Introduction to Robot Operating System

Autonomous Robotics Labs

Labs 02 (26.2./28.2. 2019)

Outline

- Quick refresher from last week
- Running ROS from Singularity container
- ROS Introduction part II
  - Python libraries
  - ROS Parameter
  - Launch files
  - Logging, recording & playing messages, debugging
  - Message files
  - ROS Service
  - ROS Actions (briefly)
- Advanced examples
- Conclusion

Quick ROS refresher

- ROS:
  - Robot Operating System
  - distributed system
  - contains a lot of “stuff” useful for developing SW for robotic applications:
    various tools (*packages*) & libraries for many robotics-related problems, SW management tools, visualization & debugging tools
- ROS Components
  - Master
  - Node
  - Topic
  - Message
  - Parameters
  - Service
  - Action
- ROS Workspace

ROS in Singularity

- Running singularity image:

```
$ singularity shell --nv /path/to/image
```

- (--nv needed for GUI, e.g. rviz)
- Existing images should be in: */local/singularity_images*, e.g.:

```
$ singularity shell --nv
/local/singularity_images/ros-melodic-robot.simg
```

- Automatic download of image from docker:

```
$ singularity shell docker://ros:melodic-robot-bionic
$ source /opt/ros/melodic/setup.sh
```

ROS Intro

...continued

ROS Python libraries

- **rospy**
  - the single most important library in Python when working with ROS
  - handles most of the interaction with ROS
- rosnode, rosservice, rosparam, rostopic,...
  - libraries that mostly do the same as their command line counterparts
- std_msgs, sensor_msgs, geometry_msgs, ... (*http://wiki.ros.org/common_msgs*)
  - libraries containing the standard set of messages
- rosbag
  - library for working with bag files
- tf
  - library for working with transformations (more about tf next week)
- actionlib
  - library for working with actions

rospy: bread and butter

```
init_node('<node_name>', [anonymous=True])
spin()
is_shutdown()

rate = Rate(<hz>); rate.sleep()

get_param('<param_name>', default=<def_val>)
set_param('<param_name>', <val>)
has_param(..)

Publisher('<topic_name>', <message_type>)
Subscriber('<topic_name>', <message_type>, <callback_function>)


loginfo, logwarn, logerr, logfatal, logdebug

 get_time()
wait_for_message; wait_for_service
```

ROS Parameter

- You can provide configuration arguments to nodes via command line:
  `$ rosrun <package> <node> arg1:=value1 arg2:=value2`
  - good for some basic stuff
  - can get messy with more complex systems
- Parameter server
  - stores configuration parameters in a network-accessible database
  - parameters are stored as key-value pairs (dictionary)
  - nodes can write or read parameters
  - parameter reusability
  - tracking who defines which parameter
  - changing parameters
- In **rospy**:

  “/global_parameter”
  “∼private_parameter”

ROS Parameters: `console commands`

`$ rosparam`

`list`	**lists** all created **parameters**
`get <param_name>`	**returns** current **value of the** specified **parameter**
`set <param_name> <value>`	**sets** the **value of the** specified **parameter**
`load <filename>`	**loads** parameters **from a file** (YAML)
`dump <filename>`	**writes** parameters **into a file**
`delete <param_name>`	**deletes** a parameter

Launch files

- XML files that automatize the start-up of nodes
- Launching of multiple nodes
- Name remapping
- (Better) argument handling
- Also offer some runtime node handling (e.g. restaring)
- And much more...
- In general, this is how ROS nodes should be started (most of the time)

Launch file elements

<launch> root element

<node> node element specifying a node that will be run, multiple nodes can be specified

- :name name of the node (any but unique)
- :ns (different) namespace
- :pkg package containing the executable
- :type executable name
- :output screen (i.e. console) or log (file)
- :respawn if true, the node will respawn if terminated
- :required if true, all other nodes in the launch file will terminate when this node is terminated

<arg> custom input argument that can be specified via console

- :name unique argument name
- :default default value that will be used if no value is supplied

- Specifying values for arguments:

```
$ roslaunch <pkg> <launch_file> <arg_name>:=<value>
```

- usage inside the launch file (including the brackets):
  `($ arg <arg_name>)`

Launch file elements

<include> element for including other launch files

:file the launch file name

- usage:
  file="($find <package_name>)/<launch_filename>"

<arg name="<arg_name>" value="<value> />" supply arguments to the external launch file

