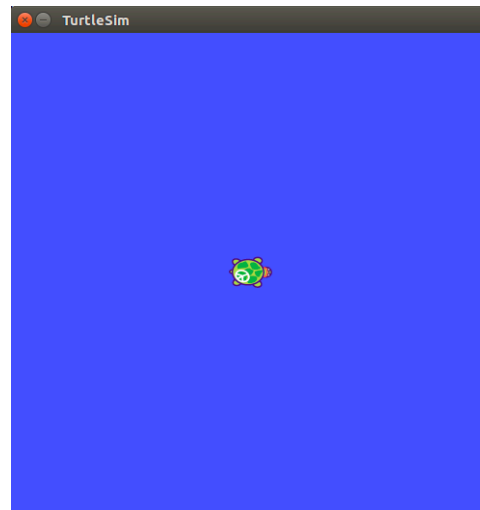


# Topics and Messages

## Recall: Exercise

If everything works correctly, you should see a graphical window similar to one below

The appearance of your turtle may differ. The simulator selects from a collection of “mascot” turtles for each of the historical distributions of ROS

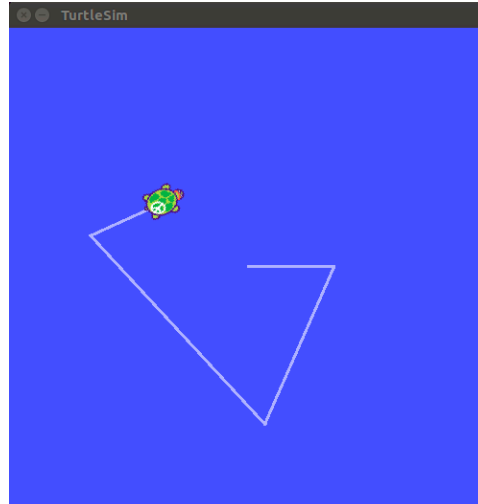


# Topics and Messages

## Recall: Exercise

Make sure your third terminal [the one executing the `turtle_teleop_key` command] is in focus [i.e. is selected].

Press the Up, Down, Left, or Right arrow key to move the turtle and leave a trail behind it.



# Topics and Messages

Use `rostopic info node-name` to get a list of the **topics** and **services** supported by the node (more on services later)

For example: `rostopic info turtlesim` yields the following

```
parallels@ubuntu: ~
parallels@ubuntu:~$ rostopic info turtlesim
-----
Node [/turtlesim]
Publications:
 * /rosout [roscpp_msgs/Log]
 * /turtle1/color_sensor [turtlesim/Color]
 * /turtle1/pose [turtlesim/Pose]

Subscriptions:
 * /turtle1/cmd_vel [unknown type]

Services:
 * /clear
 * /kill
 * /reset
 * /spawn
 * /turtle1/set_pen
 * /turtle1/teleport_absolute
 * /turtle1/teleport_relative
 * /turtlesim/get_loggers
 * /turtlesim/set_logger_level
```

turtlesim publishes on these topics

turtlesim subscribes to these topics

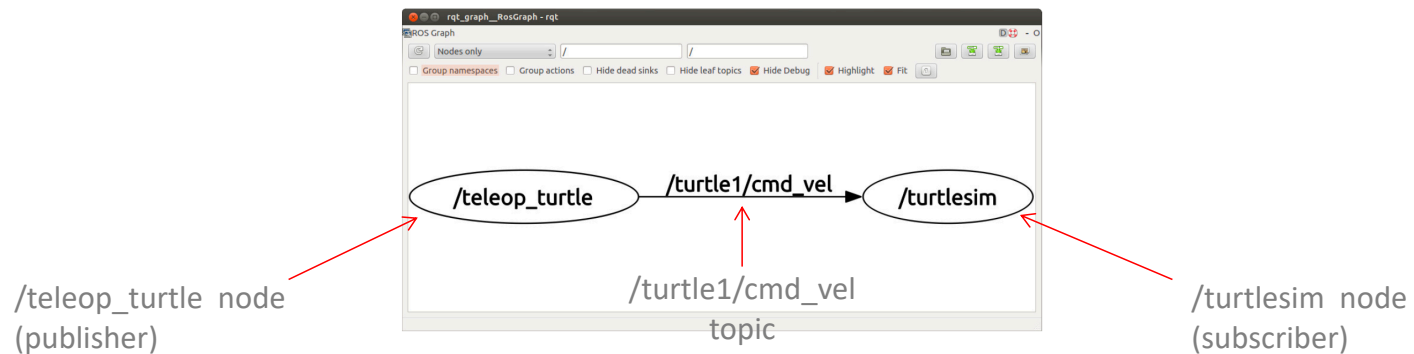
turtlesim can be configured by using these services

