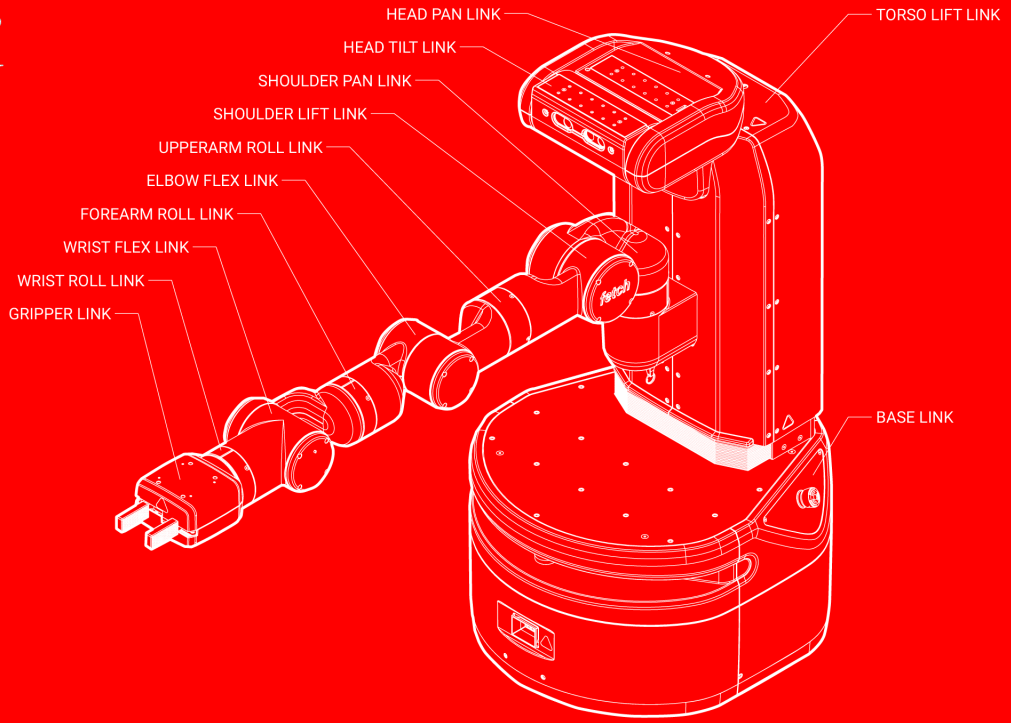
KING'S College LONDON



ROS: Robot Operating System

> Aran Sena

> Eric Schneider

> Tsvetan Jivkov

King's College London
Robot Learning Lab / Robot Interaction Lab

↑

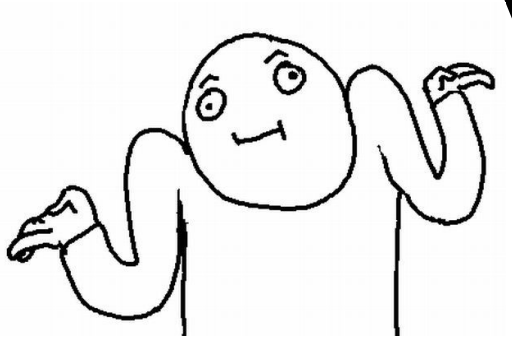
Trinity College Dublin
Human Robot Interaction

↑

National Instruments UK/IRE
Automated Test & Measurement

↑

Trinity College Dublin
Mobile Robotics



What is ROS?

- Not an operating system!
- It is a collection of tools that offer...
  - Hardware abstraction
  - Message-passing between multiple processes
  - Open-source packages for commonly-used functionality
  - Package management
  - Tools for running code across multiple computers
  - And more...

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

Where is ROS?

