Robot Operating System (ROS)

Jethro Kuan

October 17, 2019

Outline

What is ROS? [3]

- Meta-operating system, providing low level services:
  - process communication over a network
  - device control
  - hardware abstraction
- Distributed framework of processes

Why use ROS?

- "Lightweight" framework that speeds up large-scale robotic development
- Many libraries developed on top of this framework that can be reused:
  - Physics simulation (Gazebo)
  - Movement + Navigation (ROS navigation)

ROS Concepts i

Computational Graph

- All computation is organized as a peer-to-peer network of communicating processes.

ROS Concepts ii

Nodes

- Processes that perform any form of computation.
- Nodes can communicate with one another.
- Example of nodes:
  - Publish sensor readings
  - Receiving teleop commands and running them
- Written with ROS client libraries (rospy, roscpp)

ROS Concepts iii

Master (Primary) Node

- Provides name registration, node lookup to all nodes in the computational graph.
- Enables communication between nodes.

Parameter Server

- "Distributed" key-value store: all nodes can access data stored in these keys.

ROS Concepts iv

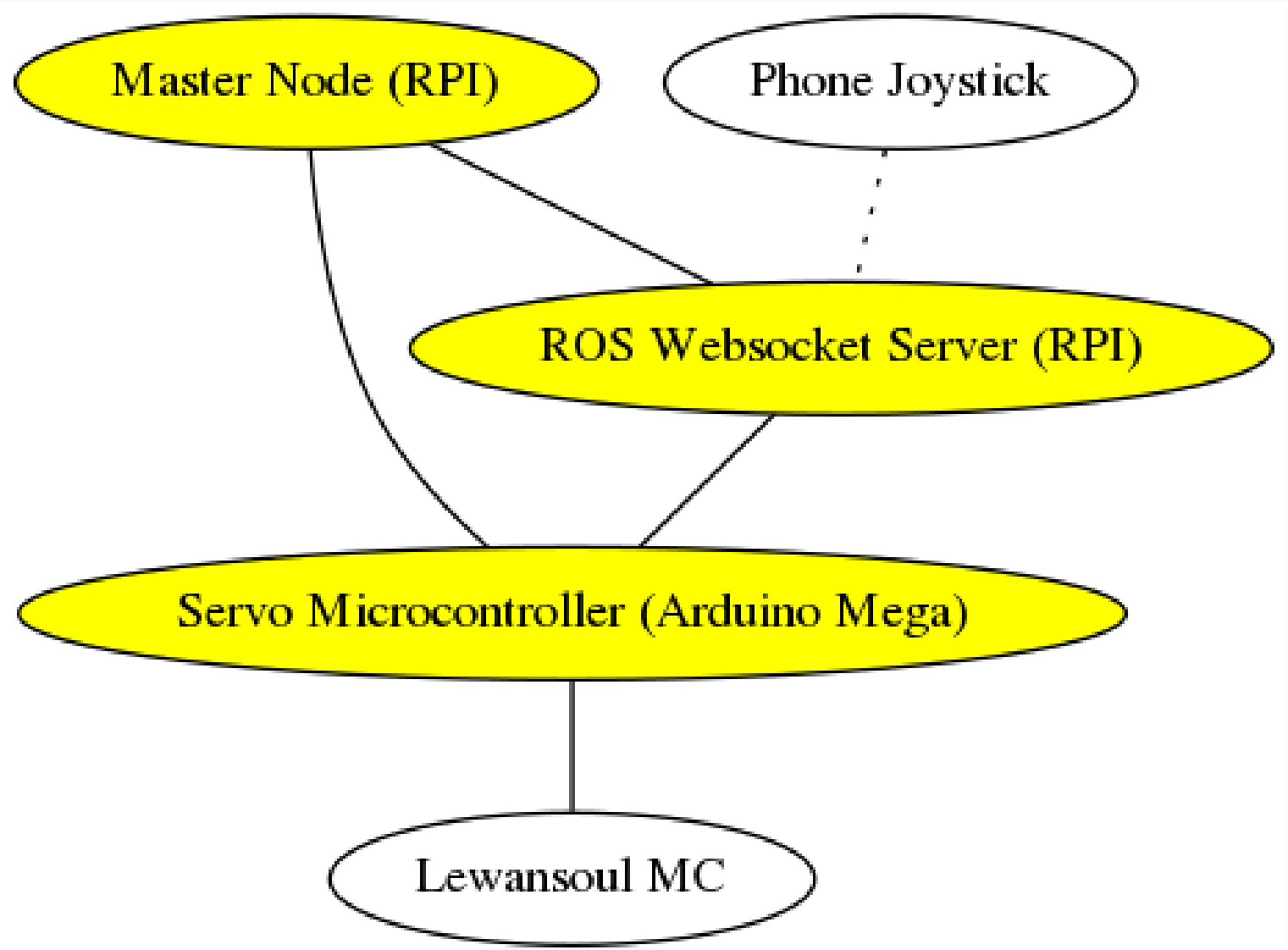
Topics

- Nodes communicating via the publish-subscribe semantics do so by publishing and subscribing to topics.
- Every topic has a name, e.g. `/sensors/temp1`
- No access permissions

Services

- Request-response semantics (think Web servers)
- Requests are blocking

Example Computational Graph



ROS Environment Setup

Here I assume you have the ROS environment set up. If not, see the appendix.

Creating a ROS Workspace

Catkin is ROS' package manager, built on top of CMake.

```
1 mkdir -p ~/catkin_ws/src          # Create the directories
2 cd ~/catkin_ws/                   # Change to the directory
3 catkin_make                       # Initial setup
```

Exploring ROS shell commands [1]

rospack

`rospack find` locates ROS packages.

```
rospack find roscpp # /opt/ros/melodic/share/roscpp
```

roscd

