Summer Course on Developing on ROS Framework — Day 1

GNU/Linux and ROS Introduction

Contents

1 Installation

For this course we will use ROS Groovy, which is the current stable version. ROS Groovy can be easily installed in Ubuntu 12.04 or 12.10. We recommend that you install Ubuntu 12.04 since it is a Long Term Support (LTS) release. It will be supported until April 2017. Ubuntu 12.10 is no longer supported (it was replaced by 13.04, which ROS does not support). We recommend that you update again when Ubuntu 14.04 is released, which will be a LTS release again.

There is a virtual machine with Ubuntu and ROS already installed that can be downloaded from `http://www.ros.org/wiki/ROS/Installation`.

Install Ubuntu by downloading it from `http://www.ubuntu.com/` or the mirror inside IST at `http://ftp.rnl.ist.utl.pt/pub/ubuntu/releases/`. Install ROS by following the instructions at `http://www.ros.org/wiki/groovy/Installation/Ubuntu` (install Desktop-Full).

Install some more packages by issuing the following command in a terminal window:

```
sudo apt-get install build-essential automake autoconf libboost-all-dev \
cmake cmake-curses-gui subversion subversion-tools git-all gitk \
byobu mmv htop rar p7zip lzop myspell-pt-pt flashplugin-installer
```

This will install:

- `build-essential automake autoconf libboost-all-dev` — Tools needed for development of programs, probably most of it was already installed by ROS.
- `cmake cmake-curses-gui` — CMake, including `ccmake`.
- `subversion subversion-tools` — Subversion.
- `git-all gitk` — Git and Gitk tool.
- `byobu` — Byobu is the terminal that we will use.
- `mmv` — Utility to move multiple files.
- `htop` — Process viewer.
- `rar p7zip lzop` — (Optional) Common compression tools.
- `myspell-pt-pt` — (Optional) Portuguese dictionary for Firefox and other programs.
- `flashplugin-installer` — (Optional) Adobe Flash Player plugin, needed by many sites.

Also install `ack` by issuing the following 3 commands:

```
mkdir -p $HOME/bin
wget -c -T 20 -t 20 'http://betterthangrep.com/ack-standalone' -O $HOME/bin/ack
chmod 0755 $HOME/bin/ack
```

2 Byobu Console

Byobu is a terminal program with some very useful enhancements. Sometimes, while working with ROS, you will have to keep a large number of terminal windows open. Byobu will be a huge help with that. Byobu is specially useful when run over SSH: it lets you have multiple terminals in the same connection, and will keep your programs running if the connection dies.

Open Byobu and lock it to the launcher. Try the following commands:

- `F2` — Create a new window
- `exit` or `Ctrl-D` — Closes the current window
- `F3` — Move to the previous window
- `F4` — Move to the next window
- `F7` — Enter copy/scrollback mode
- `F8` — Rename the current window
- `F9` — Configuration Menu, here you can access help and learn how to split windows
- `Shift-F12` — Disable F shortcuts, so you can use them in other programs

Note that Linux has two copy-paste buffers:

- The normal `Ctrl-C Ctrl-V` buffer. In the terminal window you have to use `Ctrl-Shift-C Ctrl-Shift-V`.
- Everything you select is automatically copied and can be pasted with the middle mouse button, or simultaneously pressing the left and right buttons.

3 Bash Basics

Start by launching Byobu, if not already running.

- Bash Manual
  `http://www.gnu.org/software/bash/manual/`

3.1 Navigating Directories

Filesystems are organized as *directories* or *folders*. To view the contents of the current directory, type:
`ls`

To change directory, use the command:
`cd` *directory name*

To create a directory, use the command:
`mkdir` *directory name*

To remove an empty directory, use the command:
`rmdir` *directory name*

To remove a directory and all of its contents (without possible undo), use the command:
`rm -rf` *directory name*

To know the current directory, use the command:
`pwd`

Some directory names have special meanings:

- `/` — The *root* directory

- `.` or `$PWD` — The current directory (process working directory)
- `..` — The parent of the current directory
- `-` or `$OLDPWD` — The directory where you were before the last `cd`
- `~` or `$HOME` — The current user home directory (if you type `cd` without an argument, you go here)

3.2 Managing files

To copy a file, use the command:
`cp` *source destination*

To move a file, use the command:
`mv` *source destination*

For both `cp` and `mv`, the `-r` (recursive) option makes it work on directories.