Photos of some of the robot platforms
Photos from growbotics workshop

Launching ROS

```
viki@c3po:~$ roscore
... logging to /home/viki/.ros/log/423604c6-c86a-11e7-aee3-00
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://c3po:36019/
ros_comm version 1.11.8


SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.8

NODES

auto-starting new master
process[master]: started with pid [13008]
ROS_MASTER_URI=http://c3po:11311/

setting /run_id to 423604c6-c86a-11e7-aee3-000c29051961
process[rosout-1]: started with pid [13021]
started core service [/rosout]
```

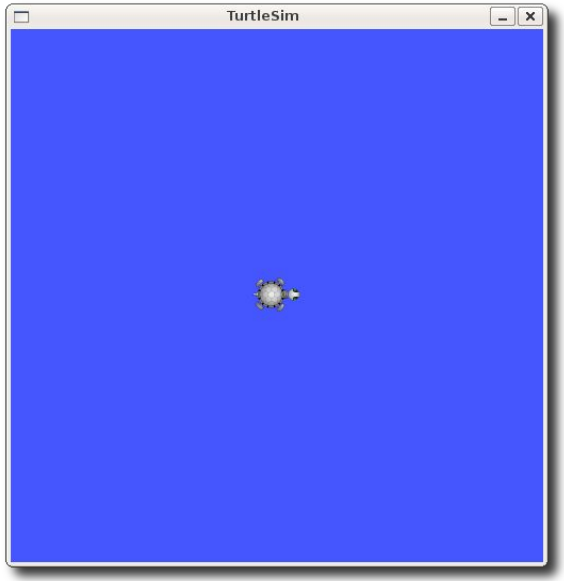
- Before using anything ROS, you must run **roscore**
- roscore starts a collection of critical nodes which manage your whole system, including the **ROS Master** which coordinates all communication.
- When working with one computer, you don't need to worry too much about this.
- When working with multiple computers connected to your ROS network, you will need to ensure that there is only one ROS master which all systems talk to, see: http://wiki.ros.org/ROS/Tutorials/MultipleMachines

http://wiki.ros.org/roscore

Packages

- ROS organises software into **packages**.
- Each package contains code/files/data/ etc. for a specific purpose.
- Hundreds of packages available from ros.org, see http://www.ros.org/browse/list.php
- To code from a package in the terminal, use **rosrun**.

rosrun turtlesim turtlesim_node



http://wiki.ros.org/turtlesim

ROS.org

About | Support | Discussion Forum | Service Status | Q&A answers.ros.org

Documentation Browse Software News

hokuyo_node

electric fuerte groovy hydro indigo jade Documentation Status

Package Summary

✔ Released ✔ Continuous integration ✔ Documented

A ROS node to provide access to SCIP 2.0-compliant Hokuyo laser range finders (including 04LX).

- Maintainer status: maintained
- Maintainer: Chad Rockey <chadrockey AT gmail DOT com>
- Author: Brian P. Gerkey, Jeremy Leibs, Blaise Gassend
- License: LGPL
- Bug / feature tracker: https://github.com/ros-drivers/hokuyo_node/issues
- Source: git https://github.com/ros-drivers/hokuyo_node.git (branch: indigo-devel)

Package Links

Code API
Tutorials
Troubleshooting
FAQ
Changelog
Change List
Reviews

Dependencies (8)
Used by (5)
Jenkins jobs (9)

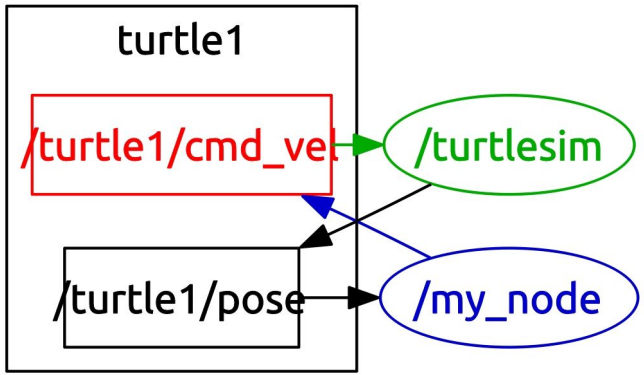
Page
Immutable Page
Info
Attachments
More Actions:

User
Login

Hokuyo node example (LIDAR node)

http://wiki.ros.org/hokuyo_node

Nodes & Topics




```
viki@c3po:~$ rosnode list
/rosout
/turtlesim
viki@c3po:~$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
viki@c3po:~$
```

