ActionLib
ROS parameters and launch file

Carmine Tommaso Recchiuto

Exercise proposed

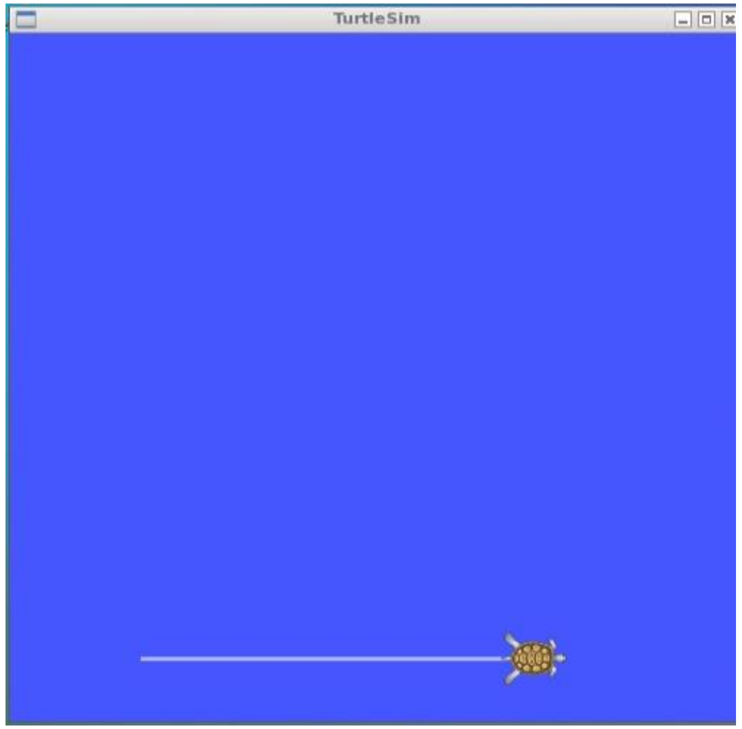
Here you can find a possible solution for the exercise proposed last week:

https://github.com/CarmineD8/my_srv

https://github.com/CarmineD8/turtlebot_controller

```
 > roscore

> rosrun turtlesim turtlesim_node
> rosrun my_srv harmonic_server
> rosrun turtlebot_controller exercise3
```

