Introduction to Robot Operating System

Autonomous Robotics Labs

Labs 01 (15.2./18.2. 2021)

ARO Labs

- For details and contacts – please see the course web page: https://cw.fel.cvut.cz/b202/courses/aro/tutorials/start
- There is a guide to get you started with ROS on **a remote faculty computer**: https://cw.fel.cvut.cz/b202/courses/aro/tutorials/remote_access
- Or you can download a *singularity image* (more on that later) and run ROS **locally**: https://cw.fel.cvut.cz/b202/courses/aro/tutorials/ros
- Main assignment:
  - Develop a program for real Turtlebot

- The first 7 labs should give you the basic knowledge needed to do it

ROS in Singularity

- Singularity = software for virtualization via containers (more details in Czech)
- 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.:
- ⟨*distro*⟩ = *melodic* (there are other distros but this one will be used throughout the labs)

$ singularity shell --nv /opt/ros-⟨*distro*⟩-desktop-full.simg

- Automatic download of image from docker:

```
$ singularity shell docker://ros:⟨distro⟩-robot-bionic
```

- Source the configuration script:

$ source /opt/ros/⟨*distro*⟩/setup.sh

Following the labs on your computer

1. Remote connection (a) or on your computer (b):
   a) See the first slide for more details

      ```
      $ ssh student@turtle.felk.cvut.cz
      $ ssh -Y -J <username>@turtle.felk.cvut.cz <username>@<target-computer>
      ```

   b) Download singularity image:

      ```
      $ wget http://ptak.felk.cvut.cz/tradr/share/robolab_melodic.simg
      ```

2. Start the singularity image
   a) On remote host:

      ```
      $ singularity shell --nv /opt/singularity/robolab/melodic
      (the arg --nv might not always work - ok if we don't need
      visual output)
      ```

   b) Locally:

      ```
      $ singularity shell --nv <whereever_it_is>/robolab_melodic.simg
      ```

3. Always source the ROS configuration file (same for local & remote):

   ```
   $ source /opt/ros/aro/setup.bash
   ```

Following the labs – example code

4. Create a "workspace" folder (more on that later):

   ```
   $ mkdir catkin_ws
   ```

5. Download the example package:

   ```
   $ wget
   https://cw.fel.cvut.cz/b202/_media/courses/aro/tutorials/incredible_package.zip
   ```

6. Unzip it into the workspace folder:

   ```
   $ unzip incredible_package.zip
   ```

7. How to *build* and use will be explained later

ROS

Very Fast & Furious ROS overview

- What is ROS?
  - Robot Operating System
  - asynchronous data processing (but can also operate in synchronous mode)
  - distributed system (but has a central“node”)
  - 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 and Python

- ROS 1 (Melodic distro) still uses Python 2.7
  - it is possible to make it work with Python 3+ or
  - switch to ROS 2 which uses Python 3+ by default
- In this course we won't do either of those (to keep things simpler – hopefully) and stick with Python 2.7
- Python 2.7 peculiarities:

```
print(7 / 2)   # = 3
print(7 / 2.0)   # = 3.5
```

- Import from __future__:

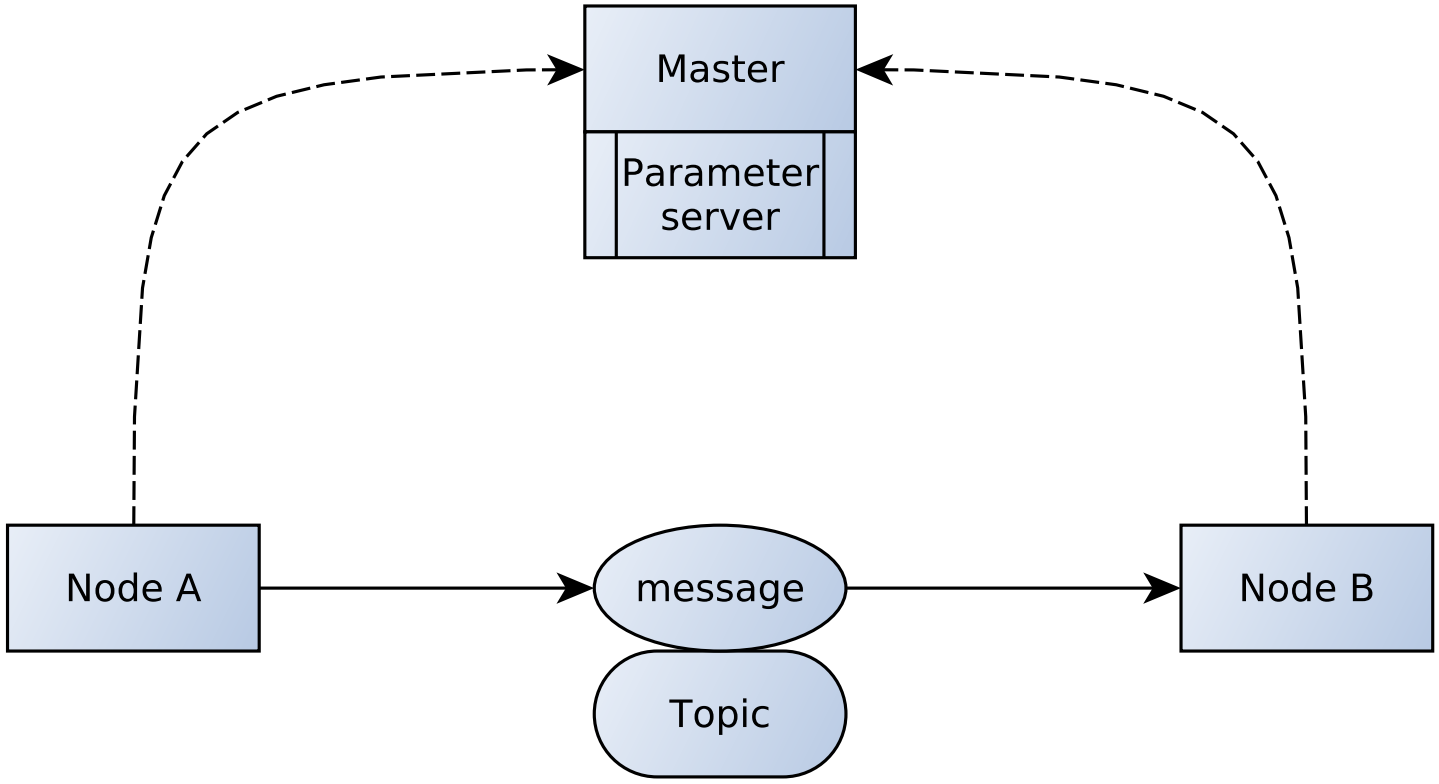
```
from __future__ import print_function, division
```

print_function enforces the use of *print()* as a function

division enables "true" division (instead of integer division)

ROS components

The simplest ROS topology:



ROS Master

- Communication "server" (ROS actually uses P2P model): **mediates communication between nodes**
  - every new node registers with the master (address where other nodes can reach it)
  - tracks topic and service publishers and subscribers
  - data is then sent directly between nodes
- Provides parameter server
- Always needs to be executed before doing anything else with ROS
  - `$ roscore`
  - launch files start master if not running already (I'll explain later...)
  - run it & forget about it (until you get to more advanced stuff)
    - reasons for restarting: new logging session, cleaning up (crashed nodes → *`$ rosnode cleanup`*, renew parameter server)
    - cost of restarting: no new connections can be established → whole system restart likely required
- Can be run on another machine on the network
  - `$ echo $ROS_MASTER_URI`
    `http://localhost:11311`
  - `$ export ROS_MASTER_URI=http://<other_machine>:11311/`

In singularity:

```
$ singularity instance start <image_name>.simg <inst_name>
$ singularity exec instance://<inst_name> bash -c "source
/opt/ros/aro/setup.bash && roscore &"
```

