Lesson 3: Starting with ROS programming (Part 1)

Jonathan Cacace

jonathan.cacace@unina.it

PRISMA Lab
Department of Electrical Engineering and Information Technology
University of Naples Federico II

www.prisma.unina.it

OUTLINE

In this lesson we will develop ROS nodes to test the plumbing (and communication) capabilities of ROS:

- Publisher/Subscriber example
- ROS Service example

Installing ROS

ROS can be installed using APT tool:

- By default, ROS software is not present in the APT stack
- Configure APT to accept software from additional repositories

Installing ROS

ROS can be installed using APT tool:

- By default, ROS software is not present in the APT stack
- Configure APT to accept software from additional repositories
- Update your `sources.list.d` in the `/etc1/apt/` folder
- Update your APT repository
- Install one of the available ROS versions
- `http://wiki.ros.org/melodic/Installation/Ubuntu`

Environment Configuration

Load the setup file included in the installation directory of ROS.

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

For example you can use the `roscd` command to move in your ROS workspace

```
$ roscd
$ pwd
$ /opt/ros/melodic
```

Environment Configuration

Load the setup file included in the installation directory of ROS.

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

This directory is owned by the super user so you should create your own workspace in the user space:

```
$ cd ~
$ mkdir -p ros_ws/src
$ cd ros_ws/src
$ catkin_init_workspace
$ cd ..
$ catkin_make
```

Environment Configuration

- `catkin_make`: compilation command
  - Must be ran in the root of your ROS workspace
  - This command compile the whole workspace (all the packages contained into the ROS workspace)
- The workspace contains three directories:
  - `src` directory you must create or download new ROS packages. If a package is not placed there, it will not be compiled
  - `build` directory instead contains the compilation file
  - `devel` folder contains the compiled libraries

Environment Configuration

This workspace must be your default workspace!

- `roscd` command must bring you in the workspace just created
- If you type `roscd` command now, you will be moved in the `/opt/ros/VERSION/` directory (in the super user space)
- You need to source the `setup.bash` file contained in the `devel` folder.

  ```
  $ echo "source ~/ros_ws/devel/setup.bash" >>
   ~/.bashrc
  ```

- Open a new terminal or source the modified `bashrc` file load the new configuration. Test it with the `roscd` command.

ROS Package

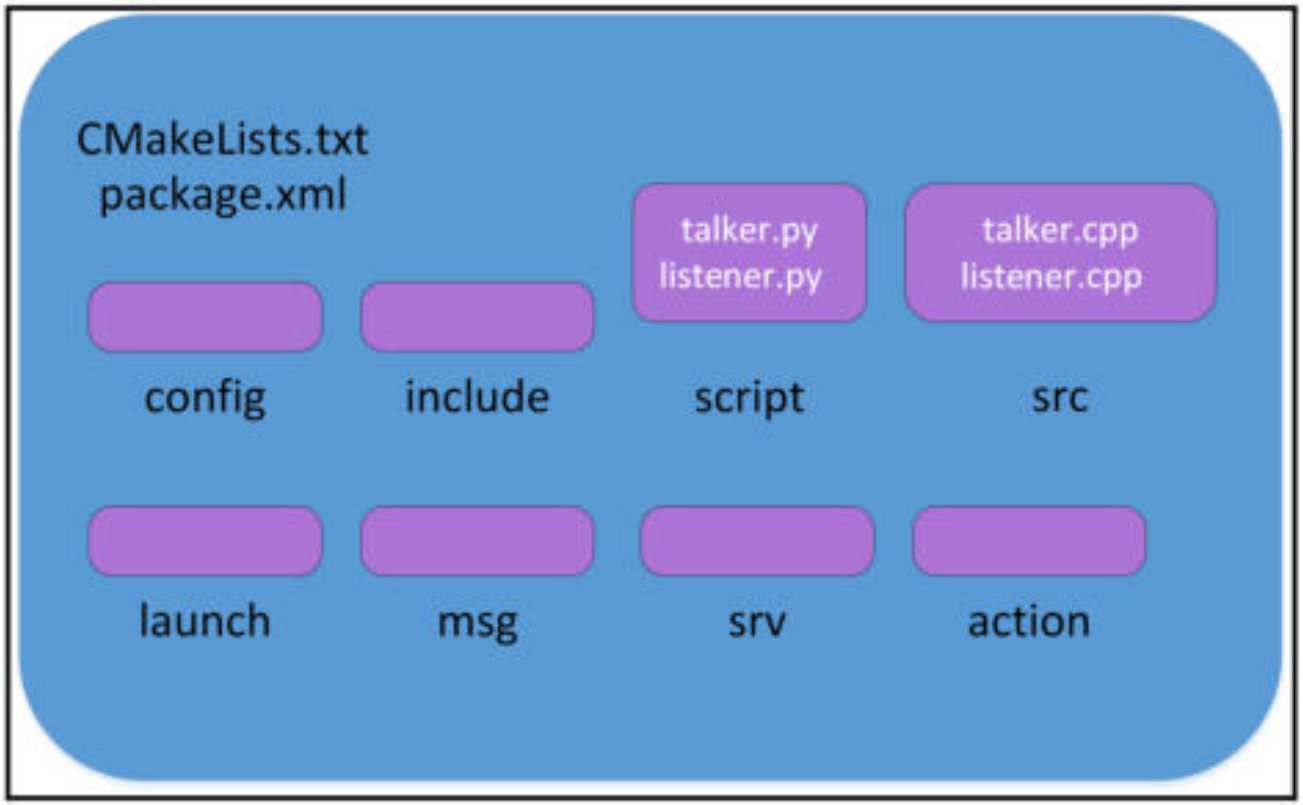
- Create a ROS package:
- A ROS package could contain different executable (ROS nodes)

  ```
  $ catkin_create_pkg package_name [dep1] ... [depN]
  ```

