

ROS: Robot Operating System

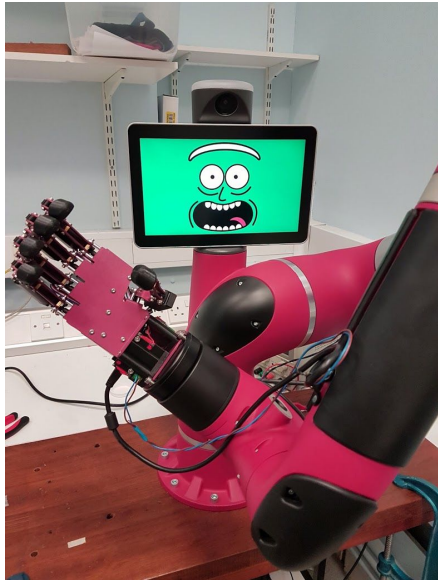
- > Aran Sena
- > Eric Schneider
- > Tsvetan Jivkov



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...





Caveat

- ROS isn't hard real-time.
- Not saying it is slow, just that it does not provide *guarantees on execution timing*, inherently due to being built on top of Linux.
- That said, ROS can interact with real-time subsystems.
- If you think ROS doesn't match up to what you need, there are other middlewares out there
 - [Player](#), [YARP](#), [Orocos](#), [CARMEN](#), [Orca](#), [MOOS](#), and [Microsoft Robotics Studio](#).



Where is ROS?

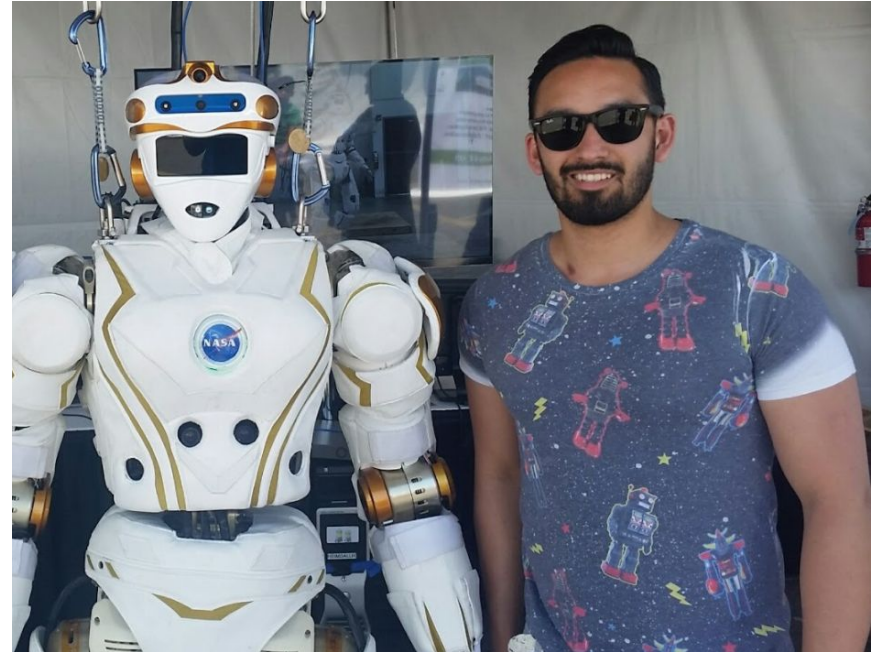
Aran Sena

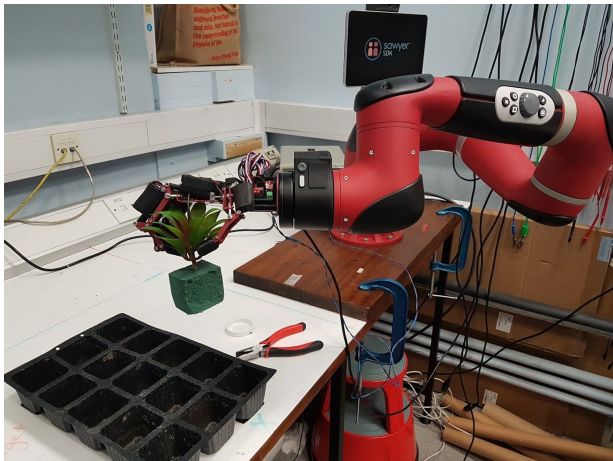
PhD Student, CoRE

Undergrad:	Trinity College Dublin
Industry:	National Instruments
Masters:	Trinity College Dublin
Current:	Robot Learning Lab KCL

Research Area

Learning from Demonstration for Robots in Horticulture.