To remove a file, without possible undo, use the command:
`rm` *file*

3.3 Command Basics

`command` $\texttt{arg}_1$ $\texttt{arg}_2$ ... $\texttt{arg}_n$

When you enter the previous line in Bash, the `command` in the current directory, receiving as arguments $\texttt{arg}_1$ to $\texttt{arg}_n$. While it executes, the command will receive whatever you type in the shell in its *Standard Input*. The command also has two output channels: *Standard Output* and *Standard Error*. Both will be displayed in the terminal, but it's important to know there are two: there are tools to process them differently.

Notice the arguments are separated by spaces. To provide an argument that has a space, you have to surround it with quotes (`"`) or escape the space with a `\`.

Let's talk about `echo`, it's one of the simplest commands. It simply takes all of its arguments and prints them to the standard output separated by spaces. Notice that Bash interprets all of the arguments before passing them to the command. Arguments are passed separated, C programs receive them in the `argv` list. That is why these two commands have exactly the same output:

```
echo one two
echo one      two
```

The command `cat` works in a very similar way. However, instead of reading from the arguments, it reads from the standard input. If it has any argument, the argument is assumed to be a file name and it reads from the file instead. It is named `cat` from *concatenate*: It concatenates all of the files passed as arguments and prints them. Try `cat` without any argument, it will duplicate whatever you type (end with `Ctrl-D`):

```
cat
```

Or try reading from a file:

```
cat /etc/issue
```

Now, let's make this interesting. Here is a list of some available redirections:

- `c1 > file` or `c1 1> file` — Standard output of `c1` is written to `file` (overwriting).
- `c1 >> file` or `c1 1>> file` — Standard output of `c1` is appended to `file`.
- `c1 2> file` — Standard error of `c1` is written to `file` (overwriting).
- `c1 2>> file` — Standard error of `c1` is appended to `file`.
- `c1 &> file` — Both standard output and standard error of `c1` is written to `file` (overwriting).
- `c1 &>> file` — Both standard output and standard error of `c1` is appended to `file`.
- `c1 < file` — Standard input of `c1` is read from `file`.
- `c1 2>&1` — Standard error of `c1` is written to standard output.

- `c1 1>&2` or `c1 >&2` — Standard output of `c1` is written to standard error.
- `c1 | c2` — Standard input of `c2` is read from standard output of `c1`.
- `c1 |& c2` or `c1 2>&1 | c2` — Standard input of `c2` is read from both standard output and standard error of `c1`.

Try to test and combine the above! Note that `less` is a good way to see the contents of a file:
`less file.txt`

3.4 Chaining commands

Commands return a number when they terminate (specified by the `return` statement in the `main` function of C programs). By convention, 0 is success and anything not 0 is an error code. You can check the return code of the previous command with:
```
cat inexistentfile
echo $?
```

There are several ways to specify multiple commands at the same time (but not necessarily in the same line):

- `c1 && c2` — Execute `c2` only if and when `c1` terminates successfully.
- `c1 || c2` — Execute `c2` only if and when `c1` terminates without success.
- `c1 ; c2` — Execute `c2` after `c1`, regardless of the return code.
- `c1 &` — Execute `c1` in the background (use `fg` to bring it to foreground while it's running).
- `c1 & c2` — Execute `c1` in the background and `c2` right away.

3.5 Environment Variables

Many features of Bash and other programs are configured using *environment variables*. These are simple pairs of name and value, both text strings. To see all mappings defined, use the command `env`, piped to `less` since the list is quite long:
```
env | less
```

To set a variable, use the `export` command:
```
export TEST="Test variable"
```

To check the value of only one particular variable, you can feed it to the `echo` command, and Bash will replace it by the actual value before executing `echo`:
```
echo $TEST
```

One of the variables is quite relevant, and should be explained here: `$PATH`. This is a colon (`:`) separated list of directories, in which Bash will search for commands. Every time you type a command, bash will search for it in each directory in `$PATH` by order, until it is found.

3.6 Initialization

When Bash starts, it reads `/etc/profile` and then `~/.profile`.

