Programming ROS nodes

Ivan Marković Matko Orsag Damjan Miklić
(Srećko Jurić-Kavelj)

University of Zagreb, Faculty of Electrical Engineering and Computing,
Departement of Control and Computer Engineering

2012

University of Zagreb
Faculty of Electrical Engineering
and Computing

Overview (Review)

By now, you should know:

- What nodes, topics, messages are
- What a ROS package is, and how it's laid out, and how to create one
- How to use ROS tools: roscd, rosnode, rostopic, rosmsg, rxgraph

In this lecture

You will remind yourself how to program real robot nodes in Python and learn how to use launch files.

Running a simulated robot

- Get STDR Simulator from apt packages

```
$sudo apt-get install ros-$ROS_DISTRO-stdr-simulator
```

- Launch the simulation

```
$ roslaunch stdr_launchers
 server_with_map_and_gui_plus_robot.launch
```

Running a simulated robot

- Get STDR Simulator from apt packages

```
$sudo apt-get install ros-$ROS_DISTRO-stdr-simulator
```

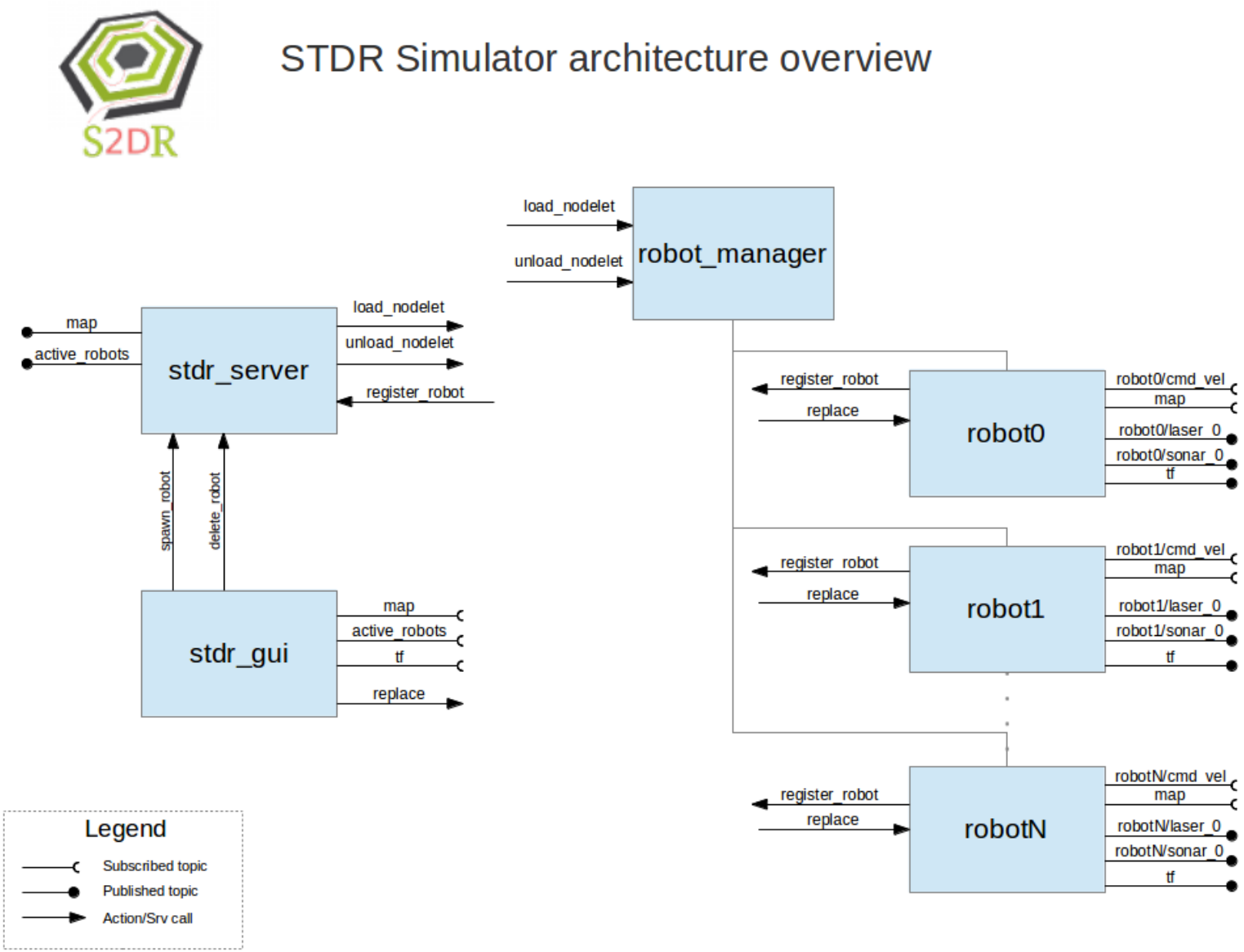
- Launch the simulation

```
$ roslaunch stdr_launchers
 server_with_map_and_gui_plus_robot.launch
```