- Packages organise application code into **nodes**. One package can have multiple nodes.
- Nodes communicate via **topics**
- Run **rqt_graph** to visualise the ROS computation graph
- Ovals indicate nodes, Rectangles indicate topics.
- The arrows indicate direction of communication - Tail: **Publisher,** Head: **Subscriber**
- You can see all nodes and topics from the terminal using **rosnode** and **rostopic** with the argument **list**

http://wiki.ros.org/Nodes

1. Start roscore
2. Start a turtlesim node (rosrun…)
3. Find a node that you can use to control the turtle (rosrun…)
4. Visualise the computation graph (rqt_graph)
5. Use rostopic to see the messages on the /turtle1/pose topic
6. Use rostopic to find out what the type of message /turtle1/cmd_vel
7. Use rostopic to verify which node is the publisher and which is the subscriber for /turtle1/cmd_vel

Tips: Use the [Tab] button and autocompletion to see what nodes a package can run and what arguments can be passed to rostopic

Exercise 1: Calling ROS nodes and inspecting them

—

1. roscore
2. rosrun turtlesim turtlesim_node
3. rosrun turtlesim turtle_teleop_key
4. rqt_graph
5. rostopic echo /turtle1/pose topic
6. rostopic [info/type] /turtle1/cmd_vel
7. Sub: /turtlesim, Pub: /teleop_turtle

*rostopic info will also show you which nodes are currently publishing/subscribing to it.

Exercise 1 Solution: Calling ROS nodes and inspecting them

Message Types

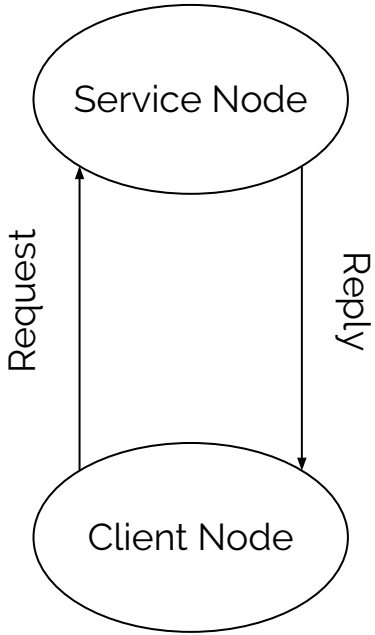
- Publishers and subscribes must have matching **message types.**
- ROS has a number of standard message types for various applications.
  - std_msgs, geometry_msgs, sensor_msgs
- Some message types are basic int/float/string/etc.
- Others provide more advanced **structure** for your application, e.g. a Pose message
- Use **rosmsg** list to see all message types (but, there are many!)
- Use **roscd** to find the std_msgs/msg folder and browse the types
  - roscd std_msgs/msg

```
viki@c3po:~$ rosmsg show geometry_msgs/Pose
geometry_msgs/Point position
  float64 x
  float64 y
  float64 z
geometry_msgs/Quaternion orientation
  float64 x
  float64 y
  float64 z
  float64 w
```

http://wiki.ros.org/msg

Services

- Topics are useful for constantly streaming data, but if we have some code that only needs to be called occasionally a node can set up a **service**.
- Service operate on a **request & reply** model - one client node sends a request to the service, the service performs the task, and then returns a reply to the client.
- Useful for offloading intensive processing to a more powerful computer; however note - **not** used if the request might take a long time to process, use **actionlib** for this (but not today)
- To call a service from the terminal, use **rosservice...**



http://wiki.ros.org/Services

Topic Remapping

- Sometimes two nodes may have matching data types for publishing/subscribing, but incorrect topic names.
- Use the ":=" operator to change the name of a topic that a node may subscribe or publish to.
- E.g. to change the teleop node command topic:

```
rosrun turtlesim turtle_teleop_key /turtle1/cmd_vel:=/cmd_vel
```

http://wiki.ros.org/Remapping%20Arguments

Record & Playback Data

- Sometimes it's useful to be able to record the data on your topics for later analysis or playback.
- **rosbag** is a tool we can call from the terminal to record and playback topic data with **bag files**.
- To record activity across all topics, simply run **rosbag record -a**
- To record specific topics, use **rosbag record /topic1 /topic2** etc.
- To playback use **rosbag play filename.bag**
- There are many options with this tool, best see http://wiki.ros.org/rosbag