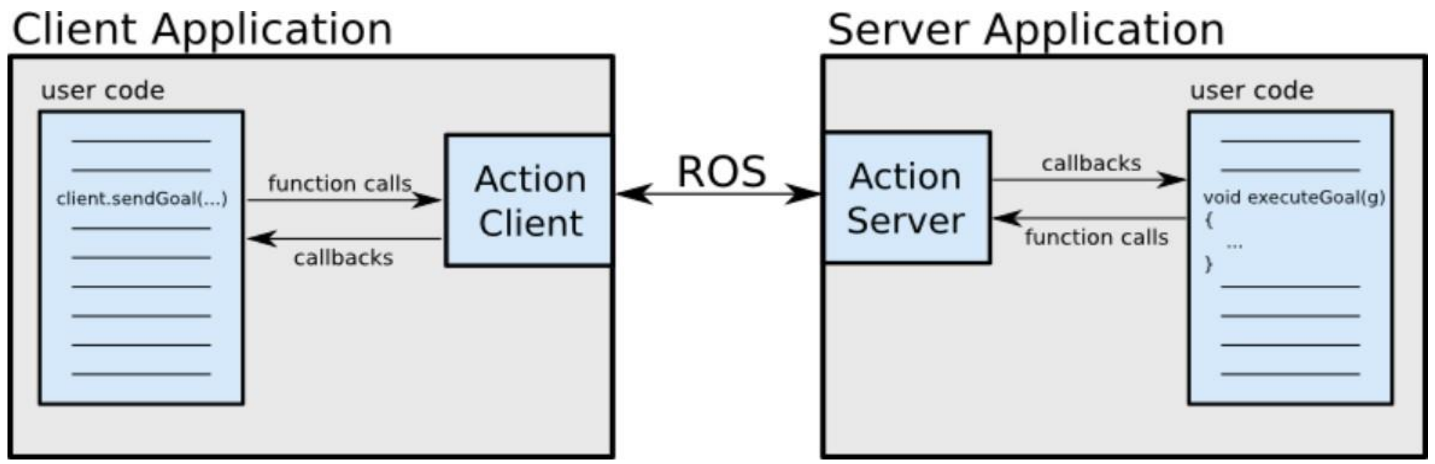


ActionLib

Imagine that Node A sends a request to node B to perform some task:

• Services are suitable if task is “instantaneous”

• Actions are more adequate when task takes time and we want to monitor, have continuous feedback and possibly cancel the request during execution



ActionLib

The actionlib package provides tools to:

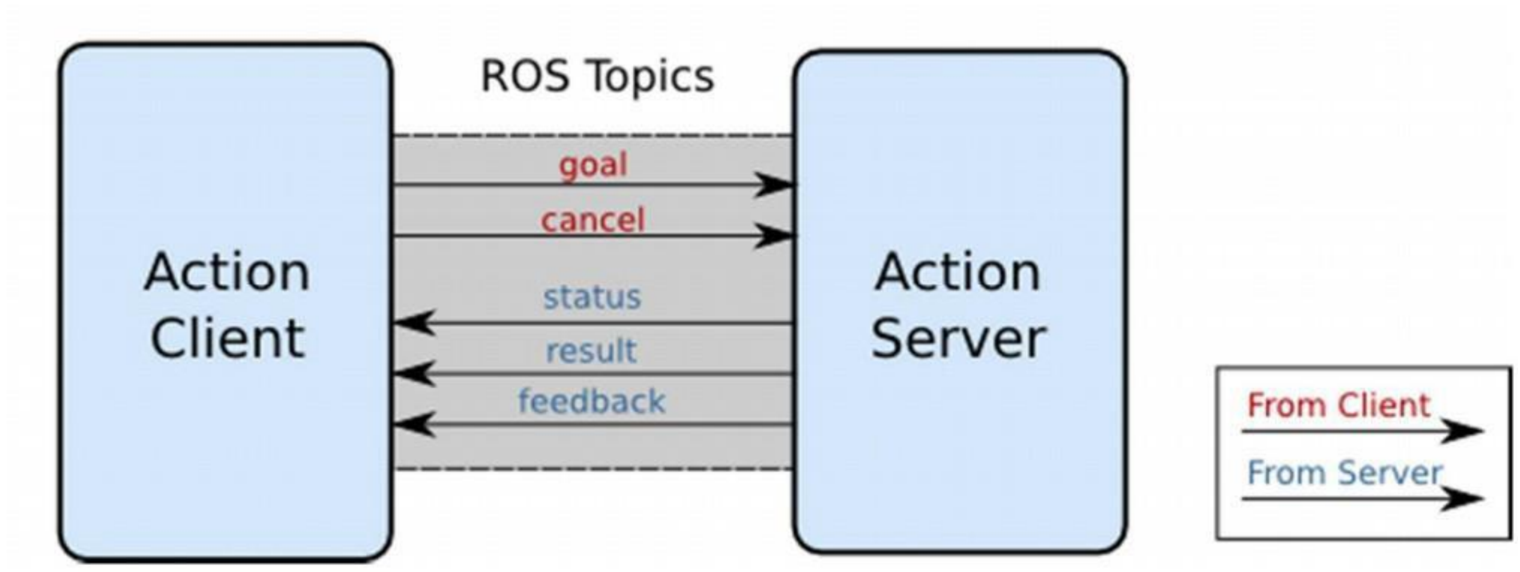
- create servers that execute long-running tasks (that can be preempted)
- create clients that interact with servers