<param> sets up a ROS parameter

:name name of the parameter

:value value to be assigned

<group> element grouping

:ns executes content in a specific namespace

:if content executes if condition holds true

Logging

- Unified way of logging (textual) outputs from nodes
- Can be printed onto the screen (console) or into a files
- Levels of severity:

  Debug
  Info
  Warn
  Error
  Fatal

- These are just messages, i.e. nothing else happens

```
rospy.logdebug()
rospy.loginfo()
rospy.logwarn()
rospy.logerr()
rospy.logfatal()
```

Bagfiles

- Recordings of ROS sessions (messages)
- Record a session:

```
$ rosbag record [-O <output_filename>] [-a] <topic_name1>
<topic_name2> ...
```

- `-a` flag records messages from all topics
- The file name is optional, default (current datetime) is used if none is specified
- Play messages from an existing bag:

```
$ rosbag play <bag_filename> [-s <start_time>] [-r <rate>]
[-l] [--clock]
```
(*rosparam set use_sim_time true*)

- `-l` flag will loop the playback
- Information about an existing bag (topics, message counts, etc...):

```
$ rosbag info <bag_filename>
```

- More options: `$ rosbag help`
- Playing/recording bag with a GUI: `$ rqt_bag`

Debugging

`rqt`	GUI with many plugins
`rqt_graph`	shows the topology of ROS components
`rqt_console`	better way of reading log messages
`roswtf`	the first question that pops into your mind when ROS is misbehaving...

Message files

- Structure:

  `<field_type> <field_name>` simple field, e.g.: `int8 someInteger`

  `<field_type>[`*N*`] <field_name>` array of basic type with length *N*, e.g.: `float32[3] xyzCoords`

  `<field_type>[] <field_name>` array of basic type with variable length, e.g.: `float64[] randomPoints`

  `<constant_type> <constant_name>=<constant_value>` an array of basic type, e.g.: `int8 luckyNumber=3`

- Example of a message file:

```
Header header
bool isMsgUseful
int8[] anotherOneBytesTheDust
uint8[3] colorRGB
float32 randomNumber
string definitelyUsefulDescription
```

Creating a message

- Create *msg* folder and a message file:
  ```
  $ mkdir msg && cd msg
  $ vim AGoodMessage.msg
  ```
  ```
  Header header
  string message
  float32 number
  ```
- Modify the package manifest
  <build_depend>message_generation</build_depend>
  <exec_depend>message_runtime</exec_depend>
- Modify the CMakeLists.txt
  - find_pacakge REQUIRED COMPONENTS + std_msgs & message_generation
  - add_message_files + AGoodMessage.msg
  - catkin_package CATKIN_DEPENDS + message_runtime
  - generate_messages DEPENDENCIES + std_msgs
- Build & source

Using the message – publisher

```
#!/usr/bin/env python2
import rospy
from incredible_package.msg import AGoodMessage
from numpy.random import rand

if __name__ == '__main__':
    rospy.init_node('publisher')
    rate = rospy.Rate(1)
    publisher = rospy.Publisher('messages',
            AGoodMessage, queue_size=10)

    while not rospy.is_shutdown():
        msg = AGoodMessage()
        msg.header.stamp = rospy.Time.now()
        msg.message = 'Hello'
        msg.number = rand()
        publisher.publish(msg)
        rate.sleep()
```

- Remember publisher from previous labs

Using the message – listener

```
#!/usr/bin/env python2
import rospy
from incredible_package.msg import AGoodMessage

def callback(msg):
    rospy.loginfo('Received a message at {}. Message:\
        "{}" and a random number: {}'.format(
                msg.header.stamp.secs, msg.message,
                msg.number
        ))

if __name__ == '__main__':
    rospy.init_node('listener')
    listener = rospy.Subscriber('messages',
                AGoodMessage, callback)
    rospy.spin()
```

ROS Service

- Topics are useful for asynchronous communication and processing
  - data are sent as they are generated and processed as they arrive
  - e.g. sensors
  - not useful in case of e.g. “on-demand” computations (needless waste of bandwidth)
- Synchronous communication model
  - request $\rightarrow$ response
  - RPC
- Useful for infrequent data transmissions & relatively quick operations
- Input arguments can be provided

ROS Service: `console commands`

`$ rosservice`

`list`	**lists** all currently available **services**
`info <service_name>`	**displays** some **info** about the service: node that will handle the service call, URI, service "message" type, input arguments
`type <service_name>`	**shows** the **service** definition file **name**
`find <service_type>`	**prints services that use** the specified **service type** definition
`call <service_name> <args>`	**calls** the **service** with the specified arguments

ROS Service files: `console commands`

```
$ rossrv
```

- ros**service** deals with currently available **services**
- ros**srv** deals with "**srv**" files

`show <message_name>`	**shows** service **input and output fields** (i.e. contents of the srv definition file)
`list`	**lists** all available **service definitions**
`package <package_name>`	**lists** all **service** definitions **in a** specific **package**
`packages`	**lists** all **packages containing** (definitions of) any **service**

Keen viewers will notice similarities with *rosmsg*.