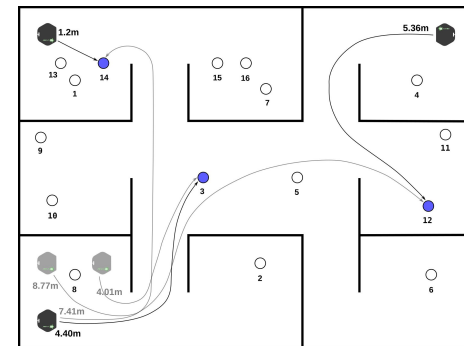
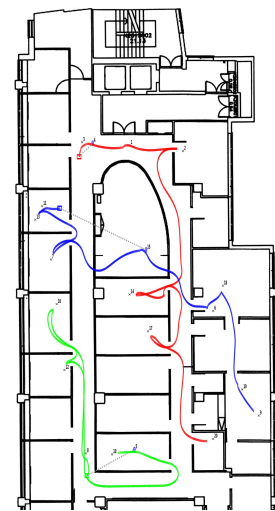
Eric Schneider

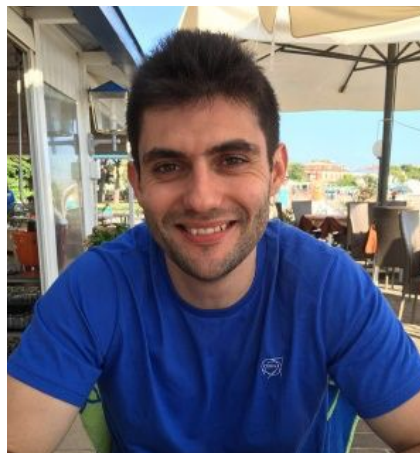
Research Assistant, CoRE

Distant past: Industry
Undergrad: City University of New York
PhD: University of Liverpool
Current: Postdoc at KCL

Research Area

Multi-robot task allocation using market-based methods (e.g., auctions). Experimental work for PhD based entirely on ROS!





Tsvetan Zhivkov

PhD student, CoRE

Undergrad: University of Liverpool

Research Area

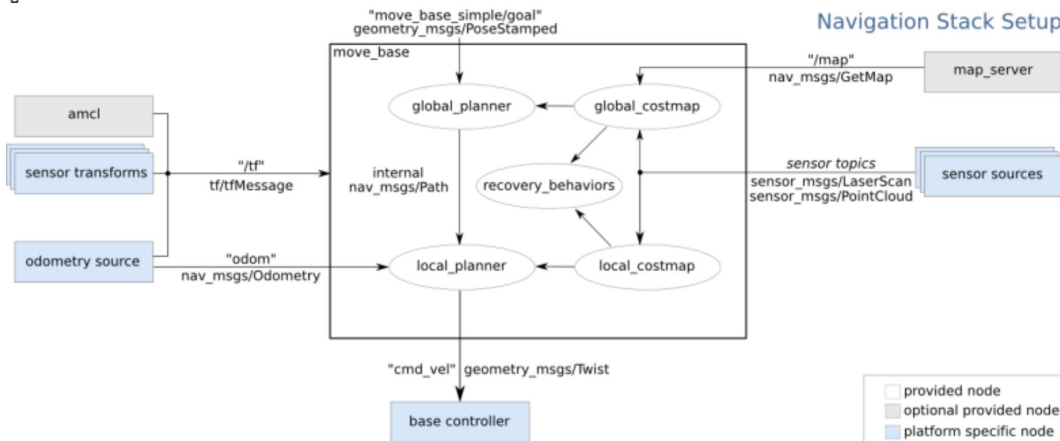
How degrading communication quality effects multi-robot team performance. How communication can influence robot behaviour and inspire formations/strategies in certain domains.



