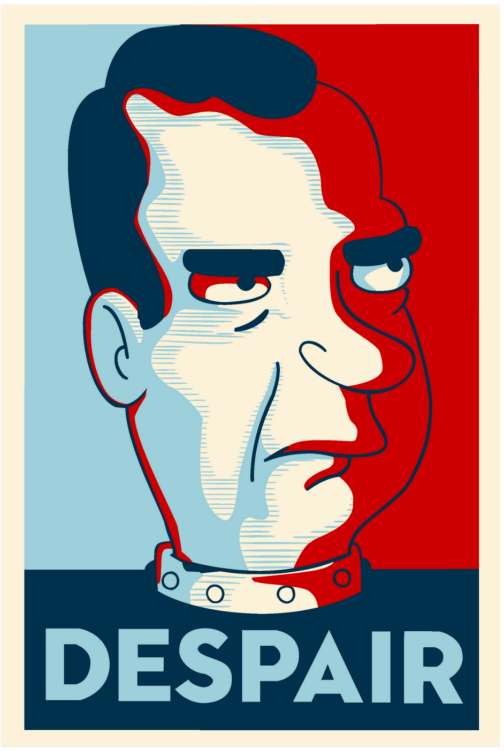
⋆⋆⋆ Make ROS Great Again ⋆⋆⋆

ICS Guide to Surviving the Robotic Apocalypse (and Incidentally Code some Cool Stuff)

M.Sc. Quentin Leboutet, M.Sc. Julio Rogelio Guadarrama Olvera, M.Sc. Constantin Uhde, and Prof. Dr. Gordon Cheng



Institute for Cognitive Systems
Faculty of Electrical Engineering and Information Technology
Technische Universität München

Contents

Part I

ROS: Fundamental Concepts

Chapter 1

Introduction to ROS

The purpose of this chapter is to provide the reader with a both conceptual and formal overview of ROS. After reading this chapter, the reader should be able to explain what ROS is, how it works, what its main features are, and above all why it has become so essential in robotics. Students who are already familiar with these concepts can directly jump to chapter 2.

1.1 ROS: Conceptual Overview

1.1.1 What is ROS ?

Initially started in 2007 by the *STanford Artificial Intelligence Robot laboratory* (STAIR) and the Willow Garage robotic institute/incubator, the **Robot Operating System** (ROS)[1] – formerly known as *Switchyard* project – is a flexible open-source framework for writing robot softwares.

Figure 1.1: ROS, Willow garage and Open Source Robotics Foundation logos

In practice, ROS is essentially a vast collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a large number of robotic platforms. Its greatest asset lies in its powerful **generalized TCP/IP communication framework** allowing to easily and efficiently handle the variety of information transiting in a robot (sensor data, video, control orders and so on). ROS is built around two main components, namely:

1. the **core software**, developed and maintained by the *Open Source Robotics Foundation* (OSRF) which provides both the communication network and the general system architecture.
2. a huge database of task-specific **libraries and packages**, freely accessible on the ROS website and maintained by a strong community made of hundreds of roboticists from all over the world.

[1] Unlike what is suggested by its name, ROS is not, strictly speaking, an "*operating system*" such as *Ubuntu*, *Debian* or *Windows* but rather a powerful open-source, cross-platform and cross-language **middleware** environment which is specifically designed and optimized for robotics applications. ROS is an opensource project, therefore you can find its entire source code at the following address: `https://github.com/ros`.

1.1.2 Why is ROS so interesting in Robotics ?

As robots become more sophisticated, the amount and complexity of the data they generate increases dramatically. As a result, conventional sequential software architectures are no longer suitable due to performance, robustness and portability issues. In this context, the use of a specific middleware becomes interesting – or even mandatory – as it allows to efficiently handle synchronous (e.g. IMU data) and asynchronous (e.g. video streams) data flux circulating within a robot, while at the same time providing a set of high-level features intended to accelerate debugging and simulation steps. Several middlewares were developed during the last decade, such as YARP, Orocos or MRPT. The properties that makes ROS different (and way more interesting) are its:

- **High-end capabilities**: such as SLAM (Simultaneous Localization and Mapping) or motion planning algorithms, allowing to speedup the implementation of a broad range of powerful applications. These capabilities are already state of the art since they are maintained by specialized research institutes all around the world.
- **Tons of high level tools**: allowing to debug, visualize robot data, and perform realistic simulations. Among these tools, the most commonly used are:
  - **rviz**: a graphical user interface allowing to display robot models, navigation maps (*octomaps*), or simply the raw data provided by most of the robot sensors (cameras, IMU, LIDARs, force/torque sensors, and so on...).
  - **rqt**: An incremental control interface, based on the plugin system of the GUI Qt library.
  - **rosbag**: A program to record and replay information streams circulating on the robot.
  - **catkin**: a package management system, providing automatic code generation and build functionalities.
  - **gazebo**: a physical engine for robot simulation...
- **Inter-platform operability**: The ROS message-passing middleware allows communicating between different nodes. These nodes can be programmed in *any language that has ROS client libraries*. We can write high performance nodes in C++ or C and other nodes in Python or Java. This kind of flexibility is not available in other frameworks.
- **Modularity**: Usually if a part (thread) of the main program crashes, the entire robot application stops. In ROS, if one process (node) crashes, the system may still remain operational. ROS also provides robust methods to resume operation even if any sensors or motors are dead.
- **Concurrent resource handling**: Suppose we want to process an image from a camera for both *face detection* and *motion detection* purposes: we can either write the code as a single entity that achieves both tasks sequentially, or as a threaded process for parallel execution. If we want to add more than two features in threads, the application behavior will get complex and will be difficult to debug (when ill-managed, sheared memory spaces may result in nasty behaviors such as dead-locks). In ROS, any number of processes (nodes) can safely subscribe to the same source of information (topic) and perform different functionalities in a concurrent manner.
- **Support of high-end sensors and actuators**: ROS is packed with device drivers and interface packages of various sensors and actuators in robotics. The high-end sensors include Velodyne-LIDAR, Laser scanners, Kinect, and so on and actuators such as Dynamixel servos.
- **Active community**: When we choose a library or software framework, especially from an open source community, one of the main factors that needs to be checked before using it is its software support and developer community. There is no guarantee of support from an open source tool. Some tools provide good support and some tools don't. In ROS, the support community is active. There is a web portal to handle the support queries from the users too (`http://answers.ros.org`).

1.2 ROS: Formal Overview

1.2.1 ROS computational graph

Computations in ROS are performed through a network of **concurrent** processes called **nodes**. Within this network – known as the **ROS computation graph** – nodes can communicate with each other in a **synchronous** or **asynchronous** manner using a set of dedicated routines[2]. The main elements of the ROS computation graph are the following:

- **Nodes**: Nodes are processes that perform computation. A robot usually contains many nodes, each one performing a set of various tasks such as path planning or sensor data processing. Nodes can communicate with each other via "**topics**", "**services**" or "**actions**". Dedicated software libraries are made available to the user so new nodes can be easily developed in C++ or in Python. In practice, nodes can be started, killed, and restarted, in any order, without inducing any error conditions.

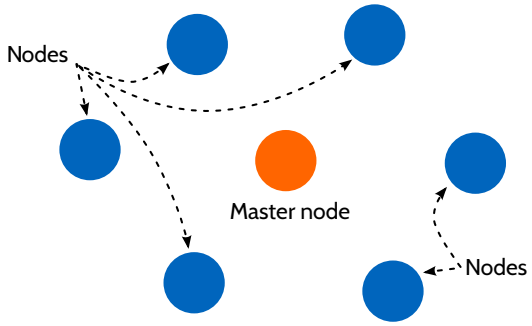
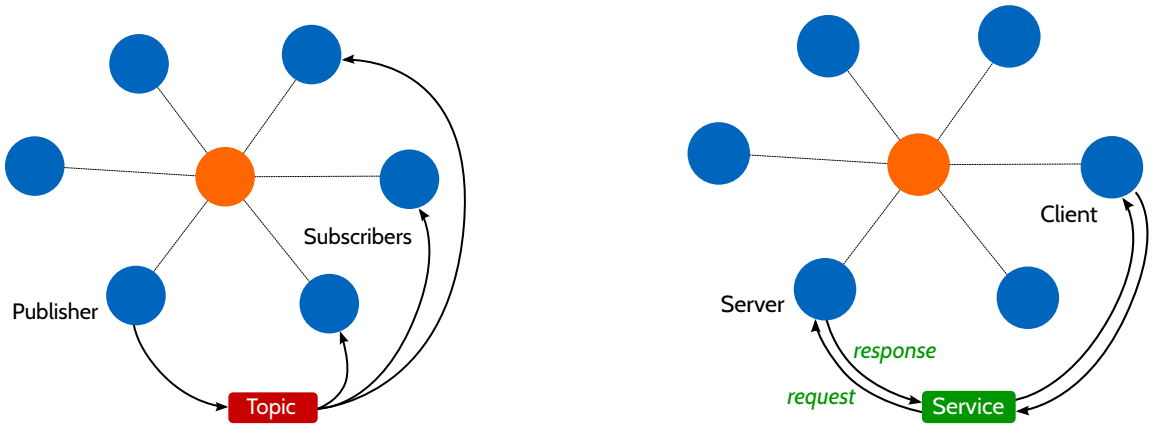


Figure 1.2: Some components of the ROS architecture.

- **Master node**: The ROS Master node aims at coordinating the different nodes of a given ROS system, acting as a name service that stores registration information from the different ROS nodes. Nodes communicate with the Master to report their connection to the data streams generated by other nodes. In practice, this is achieved automatically. More information can be found at the following address: `https://wiki.ros.org/Master`
- **Topics**: Topics can be defined as *asynchronous communication protocols* between the different nodes. An interesting characteristic of topics is that they "*decouple the production of information from its consumption*" which is particularly useful in robotics – especially for computer vision purpose – when data cannot be obtained "*on request*", but in stead, comes as a continuous stream. A node can either connect with a topic as a "*publisher*" (in order broadcast a data stream), or as a "*subscriber*" (in order to receive data from other topics). An arbitrary number of publishers and subscribers can be generated with a single topic.
- **Services**: Services are the *synchronous alternative to topics*, taking the form of a remote procedure call ("*request/answer*") instead of a "*publish/subscribe*" model. By using a service, a given node will only receive data on request. It is interesting to notice that the received data may change with the request, allowing a good flexibility.
- **Actions**: If a service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing. The `actionlib` package provides tools to create servers that execute long-running goals that can be preempted. It also provides a client interface in order to send requests to the server.

