Pumpkin basic Manual

Rafael de Paula Paiva, 08.paiva@gmail.com

November 20, 2015

Contents

1 First steps

1.1 Environment

The Pumpkin project is based on ROS (Robot Operatin System), a framework and system that creates a base layer for distributed systems.

The system is intended to be run into, at least, two different devices. One of them will hold the core system, that read from the sensors, and command the servos. The others devices will run the GUI (Graphical User Interface).

1.1.1 Installing ROS

The ROS, by default, runs on Linux, but there is possible to install ROS in other systems. The Pumpkin was developed and tested in the Ubuntu distro, so this is the recommended disto, although it can be used other distros. But the recomendation to Linux stills.

The installation guide can be found in: ROS Indigo installation guide page.

The Pumpkin was developed in the ROS Indigo, since it has more support. So, it's sugested that install this version. To be sure to have everything OK, try to install the desktop-full version.

1.1.2 Necessary ROS packages

There is others ROS packages after the basic installation that the Pumpkin system use. All the subsequent packages are debian-like installation packages. They can be installed running a installation program, like the *synaptic*, or the *Ubuntu Software Centre* programm, or running the command below in a terminal:

```
sudo apt-get install <package-name>
```

All the packages below will contain the ROS distro name. For example, if you run the Indigo distro, the packages to be installed will be named: `ros-indigo-...`, but if you are running the Jade distro, the packages will be named: `ros-jade-...`.

For each package shown below, install it using one of each option. Using a installation program, or by command-line.

The first package to install is the **actionlib**, to run the Action Service. The name of the package is:

```
ros-<distro>-actionlib
```

(override the ¡distro¿ by the distro used, as explained above).

After that, it is necessary to install the *MoveIt!*. This package if for trajectory planning. To install it, install this package:

```
ros-<distro>-moveit-ros
```

The next component is the Rosserial. It is necessary to communicate to the Arduino board. To install it, there is two packages to be installed:

```
ros-<distro>-rosserial-arduino
and
ros-<distro>-rosserial-server
```

1.1.3 QT

The ROS GUI is made in QT. This section is only necessary to do in the device to run the GUI, but it can also be done in the machine that run the Pumpkin core.

Note: It is recommended to install the QT in the machine that will run the pumpkin core, since it has some visual tools that will use, also, QT.

To run the GUI, install the package:

```
libqt4-dev.
```

1.2 Pumpkin Installation

First of all, it is necessary to download the Pumpkin project to the machines. Create a folder in the machine to the project be in.

...

Atention: These instructions above is for all machine that will run the Pumpkin system.

Note: All the commands shown below is to be ran in the same folder that the project was downloaded.

Now, it's time to build the system.

1.2.1 Building the core packages

This instructions below is for the machine that will run the Pumpkin core, that is, the machine that have direct control with the servos, and connections with the sensors.

In theory, this machine should be enbedded in the robot.

After downloading the project, run the command:

```
catkin_make
```

to build the project. Sometimes it is necessary to run the command again, if the first time some erros may have been shown.

If you don't want to install QT on the machine that will run the GUI, run this command to exclude the GUI package:

```
catkin_make --pkg pumpkin_messages pumpkin pumpkin_interface
```

1.2.2 Building the GUI packages

This steps is to build the GUI for the Pumpkin.

If both the machine are the same, and it was run `catkin_make` in the step before, nothing more have to be done. So, just run the command.

Otherwise, if the machine are not the same, there is two options:

The first option, run `catkin_make` to build everything. But this is not necessary.

The second option is just to build the GUI. To do it, run:

```
catkin_make --pkg pumpkin_messages pumpkin_qt
```

1.3 Last environment settings

For now, the both systems may run, but we need to configure the network environment to both be able to communicate.

The first thing to do is to be sure that both machines are on the same local network, i. e. they are on the same WiFi network.

1.4 Running Pumpkin

To easily run the Pumpkin, it was created two scrips. Both are on the project base directory.

For the *core*, run the `pumpkin.sh`. And for the *GUI*, run the `gui.sh`.

2 Using Pumpkin

2.1 Basics

Pumpkin is a robot made, until now, to play recorded movements, and to be able to concatenate movements.