References

- http://wiki.ros.org/actionlib
- http://wiki.ros.org/actionlib/Tutorials

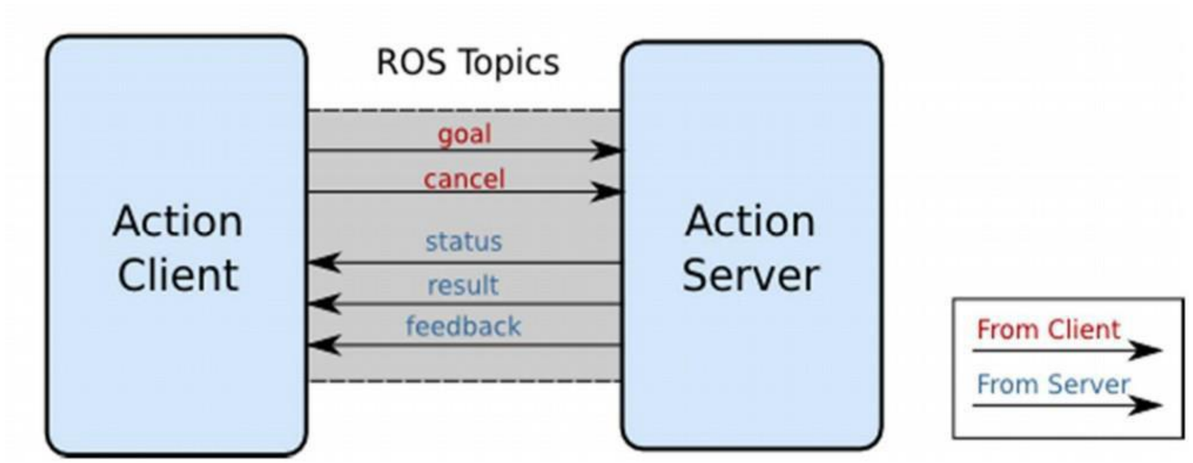
ActionLib

When using ActionLib, the Action Client and the Server communicate with each other using the ROS-Action protocol, represented in the figure below



ActionLib

- goal – used to send new goals to server
- cancel – used to send cancel requests to server
- status – used to notify clients on the current state of every goal in the system
- feedback – used to send clients periodic auxiliary information for a goal
- result – used to send clients one-time auxiliary information about the completion of a goal



ActionLib

- As for messages and services, we have a file defining the structure of the action message.
- Action templates are defined by a name and some additional properties through an .action structure defined in ROS, which is placed in the action folder of the package.
- Each instance of an action has a unique Goal ID, that provides the action server and the action client with a robust way to monitor the execution of a particular instance of an action
- Let’s see step by step how to work with actions, and how to use it in our client and server

Example: Fibonacci sequence

Let’s create a package with an action server which implements the Fibonacci sequence (Fibonacci number – Wikipedia)

Let’s start creating the package:

- new package:
    - *catkin_create_pkg my_actions actionlib actionlib_msgs roscpp*
    - create the folder *action* in the package

Example: Fibonacci sequence

The action messages are generated automatically from the .action file

- This file defines the type and format of the goal, result, and feedback topics for the action.
- It goes inside the folder action of the package

```
#goal definition
int32 order
---
#result definition
int32[] sequence
---
#feedback
int32[] sequence
```

Fibonacci.action

Example: Fibonacci sequence

To automatically generate the message files during the make process, a few things need to be added to CMakeLists.txt.

- add the actionlib_msgs package to the find_package arguments (if you have done it during the creation of the package you can skip this step)

- use the add_action_files macro to declare the actions you want to be generated:

```
add_action_files(
FILES Fibonacci.action
)
```

- call the generate_messages macro, not forgetting the dependencies on actionlib_msgs and other message packages like std_msgs:

```
generate_messages(
  DEPENDENCIES actionlib_msgs # Or other packages containing msgs
)
```

Example: Fibonacci sequence

Server:

- The server waits for a goal message, and when it receives it executes the related function.
- The action server should allow the action client to request that the current goal execution be cancelled
- Once the action has finished the server notifies the action client that the action is completed, by setting succeeded.