- Packages must be placed in the `src` directory of ROS workspace, otherwise they will not be compiled!!

ROS Package

The shape of a ROS package is a directory with the following sub-directories:



Example 1: pub/sub

- Create a ROS package with two nodes, a publisher and a subscriber:

  ```
  $ catkin_create_pkg ros_topic roscpp std_msgs
  ```

  - `roscpp`: C++ implementation of ROS. It provides APIs to C++ developers to make ROS nodes
  - `std_msgs`: standard ROS messages (int, string, array, ...)
- Navigate the new package:

  ```
  $ roscd ros_topic
  ```
- Create a new package:

  ```
  $ touch src/ros_publisher.cpp
  ```
- Goal: Publish an integer value on a topic called `/numbers`.

Example 1: pub/sub

```
1  #include "ros/ros.h"
2  #include "std_msgs/Int32.h"
3  #include <iostream>
```

Example 1: pub/sub

```
1   int main(int argc, char **argv) {
2     ros::init(argc, argv,"ros_topic_publisher");
3     ros::NodeHandle nh;
4     ros::Publisher topic_pub =
5     nh.advertise<std_msgs::Int32>("/numbers",10);
```

ROS messages

- ROS communication relies on a set of standard and custom data structures called ROS messages
- The types of data are described using a simplified message description language called ROS messages
- The message definition consists in a typical data structure composed by two main types: fields and constants

ROS messages

`geometry_msgs::PoseStamped` is used to share the pose of an object:

```
1  std_msgs/Header header
2    uint32 seq
3    time stamp
4    string frame_id
5  geometry_msgs/Pose pose
6    geometry_msgs/Point position
7      float64 x
8      float64 y
9      float64 z
10   geometry_msgs/Quaternion orientation
11     float64 x
12     float64 y
13     float64 z
14     float64 w
```

Example 1: pub/sub

The inbuilt tools called `rosmsg` is used to get information about ROS messages. Here are some parameters used along with `rosmsg`:

```
$ rosmsg show [message]
$ rosmsg list
$ rosmsg md5 [message]
$ rosmsg package [package_name]
```

Example 1: pub/sub

```
1       ros::Rate rate(10);
2       int count = 0;
3       while (ros::ok()) {
4         std_msgs::Int32 msg;
5         msg.data = count++;
```

Example 1: pub/sub

```
1         ROS_INFO("%d",msg.data);
2         topic_pub.publish(msg);
3         rate.sleep();
4       }
5       return 0;
6     }
```