Here are some terms that will be used in this project:

Movement A sequence of positions of the robot parts over time, that, combined, form a movement, as a normal movement as you can imagine. Examples are: shaking hand, doing a "bye", etc.

Playback file Is a full movement that is recorded into a file, to be played later. The file contains the value of the sensors of the robot over the time since the start of the movement.

Playback This is the action of the robot executing the movement saved in a *playback file*.

Record Is the action of create a *playback file*, saving, over a period of time, the value of sensors. The movement is done moving the robot parts, like a ragdoll.

Planning Is a computational process that create a temporary movement to be executed between two defined positions passed as the entry of this process.

Scene Is a sequence of *playback files* that are executed in order, composing a new movement. Between each pair of movement, the Pumpkin *plans* the xxx, as the overall movement appears more natural. This new movement, although, is not saved as a new movement, since this new movement depends only of the sequence of *playback files*, repeating this sequence recreates this *scene*

Note: For security reasons, all the motors starts turned off, and also they will be turned off after finishing a *playback* or *scene*. Obviously, the motors have to be turned off to be able to record a scene.

To command the robot, a prior, use the GUI, since it covers all the functionalities of the robot, and also is made to be ease to use.

2.2 The PumpkinQT

Now, for the basic usage of the Pumpkin system. The GUI was designed to help for the basic usage of the robot.

The first window that you may see is the configuration window. This window run only once as the core system still runs in the main machine, it may appear if the core system restarts.

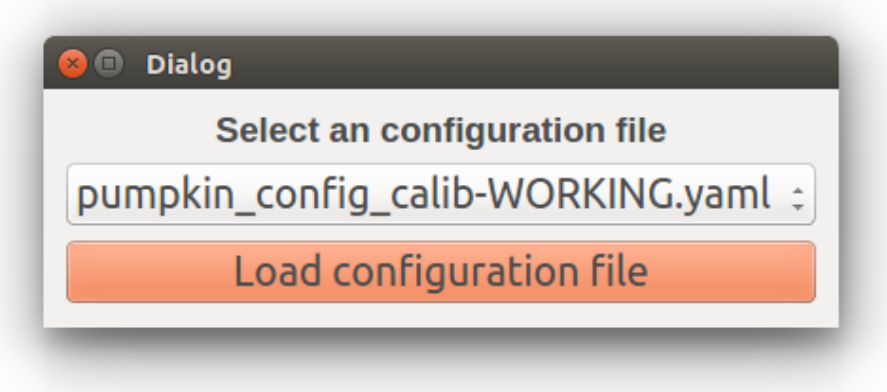

Figure 1: Load Config dialog

The only thing to do here is select a configuration file from the list, and press the button.