roscd changes you to the directory of the ros package.

```
roscd roscpp
pwd # /opt/ros/melodic/share/roscpp
```

[1] Almost all these commands have tab completion!

Creating a ROS package

We use the convenience script `catkin_create_pkg` to instantiate our package.

```
cd ~/catkin_ws/src
catkin_create_pkg workshop std_msgs rospy roscpp
# Created file workshop/CMakeLists.txt
# Created file workshop/package.xml
# Created folder workshop/include/workshop
# Created folder workshop/src
# Successfully created files in
↪  /home/jethro/catkin_ws/src/workshop. Please adjust the
↪  values in package.xml.
```

What's in a ROS package?

```
workshop
    CMakeLists.txt          # Build instructions
    include                 # For cpp deps, if any
       workshop
    package.xml             # Details about the package
    src                     # Contains source code
```

Starting ROS

We initialize the ROS master node with `roscore`.

```
roscore

# ...
# process[master]: started with pid [16206]
# ROS_MASTER_URI=http://jethro:11311/

# setting /run_id to 05bf8c5e-efed-11e9-957b-382c4a4f3d31
# process[rosout-1]: started with pid [16217]
```

To kill it, press `Ctrl-C` in the same terminal.

ROS Nodes i

rosnode
rosnode let's us inspect available nodes:

```
rosnode list                    # /rosout
rosnode info /rosout
```

What happens if master is not running?

```
rosnode list                    # ERROR: Unable to communicate
↪  with master!
```

ROS Nodes ii

Running a ROS node

A ROS package may contain many ROS nodes.

```
rosrun turtlesim <TAB>
# draw_square        mimic                turtlesim_node
↪  turtle_teleop_key
```

```
rosrun turtlesim turtlesim_node
# [ INFO] [1571214245.786246078]: Starting turtlesim with
↪  node name /turtlesim
# [ INFO] [1571214245.790986159]: Spawning turtle
↪  [turtle1] at x=[5.544445], y=[5.544445],
↪  theta=[0.000000]
```

Exercise: reinspect the node list.

ROS Topics i

Now we have a visual simulation of a turtle. How do we make it move?

```
rosrun turtesim turtle_teleop_key
```

What's going on?