Let’s download the folder: https://github.com/CarmineD8/code

Here we will focus only on the client side: if you are interested, feel free to give a look to the server.

Example: Fibonacci sequence

Client:

```
#include <actionlib/client/simple_action_client.h>
#include <actionlib/client/terminal_state.h>
#include <my_actions/FibonacciAction.h>
```

actionlib/client/terminal_state.h defines the possible goal states. FibonacciAction.h includes action messages generated from the Fibonacci.action file shown before

Example: Fibonacci sequence

Client:

```
actionlib::SimpleActionClient<my_actions::FibonacciAction> ac("fibonacci", true);
```

The action client is templated on the action definition, specifying what message types to communicate to the action server with. The action client constructor also takes two arguments, the server name to connect to and a boolean option to automatically spin a thread. If you prefer not to use threads (and you want actionlib to create a thread behind the scenes), you can set it to true.

Example: Fibonacci sequence

Client:

ac.waitForServer();

Since the action server may not be up and running, the action client will wait for the action server to start before continuing.

```
my_actions::FibonacciGoal goal;
goal.order = 20;
ac.sendGoal(goal);
```

Here a goal message is created, the goal value is set and sent to the action server.

Example: Fibonacci sequence

Client:

```
bool finished_before_timeout = ac.waitForResult(ros::Duration(30.0));
```

The action client now waits for the goal to finish before continuing. The timeout on the wait is set to 30 seconds, this means after 30 seconds the function will return with false if the goal has not finished.

```
 if (finished_before_timeout)
   {
actionlib::SimpleClientGoalState state = ac.getState();
ROS_INFO("Action finished: %s",state.toString().c_str());
   }
```

Example: Fibonacci sequence in python

Please also notice that in the cpp file, we can add

```
ac.cancelGoal();
ROS_INFO("Goal has been cancelled");
```

In case the goal has not been accomplished before the timeout ends.

Client.py provides an example of an action client in python.

Example: Fibonacci sequence

Action subscribed topics (server perspective)

fibonacci/goal

(my_actions/FibonacciActionGoal): order of the Fibonacci sequence

fibonacci/cancel

(actionlib_msgs/GoalID): a Request to cancel the specific sequence

Action published topics (server perspective)