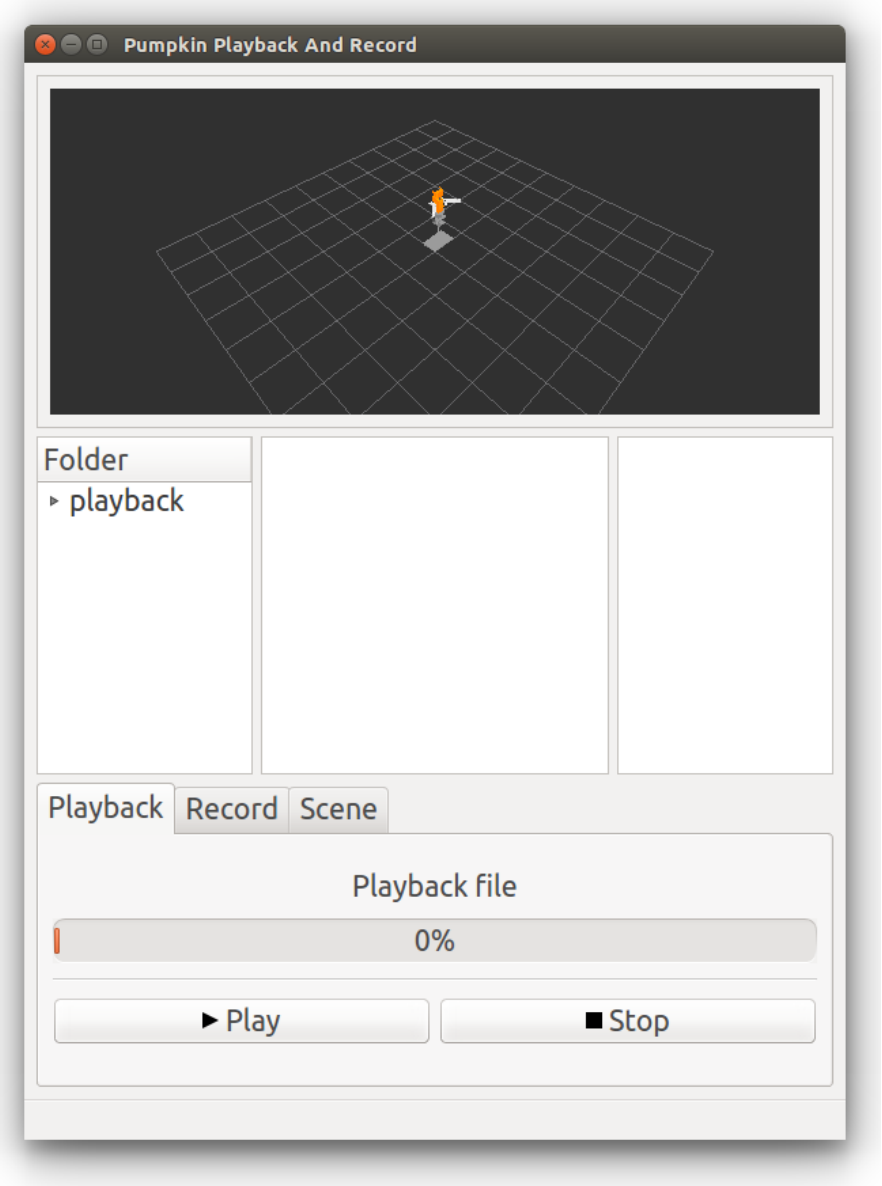

Figure 2: the PumpkinQT main window, an ease way to use the Pumpkin

As said, this dialog only show once. So, after selecting the configuration file, you go to the main window. If you have ran the configuration before, the main window will be shown directly.

At a first glance, there is, in the topmost part, a 3D model of the robot, to have a view of the Pumpkin.

Under this view, there are thre lists:

- The left one is to show the folder tree to view the folders that holds the recorded playback files, in a tree model (that is, the children folders are indented from the parent folder).
- The center one is to show the list of playback files that are inside the selected folder. This list start empty since no folder is selected. As you select a folder, the list of files are shown, for the current folder.
- The right one is the list of selected playback files to play a scene. The scene is covered after.

And the bottom part of the window are the actions controls. They're displayed in different tabs, each one representing a different functions.

2.2.1 Playback

This function is to order the robot to execute a previous recorded playback. To do this, first, an playback file must be selected. As it's selected, the name of the file will be displayed where initially is written **Playback file**.

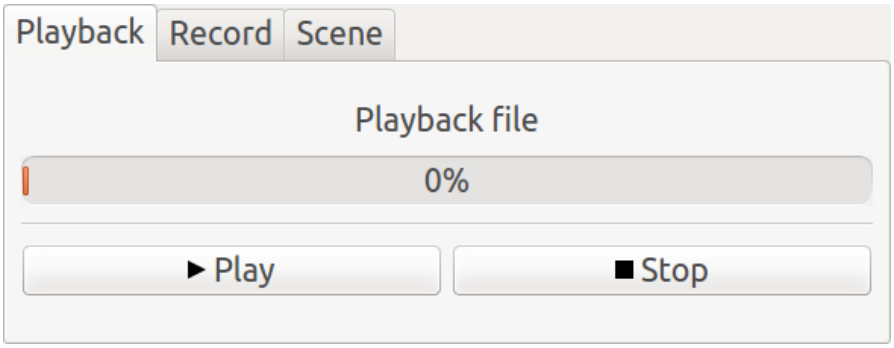

Figure 3: The playback tab is to execute one movement

To playback, just press the *Play* button (as seen in: 3). As you press this button, you will see the progress bar starting filling up. This is because this progress bar show the completed percentage of the movement. You will see, also, that the *Play* button will not be more clickable, and this is logically because you are in the middle of a movement.