—

1. Start roscore
2. Start a turtlesim node (rosrun…)
3. Call the service /spawn to create a new turtle in your window (rosservice call …)
4. Call two teleop nodes and take control of each turtle
5. Record the command inputs you send, and then play them back (rosbag)

Tips: Use the [Tab] button and autocompletion to see what arguments a service requires to run

Exercise 2: Calling services and remapping topics

```
viki@c3po:~$ rosservice call /spawn "x: 1.0
y: 1.0
theta: 0.0
name: 'my_turtle'"
```

1. roscore
2. rosrun turtlesim turtlesim_node
3. rosservice call /spawn 1 1 0 'my_turtle' (or use tab completion and fill in the arguments)
4. rosrun turtlesim turtle_teleop_key
5. rosrun turtlesim turtle_teleop_key /turtle1/cmd_vel:=/my_turtle/cmd_vel

Tips: Use the [Tab] button and autocompletion to see what arguments a service requires to run

Exercise 2 Solution: Calling services and remapping topics

Developing Your Own Code

- All of your robot code lives in a ROS **workspace** built using a tool called Catkin (a CMake build system).
- Using Python today; however C++ also widely used, especially for heavy computation work (vision etc.).
- ROS can work with many other languages (Java/Lisp/etc.), but less common and varying degrees of community support.
- Note: If using C++, some additional setup steps are required to get your executables working with ROS tools (rosrun etc.), see official tutorials.

Setting up the ROS Workspace

- In many ROS tutorials, you'll see they work with "catkin_ws", but useful to remember you can name it whatever you like.
- Example procedure:

1. mkdir -p kcl_ws/src
2. cd kcl_ws/src
3. catkin_init_workspace
4. cd ..
5. catkin_make

1. Make a folder kcl_ws and subfolder src
   a. (-p argument lets you create both at same time)
2. Go to the subfolder src we just made
3. Initialise the workspace
4. Go to the parent folder of src (kcl_ws)
5. Build the workspace (should now have folders called build and devel)

http://wiki.ros.org/catkin/Tutorials/create_a_workspace

Create your first ROS Package

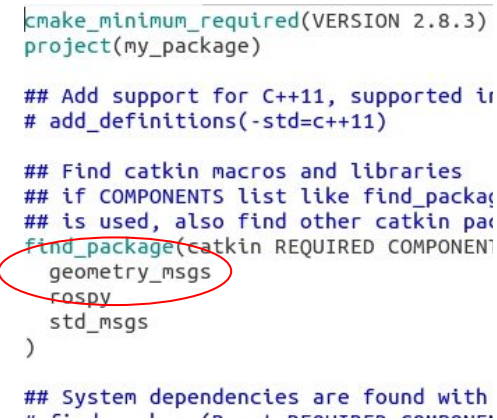
- Example procedure:

1. cd ~/kcl_ws/src
2. catkin_create_pkg my_package rospy std_msgs
3. cd ..
4. catkin_make
5. source devel/setup.bash

1. Go to the src subfolder
2. Create your package using catkin, 1st Argument is the package name, following arguments are dependencies that the package depends on.
3. Go to the parent folder of src (kcl_ws)
4. Build the workspace
5. Source the workspace - cannot use any code inside the workspace without this step, and you need to source your workspace in **every new terminal you open**.

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

Inside your package



- In the package you just created, you will find the files “CMakeLists.txt”, “package.xml”, and the folder “src”.
  - CMakeLists.txt: Used for specifying dependencies, configuring C++ executables, specifying custom message/services/etc.
  - package.xml: Meta information of package, but also specifies dependencies.
  - src: This is where you code goes.
- You can add to this folder structure however you like, but these are the core elements.
- If you are using Python you won’t need to worry so much about CMakeLists.txt for now, but if using C++, you will use this file more for executable configuring and keeping dependencies up to date.