TurtleBot 2 Platform

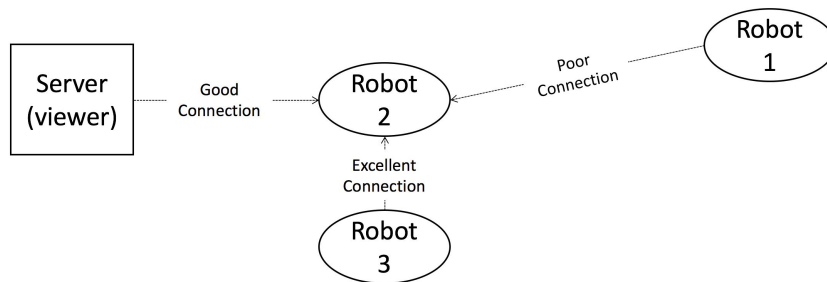
How ROS Helps in Research.

- Contains a set of drivers that allow receiving/sending of sensor data to actuators. (e.g. Turtlebot platform)
- The navigation package (also referred to as the navigation stack).

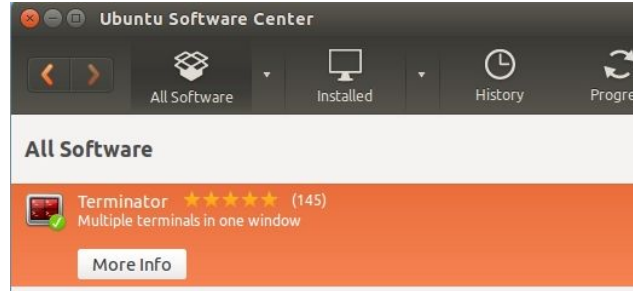


Communication between Components.

- Biggest advantage is the possibility to communicate to multiple devices using ROS. (i.e. Multi-robot communication)
- Allows for much faster prototyping of ideas and design, without starting over or writing new means for communication.
- Decentralized computation and cooperative behavior!

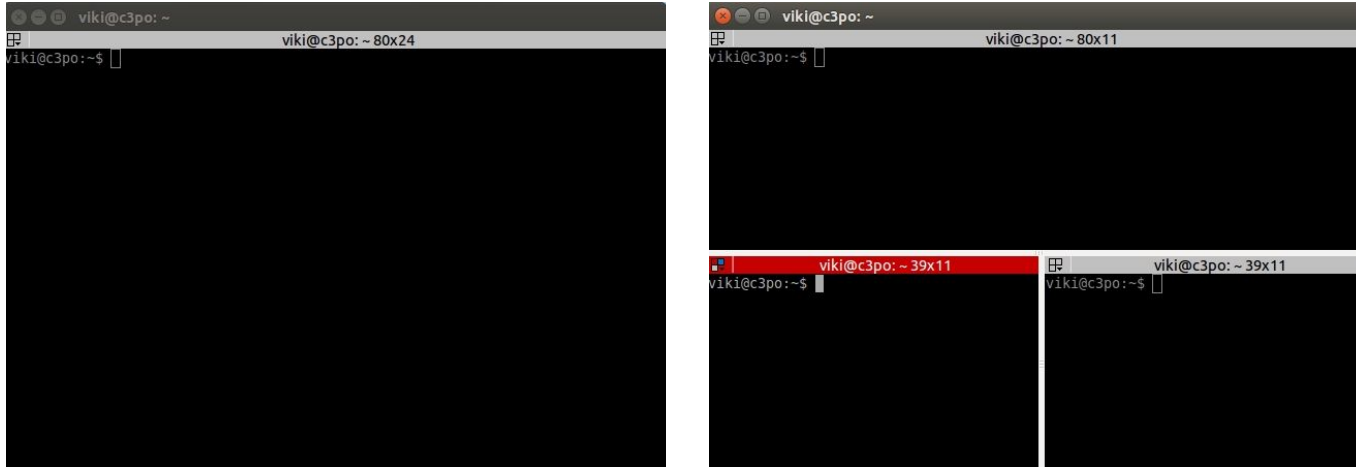


- Open the Ubuntu Software Centre
- Search for "Terminator", choose to install it
- Search for "Spyder", choose to install it (Python IDE, there are better ones out there but this is convenient for now).
- If using the Nootrix VM image, the Admin password is "viki"