While the robot is executing the movement, you can press the stop button, to stop the execution of the movement. Obviously, the button will only be clicable when the robot is executing a movement.

2.2.2 Record

This function is to record a new movement. To this, you have, first, to select a folder. You can, also, select a file in the playback file list to override a playback file.

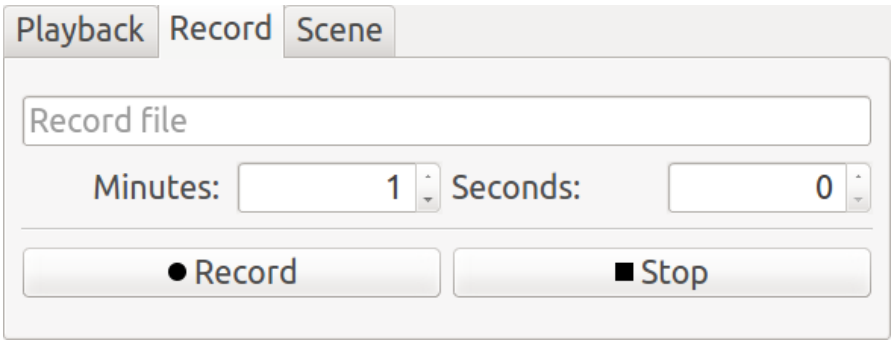

Figure 4: The record tab is to record new movements

The first thing to do here is to write the *playback file* name (in the entry, that maybe show a light gray "Record file" text.). You can, also, overrite a movement, selecting a file, and the *playback file* name will be in the entry. Note that writing an existing *playback file* (in the same folder) will also overrite that file.

After that, a time limit for the movement is needed to be set (in the tho slots, one for the minutes value, and other for the seconds value). The recording will stop when the time limit reachs, or the *Stop* button is pressed in the middle of a recording.

As the record starts, you will see that the entries for the time limit will not be able to modify, but the values in there will start changing. This is because these entries will show the elapsed time since the begining of the recording.

2.2.3 Scene

This is, for now, is the most advance function of the robot. This is the only function that will use the right list (in the middle part of the 2).

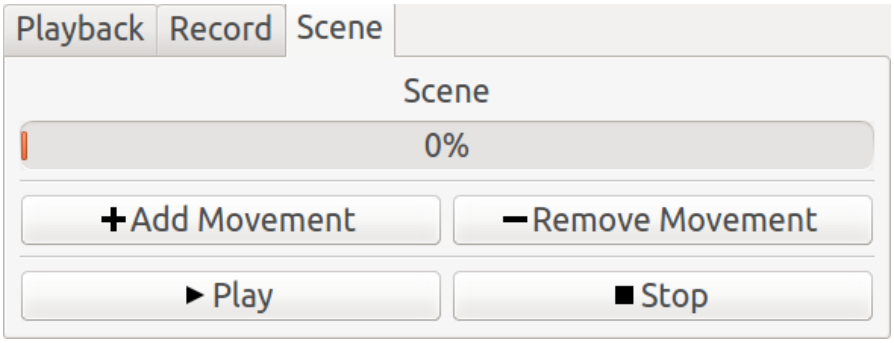

Figure 5: The scene tab is to play a sequence of movements

The first thing to do is to create the *playback file* list. To do that, the two buttons in the upper part of the *scene tab*.

To add a movement to the *scene*, select a playback file (as for *playback* this file), and then click the *Add movement* button.

To remove a movement from the scene, select a file from the scene list, and click the *Remove movement* button.

And to change the order of the movement, just "drag and drop" the *playback file* from the *scene list* and put it in the desired position.

After that, play the scene clicking the *Play* button. This is like a playback, but all the movements are executed in order they appear in the *scene list*. Remember that, as the playback, you can stop at any moment.

The progress bar and the text that you see is to indicate the overall completion of the scene. The progress bar is equal to the playback, indicate the completed percentage of the movement, or the planning, and the text indicate each movement (or planning between movements) is being executed in that moment.

2.3 Other PumpkinQT ...

There are other two ... that the PumpkinQT is able to do. They're accessed by the menu of the *Main Window* (2).