- `turtle_teleop_key` advertises on a ROS topic, and publishes each keystroke:

```
rostopic list
rostopic echo /turtle1/cmd_vel
```

ROS Topics ii

ROS Messages

- ROS messages are pre-defined formats. They are binarized and compressed before they are sent over the wire.

```
rostopic type /turtle1/cmd_vel    # geometry_msgs/Twist
```

ROS Topics iii

Monitoring the Topic

- The rate at which messages is published is good to monitor (in Hz).
- A topic that has too many messages can get congested, and buffer/drop many messages, or congest the ROS network.

```
1 rostopic hz /turtle1/cmd_vel
2 # subscribed to [/turtle1/cmd_vel]
3 # average rate: 13.933
4 # min: 0.072s max: 0.072s std dev: 0.00000s window: 2
```

ROS Topics iv

Rosbag

- A bag is subscribes to one or more topics, and stores serialized data that is received (for logging/replay)

```
1 rosbag record /turtle1/cmd_vel
2 # [ INFO] [1571294982.145679913]: Subscribing to
  ↪ /turtle1/cmd_vel
3 # [ INFO] [1571294982.168808833]: Recording to
  ↪ 2019-10-17-14-49-42.bag
```

ROS Services

- Services allow request-response interactions between nodes.

```
rosservice list
rosservice call /clear
rosservice type /spawn | rossrv show
```

ROS Params

the `rosparams` commandline interface allows us to store and manipulate data on the ROS Parameter server. [2]

```
rosparam set            # set parameter
rosparam get            # get parameter
rosparam load           # load parameters from file
rosparam dump           # dump parameters to file
rosparam delete         # delete parameter
rosparam list           # list parameter names
```

[2] can also be done programatically

When do we use topics?

Previously we looked at ready-made ROS packages and how they used topics and services. Now, we'll write our own publisher and subscriber.

The pubsub interface is useful in situations where a response for each request is not required:

- Sensor readings
- Log info

A Simple Publisher

We use `rospy`, but `roscpp` is fine as well. We create a new file in our workshop package `workshop/src/talker.py`:

```
1  #!/usr/bin/env python
2  import rospy
3  from std_msgs.msg import String
4
5  pub = rospy.Publisher('my_topic', String, queue_size=10) #
   ↪  initializes topic
6  rospy.init_node('talker', anonymous=True) # required to
   ↪  talk to Master
7
8  while not rospy.is_shutdown():
9      pub.publish("Hello")
```

Executing the Publisher Node

We need to make our Python file executable:

```
chmod +x talker.py
```

```
rosrun workshop talker.py
```

Exercise: monitor the output. What's wrong? (hint: Hz)

Setting the rate of publishing

We use the `Rate` object, and the `rate.sleep()` to set the rate of publishing:

```
1 rate = rospy.Rate(10)              # 10 hz
2 # ...
3 rate.sleep()
4 # ...
```

Good Practice

We often wrap all our logic in a function, and catch the `ROSInterruptException` exception:

```
1  #!/usr/bin/env python
2  import rospy
3  from std_msgs.msg import String
4
5  def talker():
6      pub = rospy.Publisher('my_topic', String,
       ↪  queue_size=10) # initializes topic
7      # ...
8
9  try:
10     talker()
11 except rospy.ROSInterruptException:
12     pass
```

Exercise: Write a time publisher (5 minutes)

Goal: publish the current date-time onto a topic `/datetime`.

Hint: Python has a `datetime` library.

Subscriber

We create a listener in `workshop/src/listener.py`

```
1  #!/usr/bin/env python
2  import rospy
3  from std_msg.msg import String
4
5  def echo(data):
6      print(data.data)
7
8  def listener():
9      rospy.init_node("listener", anonymous=True)
10     rospy.Subscriber("my_topic", String, echo)
11     rospy.spin() # prevents python from exiting
12
13 listener()
```

Summary

```
1  rospy.init_node(name)                # create node
2  rospy.Publisher(topic_name, msg_type) # create publisher
3  rospy.Subscriber(topic_name, msg_type, callback) # create
   ↪  subscriber
4  rospy.Rate(10)                       # rate object
5  rospy.spin()                         # spin
```