fibonacci/status

(actionlib_msgs/GoalStatusArray): status information of the server

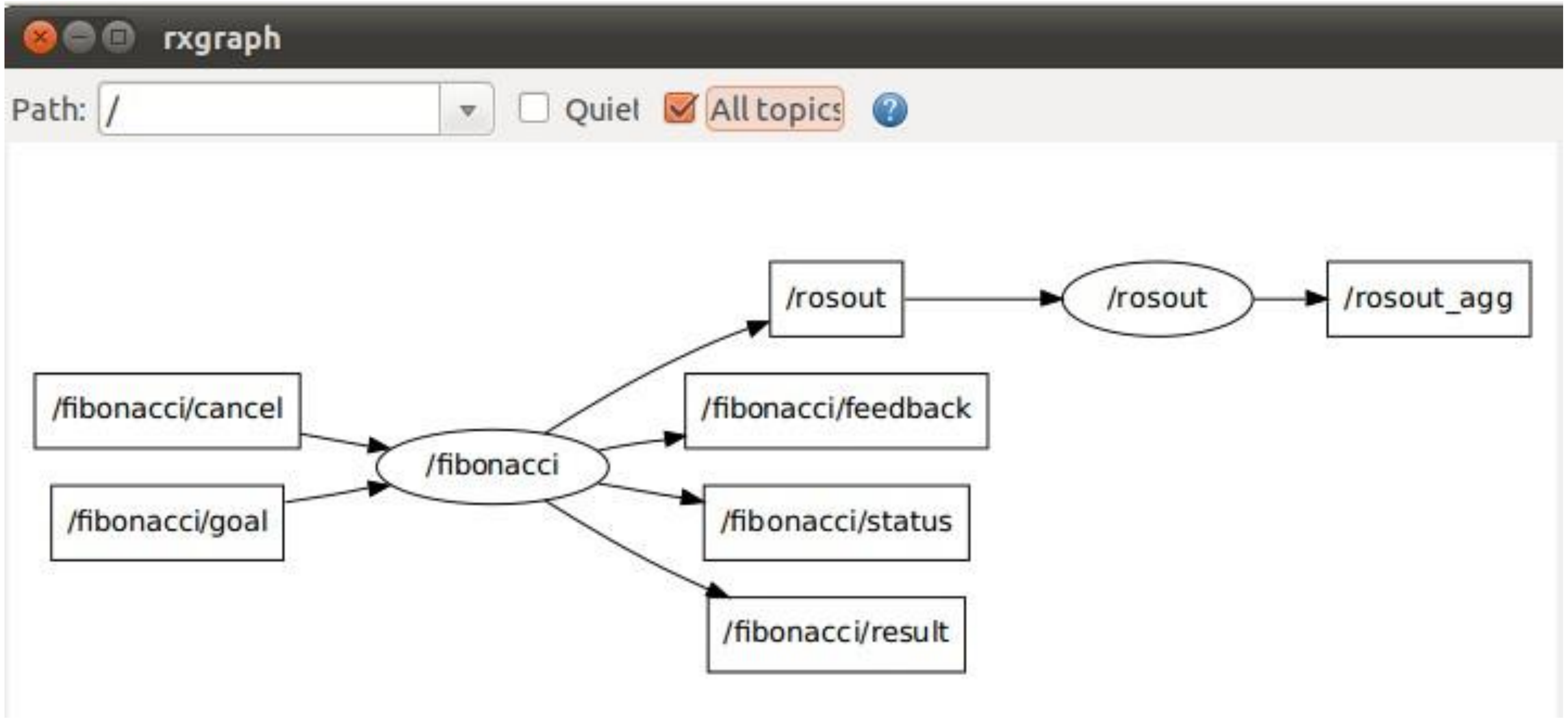
fibonacci/feedback

(my_actions/FibonacciActionFeedback): current sequence

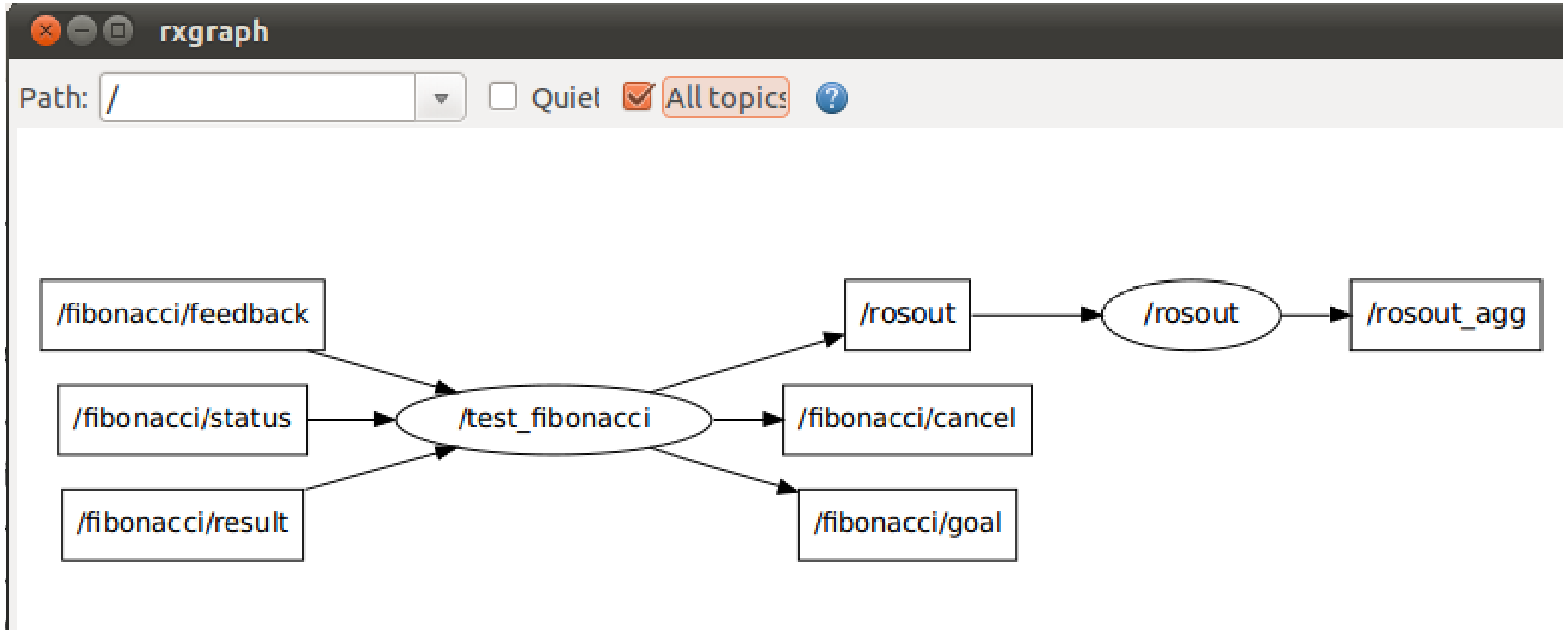
fibonacci/result

(my_actions/FibonacciActionResult): final Fibonacci sequence

Example: Fibonacci sequence



Example: Fibonacci sequence



ROS – Parameter Server

The ROS parameter server is a shared, multi-variable dictionary that is accessible via network APIs. Nodes use this server to store and retrieve parameters at runtime. As it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters.

It is meant to be globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary.

The Parameter Server uses standard data types for parameter values, which include:

- 32-bit integers
- booleans
- Strings
- doubles
- iso8601 dates (November 26, 2020)
- lists
- …

ROS – Parameter Server

The rosparam command-line tool enables you to query and set parameters on the Parameter Server using YAML* syntax.

rosparam set <parameter-name>	set parameter
rosparam get <parameter-name>	get parameter
rosparam load <yaml file>	load parameters from file
rosparam delete <parameter-name>	delete parameter
rosparam list	list all parameters name
rosparam list /namespace	list all parameters in a particular namespace

* YAML stands for Yet Another Markup Language, that supports all parameter types

ROS – Parameter Server

Example:

- rosrun turtlesim turtlesim_node
- rosparam get /background_b

Parameters may be retrieved (or modified) also within a node:

- in python, by using rospy.get_param or (rospy.set_param)
- in cpp, by using ros::param::get (or ros::param::set).

E.g., package CarmineD8/parameters: Package for learning how to deal with ROS parameters (github.com)

How to launch a node together with some parameters?

ROS – Launch Files

roslaunch is a tool for easily launching multiple ROS nodes locally, as well as setting parameters on the Parameter Server. It includes options to automatically respawn processes that have already died.

roslaunch takes in one or more XML configuration files (with the .launch extension) that specify the parameters to set and nodes to launch

The roslaunch package contains the roslaunch tools, which reads the roslaunch .launch/XML format. It also contains a variety of other support tools to help you use these files.

Many ROS packages come with "launch files", which you can run with:

```
$ roslaunch <package_name> <file.launch>
```

ROS – Launch Files

Some flags:

--wait

Delay the launch until a roscore is detected. By default, roslaunch will also launch the roscore master, the option “wait” may be added to force the launcher to wait for a master to be executed.

--local

Launch of the local nodes only. Nodes on remote machines will not be run.

--screen

Force all node output to screen. Useful for node debugging.

-v

Enable verbose printing. Useful for tracing roslaunch file parsing.