Example 1: pub/sub

- To compile this package you must edit the `CMakeLists.txt` file
- You must specify the source files that must be compiled and its dependencies

```
1 add_executable(topic_publisher src/ros_publisher.cpp)
2 target_link_libraries(topic_publisher ${catkin_LIBRARIES})
```

- Build `ros_topic` package as follows:

```
$ cd ~/ros_ws
$ catkin_make
```

catkin_make

- `catkin_make` compiles all the package in your workspace.
- Compilation errors cause the failing of the compilation of all packages
- To compile only one package you can use the `DCATKIN_WHITELIST_PACKAGES` argument. With this option, it is possible to set one or more packages enabled to be compiled.

```
$ catkin_make -DCATKIN_WHITELIST_PACKAGES="pkg1,pkg2,..
```

- It is necessary to revert this configuration to compile other packages not specified in the `WHITELIST`.

```
$ catkin_make -DCATKIN_WHITELIST_PACKAGES=""
```

Run pub node

- Activate a `roscore` in your system

  ```
  $ roscore
  ```

- Run the publisher node

  ```
  $ rosrun ros_topic topic_publisher
  ```

The output on the linux shell shows the `INFO` about the integer that are going to be published on `/numbers` topic. We can use two additionally commands to debug and understand the working of the nodes: `rosnode` and `rostopic`.

rosnode command

```
$ rosnode info [node_name]
$ rosnode kill [node_name]
$ rosnode list
```

For example, `rosnode info /ros_topic_publisher` will provide information about the published and subscribed topics:

```
Node [/ros_topic_publisher]
Publications:
* /numbers [std_msgs/Int32]
* /rosout [rosgraph_msgs/Log]

Subscriptions: None
```

This is useful to understand the I/O of a node.

rostopic command

```
$ rostopic list
$ rostopic echo /topic
$ rostopic hz /topic
$ rostopic info /topic
$ rostopic pub /topic message_type args
$ rostopic type /topic
```

Using `rostopic` command you can check which kind of topics are published by the node:

```
$ rostopic list
Output:
/numbers
/rosout
/rosout_agg
```

rostopic command

And check its content:

```
$ rostopic echo /numbers
```

```
Output:
data: 609
---
data: 610
---
data: 611
```

Example 1: pub/sub

- Create a ROS package with two nodes, a publisher and a subscriber
- Goal: Subscribe to the `/numbers` topic and print out its contents

```
$ roscd ros_topic/src
$ touch ros_subscriber.cpp
```

- We will implement a new C++ class called `ROS_SUB`.
  - Multiple functions can easily share information get by topics

Example 1: pub/sub

```
1  #include "ros/ros.h"
2  #include "std_msgs/Int32.h"
3  #include <iostream>
```

Example 1: pub/sub

```
1  class ROS_SUB {
2    public:
3      ROS_SUB();
4      void topic_cb( std_msgs::Int32ConstPtr data);
5
6    private:
7      ros::NodeHandle _nh;
8      ros::Subscriber _topic_sub;
9  };
```

Example 1: pub/sub

```
1  ROS_SUB::ROS_SUB() {
2    _topic_sub = _nh.subscribe ("/numbers", 0, &ROS_SUB::topic_cb, this
         );
3
4  }
```

- `subscribe` function is part of the `ros::NodeHandle` class.
- No class function:

```
1  subscribe( string topic_name, int queue, void*
       function_callback)
```

- Class function:

```
1  subscribe( string topic_name, int queue,void(T::*)(M)
       callback function, T * obj)
```

  - inside a member function we can used the `this` pointer to refer to the invoking object.

Example 1: pub/sub

```
1  void ROS_SUB::topic_cb( std_msgs::Int32ConstPtr data) {
2    ROS_INFO("Listener: %d", data.data);
3  }
```

- What is a `Int32ConstPtr`??

```
1  melodic/include/std_msgs/Int32.h:
2    typedef boost::shared_ptr< ::std_msgs::Int32 const> Int32ConstPtr;
```

Example 1: pub/sub

```
1   int main( int argc, char** argv ) {
2     ros::init(argc, argv, "ros_subscriber");
3     ROS_SUB rs;
4     ros::spin();
5     return 0;
6   }
```