# Topics and Messages

- The publish and subscribe relationships between nodes can be represented and visualized as a directed graph

r for ROS and qt for the Qt GUI toolkit

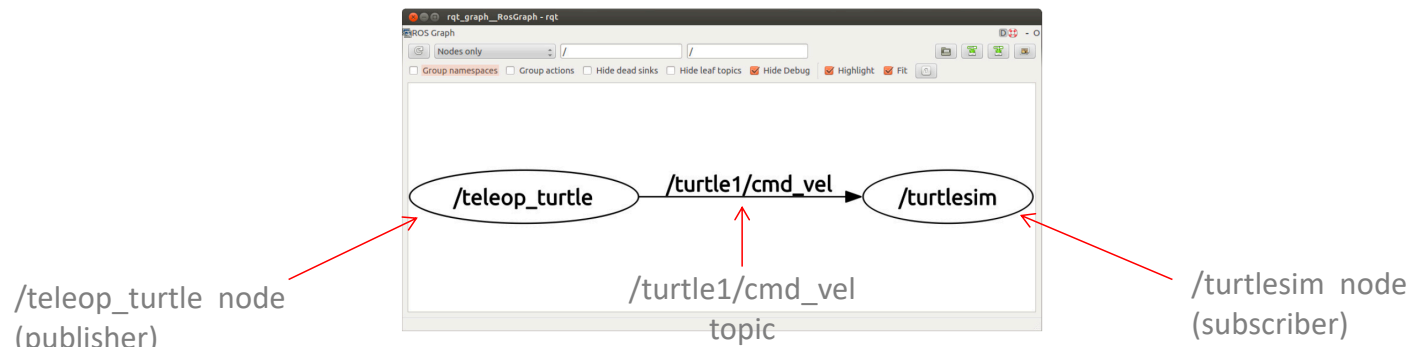
- The `rqt_graph` command allows you to draw this graph



Credit: J. M. O’Kane, A Gentle Introduction to ROS, 2014  
<https://cse.sc.edu/~jokane/agitr/>

# Topics and Messages

- `/teleop_turtle` publishes messages on a topic called `/turtle1/cmd_vel`
- `/turtlesim` subscribes to those messages
  - `cmd_vel` is short for “command velocity”

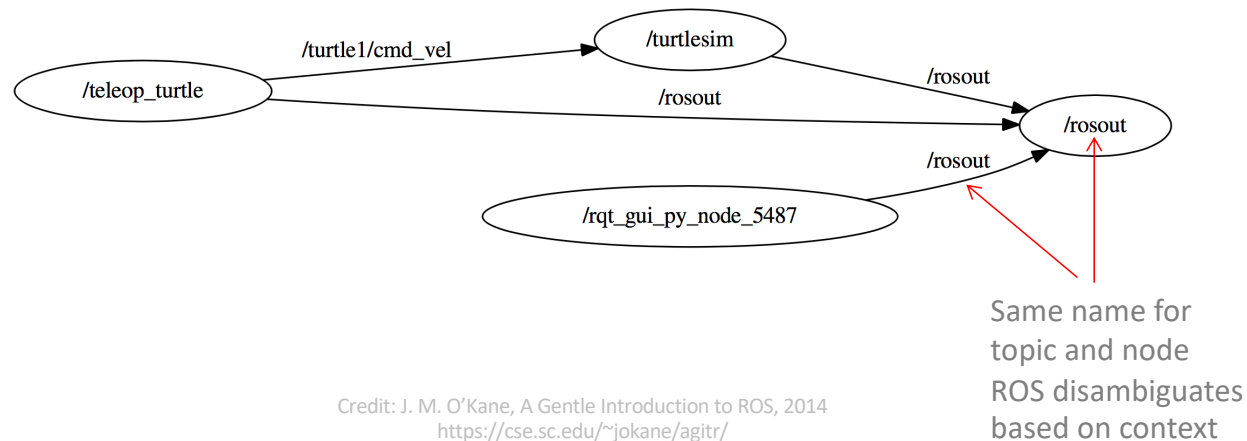


Credit: J. M. O’Kane, A Gentle Introduction to ROS, 2014  
<https://cse.sc.edu/~jokane/agitr/>

# Topics and Messages

By default, `rqt_graph` hides nodes that are usually used for debugging, e.g. `rosout`