ROS – Launch Files

roslaunch .launch files are written in the XML format

roslaunch evaluates the XML file in a single pass. Includes are processed in depth-first traversal order. Tags are evaluated serially and the last setting wins. Thus, if there are multiple settings of a parameter, the last value specified for the parameter will be used.

Minimal example:

```
<launch>
		<node name="talker" pkg="rospy_tutorials" type="talker" />
</launch>
```

The **<node>** tag specifies a ROS node that you wish to have launched. This is the most common roslaunch tag as it supports the most important features: bringing up and taking down nodes.

ROS – Launch Files

Within the <node> tag, we may have additional (optional) attributs:

args= “arg1 arg2 arg3” -> arguments to be passed to the node.

respawn = “true” (default is false) -> restart the node automatically if it quits

required = “true” (default is false) -> if node dies, kill entire roslaunch

output = “screen” (default is log) -> if screen, stdout/stderr and ROS_INFO will be visualized on the terminal. If ‘log’, the stdout/stderr will be sent to a log file in $ROS_HOME/log.

cwd = “node” (default is ROS_HOME) -> if ‘node’, the working directory of the node will be sed to the same directory ad the node’s executable.

ROS – Launch Files

The <include> tag enables you to import another roslaunch XML file into the current file.

```
<launch>
        <include file=$(find pkg-name)/path/filename.xml />
        <node name="talker" pkg="rospy_tutorials" type="talker" />
</launch>
```

Roslaunch tag attributes can make use of *substitution args*, which roslaunch will resolve prior to launching nodes.

e.g. $(find pkg-name):

The filesystem path to the package directory will be substituted inline. Use of package-relative paths is highly encouraged as hard-coded paths inhibit the portability of the launch configuration.

ROS – Launch Files

<remap>

Remapping allows you to "trick" a ROS node so that when it thinks it is subscribing to or publishing to /some_topic it is actually subscribing to or publishing to /some_other_topic, for instance.

Sometimes, you may need a message on a specific ROS topic which normally only goes to one set of nodes to also be received by another node. If able, simply tell the new node to subscribe to this other topic. However, you may also do some remapping so that the new node ends up subscribing to /needed_topic when it thinks it is subscribing to /different_topic.

This could be accomplished like so:

```
<remap from="/different_topic" to="/needed_topic"/>
```

The remap tag can be used within a <node> tag, and in that case it will remap will apply just to that specific node, or generally in the launch file, and in this case it will apply to the lines following the remap.

ROS – Launch Files

How to set parameters in the launch file?

- ✓ Tags <param> and <rosparam>

The tag <param> defines a parameter to be set on the Parameter Server. The <param> tag can be put inside of a <node> tag, in which case the parameter is treated like a private parameter.

Es.

```
<param name="publish_frequency" type="double" value="10.0" />
```

Type may be str, int, double, bool or yaml.

ROS – Launch Files

How to set parameters in the launch file?

- ✓ Tags <param> and <rosparam>

Similarly, the tag <rosparam> gives the possibility of loading or deleting parameters from the ROS Parameter Server.

```
<rosparam command="load" file="$(find rosparam)/example.yaml" />
<rosparam command="delete" param="my/param" />
```

Example

Example: package parameters -> **roslaunch parameters param.launch**

```
int main(int argc, char **argv)
{
ros::init(argc, argv, "getting_params");

int int_var;
double double_var;
std::string string_var;

ros::param::get("/my_integer", int_var);
ros::param::get("/my_float", double_var);
ros::param::get("/my_string", string_var);

ROS_INFO("Int: %d, Float: %lf, String: %s",
int_var, double_var, string_var.c_str());
}
```

```
#! /usr/bin/env python

import rospy

int_var =  rospy.get_param("/my_integer")
float_var = rospy.get_param("/my_float")
string_var = rospy.get_param("/my_string")
print("Int: "+str(int_var)+", Float: "+str(float_var)+"
String: "+str(string_var))
```