Example 1: pub/sub

- Modify the `CMakeLists.txt` file to add this new node (executable) and compile it with the `catkin_make` command

```
$ roscore
$ rosrun ros_topic topic_publisher
$ rosrun ros_topic topic_subsciber
```

Example 1: pub/sub

```
rostopic info /numbers

Output:
Type: std_msgs/Int32

Publishers:
* /ros_topic_publisher
(http://jcacace-Inspiron-7570:41703/)

Subscribers:
* /ros_subscriber
(http://jcacace-Inspiron-7570:34901/)
```

Example 1: pub/sub

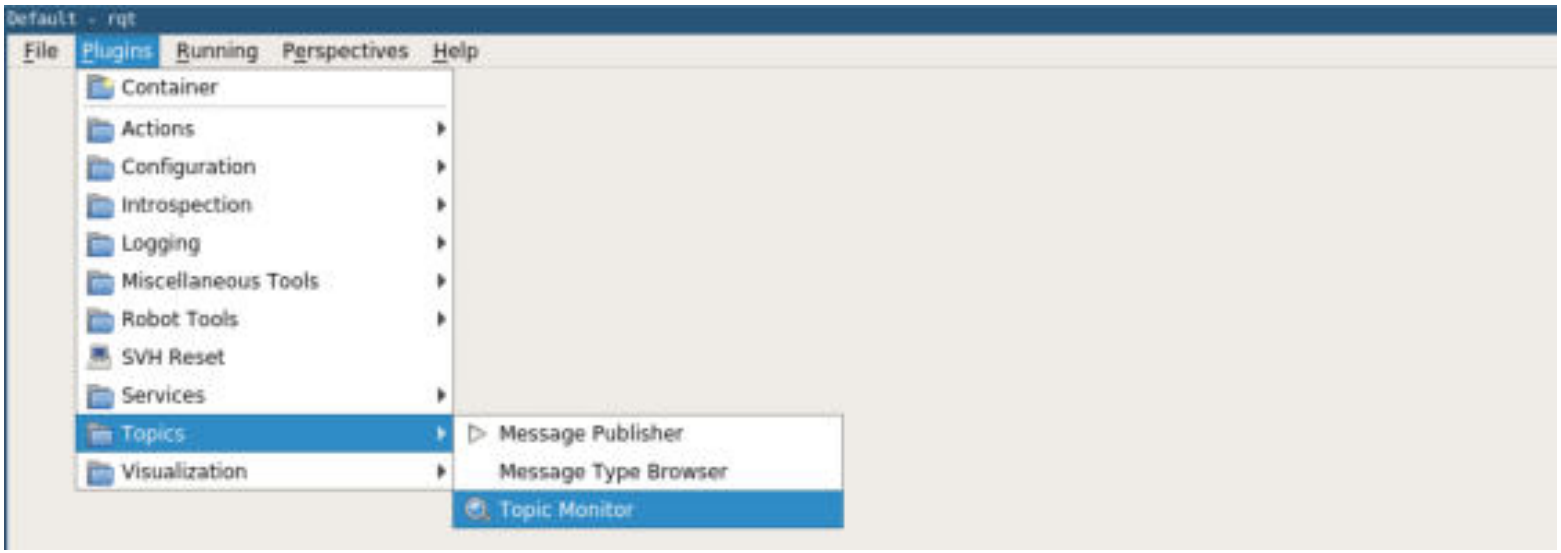
- Sometimes you just need to publish some data used to test a subscriber node
- Implement a ROS node from scratch could be a waste of time
- use `rostopic pub`

```
$ rostopic pub /numbers std_msgs::Int32
"data: 13" -r 10
```

This command publishes the number 13 on the topic `numbers` with a publishing rate of 10 Hz.