1. In Spyder, create a new python file in ~/kcl_ws/src/my_package/src
2. Ensure that the first line in the file is this: **#!/usr/bin/env python**

```
import rospy

rospy.init_node('my_node')

r = rospy.Rate(1)

while not rospy.is_shutdown():
    print "hello"
    r.sleep()
```

3. Enter this code into the file.
4. After entering this code, you will need to give it *permission to run*. From a terminal,
   a. **cd ~/kcl_ws/src/my_package/src**
   b. **sudo chmod a+x my_node.py**
   c. Enter the admin password
5. Now you must make and source your workspace
   a. **cd ~/kcl_ws**
   b. **catkin_make**
   c. **source devel/setup.bash**
6. Finally, you can run your node
   a. **rosrun my_package my_node.py**

Exercise 3: Your First Node

1. Open the code for the node you just created
2. Add the following lines in **bold**
3. Run the updated node and verify it's working using rostopic echo

```
import rospy
from std_msgs.msg import String # Using a standard ROS message type, String

rospy.init_node('my_node')

# Topic name, topic type, publisher queue size
pub = rospy.Publisher('/my_topic', String, queue_size=10)

r = rospy.Rate(1)

while not rospy.is_shutdown():
    pub.publish("hello") # Send our message to the topic
    r.sleep()
```

Note: Queue size can be important if you have a publisher temporarily producing information faster than a subscriber can read the messages.

If the subscriber lets too many messages build up in the queue, you will start losing information.

Exercise 4: Adding a publisher

http://wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers#Choosing_a_good_queue_size
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

1. Open the code for the node you just created
2. Add the following lines (note, not all lines are shown here)
3. Run the updated node and verify it's working by observing console output and using rostopic info on /my_topic

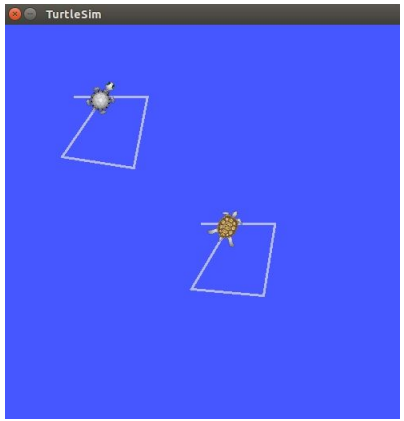
```
def sub_cb(msg): # Callback function that runs everytime a message is received
        print msg.data

rospy.init_node('my_node')

rospy.Subscriber('/my_topic', String, sub_cb) # Subscriber setup
```

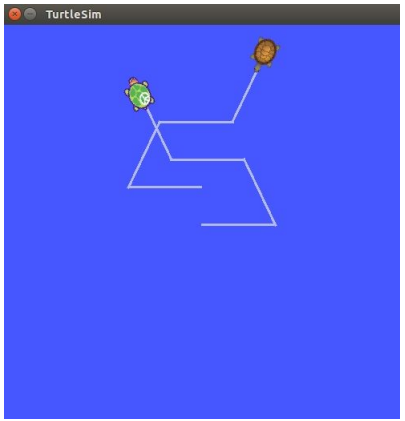
Note: You can communicate data from your callback to other parts of your program using standard python methods, but a commonly used approach is to use global variables.

Exercise 4: Adding a subscriber

http://wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers#Choosing_a_good_queue_size
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

- Call a turtlesim node and spawn in a second turtle.
- Call a turtlesim teleop node.
- Write a node of your own that takes the teleop commands going to the first turtle, and *copies* the input command for the second turtle. **Extra:** Make the second turtle *mirror* the motion of the first turtle.





Tips: Remember to check message types!

Challenge

http://wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers#Choosing_a_good_queue_size
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

‘Pro’-tips

1. Use Terminator (a terminal emulator)
    a. Lets you have multiple terminals open in one window - comes up very often during ROS development and saves headaches
2. Learn to use the terminal
    a. If you aren’t familiar with terminal commands, it is worth spending time becoming familiar with the basics, again comes up often (e.g. ls, mv, rm, etc.)

See https://github.com/aransena/kcl_ros_tut for more details on the material covered, along with links to the main tutorials to work through.

KING’S College LONDON