[2]The ROS communication related packages, including the core client libraries, such as `roscpp` and the implementation of concepts such as topics, nodes, parameters and services, are included in a meta-package called "`ros_comm`".



(a) ROS Topic (b) ROS Service

Figure 1.3: Illustration of the difference between topics and services

- **Bags**: Bags refer to a format used for saving and playing back ROS message data. ROS bags can for example be imported on Matlab in order to display the data collected dutring an experiment.
- **Parameter Server**: Programming a robot often requires to define a set of parameters, such as the gains of a PID controller. When the number of parameters increases, it becomes interesting to store them withing dedicated files. Moreover, in some situation, parameters have to be shared between two or more programs. The ROS parametrer server is a **shared server providing access to the parameters to every ROS node**. A node can read, write, modify and delete parameter values from the parameter server. The parameter server is currently part of the Master.
- **Messages**: Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs).

1.2.2 ROS compiler

ROS uses its own build system – named **catkin** – combining CMake macros and Python scripts to provide additional functionalities. A build system is responsible for generating "*targets*" from raw source code that can be used by an end user. These targets may be in the form of libraries, executable programs, generated scripts, exported interfaces (e.g. C++ header files) or anything else that is not static code. In ROS terminology, source code is organized into "**packages**" where each package typically consists of one or more targets when built. To build targets, the build system needs information such as the locations of tool chain components (e.g. `gcc, g++`...), source code locations, code dependencies, external dependencies, where those dependencies are located, which targets should be built, where targets should be built, and where they should be installed. This is typically expressed in some set of configuration files read by the build system. With CMake (and catkin), these informations are specified within a file named "**CMakeLists.txt**". The build system uses these informations to process and build source code in the appropriate order, and eventually to generate targets (c.f. `https://wiki.ros.org/catkin/conceptual_overview`).

1.2.3 ROS file system

The ROS filesystem refers to the resources you may encounter on your disk, once ROS is installed:

- **Packages**: The atomic item in the ROS software. The packages combine the runtime processes (nodes) with their associated libraries, configuration files, headers (and so on) in a single unit.
- **Metapackages**: The term meta package refers to a group of packages used for a special purpose. An the examples of a meta package is the ROS navigation stack, where each package provides dedicated navigation functionalities.
- **Repositories**: Most of the ROS packages are maintained using a Version Control System (VCS) such as git, subversion (svn), or mercurial (hg). A collection of packages that share the same VCS can be called repositories. The package contained in a repository can be released using a catkin release automation tool called bloom.
- **Message (msg) types**: ROS messages refers to the information circulating from one ROS node to the other. It is possible to define a custom message inside a package[3]. The extension of the message file is *.msg*.
- **Service (srv) types**: ROS services are a kind of **request–reply** interaction between ROS nodes. The reply and request data types can be defined inside the srv folder inside the package[4].
- **Action (action) types**: ROS actions can be seen as services with special features. The goal, result and feedback data types can be defined inside the action folder inside the package[5].
- **Package Manifests**: A manifest (`package.xml`) provide metadata about a given ROS package, including its name, version, description, license information, and dependencies.
- **Meta packages manifest**: Similar to the package manifest; differences are that it might include packages inside it as runtime dependencies and declare an export tag.

ROS packages can be built as standalone projects – simillarly to normal cmake projects – but catkin also provides the concept of **workspaces**, where it is possible to build multiple, **interdependent** packages together all at once. A schematic overview of a classic ROS workspace is provided in Fig. 1.4. A good practice for organizing ROS packages within a workspace is to group them together for **functional consistency**. For example, if a group of roboticists is developing a set of control algorithms while another group is working on advanced computer vision, it is better to group the corresponding packages into two separated meta-packages: one for control and one for computer vision.

1.2.4 ROS community

One of the main ideas behind ROS is to speedup the development of cool robotic applications by allowing separate communities (i.e. laboratories, companies or individuals) to exchange software and knowledge based on the same set of conventions. The resources provided by the ROS community include:

- **Repositories**: ROS relies on a federated network of code repositories, where different institutions can develop and release their own robot software components.
- **ROS Wiki**: The ROS community Wiki is the main forum for documenting information about ROS. Anyone can sign up for an account and contribute their own documentation, provide corrections or updates, write tutorials, and more.

[3] `my_package/msg/MyMessageType.msg`
[4] `my_package/srv/MyServiceType.srv`
[5] `my_package/action/MyActionType.action`

- **ROS Distributions**: ROS Distributions are collections of versioned metapackages that you can install. Distributions play a similar role to Linux distributions: they make it easier to install a collection of software, and they also maintain consistent versions across a set of software. The current LTS release is **ROS Kinetic**. It is compatible with ubuntu 16.04, and will be supported until 2021.

1.3 (Not so) dumb questions about ROS

1.3.1 Why has ROS a custom build system?

This is a long story. For any metaphysical question about catkin, please refer to `https://wiki.ros.org/catkin/conceptual_overview`.