Msg and Srv

msg message files that define the format of a ROS message. These generate source code for different languages (think Apache Thrift, Protobuf).

srv describes a service (request/response)

Creating a msg

```
1 mkdir -p workshop/msg
```

Create a file `workshop/msg/Num.msg`:

```
1 int64 num
```

Compiling the msg i

In package.xml:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

In CMakeLists.txt:

```
find_package(catkin REQUIRED COMPONENTS
   roscpp
   rospy
   std_msgs
   message_generation
)

```

Compiling the msg ii

```
8  catkin_package(
9    ...
10   CATKIN_DEPENDS message_runtime ...
11   ...)
12
13 add_message_files(
14   FILES
15   Num.msg
16 )
17
18 generate_messages()
```

Compile the message:

Compiling the msg iii

```
cd ~/catkin_ws
catkin_make
catkin_make install
# ...
# [100%] Built target workshop_generate_messages_cpp
# [100%] Built target workshop_generate_messages_py
# [100%] Built target workshop_generate_messages_eus
# Scanning dependencies of target
↪  workshop_generate_messages
# [100%] Built target workshop_generate_messages
```

Using the ROS msg

```
1  rosmsg list                          # ... workshop/Num
2  rosmsg show workshop/Num             # int64 num
```

Creating a ROS srv

```
mkdir -p workshop/srv
```

In `workshop/srv/SumInts.srv`:

```
int64 a
int64 b
---
int64 sum
```

Compiling the ROS srv

Since srv files are also compiled, the setup is similar to compiling msgs.

Writing a Service Node

We can create a server that uses the service file we defined earlier:

```
1  #!/usr/bin/env python
2  from workshop.srv import SumInts, SumIntsResponse
3  import rospy
4
5  def handler(req):
6      return SumIntsResponse(req.a + req.b)
7
8  def sumints_server():
9      rospy.init_node("sumints_server")
10     s = rospy.Service("sumints", SumInts, handler)
11     rospy.spin()
12
13 sumints_server()
```

Writing a Client i

```
1  #!/usr/bin/env python
2  import sys
3  import rospy
4  from workshop.srv import SumInts
5
6  def sumints_client(x, y):
7      rospy.wait_for_service('sumints')
8      try:
9          sumints = rospy.ServiceProxy('sumints', SumInts)
10         resp1 = sumints(x, y)
11         return resp1.sum
12     except rospy.ServiceException, e:
13         print "Service call failed: %s"%e
```

Writing a Client ii

```
14
15  x = int(sys.argv[1])
16  y = int(sys.argv[2])
17  print "%s + %s = %s"%(x, y, sumints_client(x, y))
```

```
1  rosrun workshop sumint_client.py 1 2
2  # 1 + 2 = 3
```

Exercise: Time Service (15 minutes)

Write a service that:

- requests nothing
- responds with the current time

Write a client that sends the request and prints this response.

What's Next?

- Run a simulator, model the robot using URDF
- Look at community ROS packages

  tf2 maintain robotic coordinate frames (pose estimation)

  gmapping/slam etc. navigation
- Look at ROS 2

Common Pitfalls

1. Not sourcing your `devel/setup.bash`:

```
source devel/setup.bash
```

1. This is necessary to make available all the C++ and python ROS packages that you have built
2. I recommend using direnv, and sourcing it every time you enter the Catkin workspace.

ROS Installation

Ubuntu
Follow the instructions on ROS Wiki. [4]

VM
Download the VM image and load it.


nil.
Ros/tutorials/writingpublishersubscriber(python) - ros wiki.
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29, nil.
Online; accessed 17 October 2019.


nil.
Ros/tutorials/writingserviceclient(python) - ros wiki.
http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29, nil.
Online; accessed 17 October 2019.


nil.
Ros/introduction - ros wiki.

http://wiki.ros.org/ROS/Introduction, nil.
Online; accessed 15 October 2019.

nil.

melodic/installation/ubuntu - ros wiki.

http://wiki.ros.org/melodic/Installation/Ubuntu, nil.
Online; accessed 16 October 2019.