- `/etc/profile`, by default, executes `/etc/bash.bashrc` and every script inside `/etc/profile.d/`.
- `~/.profile`, by default, executes `~/.bashrc`. Note that it also adds `~/bin` to `$PATH`, so you can have your executables easily placed in `~/bin`.

When you want to execute something every time Bash starts, you should add it to the end of `~/.bashrc`. Our ROS configuration will be put there!

If you want to run one of these files manually, you have several alternatives:

- `source file` — Reads file and executes every line in the current shell (this is the correct way to load `~/.bashrc` after changing it).
- `. file` — Same as above.
- `bash file` — Starts a new Bash. The only thing that new Bash will do is execute the script and exit. Therefore, any exports done there will not be applied to the current Bash.
- `sh file` — Almost the same as above, but uses `sh` instead of Bash. `sh` is the system default shell. By default, in Ubuntu, it is `dash`, a lightweight shell.

3.7 Nano Editor

Ubuntu comes with a very simple editor called `nano`. Try it:

```
nano test.txt
less test.txt
```

Notice the commands in the bottom lines, `Ctrl-X` will exit asking you if you want to save.

We will use `nano` in all command examples, but feel free to use whatever editor you are more comfortable with.

The default editor is defined in the `$EDITOR` environment variable. You can check yours with:

```
echo $EDITOR
```

Now set the default editor to `nano`. Edit `~/.bashrc` with:

```
nano ~/.bashrc
```

and, in the very end, add:

```
export EDITOR=nano
```

exit and save. Now reload `~/.bashrc`:

```
source ~/.bashrc
```

3.8 Managing Processes

To get a list of running processes, use:

```
ps aux
```

Htop is a nice program to view all running processes (some appear in duplicate because they have multiple running threads):

```
htop
```

To kindly ask one or more processes to terminate, you can use either of the commands:

`kill` *process numbers*
`killall` *process names*

When killed with the previous commands, the processes might not terminate. To kill them without giving a chance to refuse, use:

`kill -9` *process numbers*
`killall -9` *process names*

4 Shell Tools

4.1 Grep

`grep` is a simple tool to filter a stream. Use it as:

```
grep pattern file
```

or

```
cat file | grep pattern
```

The pattern is a regular expression. Find out more about regular expressions here: `http://www.grymoire.com/Unix/Regular.html`. Special characters have special meanings in regular expressions, but text characters have no special meaning. You can search for strings in text files easily:

```
grep name /proc/cpuinfo
```

4.2 Ack

For programmers, `ack` is a very useful tool. It is similar to `grep`, but automatically searches in the current directory and all subdirectories, only it source code files. For example, to find where the Boost Circular Buffer class is defined, you can use:

```
cd /usr/include/boost/
ack "class circular_buffer"
cd -
```

4.3 Sed

`sed` is a very flexible stream editor. The most common usage it to replace text in files. By default, it reads from standard input or a file specified as an argument, and prints the results to standard output:

```
sed 's/from/to/g' < file
sed 's/from/to/g' file
```

To change a file in place, use `-i`:

```
sed 's/from/to/g' -i file
```

Try this simple example:

```
sed 's/Ubuntu/OS/g' /etc/issue
```

A good tutorial can be found at: `http://www.grymoire.com/Unix/Sed.html`.

4.4 Awk

`awk` is a more complex tool. It has a simple programming language, which is very good to quickly process files. Invoke it as:

```
awk 'SCRIPT' < file
awk 'SCRIPT' file
```

The script usually looks like:

```
BEGIN {...} {...} END {...}
```

The `BEGIN` section runs before anything else and the `END` section runs in the end. The middle section runs once for every line. All three are optional. In the middle section, the variable `$0` represents the whole line, and variables `$1` to `$n` represent columns.

Let's see how to generate a sequence of numbers from 1 to 19 and calculate the average:

```
seq 19 | awk '{sum+=$1; count++} END {print "Average: " sum / count}'
```

4.5 Mmv

`mmv` is a simple tool to move multiple files. Invoke it as:

`mmv "`*source*`" "`*destination*`"`

In the source, `?` matches a single character and `*` matches as many characters as it can. In the destination, you use `#1` to `#n` for each `?` and `*` in the source.