1.3.2 Is ROS real-time safe ?

While ROS is **fast**, it does not provide **guarantees about the timing of operations** (c.f definition of a real-time system)[6]. Therefore it should not – in theory – be used for operations that have strict timing requirements, such as high-frequency PID and motion control. Many ROS robots will implement their timing-sensitive operations either on an embedded controller which communicates with a computer running ROS, or as a ROS node with separate real-time threads that communicate through a non-blocking API to ROS threads within the same node. The latter approach here requires a Linux kernel with special real-time extensions that help guarantee real-time scheduling of user-space threads. Some of the particular things that ROS does that can violate real-time constraints are:

- Using a network-based transport:
  - Ethernet and IP are unreliable, but have minimal transport latency.
  - Standard switches and routers introduce a non-deterministic amount of latency, particularly when there is other traffic on the network.
  - The handshake and retry mechanisms that make TCP reliable also introduce a significant amount of latency.
- Many of the ROS message types and other APIs do memory allocation internally. In a Linux system, this can cause a context switch[7] which has the potential to break real-time.
- The ROS message queue will drop messages when they become full.

Some dedicated branches of ROS such as "ROS 2.0" `https://github.com/ros2` are addressing this issue. However they are still (very) experimental and currently under heavy development.

1.3.3 Is ROS robust enough for industrial environments ?

The main focus of ROS is robotics research. The idea is to rapidly develop cross-platform applications that can be shared with a broad community. Industrial environment has other imperatives. A special branch of ROS named "*ROS industrial*", (`https://rosindustrial.org/`) was specifically developed in order to deal with these considerations. So far, it is still "experimental" and under active development.

[6] This part is extracted from the ros wiki, at the following address: `https://answers.ros.org/question/134551/why-is-ros-not-real-time/`

[7] A context switch is the process of storing the state of a process or of a thread, so that it can be restored and execution resumed from the same point later. This allows multiple processes to share a single CPU, and is an essential feature of a multitasking operating system. (c.f. `https://en.wikipedia.org/wiki/Context_switch`)