Service files

- Uses the same data types as messages
- Similar definition files
  request fields
  - - -
  response fields
- Request and response fields separated with three dashes without spaces

- E.g.:

```
field_typeA request_field_name1
field_typeB request_field_name2
field_typeC request_field_name3
- - -
field_typeD response_field_name1
field_typeE response_field_name2
```

Creating services

```
mkdir srv && cd srv
```

- Create the service files
  - Random.srv:

    ```
    - - -
    uint8 randNumber
    ```

  - MatOp.srv:

    ```
    string op
    float32[9] matrixIn
    float32[] args
    - - -
    float32[9] matrixOut
    ```

Modify the configuration

- Most of the changes are similar to the messages
- We already did that, so we don't need to
  - but **remember to do it** in your project
- The only difference is in CMakeLists.txt:

```
add_service_files(
    FILES
    Random.srv
    MatOp.srv
)
```

- Build & source the WS

Creating service server

```
#!/usr/bin/env python2
import rospy
import numpy as np
from incredible_package.srv import Random, RandomResponse

def randGen(request):
        randNumber = np.random.randint(256)
        rospy.loginfo('Somebody called me and I sent'
              'this number: {}'.format(randNumber))
        return RandomResponse(randNumber)

if __name__ == "__main__":
        rospy.init_node('random_server')
        s = rospy.Service('generate_number', Random, randGen)
        rospy.spin()
```

Creating service caller

```
#!/usr/bin/env python2
import rospy
from incredible_package.srv import Random,
                RandomRequest, RandomResponse
from time import sleep

SERVICE_NAME = 'generate_number'

if __name__ == '__main__':
        rospy.wait_for_service(SERVICE_NAME)
        generatorService = rospy.ServiceProxy(SERVICE_NAME, Random)
        while not rospy.is_shutdown():
                response = generatorService.call()
                print ('Called the {} service, received '
        'response: {}'.format(SERVICE_NAME, response.randNumber))
        sleep(1)
```

- Always use "*wait_for_service*", don't just hope for synchronized start-up sequence!
- No need to initialize the node if you just want to call the service
  - but then you can't use `Rate`, logging and other ROS goodies
- Empty call for service without the request part

ROS Actions

- Uses for *services*:

  `Node A requires some information or (almost) instant operation from Node B. Node A calls a service on Node B and (almost) immediately receives a response.`

- Uses for ***actions***:

  `Node A wants Node B to perform some (time consuming) operation. Node A initiates an action of Node B and is notified about the progress until the operation is complete. It is possible to cancel the operation.`

- Actions can be (and actually are) implemented using only services
- "***actionlib***" provides functionality to use the actions
- Custom actions can be generated the same way as messages and services

Advanced examples – Service and Action

- Following are advanced examples of service and action server
- We won't go through it on the labs but it is recommended that you try it!

Slightly more complex service – server

- Use the same Python script for service server as before
- Add `MatOp`, `MatOpResponse` to the service imports
- Add new service handling function:

```
def matOp(request):
        if request.op == 'translate':
                mat = np.diagflat([1., 1., 1.])
                t = request.args mat[:2, 2] = t
                if request.op == 'scale':
                        s = list(request.args)
                        mat = np.diagflat(s + [1])
                elif request.op == 'rotate':
                        angle = np.deg2rad(request.args[0])
                        mat = np.diagflat([0., 0., 1.])
                        mat[:2, :2] = [[np.cos(angle), -np.sin(angle)],
                       [np.sin(angle), np.cos(angle)]]

        matrixOut = mat.dot(np.reshape(request.matrixIn, (3, 3)))
        response = MatOpResponse(matrixOut.flatten())
        return response
```

- Advertise the new service:

```
rospy.Service('mat_op', MatOp, matOp)
```

- Restart the service server node

Slightly more complex service – client

- Here we will need a new file for the script
- Import the service classes `MatOp` and `MatOpRequest` and also `rospy`, `numpy`, and `pylab` (or anything else for plotting)
- Define plotting function (so we save some space and time):

```
def plot(data, subplot):
    pylab.subplot(subplot)
    pylab.scatter(data[0, :], data[1, :])
    pylab.xlim(-150, 150), pylab.ylim(-150, 150)
```