For example:

```
mmv "Chapter ? - *.pdf" "Chapter about #2 number #1.pdf"
```

4.6 Tar

In Unix environments, compression of files usually happens in two steps: grouping files to compress and actually compressing. Compressing tools work only with one file. `tar` is a tool to group files. It is so flexible that it can invoke the compression command simultaneously. To compress, the best options are:

- `xz` - Extreme compression.
  `tar cavf archive.tar.xz file1 ... filen`
- `lzop` - Fastest compression, but not installed by default in Ubuntu.
  `tar cavf archive.tar.lzo file1 ... filen`

- gz - Old program, relatively fast, can be found everywhere.
  `tar cavf archive.tar.gz file1 ... filen`

To decompress, use:
`tar xavf archive.tar.??`

5 SVN

`http://subversion.apache.org/`
Subversion is a centralized version control system. It uses a single central repository from which users can create working copies.

- Version Control With Subversion
  `http://svnbook.red-bean.com/`

5.1 Setting up a repository

- Create a repository for these examples
  `svnadmin create /tmp/svndemo`
- Checkout the empty repository
  `svn checkout file:///tmp/svndemo`
  `cd svndemo`
- Create the initial structure
  `mkdir trunk tags branches`
  `svn add trunk tags branches`
- Commit (this will use your `$EDITOR`)
  `svn update`
  `svn status`
  `svn diff`
  `svn commit`
- Clean up
  `cd ..`
  `rm -rf svndemo`
- Now the repository is ready for work.

5.2 Using SVN

- We usually checkout only the trunk directory, and keep it around to work. For this tutorial, the repository is stored in `/tmp/svndemo/` but usually this in a SVN server.
  `svn checkout file:///tmp/svndemo/trunk svndemo`
  `cd svndemo`
- Edit some file, write some text
  `nano test.txt`
- SVN status will display an unknow file
  `svn status`
- So we must add it before commiting
  `svn add test.txt`
- And commit it
  `svn update`
  `svn status`
  `svn diff`
  `svn commit`

- If someone else was working on the repository, you should update frequently to get the latest changes
  `svn update`
- Edit the file again, change or add some text
  `nano test.txt`
- SVN status will display a modified file
  `svn status`
- The files must only be added once, so no need to add it again
- Then you can commit it right away
  ```
  svn update
  svn status
  svn diff
  svn commit
  ```
- Try checking what you have been doing
  ```
  svn update
  svn log -v | less
  ```

5.3 Saving a patch

This is a good metod to save work without committing it, so you can give it directly to someone else or keep it somewhere.

- Edit something, change or add some text
  ```
  nano test.txt
  svn status
  ```
- The file must be added so it appears as added or modified in `svn status`
- Save a patch (optionally compressing it)
  ```
  svn diff > work.patch
  xz -9ev work.patch
  ```
- Now you can undo your changes, so the local copy reflects the repository again
  ```
  svn revert -R .
  svn status
  ```
- Apply the patch, and your work should be back
  ```
  xzcat work.patch.xz | patch -p0
  svn status
  ```

6 Git

`http://git-scm.com/`

Git is a distributed version control system. Although central repositories can be created in servers, every copy contains the full repository and users can synchronize their copies directly without passing through the server.

- A successful Git branching model
  `http://nvie.com/posts/a-successful-git-branching-model/`
- GitHub help
  `https://help.github.com/`
- Pro Git
  `http://git-scm.com/book/`

- Git Magic
  `http://www-cs-students.stanford.edu/~blynn/gitmagic/`

6.1 Setting up a repository

- First, configure some global Git variables.
  ```
  git config --global user.name "Your Name Here"
  git config --global user.email "your_email@example.com"
  git config --global color.ui auto
  ```
- To create an empty repository, just run `git init` on your working directory
  ```
  git init gitdemo
  cd gitdemo
  ```

6.2 Using Git

- Edit some file, write some text
  `nano test.txt`
- Everytime you want to commit a file, you must first add it to the stage
  ```
  git status
  git add test.txt
  git diff --staged
  ```
- Only then you can commit
  `git commit`
- Edit the file again, change or add some text
  `nano test.txt`
- If it's only a few files, you can add to the stage and commit in the same command, but only if the files are already known to Git
  ```
  git status
  git diff
  git commit test.txt
  ```
- Adding to the stage may seem an unecessary extra step, but it allows us to have more control over commits. Commits should be simple and as small as possible, to better describe the changes and to make it easier to find bugs. You can add only the necessary files to the stage. You can even add only some changes inside files!
- You can check what you have been doing with
  `git log | less`
- Or use the `gitk` tool:
  `gitk --all`

6.3 Using branches

Git makes it very easy to work with branches, like different versions of the same repository. Git organizes commits as a tree, and branches are labes to a specific commit.