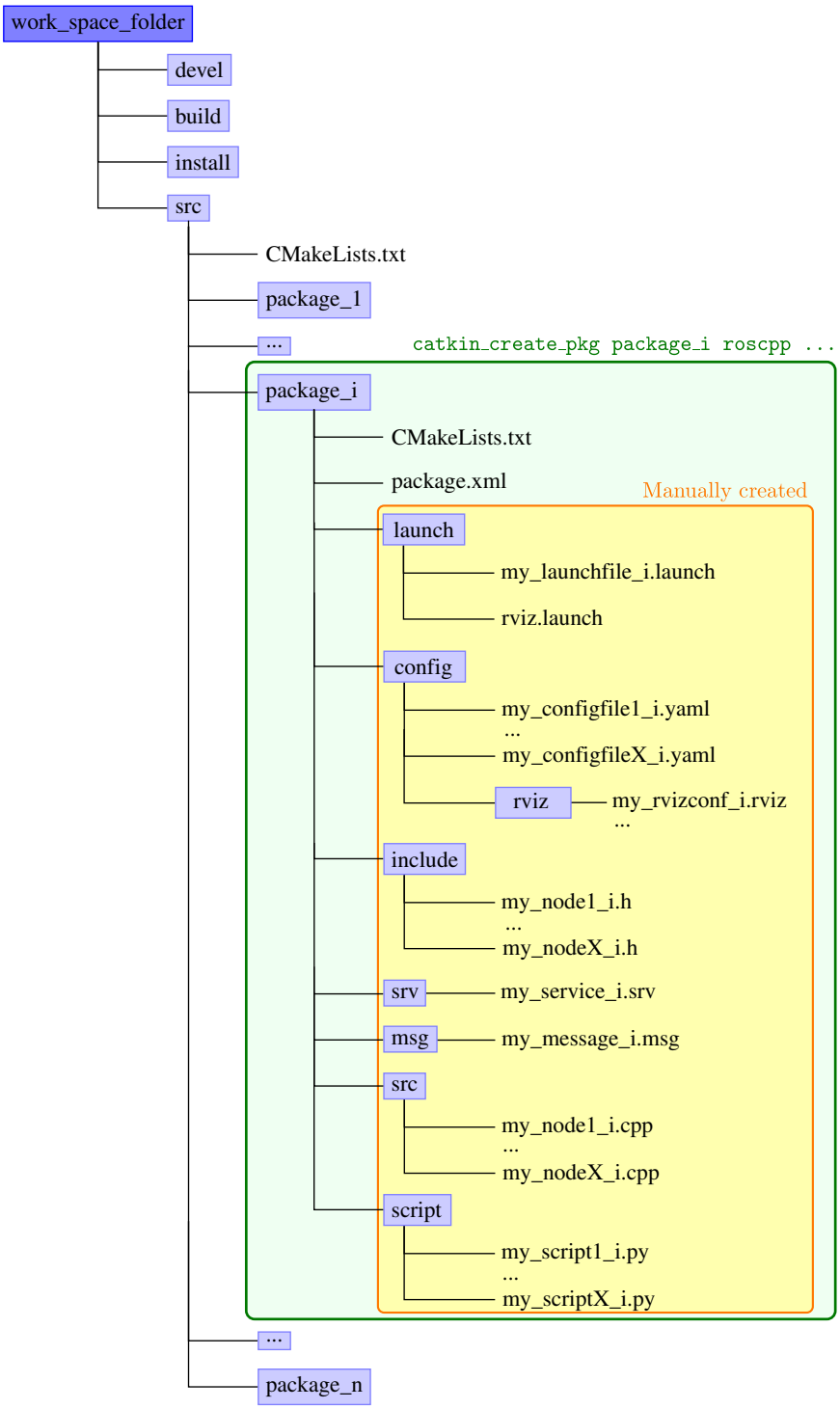


Figure 1.4: General architecture of a ROS package, within a ROS workspace (credits: LAAS, Dr. Olivier Stasse). Only the CMakelist.txt and the package.xml are required within the package folder. The other blocks can be added if necessary. Note that there can be an arbitrary number of launch files, source files, messages, services, actions and scripts within a single package, and an arbitrary number of packages or meta packages within a given workspace.

Chapter 2

Getting started with ROS

The purpose of this chapter is to make the reader capable of creating and configuring a ROS development environment. The emphasis is put on understanding the ROS package system. At the end of this chapter, the reader should be able to create its own packages and execute its own ROS nodes. Many concepts, such as CMakeLists or launch files are only briefly presented and will be discussed in more detail in chapter 5.

2.1 Setting up a new ROS development environment

2.1.1 ROS installation

The current Long Term Support (LTS) release of ROS is "**ROS Kinetic**". It is fully compatible with Ubuntu 16.04. However many applications or lectures still use the previous version of ROS (i.e. "**ROS indigo**"), which is fully compatible with Ubuntu 14.04. The installation procedure of ROS kinetic is described in details in the following tutorial: `https://wiki.ros.org/kinetic/Installation/Ubuntu`. **Important:** ROS has backwards compatibility issues. Therefore, packages that are labelled for a specific distribution (e.g. ROS fuerte, hydro, indigo, kinetic...) may have issues with other distributions. This is explained in details in the following thread: `https://wiki.ros.org/Distributions`.

(a) ROS fuerte (2012)

(b) ROS hydro (2013)

(c) ROS indigo (2014)

(d) ROS kinetic (2016)

2.1.2 ROS environment variables configuration

2.1.2.1 Make Linux aware of ROS

Using ROS requires a proper configuration of the shell environment. Under Linux, it is necessary to add the following line to the `.bashrc` file:

```
source /opt/ros/kinetic/setup.bash
```

If this is not done properly, Linux will not recognize the different ROS commands (e.g. `roscore`, `roslaunch`...)

2.1.2.2 The ROS Environment Variables on Linux

On Linux, the ROS environment variables can be displayed using the following command:

```
$ env | grep ROS # Display the environment variables related to ROS
```

A detailed overview of the different ROS environment variables is given at the following address: `https://wiki.ros.org/ROS/EnvironmentVariables` In practice, a set of tree very important environment variables have to be checked, namely the `ROS_MASTER_URI`, the `ROS_PACKAGE_PATH` and the `PYTHONPATH`:

1. `ROS_MASTER_URI` tells nodes where they can locate the master. It has the following structure: `ROS_MASTER_URI=http://hostname:port_number`. Here **hostname** indicates the computer on which roscore is launched, while **port_number** indicates the port where roscore is waiting for connections. The basic setting is "`http://localhost:11311`", which means that roscore will be executed locally.
2. `ROS_PACKAGE_PATH` indicates where to look for packages. By default it indicates the system packages of the ROS release: `/opt/ros/kinetic/share`. This variable will be updated during the overlaying process of a workspace (c.f. section 2.1.3.2). If there are multiple packages of the same name, ROS will choose the one that appears on `ROS_PACKAGE_PATH` first.
3. `PYTHONPATH` is mandatory since many ROS infrastructure tools rely on Python. The pythonpath must be properly set even if Python is not used within the developed nodes.

2.1.2.3 How to read and set ROS Environment Variables on Linux

- Environment variables can be read using the `echo` command with the following syntax:

```
$ echo $PYTHONPATH
# Provides a colon-separated list of directories in which the shell
    looks for commands.
```

- Environment variables can be set using the `export` command. For example, the command line to add the directory "`/home/<myuser>/Documents`" at the end of the `PYTHONPATH` is:

```
$ export PYTHONPATH=${PYTHONPATH}:/home/<myuser>/Documents
```

- **These settings are only temporary !** To make them permanent, copy the corresponding export commands into the `.bashrc` file.

2.1.3 Creating, Building and Overlaying a ROS Workspace

2.1.3.1 Creating and initializing a ROS workspace

The first thing to do after a proper ROS installation is to create a general folder, which will contain the different *workspaces*:

```
$ cd                          #Move to the home directory
$ mkdir -p ros/workspaces     #Create a new directory for your ROS workspaces
$ cd ros/workspaces           #Move to this subdirectory
```

A new ROS workspace can be created and then initialized using `catkin_init_workspace`:

```
$ mkdir -p myWorkspace/src    #Create your workspace directory
$ cd myWorkspace/src          #Switch to the src subfolder.
$ catkin_init_workspace       #Initialize your new workspace
```

The `catkin_init_workspace` command will create a `CMakeLists.txt` file within the "`src`" folder of the considered workspace (actually it just create a symbolic link, that points at the following location: `CMakeLists.txt` → `/opt/ros/kinetic/share/catkin/cmake/toplevel.cmake`).