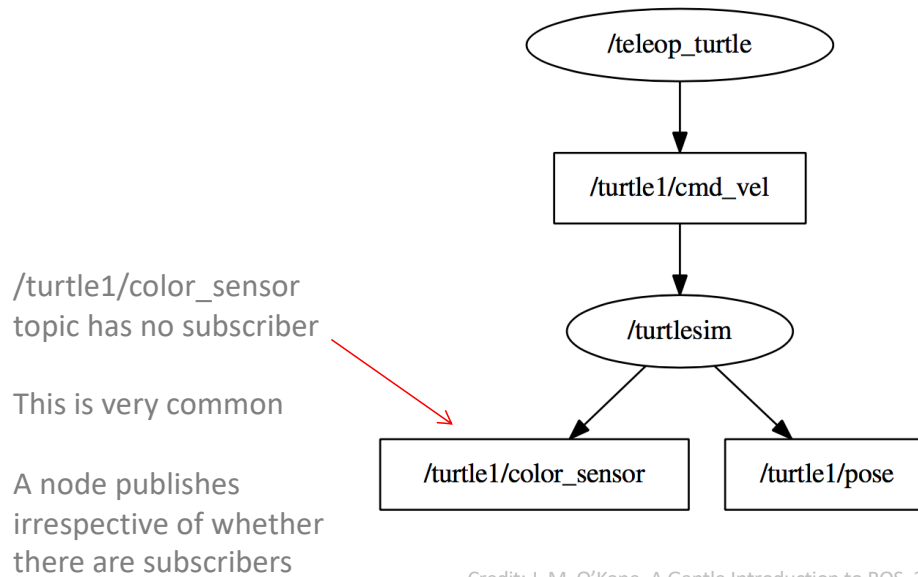
- You can disable this by unchecking the “Hide debug” box
- `rqt_graph` will then draw the following graph



Credit: J. M. O’Kane, A Gentle Introduction to ROS, 2014  
<https://cse.sc.edu/~jokane/agitr/>

# Topics and Messages

Alternative `turtlesim` graph, showing all topics, including those with no publishers or no subscribers, as distinct entities



Credit: J. M. O’Kane, A Gentle Introduction to ROS, 2014  
<https://cse.sc.edu/~jokane/agitr/>

# Topics and Messages

## Use

- `rostopic list` to get a list of active topics
- `rostopic echo topic-name` to see the messages that are being published on a topic
- `rostopic hz topic-name` to see how often the messages are sent
- `rostopic bw topic-name` to see the bandwidth used by the messages
- `rostopic info topic-name` to see the information about a topic



# Topics and Messages

- A messages has a **message type** (i.e. a data type) that determines the information in a message
- The message type is printed when you use `rostopic info topic-name`
- The message type is also printed when you use `rostopic echo topic-name`
- Use `rosmmsg show message-type-name` to see details about a message type

# Topics and Messages

- A message comprises one or more **fields**
- Each field has a built-in data type (e.g. `int8`, `bool`, `string`)
- A field can be a **composite field**, each with component fields
- Composite fields can be nested:
  - The fields in a composite field can also be a composite field
- The data types of composite fields are message types in their own right

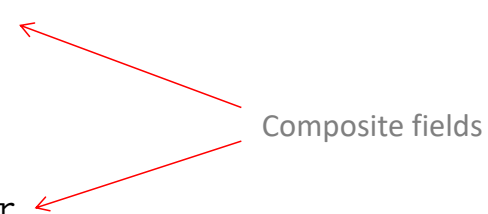
# Topics and Messages

For example, the message type for the `/turtle1/cmd_vel` topic is `geometry_msgs/Twist`

This is a composite message type, comprising two (composite) fields of type `geometry_msgs/Vector3`

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

Composite fields



This message comprises exactly six numbers of type `float64`

# Topics and Messages

If we declare a message of this type

```
geometry_msgs::Twist msg;
```

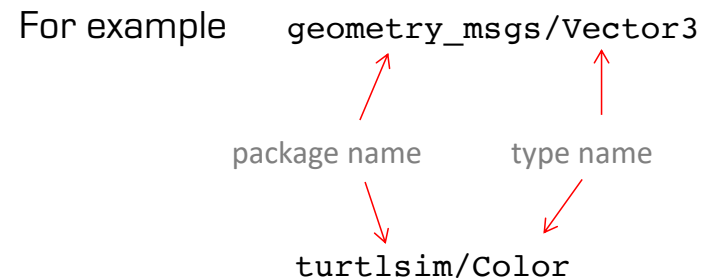
We can assign a value as follows:

```
msg.linear.x = 0;
```