- `rqt` is a software framework that implements the various GUI tools as plugins
- To start `rqt` just type this command in your linux shell:

```
$ rqt
```

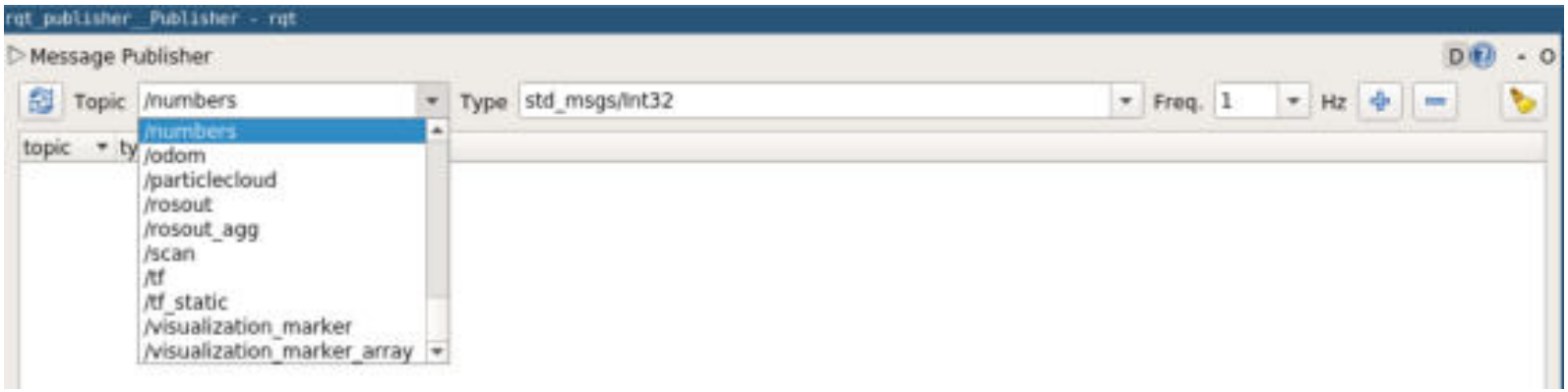


- In the `rqt` start window you can load any desired plugin present in your system.
- You can also add custom plugins.

Topic monitor:

- In the `rqt` start window you can load any desired plugin present in your system.
- You can also add custom plugins.

Topic publisher:



Notice that the same plugin can be launched directly from Linux shell:

```
$ rosrun rqt_publisher rqt_publisher
```

Example 2: ROS Service

- Create a ROS package with two nodes, a service server and a service client
- Goal: client node have to send a string to the server. The server must reply with a string
- ROS services doesn't rely on standard (already implemented) messages.
- We must implement our service message

```
$ roscd && cd ..
$ cd src
$ catkin_create_pkg ros_service roscpp std_msgs
message_generation message_runtime
```

- `message_generation` and `message_runtime` packages are used to handle the building and run-time usage of custom messages.

Example 2: ROS Service

- Create a new folder called `srv` in the package directory and add a `srv` file called `service.srv`

```
string in
---
string out
```

- Compile the message!

```
## Generate services in the 'srv' folder
add_service_files(
FILES
service.srv
)

generate_messages(
DEPENDENCIES
std_msgs
)
```

Example 2: ROS Service

- After compiled with `catkin_make` we can verify that everything works properly:

  ```
  $ rossrv show ros_service/service
  ```

  If we see the same content as we defined in the file, we can confirm it's working.

- Crate the source for client and server

  ```
  $ roscd ros_service/src
  $ touch service_server.cpp
  ```

Example 2: ROS Service

```
1  #include "ros/ros.h"
2  #include "ros_service/service.h"
3  #include <iostream>
4  #include <sstream>
5
6  using namespace std;
```

Example 2: ROS Service

```
1   bool service_callback
2   (ros_service::service::Request &req, ros_service::service::Response &
        res) {
3
4     std::stringstream ss;
5     ss << "Received Here";
6     res.out = ss.str();
7     ROS_INFO("From Client [%s], Server says
8     [%s]",req.in.c_str(),res.out.c_str());
9     return true;
10
11  }
```

Example 2: ROS Service

```
1  int main(int argc, char **argv) {
2    ros::init(argc, argv, "service_server");
3    ros::NodeHandle n;
4    ros::ServiceServer service = n.advertiseService("service",
         service_callback);
5    ROS_INFO("Ready to receive from client.");
6    ros::spin();
7    return 0;
8  }
```

Example 2: ROS Service

- Compile and run it!
- Check that the service instantiated in this node is active

```
$ rosservice list

Output:
/rosout/get_loggers
/rosout/set_logger_level
/service
/service_server/get_loggers
/service_server/set_logger_level
```

You can also call this service using the following command:

```
$ rosservice call /service "in_: 'Call'"

out: "Received Here"
```