2.3.1 Manage files and folders

The *Files manager dialog* (Figure 6) is used to manage the files and folders. To organize the *playback files* better, they're disposed in different folders. So, you are able to create, and delete these folders. And also delete any movement file thay you doesn't want anymore.

ATENTION! As this system runs in a network environment, all these actions are request to the OS that runs the core of the Pumpkin. So, it is not possible to recover a deleted movement, or folder.

If you open the dialog without selecting any folder, you will see it as shown in the figure 6. But, as soon you select a folder, in the space you see "No folder selected", you will see the name of the selected folder. In the same way, as soon you select a file, you see the name of the selected file (in this case, with the full path), where is shown "No file selected".

The entry, in the botton left part of the dialog, is where you write the name of a new folder, in case you want to create one.

To do any job, just click on the respect button in the right part of the dialog window:

Delete File Delete the selected file. This button will not be clickable until you select a file.

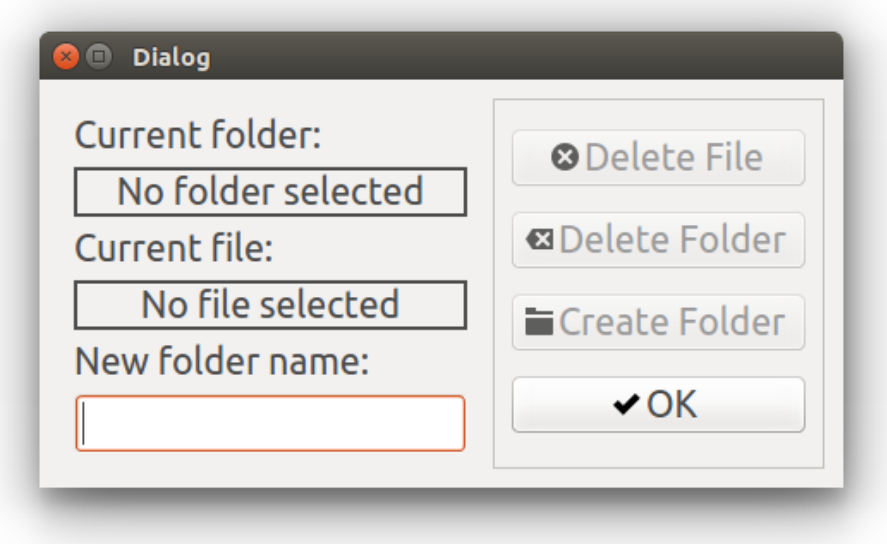

Figure 6: The files manager dialog is to manage the files and folders of playback

Delete Folder Delete the selected folder. This button will not be clickable until you select a folder.

Create Folder Create a new folder under the selected folder (as a child folder). First, you have to select a folder to do that.

To close the dialog, just click the *OK* button.

Note: Creating a new playback file is done in the *Record* (Section 2.2.2).

2.3.2 SSC Move Command Sender

The other resource is more advanced, and was created to debugging purpose. You can test it too, but is **HIGHLY NOT RECOMMENDED** in case you are not developing for Pumpkin.

First of all, if you don't know what is a Servo, it probably is not for you. Just don't touch in this window. Press the *Cancel* button and close it.

This is a raw command sender to the robot servos. As, until now, the robot use an SSC to send command to this joints, this dialog is used to that: send commands to the joint.

The command dialog is divided into tabs, each tab contains a robot part (this is just to organize better the joint panel). Each pannel inside a tab is designed for one joint.

The first thing to notice, for each panel, is there is a check box. Leave it checked to send the specific command to the servo related to that joint, but if you want to leave that servo out of the command, uncheck it. After that, just set the value for the pulse width. If you wish to set a speed to the servo, just set the speed value over 0 (zero).

The last thing before sending the command is the time value (the time limit to each servo have to be). If you want to send the time among the info, just set the value above 0 (zero).

Click the *Send Command* button to send the button. You can send any ammount of commands you wish. After that, click on the *Cancel* button to close the dialog.