Preliminaries

-
- After installing Terminator, use Ctrl+t to bring up a terminal window
 - Use Ctrl+Shift+O to split the window horizontally into subterminals, use Ctrl+Shift+E to split it vertically
 - Use Ctrl+Tab to switch between subterminals (active one highlighted red)



Preliminaries

-
- After installing ROS, you can often install other packages listed on <http://wiki.ros.org> just using:
 - **sudo apt-get install ros-indigo-[name of package to install]**
 - Ensure you have the *turtlesim* packages installed now:
 - **sudo apt-get install ros-indigo-turtlesim**
 - If you are unsure of the name of a package, or if there are multiple parts to install, you can use a * as a wildcard symbol, meaning the system will try install any package that matches the start of the expression, e.g. "ros-indigo-turtles*"
 - To see what packages start with "ros-indigo-turtlesim", you can use tab completion, ie. **sudo apt-get install ros-indigo-turtlesim****[Press Tab]** gives the same result as the above command
 - sudo is appended onto the front of terminal commands to run them as an admin - be very careful with using it if you're not familiar with the terminal.

Preliminaries

Launching ROS

```
viki@c3po:~$ roscore
... logging to /home/viki/.ros/log/423604c6-c86a-11e7-ae3-00
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
* /roscdistro: 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-ae3-000c29051961
process[roscout-1]: started with pid [13021]
started core service [/roscout]
```

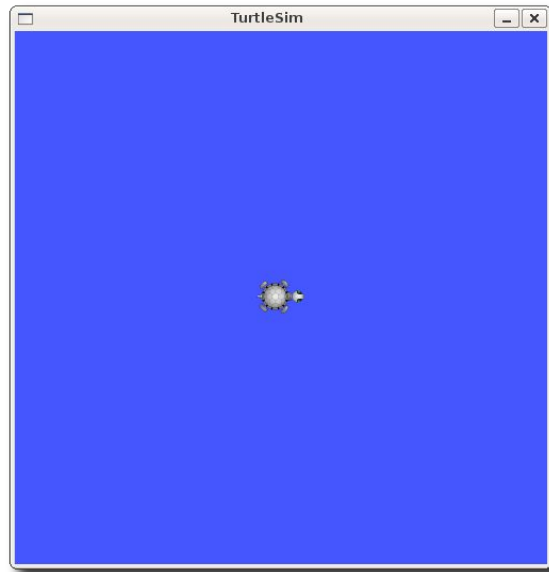
- Before using anything ROS, you must run **roscore**
- roscore starts a collection of critical nodes which manage your whole system, including the **ROS Master** which coordinates all communication.
- When working with one computer, you don't need to worry too much about this.
- When working with multiple 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>

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 run a package in the terminal, use **roslaunch**, e.g.:

```
roslaunch turtlesim turtlesim_node
```



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](#)

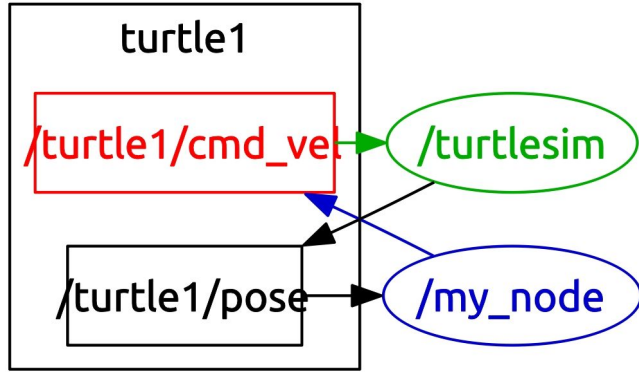
More Actions: ▼

User

[Login](#)