# Topics and Messages

Message type names comprise two parts (separated by /)

1. the name of the package to which it belongs
2. the name of the type



# Topics and Messages

- We note in passing that ROS allows you to publish messages from the command line in a terminal

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

- We won't use this facility but it can be helpful on occasion

# Services

Service calls: an alternative way of communicating with nodes

– Bi-directional

- One node **sends** information to another node (e.g. requesting information)
- The other node **responds** (e.g. with the required information)
- In contrast, when a message is published, there is no concept of a response, and no guarantee that there is even a node subscribing to topic and receiving the messages

– One-to-one

- Each service call is **initiated by one node** and the **response goes back to it**
- In contrast, topics and message may have many publishers and many subscribers

# Services

## Terminology

- Client node sends some data called a **request** to a **server** node
  - Waits for a reply
- **Server** node receives the **request**
  - Takes some action
  - Sends some data called a **response** back to the **client**
- The content of the request and the response is determined by the **service data type**
  - Similar to the message type associated with a topic
  - Two parts (and possibly two different types): request and response



# Services

## Two basic types of service

- General services that can be used with different nodes
- Services that are for specific nodes

As we have seen, use `roscall info node-name` to list the services offered by a node

```
parallels@ubuntu: ~
parallels@ubuntu:~$ roscall info turtlesim
-----
Node [/turtlesim]
Publications:
* /rosout [roscpp_msgs/Log]
* /turtle1/color_sensor [turtlesim/Color]
* /turtle1/pose [turtlesim/Pose]

Subscriptions:
* /turtle1/cmd_vel [unknown type]

Services:
* /clear
* /kill
* /reset
* /spawn
* /turtle1/set_pen
* /turtle1/teleport_absolute
* /turtle1/teleport_relative
* /turtlesim/get_loggers
* /turtlesim/set_logger_level
```

# Services

Use `rosservice info service-name`  
to determine the service data type of a service

For example, `rosservice info /spawn` gives

The service data type is  
`turtlesim/Spawn` →

```
parallels@ubuntu: ~  
parallels@ubuntu:~$ rosservice info /spawn  
Node: /turtlesim  
URI: rosrpc://ubuntu:53282  
Type: turtlesim/Spawn  
Args: x y theta name  
parallels@ubuntu:~$
```

There are four arguments in this data type:  
`x`, `y`, `theta` (specifying the turtles pose ... location and orientation)  
`name` (specifying the name of the new turtle being spawned)

# Services

Use `rossrv show service-data-type-name` to get details about the service data type

For example, `rossrv show turtle1/Spawn` gives

The part is the request  
i.e. information sent to  
the server by the client

This part is the response  
i.e. information sent back  
to the client by the server

```
parallels@ubuntu: ~  
parallels@ubuntu:~$ rossrv show turtlesim/Spawn  
float32 x  
float32 y  
float32 theta  
string name  
---  
string name  
parallels@ubuntu:~$
```

# Checking for Problems

- When ROS is not behaving the way you expect, use the command line tool

`roswtf`

- It performs a variety of sanity checks
  - For example, `roswtf` checks whether the `rosdep` portions of the install process have been completed (see later)

# ROS Resources

Wiki	<a href="http://wiki.ros.org/">http://wiki.ros.org/</a>
Installation	<a href="http://wiki.ros.org/ROS/Installation">http://wiki.ros.org/ROS/Installation</a>
Tutorials	<a href="http://wiki.ros.org/ROS/Tutorials">http://wiki.ros.org/ROS/Tutorials</a>
Tutorial Videos	<a href="http://www.youtube.com/playlist?list=PLDC89965A56E6A8D6">http://www.youtube.com/playlist?list=PLDC89965A56E6A8D6</a>
ROS Cheat Sheet	<a href="http://www.tedusar.eu/files/summerschool2013/ROScheatsheet.pdf">http://www.tedusar.eu/files/summerschool2013/ROScheatsheet.pdf</a>

# Recommended Reading

[http://wiki.ros.org/catkin/Tutorials/create\\_a\\_workspace](http://wiki.ros.org/catkin/Tutorials/create_a_workspace)

<http://wiki.ros.org/ROS/Tutorials/CreatingPackage>

<http://wiki.ros.org/roscpp/Overview/InitializationandShutdown>

<http://wiki.ros.org/roscpp/Overview/NodeHandles>

<http://wiki.ros.org/ROS/Tutorials/BuildingPackages>

[http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber\(c++\)](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c++))

J. M. O'Kane, A Gentle Introduction to ROS, 2014.

<https://cse.sc.edu/~jokane/agitr/>