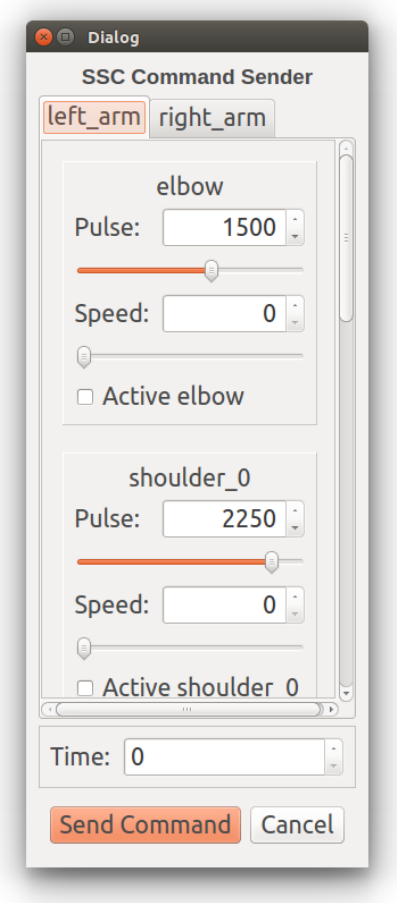

Figure 7: The SSC Move Command Sender is a more advanced resource, to send commands directly to the servos

3 Understanding the project

For now on, the basic usage is already covered. This part is for the ones that are developing for the pumpkin, or for enthusiasts that want to know more about.

The ROS way of working is by a distributed system that run by a web of nodes. A node is just a small program that do a specific task. The concept of small programs come by the idea: if any program is enough big, it should have high computational time that may compromite the communication among the nodes.

The ROS have, besides, another program that controls the nodes execution, and do the communication among them (all the network task and so on), that is the *ROS Core*, or *roscore*.

The communication is a key of ROS. By the ROS communication system, each node can send and receive info from the others. There are two tipes of communication that ROS can do:

Topic The ROS topic system is composed as a Queue server. Some nodes send the information to the *roscore*, into a specific topic (like each topic is a queue), and other nodes listen to the refered topic. Each time a new information is available in the topic, the *roscore* send them to the nodes that are listening. It can be compared to a newsletter: when you subscribe to one of them, you receive the each news that are published.

Service The ROS Service acts more like an RPC. A node tells the *roscore* that it is running a specicic service, becoming the server node, and then some nodes can call this service, sending some info to the server node (as the parameters for the RPC), and the server node processes and send the response to the call (as a return). Is is compared to a function call in a code.

Each information that goes across the nodes (in topic or services) are called **messages**. A message is a data structure that can be serialized. The ROS have a message build system to easily create a message, and the code to handle it in a specific programming language (naturally in C++ and Python).

And, also, the ROS have a system to save some information directly into the *roscore*, the ROS Parameter Server. It is like a properties part of the system. And it is more often used to save configuration parameters.

For futher understanding on the ROS, please follow the tutorials in this page: ROS Tutorials.

3.1 Pumpkin nodes overview

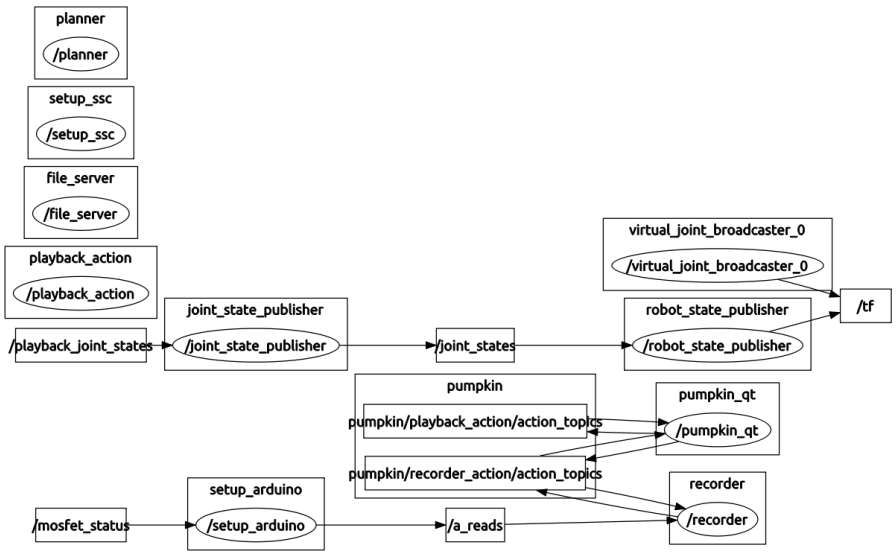

Figure 8: This graph represents the Pumpkin nodes before loading the configuration, obtained in rqt-graph