ROS Packages Online: Check the version and look at the tutorials

Nodes & Topics



```
viki@c3po:~$ rostopic 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 **rostopic** and **rostopic** with the argument **list**

-
1. Start roscore
 2. Start a turtlesim node (roslaunch...)
 3. Find a node that you can use to control the turtle (roslaunch...)
 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. `roslaunch turtlesim turtlesim_node`
3. `roslaunch turtlesim turtle_teleop_key`
4. `rqt_graph`
5. `rostopic echo /turtle1/pose`
6. `rostopic info /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

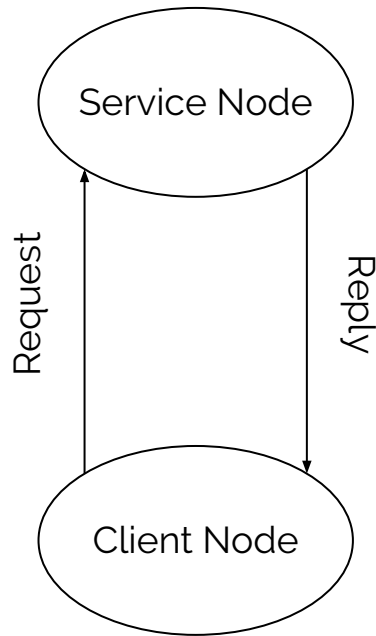
Message Types

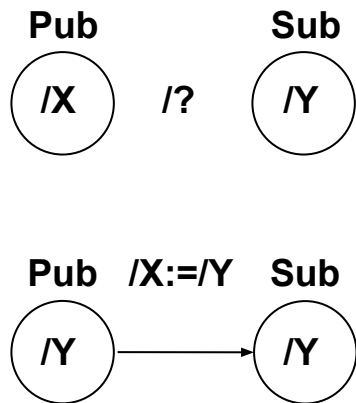
- Publishers and subscribers must have matching **message types**.
- ROS has a number of standard message types for various applications.
 - std_msgs, geometry_msgs, sensor_msgs
- Some message types are basic int/float/string/etc.
- Others provide more advanced **structure** for your application, e.g. a Pose message
- Use **rosmmsg** 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:~$ rosmmsg 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
```

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**.
- Services 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 covering this today)
- To call a service from the terminal, use **rosservice...**





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:

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

Record & Playback Data

- Sometimes it's useful to be able to record the data on your topics for later analysis or playback.
- **rosvbag** 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 **rosvbag record -a**
- To record specific topics, use **rosvbag record /topic1 /topic2** etc.
- To playback use **rosvbag play filename.bag**
- There are many options with this tool, best see <http://wiki.ros.org/rosvbag>

-
1. Start roscore
 2. Start a turtlesim node (roslaunch...)
 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 (roslaunch ...)

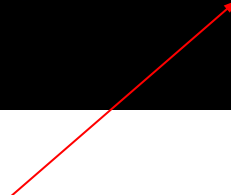
Tips: Use the [Tab] button and autocompletion to see what arguments a service requires to run

Exercise 2: Calling services and remapping topics

—

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

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



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 (roslaunch 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`)

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**.

Inside your package

```
cmake_minimum_required(VERSION 2.8.3)
project(my_package)

## Add support for C++11, supported in
# add_definitions(-std=c++11)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  geometry_msgs
  roscpp
  std_msgs
)

## System dependencies are found with
```

- 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. **`roslaunch 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