ROS Node

- Basic building block of ROS
- Executable programs and scripts (Python)
  - write a script
  - make it executable:
    `$ chmod u+x <filename>.py` or `$ chmod +700 <filename>.py`
  - run it:
    `$ rosrun <package_name> <node_name>.py`
    - simply executes an executable program or script
- *A node* is an **instance of** a **ROS program**
  - multiple instances of the same program can run simultaneously (with different names)
  - names separated into namespaces (/)
- Nodes can do anything you want them to (or anything you can program them to do)
- Communicate with other nodes via topics and services
  - can be all on one machine or distributed across the Universe, as long as they can all reach the *master* and each other
- Each node can be written in any language with ROS support: C++, Python, MATLAB, Java, Lisp

ROS Node: `console commands`

`$ rosnode`	
`list`	**lists** currently active **nodes**; hint: `<command> | grep <expression>` outputs only lines containing the expression and highlights the occurrences
`info <node_name>`	shows **info about** a specific node: **topics** where the node **publishes** and to which it is **subscribed** to, **services**, and node **address**
`ping <node_name>`	tests node **reachability** and response time
`machine [machine_uri]`	**lists machines** with nodes connected to the master **or nodes** running **on a** specific **machine**
`kill <node_name>`	does what it says on the cover...

Help will always be given to those who ask for it:

- `$ rosnode help`
- `$ rosnode <command> -h`

Or in general:

- `$ ros<whatever> help`
- `$ ros<whatever> <some_sub_command> -h`

And use TAB key for command completion!

- Trivia: Every time someone does not use command completion a cute bunny eats a fluffy unicorn! And bunnies have a lethal allergy to unicorn fur!

ROS Topic

- Communication channels used by the nodes to send and share information
- Publisher & Subscriber model
  - every node can publish or subscribe/listen to a topic
- Each topic has a specific data type that can be sent over it

ROS Topic: `console commands`

`$ rostopic`

`list`	**lists existing topics**; existing topic = any topic that was registered with the master, i.e. existing does not mean active (useful to know when debugging); use grep...
`info <topic_name>`	prints **info about** a specific topic: **nodes publishing** in the topic, **subscribed** nodes and **type of message** that can be transferred via the topic (data type)
`hz <topic_name>`	shows **publishing rate** of a topic (better than echo if you just want to see whether something is being published over a topic)
`echo <topic_name>`	**writes out messages** transmitted over a topic (useful for debugging of topics with low rate and small messages); specific parts of the message can be printed by appending"/<msg_part>/..." `-noarr` flag will suppress printing of arrays (e.g. images that can "flood" the console)
`type <topic_name>`	prints the **type of the messages** transmitted via the topic
`bw <topic_name>`	**bandwidth** used by the topic, i.e. the amount of data transmitted over it per second (on average) – useful to check when sending a lot of data
`pub <topic_name> <message_type> <msg>`	can be used to **publish a message** over a topic when debugging – obviously, only usable for topics with simple messages
`find <message_type>`	**lists** all **topics that use the** specified **message type**

ROS Message

- Data structures used to send data over topics
  - simple: bool, int<N>, uint<N>, float<N>, string, time, duration ($N \in \{8, 16, 32, 64\}$~variable bit size)
  - complex: composed of simple types, can contain other message types and a header
- Message header

`seq`	sequence number – unique ever-increasing ID
`stamp`	message timestamp – epoch seconds & nanoseconds
`frame_id`	frame ID – frame associated with the message

  - `$ rostopic echo /<some_interesting_topic>/header` – will display just the headers of the messages
- Messages are defined in "message files"

ROS Message: `console commands`

`$ rosmsg`

`show <message_name>`	**shows message fields** (msg definition file)
`list`	**lists** all available **message types**
`package <package_name>`	**lists** all **message types in a** specific **package**
`packages`	**lists** all **packages containing** (definitions of) any **messages**

ROS Packages

- ROS files are organized into packages
- Structure of a package:

`<some_package>`

`[src]/package_name/` source code – scripts; normal "Pythonic" code structure

`[scripts]` usually (non-Python/non-C++) scripts or (standalone) executables (e.g. for rosrun)

`[launch]` launch files

`[config]` configuration files, yaml param files for param server

`[include]` additional libraries; include headers for C++

`[msg]` message definitions

`[srv]` service definitions

`[action]` action definitions

`CMakeLists.txt` CMake config file (used by catkin)