2.1.3.2 Building a ROS workspace

Once a workspace is created, it is necessary to compile it:

```
$ cd ~/ros/workspaces/myWorkspace
$ catkin_make #Build the empty workspace
```

This works well even if the workspace is empty (i.e. there are no packages in the "`myWorkspace/src`" folder). Notice the "`build`" and "`devel`" folders within the root directory of the workspace (c.f. Fig.1.4):

- The "`build`" folder mainly contains executables of the nodes that are placed inside the respective packages "`src`" folders.
- The "`devel`" folder contains bash script, header files, as well as different executables generated during the build process.

2.1.3.3 Overlaying a ROS workspace

After building the workspace, it is necessary to make it **visible to the ROS system**. This process is called **overlaying**. There are several "`setup.*sh`" files within the "`devel`" folder of any freshly built workspace. Sourcing any of these files will overlay the workspace on top of the environment so that it can be used by ROS:

```
$ cd ~/ros/workspace/myWorkspace
$ source devel/setup.bash #Source your workspace so that ROS is aware of it
```

To make sure that a workspace is properly overlayed, it is possible to check that the `ROS_PACKAGE_PATH` environment variable includes the workspace directory:

```
$ echo $ROS_PACKAGE_PATH
#if worskpace is not overlayed, should give something like:
#/opt/ros/kinetic/share
#if worskpace is overlayed, should give something like:
#/home/youruser/ros/workspaces/myWorkspace/src:/opt/ros/kinetic/share
```

2.1.4 Setting up a ROS-compatible development environment

2.1.4.1 Getting standard QtCreator to work with ROS

QtCreator is a very powerful IDE, with advanced debugging features. It is free and can be installed from the official ubuntu repository using the following command line:

```
$ sudo apt-get install qtcreator
```

QtCreator can be easily interfaced with a ROS workspace provided that the associated *CMakeLists.txt* file is **no longer a symbolic link**:

```
$ cd ~/ros/worspace/myWorkspace/src
$ mv CMakeLists.txt CMakeLists.txt.old
$ cp CMakeLists.txt.old CMakeLists.txt
```

The following commands open Qtcreator and make it point to the "`src`" folder of a specific workspace:

```
$ cd ~/ros/worspace/myWorkspace/src
$ qtcreator CMakeLists.txt & #Start QtCreator as a background task
```

It is then possible to start developing and compiling applications using the Qtcreator IDE.

2.1.4.2 Qt Creator Plugin for ROS

Depending on the version of QtCreator provided by the official packages repositories of Ubuntu, some undesired behaviors (crashes) may be noticed. This can be avoided by using a special branch of the QtCreator project, specifically designed to be entirely compatible with ROS. The detailed installation procedure is described at the following address: `https://ros-industrial.github.io/ros_qtc_plugin/_source/How-to-Install-Users.html` Using this improved IDE, it is possible to directly import a ROS workspace as a new project.

Figure 2.1: QTcreator ROS plugin

2.2 Mastering the ROS package system

2.2.1 General structure of a ROS package

Packages contain the ROS nodes, libraries, configuration files, and headers, organized together as a single unit. As depicted in Fig. 1.4, a ROS package contains at least two files:

1. **package.xml**: is the file describing the package identity and dependencies. If dependencies are missing or incorrect, **it may still be possible to build from source and run tests on a machine**, but the package will not work correctly when released to the ROS community.
2. **CMakeLists.txt**: is the file indicating how to compile and install the package. **99% of the build errors on a ROS system comes from an ill-configured CMakeLists file.**

These files must be maintained by the developer in charge of the package. Their structure is studied in details in chapter 5. Note that in case a package is faulty and cannot be fixed rapidly, it is still possible to compile the other packages of the workspace. To do so, a file named `CATKIN_IGNORE` must simply be added to the considered package root:

```
$ cd ~/ros/worspace/myWorkspace/src/package_i
$ touch CATKIN_IGNORE
```

When parsing the workspace, the compiler will simply ignore the considered package.

2.2.2 Navigating the ROS Package System

ROS provides a set of powerful tools that allows one to efficiently navigate across a vast collection of packages. The following gives a summary of the most useful commands:

- Find a package: `rospack find [myPackage]`. Returns the address of the package.
- Switch to a package: `roscd [myPackage]`. Essentially prevents from writing the entire path to the package (e.g. `roscd roscpp` is equivalent to `cd /opt/ros/kinetic/share/roscpp`)

A complete tutorial is dedicated to this on the ROS wiki: `http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem`

2.2.3 Creating a ROS package

The `catkin_create_pkg` command is used to create a ROS package automatically, just by setting the name of the package and its dependencies. It has the following syntax: `catkin_create_pkg [package_name] [dependency 1]...[dependency n]`. For instance, creating a new ROS package within the previously created workspace can be achieved using the following commands:

```
$ cd myWorkspace/src  #Switch to the src subfolder of the workspace.
$ catkin_create_pkg package_i std_msgs rospy roscpp  #Create a new package
```

In this example, the created package "`package_i`" has the following dependencies:

- **roscpp**: is a ROS library providing a C++ API to develop ROS nodes, topics, services, parameters, and so on.
- **rospy**: is the equivalent of roscpp for Python.
- **std_msgs**: contains basic ROS primitive data types such as integer, float, string, array, and so on. These data types can be directly used in ROS nodes without defining a new ROS message.