-
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')
```

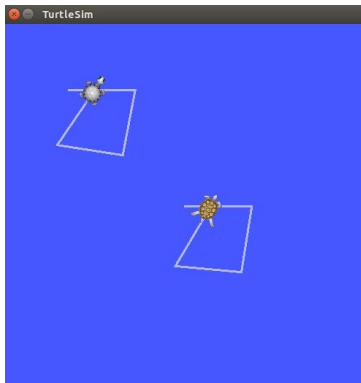
```
rospy.Subscriber('/my_topic', String, sub_cb) # Subscriber setup
```

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.

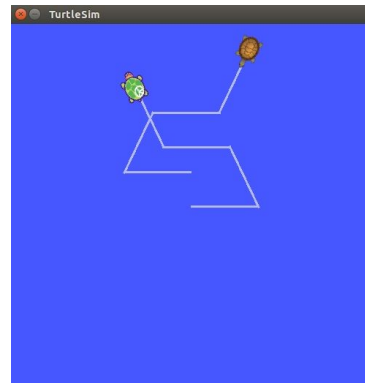
Exercise 4: Adding a subscriber

- 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.

Copying



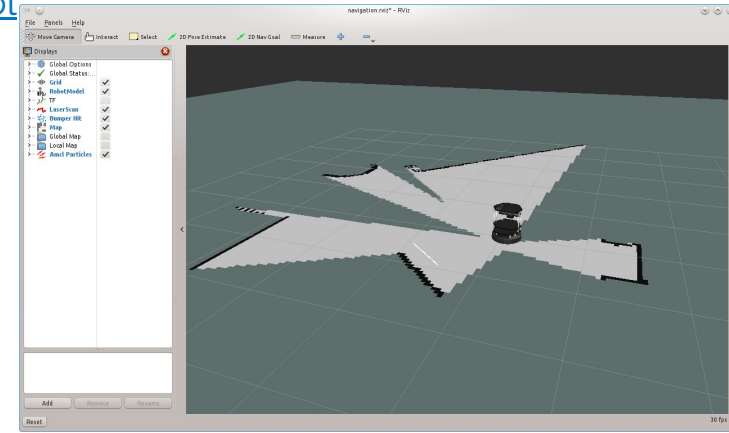
Mirroring



Tips: Remember to check message types!

Challenge

- Learn to use launch files
 - [http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleeRoslaunch)
 - <http://wiki.ros.org/roslaunch#Tutorials>
 - <http://wiki.ros.org/roslaunch/XML>
- Try out more advanced features in ROS, like Gazebo
 - http://wiki.ros.org/turtlebot_gazebo/Tutorials



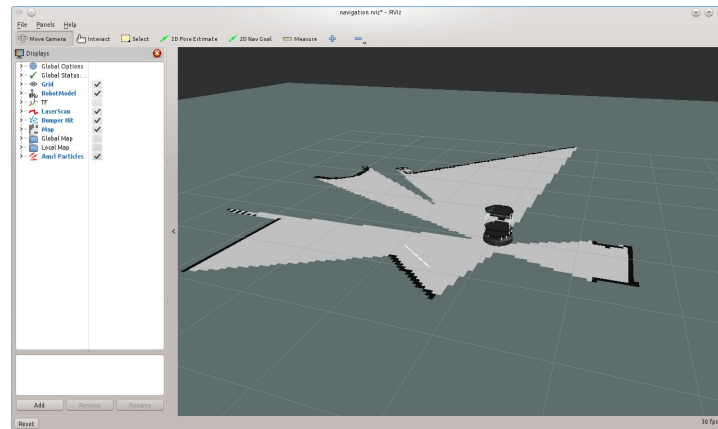
Next Steps

Next Steps

- Learn to use launch files
 - <http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch>
 - <http://wiki.ros.org/roslaunch#Tutorials>
 - <http://wiki.ros.org/roslaunch/XML>
- Try out more advanced features in ROS, like Gazebo
 - http://wiki.ros.org/turtlebot_gazebo/Tutorials

Next Steps

- Learn to use launch files
 - <http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch>
 - <http://wiki.ros.org/roslaunch#Tutorial>
 - <http://wiki.ros.org/roslaunch/XML>
- Try out more advanced features in ROS, like using Gazebo to simulate a turtlebot and build a map
 - http://wiki.ros.org/turtlebot_gazebo/Tutorials
- Figure out how to get ROS onto a Raspberry Pi
- Build your own robot!



'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.