- Define a function that will be calling the service (for convenience):

```
def transform(matrix, operation, *args):
    request = MatOpRequest(operation, matrix.flatten(), args)
    response = matService.call(request)
    return np.reshape(response.matrixOut, (3, 3))
```

- Define the service name, wait for it and create proxy, when it shows up:

```
SERVICE_NAME = 'mat_op'
rospy.wait_for_service(SERVICE_NAME)
matService = rospy.ServiceProxy(SERVICE_NAME, MatOp)
```

- Initiate some random points and the transformation matrix (identity for now):

```
n = 7
points = np.vstack((np.random.randint(0, 50, size=(2, n)), np.ones((1, n))))
matrix = np.diag([1, 1, 1])
```

Slightly more complex service – client

- First, let's plot the original points:

```
plot(points, '151')
```

- Append a transformation to the matrix using the service:

```
matrix = transform(matrix, 'rotate', 45)
```

- Apply the transformation to the points and plot them:

```
plot(matrix.dot(points), '152')
```

- Now apply several more transformations to your liking
  - don't forget to plot the points after each transformations (trust me, it's better that way)

```
matrix = transform(matrix, 'translate', 30, -30)
plot(matrix.dot(points), '153')

matrix = transform(matrix, 'rotate', -45)
plot(matrix.dot(points), '154')

matrix = transform(matrix, 'scale', 2.5, -0.5)
plot(matrix.dot(points), '155')
```

- And show the plots:

```
pylab.show()
```

Creating Action definition file

```
$ roscd incredible_package
$ mkdir action && cd action
```

- Create the action file NumberCrunching.action and fill with the following content:

```
float32[] data
int32 repetitions
uint8 sleepTime
---
float64 sum
uint64 totalSleepTime
---
uint32 numbersCrunched
uint32 numbersToBeCrunched
```

- Modify the *CMakeLists.txt*
  - find_package + actionlib_msgs
  - generate_messages DEPENDENCIES + actionlib_msgs
  - catkin_package CATKIN_DEPENDS + actionlib_msgs
  - add_action_files DIRECTORY action
  - add_action_files FILES + NumberCrunching.action
- Modify the package.xml:

```
<depend>actionlib_msgs</depend>
```

Actions server I

- Imports:

```
import rospy
import actionlib
from incredible_package.msg import NumberCrunchingAction,
     NumberCrunchingGoal, NumberCrunchingResult,
        NumberCrunchingFeedback
import numpy as np
from math import factorial
```

Action server II

```
def crunch(goal):
    numbers = np.array(goal.data)
    result = np.zeros(numbers.shape)
    reps = goal.repetitions
    pause = rospy.Duration(goal.sleepTime)
    pauseCount = 0
    success = True
    for r in range(reps):
        rospy.loginfo('Starting repetition number {}'.format(r))
        for i, num in enumerate(numbers):
            if server.is_preempt_requested():
                server.set_preempted()
                success = False
                break
            result[i] += factorial(num)
            server.publish_feedback()
            rospy.sleep(pause)
            pauseCount += 1
        if not success:
            break
    if success:
        actionResult = NumberCrunchingResult(np.sum(result),
                                             pauseCount * pause.secs)
        server.set_succeeded(actionResult)
        rospy.loginfo('Action successfuly completed')
```

Action server III

```
if __name__ == '__main__':
        rospy.init_node('action_maker')
        server = actionlib.SimpleActionServer('number_cruncher',
                 NumberCrunchingAction, crunch, False)
        server.start()
        rospy.spin()
```

Action client

```
import rospy
import actionlib
from incredible_package.msg import NumberCrunchingAction,
     NumberCrunchingGoal, NumberCrunchingResult
import numpy as np

if __name__ == '__main__':
        rospy.init_node('action_client')
        client = actionlib.SimpleActionClient('number_cruncher',
                                    NumberCrunchingAction)
        client.wait_for_server()

        goal = NumberCrunchingGoal([1, 2, 3], 10, 128)
        client.send_goal(goal)

        client.wait_for_result()
```

What's next?

- ROS tutorials: `http://wiki.ros.org/rospy/Tutorials`
- ...or in general *http://wiki.ros.org/*
  - a lot of helpful tutorials & documentation
- Visualization tools:
  - more *rqt*
  - **rviz**
- Simulation tools:
  - e.g. Gazebo

Thank you for your attention