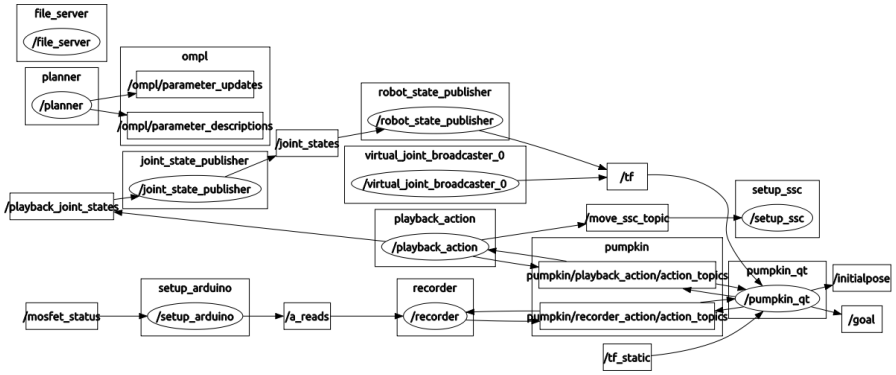

Figure 9: This graph represents the Pumpkin nodes after loading the configuration, obtained in rqt-graph

The images 8 and 9 show the nodes that make the pumpkin running. The first (8) one is when the configuration was not done yet (Is when the 1 dialog is shown). The second (9) one is when the configuration is done, and the system is running.

Note: To understand better the graphs, there are what the forms represents:

- The small boxes represents a topic.

- The ellipsis represents a node.
- The larger boxes (the ones that have others elements inside) represents namespace (and it's just for better organization of the nodes).
- The arrows represents the way that the nodes communicates (the tip of the arrow indicate the receiver, the other side, the sender). If the arrow connects two nodes, it represents the two nodes are in a service communication. If the arrow connect a node and a topic, it represents this node publishes to it, or subscribe from it (the way of the arrow indicates each one is).

It's clear that, before load the configuration, the system is much simpler. Some topics doesn't even exist, and a lot of nodes are not communicating to each other. Nonetheless, even the second graph (9) is not complete, since the `playback-action` is not in a service to the `planner`, as the service is only done if a scene is in execution.

The Pumpkin are divided into five packages:

pumpkin This main package controls the robot behavior and functionalities

pumpkin_interface This package contains interface nodes, or interface between the ROS and the robot sensors and servos, or between the GUI and the OS (since they maybe in different machines).

pumpkin_qt This package is for the robot GUI

pumpkin_messages This package holds the messages for topics, services, or action services, that pumpkin has and uses.

pumpkin_moveit This package has the *MoveIt!* configuration files and sources, that was made in a setup for the robot. There are, also, some movements from a first stage of the planning.

Note: There are another folder under the `src` that was not covered above: the `pumpkin_description`. That is because this is not a ROS package. It is the 3D data of the robot, to be rendered or to be used in the kinectics processing (as the planning).

The nodes of the Pumpkin are described below, except those generated from the MoveIt! setup, since it is almost automatic code.

3.2 Pumpkin Interface

This package is the simplest, only has three nodes. Its meaning is already viewed in the above section

3.2.1 The "file_server" node

This node is intended for making some tasks related to files. It has two different services:

file_lister This service is to seek files in a specific directory (and all of inside folders). It is intended to show specific file types, as: playback files and configuration files.

file_handle This service is to create and delete a file or a folder.

Atention! Before running this node correctly, it is needed to pass to the node the path of the pumpkin package folder. To do that, there is two options: by environment variable, or by ROS parameter. If you run the system by the script (1.4), this is done automatically.

3.2.2 The "setup_arduino" node

This node does all the communication to the Arduinos that are connected to the core system.