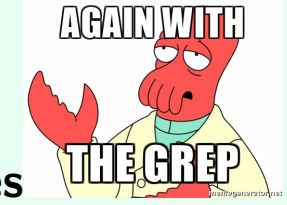
`package.xml` package manifest – catkin/ROS package config file

`logs` build logs

ROS Packages: `console commands`

`$ rospack`

`list`	**lists all** currently available **packages**
`find <message_name>`	prints **path to** a specific **package**



`$ roscd <package_name>` – *cd* into a package

`$ rosls <package_name>` – *ls* a package directory content

`$ rosed <package_name>/<some_file>` – launch a text editor and open the specified file in it (a quick way to adjust small details in a file while debugging)

Workspace

Workspace

- Collection of folders with related ROS files
- Source files, definitions, configuration files, scripts, and other files are organized into packages
- Compilation done **only** via the ROS build system (catkin tools)

ROS Build system

- **catkin**
- a.k.a. *catkin command line tools*
  https://catkin-tools.readthedocs.io/en/latest/cheat_sheet.html
- Extension of CMake – can build libraries, executables, ... (C++)
  - collection of CMake macros and Python scripts
- Auto-generates message/service/action related functions based on their definitions

`init`	initializes a workspace in the current folder
`config`	show current WS configuration (additional args to change the current config)
`create pkg <package_name>`	creates a new package (in the current folder); additional args to provide package dependencies, author, description, ...
`build [package_name]`	builds the current WS/package
`clean [package_name]`	cleans catkin products (build, devel, logs)

- Building a WS with catkin creates these folders in the WS:
  - `build` build targets
  - `devel` (as in "development") – contains setup script
  - `logs` build logs

Creating a workspace

- **Create** folder and *cd* into it
  `$ mkdir example_ws && cd example_ws`
- Create **src** folder
  `$ mkdir src`
- **Init** the workspace
  `$ catkin init`
- **Build** the WS (builds just the catkin tools)
  `$ catkin build`
- Look at it (just to make you feel happy)
  `$ ll` or `$ ls -la` (if the first command does not work
- Go into the src folder
  `$ cd src`

Creating a package

- Create a package
  `$ catkin create pkg incredible_package --catkin-deps rospy`
- CD into the package
  `$ cd incredible_package`
- Check and modify the manifest
  `$ vim package.xml` (or just use GUI based editor)
- Check the CMakeLists.txt (just look at it for now)
- Create a src folder (if it does not exist)
  `$ mkdir src/`

Creating a node

- Fire up your favorite editor and create publisher.*py*:

```
#!/usr/bin/env python2
import rospy
from std_msgs.msg import Float32
from numpy.random import rand

if __name__ == '__main__':
    rospy.init_node('publisher')
    rate = rospy.Rate(2)
    publisher = rospy.Publisher('random',
                Float32, queue_size=10)
    while not rospy.is_shutdown():
        publisher.publish(rand())
        rate.sleep()
```

- Make executable

  `chmod u+x publisher.py`
- Build & source

  `$ catkin build`

  `$ source ~/example_ws/devel/setup.bash`

Creating another node

- *listener.py*

```
#!/usr/bin/env python2
import rospy
from std_msgs.msg import Float32

def callback(msg):
    print('Received a message: {}'.format(msg))
    # rospy.loginfo('Received a message:\
    # {}'.format(msg))

if __name__ == '__main__':
    rospy.init_node('listener')
    listener = rospy.Subscriber('random',
                Float32, callback)
    rospy.spin()
```

You first ROS package

- Run the nodes and observe the beauty of messages being transmitted:

```
$ roscore
```

```
$ rosrun my_package publisher.py
```

```
$ rosrun my_package listener.py
Received a message: data: 0.312089651823
Received a message: data: 0.984019577503
Received a message: data: 0.142692148685
Received a message: data: 0.230828240514
Received a message: data: 0.27526524663
```

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 between coordinate systems (very important in ROS)
- 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 (parameters can be configured via a launch file instead)
- 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 file
- Levels of severity:

  Debug
  Info
  Warn
  Error
  Fatal

- These are just messages, i.e. nothing else happens (e.g., logerr() does not raise or handle an error, it can just report it)

```
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...

Thank you for your attention