- The default branch is called `master`. You can check the branches with
  `git branch -a`
- Create a new branch from `master`, name it `test`
  `git checkout -b test`
  The `-b` flag creates a new branch. Now `master` and `test` point to the same commit. Only when you commit something will the commit tree branch.

- Edit some file, write some text and commit it
  `nano test.txt`
  `git status`
  `git diff`
  `git commit test.txt`
- Switch back to the `master` branch
  `git checkout master`
- To merge, you have two options:
  - Without *fast-forward*, the merge will always be a commit with the merged code. This will make it obvious that a branch was used to develop the code when looking at the commit history. This is the recommended way to merge the `test` branch, so that commits done to that branch will be easier to spot when looking at the commit history.
    `git merge --no-ff test`
  - Using *fast-forward* (only when possible), if the current branch is a predecessor of the branch to be merged, the current branch will simply be fast-forwarded. You will not have a commit for the merge and you will not be able to tell in which branch the code was developed. This is useful when merging an updated version of the branch from someone else.
    `git merge test`
- Then you can delete the test branch
  `git branch -d test`
- Create a new branch to be used as an example later, with some modifications, and leave it in the `master` branch.
  `git checkout -b test2`
  `nano test.txt`
  `git status`
  `git diff`
  `git commit test.txt`
  `git checkout master`

6.4 Using remotes

- Git repositories stored in servers, like GitHub, are *bare* repositories that are not prepared to work there directly, but to receive modifications from others. Let's create one of those. `cd ..`
  `git clone --bare gitdemo/ gitdemo.git`
- Now let's try cloning the bare repository. This is the same process used for public repositories, `gitdemo.git` could be a web link.
  `git clone gitdemo.git gitdemoclone`
  `cd gitdemoclone`
- Notice this repository has only the master branch, but knows about the remote existance of `master` and `test2`. The local `master` branch is configured to track the remote `origin/master`. That is, it will push and pull from that remote branch later.
  `git branch -avv`
  `git remote show origin`
- Now you can make some changes and commit them
  `nano test.txt`
  `git status`
  `git diff`
  `git commit test.txt`
- After all your changes, push your commits to the remote repository. This will only push the current branch.
  `git push origin`

- To ckeckout a branch that does not exist locally, use
  `git checkout -b test2 origin/test2`
- You can make some changes in this branch and the push all branches to the remote repository with
  `git push --all`
- To get updated code from the server, you must first *fetch* to download the origin branches, and then merge
  `git fetch origin`
  `git merge test2 origin/test2`
- Or, if the local branch is tracking the remote branch, you can do this in one step only with the shortcut
  `git pull`

6.5 Working model

Git is very flexible, and using branches can be confusing. The branching model linked above proposes the following:

- The `master` branch should always be stable, tested and demo-ready.
- The `develop` branch contains code that should be working, but we are not so sure it is correct and bug-free as the `master` branch. Merge `develop` into `master` only when you are sure it is stable and well tested.
- Create feature branches when needed. When implementing a new feature, create a new branch with a descriptive name. Use that branch while working and testing. You can merge from `develop` frequently to keep the branch updated. Merge into `develop` when the implementation is stable and tested.

When working on servers like GitHub, the official version of a project usually has a well-known repository. To contribute to it, you fork the repository online, creating your own copy in your GitHub account. Your copy is the remote `origin` while the main repository of the project is usually known as `upstream`. When you clone your repository to your computer to work, you can push all your branches to GitHub. From `upstream` you only fetch. To contribute to the main repository of the project, you create a *pull request* online in GitHub, that whoever decides for the project will decide to accept or not.

7 ROS Setup

Now that you know about `~/.bashrc`, the setup instructions of ROS should make more sense. Make sure your `~/.bashrc` ends with a line including the ROS setup script.

To make sure everything is ok, try:

```
rospack profile
env | grep ROS
```

You should see a list full of `/opt/ros/groovy` directories.