The functionalities that the Arduino does are coded inside them. This node is just to maintain the communication online.

3.2.3 The "setup_ssc" node

This node does almost the same thing that the `setup_arduino` does, but this one also process the information, since there is no program running the SSC. The SSC just receive a string command via serial port to create a PWM pulse. This pulse sets the position of a servo.

To send a command to SSC, there is two ways: by service or by node. The service has preference over the topic, but the topic is better to ordered sequencial commands, like the ones in a movement. So, the service is better to send a secure order, or by debugging/testing.

Note: To run this node, the SSC must be powered on. This is automatically done if the robot is powered on. This is for security, since it alerts the robot, or the SSC, is powered, showing a possible disconected cable (or other problem).

The node tries to maintain the communication to the SSC on. So, after the node is running, the SSC can be powered off, and powered on later (The only exception is in the node initialization, as seen above).

Also, this node checks the command limits, to prevent harming the robot. But it not process the command, just get the message and transform it in a SSC understandanble command. To know better how the SSC command works, see the SSC reference manual.

3.3 Pumpkin

3.3.1 The "load_config" node

This node is for load the robot configuration file. When using the *GUI*, this is done by selecing a file in the *Load config dialog* (1).

After selecting a configuration file, this node loads it in the ROS Parameter Server, and closes itself.

3.3.2 The "recorder" node

This node records a movement into a *playback file*. This node is run when the user starts a record (see 2.2.2).

It saves the values comming from the Arduino(s) into a *playback file* when in a record. It controls the file handling, and other stuffs.

The files are, now, in the YAML file type.

3.3.3 The "playback_action" node

This node handles the *playback* and *scene*. When the user commands a playback (see 2.2.1) or scene (see 2.2.3), this node starts sending commands to the SSC based on the values inside the *playback file*.

Note: It also publishes the commands as joint values, to update the 3D robot model.

Note(2): The conversions to one value to another is made linearly.

This node also does the communication to the *planner* node, to get the planned trajectory between two movements, and send them to the SSC.

To run this node, first the configuration must be set (as this node will wait for it), and other parameters. This is done automatically if you run the script (1.4).

3.3.4 The "planner" node

This node uses the *MoveIt!* planner pipeline to plan the robot trajectory between two movements.

It initialize all the *MoveIt!* stuff, and set up a ROS service server. This service is for receive a request from the *playback_action*, plan, and response with the joint positions trajectory.

To run this node, first the configuration must be set, and a bunch of ROS parameters (required for the *MoveIt!*). This is also done automatically if you run the sript (1.4).

3.4 PumpkinQT

This package contains only the *pumpkin_qt* node. It holds all the GUI to run the Pumpkin.

It has classes to comunicate with the core, and classes that displays the windows with the widgets.

3.5 Pumpkin description & Pumpkin Moveit

The `pumpkin_moveit` package is created automatically with the MoveIt! setup. But it also contains some codes to send specific commands to the robot, and try to plan them.

The `pumpkin_description` has the *URDF* model for the Robot, so the ROS and *MoveIt!* can recognize the robot. It has, also, the 3D model meshes, so the robot can be displayed, and the collision detection can work; and it has other configuration files used in *MoveIt!*, like the kinectics configuration.

3.6 Pumpkin Messages

This package holds the message that all the *Pumpkin* uses. It has no code.

The message are data structure that can be serialized. To help creating classes, the ROS have a message generator package. To generate a package, a "message description file" must be created, inside a specific folder (indicating the message type):

msg For a simple message, that will be used into a Topic, or to be inside another message.

srv Service message, containing exactly two messages: the first for the request, and the second for the response.

action For Action Service, containing exactly three messages: the first for the goal (like a request), the second for the response, and the third for the feedback.

To understand what the messages are, the easiest way is to see the documentation files.

4 Futher Information

For futher information about the code, you can see the documentation about the source code.

The documentation is in doc folder. Each package has its own documentation. This doc is auto-generated.

The documentation is in HTML format, so, to view it, open the *index.html* files. There are links to the indexes in each package to make it easier to find the file and open it.

Note: To the `pumpkin_messages` package, the index page is the *index-msg.html*, to the messages documentation.