This is **obviously not an exhaustive list**: in practice, it is the programmer's responsibility to maintain the list of packages required to run the application (and therefore to update the CMakeList and package manifest periodically). Note that the package manifest and the CMakelist file are automatically populated by the `catkin_create_pkg` command[1]. However they have to be **updated by hand**, following the conventions presented in chapter 5.

2.2.4 Populating a ROS package

2.2.4.1 How to create a C++ ROS node

Switch to the `src` folder of the considered package and create the node source file "`my_nodeX_i.cpp`":

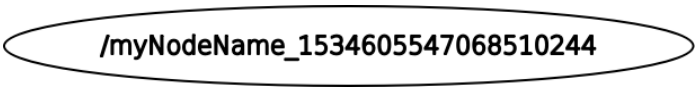
```
$ cd myWorkspace/src/package_i/src   #Switch to the src subfolder.
$ gedit my_nodeX_i.cpp       #Edit the source file which contains the node
```

The basic structure of every C++ ROS node is the same:

```
#include "ros/ros.h"
//...
int main(int argc, char **argv)
{
    ros::init(argc, argv, "myNodeName", ros::init_options::AnonymousName);
    ros::NodeHandle n;
    //...
    ros::Rate loop_rate(10);
    //...
    while (ros::ok())
    {
        //Do some computations...
        ros::spinOnce(); // NEVER forget that.
        loop_rate.sleep();
    }
    return 0;
}
```

- `#include "ros/ros.h"` allows to include all necessary ROS headers in a compact way.
- The `ros::init()` function needs to see `argc` and `argv` so that it can perform any ROS arguments and name **remapping**[2] that were provided at the command line[3]. The third argument to `ros::init()` is the **name of the node in the ROS graph**[4]. The last argument is optional. In this example the field `ros::init_options::AnonymousName` makes the node "anonymous" by adding a random number to the end of its name, in order to make it unique on the ROS computational graph:

(a) Node name on the ROS computation graph



(b) Anonymous node name on the ROS computation graph, using the option `ros::init_options::AnonymousName`

[1] It is of course possible to create a package "by hand" but this is **STRONGLY DISCOURAGED** as it is the source of so many errors and painful debugging hours. This makes students frustrated and supervisors angry.

[2] See `https://wiki.ros.org/Remapping%20Arguments`

[3] See subsection 3.6 for further details on the arguments of a C++ program.

[4] The name of a node in the ROS graph must be unique. The use of special characters must moreover be avoided.

- `ros::NodeHandle` creates a link between ROS and the node. The first instantiated NodeHandle object handles the initialization of the node. The last destroyed deals with cleaning up all the resources that the node has used.
- `ros::Rate` object allows to specify the loop frequency (in Hz). The object monitors the time between two calls to `Rate::sleep()` and makes the program wait for the time needed to complete the loop (**unless of course the return to `Rate::sleep()` is not fast enough**).
- By default roscpp sets a redirection of interrupt signals, such as `SIGINT` (Ctrl-C). If this signal is issued, then `ros::ok()` returns false. More precisely `ros::ok()` returns false if:
  - a `SIGINT` interrupt signal is received (Ctrl-C).
  - another node with the same name has taken the node out of the application graph.
  - `ros::shutdown()` has been called by another part of the application.
  - all `ros::NodeHandle` objects have been destroyed.
- `ros::spinOnce()` will call every callback[5] function defined in the code. Several variants of this function can be used in practice, as explained in `https://wiki.ros.org/roscpp/Overview/Callbacks%20and%20Spinning`:
  - **Single-threaded Spinning**
    - `ros::spin()` will not return until the node has been shutdown, either through a call to `ros::shutdown()` or a `Ctrl-C`:

```
ros::init(argc, argv, "myNodeName");
ros::NodeHandle n;
//...
ros::spin();
```

    - `ros::spinOnce()` has to be called periodically in a while loop:

```
ros::init(argc, argv, "myNodeName");
ros::NodeHandle n;
//...
ros::Rate loop_rate(10);
//...
while (ros::ok())
{
    //Do some computations...
    ros::spinOnce();
    loop_rate.sleep();
}
```

    - Implementing a `ros::spin()` of our own is quite simple:

```
#include <ros/callback_queue.h>
ros::NodeHandle n;
while (ros::ok())
{
  ros::getGlobalCallbackQueue()->callAvailable(ros::WallDuration
    (0.1));
}
```

      and `ros::spinOnce()` is simply:

[5] An example of callback function is provided in subsection 3.2.4.

```
#include <ros/callback_queue.h>
ros::getGlobalCallbackQueue()->callAvailable(ros::WallDuration
    (0));
```