8 ROS Concepts

8.1 Topic

Topics have two major modes of operation:

- *Latched* — When a publisher publishes a message, it is sent to every connected subscriber. The publishers keep the last sent message in memory. When a new subscriber connects, the last published message is immediately sent to it. Use this when you want the last published message, independently of how old it is.

- *Not latched* — When a publisher publishes a message, it is sent to every connected subscriber and forgotten about. Use this when you want messages to be events, that are only important in the moment when they are produced and not in the future.

Topics have a queue size, set it to:

- 1 — If you only care about the most recent message (ex. when publishing messages). This is the most common situation.
- 100 (or some other large value) — If you care about the sequence of messages or want to receive all messages (ex. detected persons in a surveillance system). Note that you might still loose messages if the queue gets full. This only happens if the system is overloaded, and there is no way around this limitation.

8.2 Parameters

Use parameters only for configuration parameters. These are either:

- Things that do not change during execution (ex. the robot number);
- Things that can only be changed manually and not likely to change (ex. parameters of some algorithm that you might want to finetune, but will be static after this).

Note that parameters cannot be stored in ROS Bags. Therefore, if two nodes communicate using parameters, offline development and debugging will be much more difficult.

Parameters can be cached in the nodes to make access much faster. Note, however, that this places a great load on the master: it has to know what parameters the nodes have cached, and update them every time a parameter gets updated. This is the best option for parameters that are accessed often and rarely change: those of the second type listed above.

8.3 Services

Services are not that common in practice. Most times, when you want to transmit a message directly to a node, you still use a topic. Services cannot be stored in ROS Bags. Use services only when you want to make a request and have a specific response to that request. Services are useful to query other nodes for exact data or offload computation to another node (which has to be shared or reside in a different computer, or else a library is a better option).

8.4 Infrastructure

A ROS Core is composed of three components:

- ROS Master — Central nameserver, knows about all nodes, topics and services and where to find them. Note that topics and services communicate directly between nodes, but first have to ask the master the address of the publisher/server.
- ROS Parameter Server — Stores the parameters.
- `rosout` node — Merges logging output from all nodes so that it can be easily read.

A ROS Core must always be running in a ROS System. Using the `roslaunch` command automatically launches the ROS Core, but in other situations it has to be launched with the `roscore` command.

8.5 Changes in ROS Groovy

From ROS Fuerte to ROS Groovy, several big changes took place. Two are specially important:

- Change from `rosbuild` to `catkin` — The packaging system was redesigned. The concept of *stacks* disappeared, being replaced by *metapackages*. The manifest file in packages was renamed from `manifest.xml` to `package.xml`. Some tools to create and manage packages were also changed.

- Change to Qt — Graphical tools were changed to use the Qt framework. Names changed: Before, graphical tools started with `rx` (ex. `rxconsole`). Now, they start with `rqt_` (ex. `rqt_console`). There is a main tool `rqt` from which all others can be accessed.

9 ROS Basics

Now, all those F-keys of Byobu will start to be pretty useful, since you have to run many commands simultaneously. Remember: `F2` creates a new window, `F3` and `F4` alternate windows, `Ctrl-C` terminates a program (ROS terminates cleanly with `Ctrl-C`) and `Ctrl-D` closes the standard input of Bash, terminating it and closing the window.

In this section, I'll summarize a few ROS tutorials about the basics:
`http://www.ros.org/wiki/ROS/Tutorials/UnderstandingNodes`
`http://www.ros.org/wiki/ROS/Tutorials/UnderstandingTopics`
`http://www.ros.org/wiki/ROS/Tutorials/UnderstandingServicesParams`
`http://www.ros.org/wiki/ROS/Tutorials/UsingRqtconsoleRoslaunch`

9.1 Core

In one Byobu window run
`roscore`
and leave it running.

9.2 Nodes

In another Byobu window run
`rosrun turtlesim turtlesim_node`
and leave it running. `rosrun` is one way to launch nodes in ROS. It is the same as running the node executable directly, but with `rosrun` you don't have to know the full path, only the package name is necessary.

In another Byobu window run
`rosrun turtlesim turtle_teleop_key`
and leave it running. Now you can move the turtle with the arrow keys!

In another Byobu window run
`rosnode list`

This will list all running nodes, which is very useful in more complex situations. You can also try
`rosnode info /turtlesim`
to get more information about the node.