- Examine the ROS computational graph

```
$ rosrun rqt_graph rqt_graph &
$ rostopic list
$ rostopic info /cmd_vel
$ rosmsg info Twist
```

STDR Simulator architecture



STDR Simulator architecture

- server: Implements synchronization and coordination functionalities of STDR Simulator.
- robot: Provides robot, sensor implementation, using nodelets for server to load them.
- parser: Provides a library to STDR Simulator, to parse yaml and xml description files.
- gui: A gui in Qt for visualization purposes in STDR Simulator.
- msgs: Provides msgs, services and actions for STDR Simulator.

Publishing velocity commands (1/2)

- Create a package

```
$ catkin_create_pkg ros_liv_wander rospy geometry_msgs
sensor_msgs stage
$ mkdir ros_liv_wander/scripts
$ gedit ros_liv_wander/scripts/vel_pub.py
```

- Write the publisher code! Check publisher.py from turtlecontrol package :)

Publishing velocity commands (1/2)

- Create a package

```
$ catkin_create_pkg ros_liv_wander rospy geometry_msgs
sensor_msgs stage
$ mkdir ros_liv_wander/scripts
$ gedit ros_liv_wander/scripts/vel_pub.py
```

- Write the publisher code! Check publisher.py from turtlecontrol package :)

```
#!/usr/bin/env python
import roslib
import rospy
from geometry_msgs.msg import Vector3, Twist

def publish_velocities(v, w):
  tw = Twist(Vector3(v,0,0), Vector3(0,0,w))
  pub.publish(tw)
  rospy.sleep(1.0)
```

Publishing velocity commands (2/2)

```
if __name__ == '__main__':

  pub = rospy.Publisher('cmd_vel', Twist)
  rospy.init_node('vel_pub')

  v = 0.5; w = 0.5
  try:
  while not rospy.is_shutdown():
    publish_velocities(v, w)
    except rospy.ROSInterruptException:
  pass
```

Running the velocity command publisher

```
$ chmod +x cmd_vel.py
$ rosrun ros_liv_wander cmd_vel.py
```

Assignment

Modify the code, so that velocities are passed to the node as command-line arguments. (Hint: You will need the argv object from the sys module)

Assignment

Modify the velocity command publisher, so that robot motion can be controlled from the keyboard. (Hint: The fastest way to do this is by utilizing turtle_teleop_key from turtlesim package.)

Launch files: running multiple nodes in one command

- Create a launch file

```
$ mkdir ros_liv_wander/launch
$ gedit ros_liv_wander/launch/server_with_map_and_gui_plus
_robot_with_keyboard.launch
```

```
<launch>
<launch><!-- standard XML blocks -->
<!-- We can start other launch files -->
<include file="$(find stdr_launchers)/launch/server_with_map_
and_gui_plus_robot.launch" />
<!-- We can start different nodes -->
<node type="turtle_teleop_key" pkg="turtlesim"
name="robot_teleop">
        <!-- remaping in launch files -->
        <remap from="turtle1/cmd_vel" to="robot0/cmd_vel"/>
</node>
</launch>
```

Namespace: the correct way of running multiple nodes

- turtle1 and robot0 are actually namespaces of robots
- Changing the namespace of a node is an easy mechanism for integrating code, as all names within the node (name, topics, etc) will be rescoped.
- For this feature to work properly, it's important that your program avoids using global names and instead uses relative and private names.

```
<!-- Turtlesim uses global name turtle1/cmd_vel -->
<node type="turtle_teleop_key" pkg="turtlesim"
name="robot_teleop" ns="robot0">
        <!-- because of global name we have to remap
        turtle1/cmd_vel to a private name cmd_vel-->
        <remap from="turtle1/cmd_vel" to="cmd_vel"/>
</node>
```