Example 2: ROS Service

- Create the client node

```
$ roscd ros_service/src
$ touch service_client.cpp
```

```
1  #include "ros/ros.h"
2  #include <iostream>
3  #include "ros_service/service.h"
4  #include <iostream>
5  #include <sstream>
6
7  using namespace std;
```

Example 2: ROS Service

```
1  int main(int argc, char **argv) {
2  ros::init(argc, argv, "service_client");
3  ros::NodeHandle n;
4  ros::Rate loop_rate(10);
5  ros::ServiceClient client =
6  n.serviceClient<ros_service::service>("service");
```

Example 2: ROS Service

```
1       while (ros::ok()) {
2         ros_service::service srv;
3         std::stringstream ss;
4         ss << "Sending from Here";
5         srv.request.in = ss.str();
```

Example 2: ROS Service

```
1         if (client.call(srv)) {
2           cout << "From Client:
3           ["<<  srv.request.in << "],
4           Server says [" <<
5           srv.response.out << "]" << endl;
6         }
7         else {
8           ROS_ERROR("Failed to call service");
9           return 1;
10        }
11        ros::spinOnce();
12        loop_rate.sleep();
13      }
14      return 0;
15    }
```

Example 2: ROS Service

You can test the service connection running the server and later the client:

```
$ roscore
$ rosrun ros_service service_server
$ rosrun ros_service service_client
```

Additional commands to handle ROS service are reported in the following:

```
$ rosservice type /service: This will print the
message type of /service
$ rosservice info /service: This will print the
information of /service
```

Create custom messages

- ROS already provides a comprehensive set of messages for robotic programming
- In some situation could be useful to define your own ROS messages.
- ROS message definitions are stored in a `.msg` file in the `msg` folder of your package.

```
$ roscd ros_topic
$ mkdir msg && cd msg
$ touch demo.msg
```

```
1  string name
2  int32 data
```

Create custom messages

- When the package has been created custom messages were not considered
- Manually add the `message_generation` dependency in the `CMakeLists.txt`.

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
  message_generation

)
```

Create custom messages

- Uncomment the following line and add the custom message file:

```
1    add_message_files(
2      FILES
3      demo.msg
4    )
5    generate_messages(
6      DEPENDENCIES
7      std_msgs
8    )
```

- As usual, to use the added message, you have to compile the `ros_topic` package

Exercise 1

- Parrot package: develop a ROS package with a publisher and a subscriber. The publisher node accepts as input string or characters using the keyboard and publish such data on a ROS topic. The subscriber prints out the published data.
  - Test publisher and subscriber also using `rqt` and command line ROS commands.

Exercise 2

- Fibonacci Service: The Fibonacci Sequence is the series of numbers: `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`. The next number is found by adding up the two numbers before it. Develop a ROS Service taking 2 numbers (**`index`**, **`length`**) as input and returning a portion of the fibonacci sequence:
  - The first argument of the service represents the index of an element along the fibonacci sequence:

    `f[0] = 0, f[1] = 1, f[2] = 1, f[3] = 2, ...`
  - The second argument represent the number of element to return

  Starting from the **`index`** element of the sequence, return the next **`length`** element of the sequence.

Exercise 2

- Fibonacci Service: The Fibonacci Sequence is the series of numbers: `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ....` The next number is found by adding up the two numbers before it. Develop a ROS Service taking 2 numbers (`index`, `length`) as input and returning a portion of the fibonacci sequence:

  `Example:`

  ```
  Input: ( 8, 3 ), Output: 21, 34, 55
  f[8] = 21, f[7] = 13 -> f[9] = 21+13 = 34
  ```

Exercise 3

- Adder package: develop a ROS package that adds and publishes two random numbers.
  - Node1: in an infinite loop generates two random float numbers and publishes them on a Topic using a custom message
  - Node2: subscribes to the topic of Node1, sums the two numbers and republishes the result on another topic using custom message with three field:
    - Field 1: first random number
    - Field 2: second random number
    - Field 3: result of the sum

Exercise

- Try to use Object oriented programming (classes) and modularity
- Share with me your solution on github
- We can discuss your solutions during next lesson
- This is not part of the evaluation (for better or worse)