9.3 Topics

To get an overview of the system, you can try
`rqt_graph`
or simply
`rqt`
and then select the *ROS Graph* plugin.

`rostopic` is one of the most useful commands in ROS. Get the list of what it can do with
`rostopic help`

Listing topics is an everyday operation in ROS:
`rostopic list`

You can see in real time what is passing in a topic:
`rostopic echo /turtle1/command_velocity`
then go to the window where teleop is running and give it some command.

There is a nice way to see numerical topics:
`rqt_plot /turtle1/command_velocity/linear`

Publishing to a topic is quite easy. First you must know the type of the topic:
`rostopic info /turtle1/command_velocity`
then, check the message fields so you know what to publish:
`rosmsg show turtlesim/Velocity`
then you can publish directly from the command line:
`rostopic pub -1 /turtle1/command_velocity turtlesim/Velocity "angular: 1"`
or, if you want to specify the whole message:
`rostopic pub -1 /turtle1/command_velocity turtlesim/Velocity -- 0 1`

The data to publish in in YAML language. See `http://www.ros.org/wiki/ROS/YAMLCommandLine` for details.

9.4 Services

The `rosservice` command works similarly to `rostopic`. Try:

```
rosservice help
rosservice list
rosservice info spawn
rossrv show turtlesim/Spawn
rosservice type spawn | rossrv show
rosservice call spawn 2 2 0.2 ""
```

Note that it is not possible to echo services.

9.5 Parameters

Try these examples:

```
rosparam help
rosparam list
rosparam get background_g
rosparam get /
rosparam set background_g 200
```

Turtlesim is programmed in a way that we have to redraw for this to take effect:

```
rosservice call clear
```

There are dump and load commands, that dump and load parameters from files. The load command is very useful to load robot configuration on initialization. `roslaunch` has a command to make this very simple.

9.6 Logging

Run these two commands simultaneously:

```
rqt_logger_level
rqt_console
```

Now, go to the teleop window and hit a wall. In the `rqt_logger_level` window, check the nodes `/teleop_turtle` and `/turtlesim`. In the loggers list, both have a logger named `ros.turtlesim`. This is the logger controlled by the program. Try changing each to *Debug* and move the turtle.

About the levels: `http://www.ros.org/wiki/Verbosity%20Levels`.

9.7 Bag Files

ROS bags are a easy way to record topic messages. You can record multiple topics into the same bag. Try it with:

```
rosbag record /turtle1/command_velocity /turtle1/pose -o testbag
```

Now you have a bag recorded. Note that `testbag` is the prefix to the bag file name, you can record multiple bags with this command and the output is not overwritten. There is also a `-a` option that records every topic, but this is not good because it records all the logging output (and with image publishers it records the images in multiple formats) making the bag huge.

Now try to play the bag. The `-l` option makes the bag play in loop, this is very useful during development.

```
rosbag play -l testbag_*.bag
```

There is a nice tool to inspect bags:

```
rqt_bag testbag_*.bag
```

Note that to compress bags for storage, you don't have to use `tar`, but can use `xz` directly:

```
xz -9ev testbag_*.bag
xz -d testbag_*.bag.xz
```

10 Creating Packages

ROS is organized in *packages*, that might contain nodes, messages types, scripts, and so on. In ROS Groovy, the system to manage packages is called *catkin*. You have to create a catkin workspace in your home, and inside of it you can create packages.

10.1 Catkin Workspace

Let's create the catkin workspace:

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
cd ~/catkin_ws/
catkin_make
source devel/setup.bash
```

This last line sets up ROS to use your workspace. Add it to your `~/.bashrc` after the ROS initialization.

10.2 Creating a package

To create a package, go to the `src` directory of your workspace and use catkin:

```
cd ~/catkin_ws/src
catkin_create_pkg test_package roscpp
```

In this case, `test_package` is the name of the package and it depends on `roscpp`. C++ code for ROS uses libraries that are in the package `roscpp`. Python code will need `rospy`. A package can depend on both simultaneously, and any other packages. To add dependencies after a package is created, edit the file `package.xml`.

10.3 ROS Package Tools

There are two useful commands to navigate packages: `roscd` and `rosls`. From any directory you can do:

```
roscd test_package
```

and it changes to the package directory. Similarly, to list contents use:

```
rosls test_package
```