Listening to sensor data (1/2)

- Let's examine the topics published by our robot

Listening to sensor data (1/2)

- Let's examine the topics published by our robot
- Data is received in callback functions

Listening to sensor data (1/2)

- Let's examine the topics published by our robot
- Data is received in callback functions

```
#!/usr/bin/env python
import roslib
import rospy
from sensor_msgs.msg import LaserScan

def scan_callback(scan):
  rospy.loginfo((len(scan.ranges), min(scan.ranges)))
def listener():
  rospy.init_node('laser_listener')
  rospy.Subscriber('scan', LaserScan, scan_callback)
  rospy.spin()

if __name__ == '__main__':
  listener()
```

Running the sensor data listener

- Let's run the listener

```
$ rosrun ros_liv_wander laser_listener.py
```

Running the sensor data listener

- Let's run the listener

```
$ rosrun ros_liv_wander laser_listener.py
```

- Are we getting the expected results?

Running the sensor data listener

- Let's run the listener

```
$ rosrun ros_liv_wander laser_listener.py
```

- Are we getting the expected results? Hint: check rxgraph

Running the sensor data listener

- Let's run the listener

```
$ rosrun ros_liv_wander laser_listener.py
```

- Are we getting the expected results? Hint: check rxgraph
- Topics are mapped by name!

Running the sensor data listener

- Let's run the listener

```
$ rosrun ros_liv_wander laser_listener.py
```

- Are we getting the expected results? Hint: check rxgraph
- Topics are mapped by name!
- Run the listener with argument remapping

```
$ rosrun ros_liv_wander laser_listener.py
scan:=laser_0
```

Running the sensor data listener

- Let's run the listener

```
$ rosrun ros_liv_wander laser_listener.py
```

- Are we getting the expected results? Hint: check rxgraph
- Topics are mapped by name!
- Run the listener with argument remapping

```
$ rosrun ros_liv_wander laser_listener.py
scan:=laser_0
```

- Run the listener with argument remapping

```
$ rosrun ros_liv_wander laser_listener.py
scan:=laser_0 __ns:=robot0
```

Running the sensor data listener

- Let's run the listener

```
$ rosrun ros_liv_wander laser_listener.py
```

- Are we getting the expected results? Hint: check rxgraph
- Topics are mapped by name!
- Run the listener with argument remapping

```
$ rosrun ros_liv_wander laser_listener.py
scan:=laser_0
```

- Run the listener with argument remapping

```
$ rosrun ros_liv_wander laser_listener.py
scan:=laser_0 __ns:=robot0
```

- Do not use /laser_0 since / makes it global i.e. root

Running the sensor data listener (launch file)

```
<launch><!-- standard XML blocks -->
<!-- We can start other launch files -->
<include file="$(find stdr_launchers)/launch/server_with_map_
and_gui_plus_robot.launch" />
<!-- We can start different nodes -->
<node type="turtle_teleop_key" pkg="turtlesim" name=
"robot_teleop" ns="robot0">
        <!-- remaping in launch files -->
        <remap from="turtle1/cmd_vel" to="cmd_vel"/>
</node>
<node type="laser_listener.py" pkg="ros_liv_wander"
    name="laser_listener" ns="robot0" output="screen">
        <!-- output allows ROS_INFO in terminal -->
        <remap from="scan" to="laser_0"/>
</node>
</launch>
```

Letting the robot out to play :)

Assignment (Homework)

1. Write a launch file for your previous homework problem (hw3) so that it starts all the nodes necessary to play the game of turtlecatch.
2. Write a node that subscribes to topic laser_0 and drives the robot safely through the map (publishes on topic cmd_vel). The robot should use laser_0 data in order to figure out obstacles in front of it $min\_ahead = min(scan.ranges[left\_angle : right\_angle]) < min\_ahead\_tresh$. If *min_ahead* falls below threshold, the robot should steer left or right (you choose), otherwise it should drive straight. Figure out what parameters *left_angle*, *right_angle* and *min_ahead_tresh* work best for your robot.

Useful links

- http://www.ros.org/wiki/ROS/Tutorials/WritingPublisherSubscriber
- http://www.ros.org/wiki/geometry_msgs
- http://www.ros.org/wiki/sensor_msgs
  http://www.ros.org/wiki/gmapping
- http://www.ros.org/wiki/dynamic_reconfigure