– **Multi-threaded Spinning** `ros::spin()` and `ros::spinOnce()` are really meant for single-threaded nodes applications, and are not optimized for being called from multiple threaded nodes at once. Fortunately `roscpp` provides built-in support for calling callbacks from multiple threaded nodes. There are two built-in options for this:

* `ros::MultiThreadedSpinner` is a blocking spinner, similar to `ros::spin()`. You can specify a number of threads in its constructor, but if unspecified (or set to 0), it will use a thread for each CPU core.

```
ros::MultiThreadedSpinner spinner(4); // Use 4 threads
spinner.spin(); // spin() will not return until the node has
    been shutdown
```

* `ros::AsyncSpinner` is a more useful threaded spinner. Instead of a blocking `spin()` call, it has `start()` and `stop()` calls, and will automatically stop when it is destroyed.

```
ros::AsyncSpinner spinner(4); // Use 4 threads
spinner.start();
ros::waitForShutdown();
```

2.2.4.2 How to link a C++ node to ROS

Compiling a ROS package first requires to make it visible to catkin. To do so, it is necessary to edit the package `CMakelist.txt` file. At the current stage, this file should contain the following lines:

```
cmake_minimum_required(VERSION 2.8.3)
project(package_i)
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
)
catkin_package()
include_directories(${catkin_INCLUDE_DIRS})
```

To make catkin aware of the newly created node "`my_nodeX_i.cpp`", just add the following two lines at the end of the `CMakelist.txt` file:

```
add_executable(testNode src/my_nodeX_i.cpp)
target_link_libraries(testNode ${catkin_LIBRARIES})
```

The detail of the different commands is given in chapter 5. Note that "`testNode`" is the target name, i.e. the name by which the node is referred to when executed from the terminal. This is not the same name as on the ROS computation graph (where the node is referred to as "myNodeName").

2.2.4.3 How to debug C++ source code in ROS using rosconsole

ROS has a complete C++ API, allowing to display messages in the terminal and to record them in a log file for debug purpose (see `https://wiki.ros.org/rosconsole`). The main functions are:

```
ROS_DEBUG_STREAM("Hello world"<<myVariable);
ROS_INFO_STREAM("Hello world"<<myVariable);
ROS_WARN_STREAM("Hello world"<<myVariable);
ROS_ERROR_STREAM("Hello world"<<myVariable);
ROS_FATAL_STREAM("Hello world"<<myVariable);
```

Using these commands, messages are printed on the screen, written in the Node log file, and written in `rosout`. `rosout` is useful for debugging: it is possible to analyze messages using `rqt_console` instead of finding the node display window. For each node, a log file is created in `/.ros/log` and can be accessed using:

```
$ roscd log
```

For debugging purpose, the classic approach is to use the following:

```
ROS_INFO_STREAM(__FILE__<<" "<<__LINE__); // Display the line being executed
```

Note that these functions are NOT real-time safe. They must be used wisely for control loop implementations.

2.2.4.4 How to create a Python ROS node

Switching to the `scripts` folder of the considered package and create the node source file "`my_scriptX_i.py`":

```
$ cd myWorkspace/src/package_i/scripts   #Switch to the script subfolder.
$ gedit my_scriptX_i.py     #Edit the source file which contains the node
```

The basic structure of every Python ROS node is the same:

```
#!/usr/bin/env python
# license removed for brevity
import rospy
# ...
def myFunction():
    rospy.init_node('myNodeNamePython', anonymous=True)
    rate = rospy.Rate(10)
    while not rospy.is_shutdown():
        # Do some computation ...
        # ...
        rate.sleep()

if __name__ == '__main__':
    try:
        myFunction()
    except rospy.ROSInterruptException:
        pass
```

- `#!/usr/bin/env python` makes sure the script is executed as a Python script.
- `import rospy` allows to include all necessary ROS headers in a compact way.
- The `rospy.init_node()` takes as argument the **name of the node in the ROS graph**. Similarly to the C++ example, it can be made anonymous by setting an optional parameter.

- `rospy.Rate` object allows to specify the loop frequency (in Hz). The object monitors the time between two calls to `rate.sleep()` and makes the program wait for the time needed to complete the loop (**unless of course the return to `rate.sleep()` is not fast enough**).
- In addition to the standard Python `__main__` check, the last piece of code catches a `rospy.ROSInterruptException` exception, which can be thrown by `rospy.sleep()` and `rospy.Rate.sleep()` methods when `Ctrl-C` is pressed. This exception is thrown so that the node does not keep executing code after the `sleep()`.

2.2.4.5 How to link a Python node to ROS

The first step is to make the python script executable:

```
chmod a+x scripts/my_scriptX_i.py
```

To mark the executable script "`my_scriptX_i.py`" for installation, just add the following lines at the end of the `CMakelist.txt` file:

```
install(PROGRAMS
  scripts/my_ScriptX_i.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

2.2.4.6 How to debug Python source code in ROS

ROS has a completePython API, allowing to display messages in the terminal for debug purpose (see `https://wiki.ros.org/rospy/Overview/Logging`). The main functions are:

```
rospy.logdebug(msg, *args)
rospy.loginfo(msg, *args)
rospy.logwarn(msg, *args)
rospy.logerr(msg, *args)
rospy.logfatal(msg, *args)
```

Using these commands, messages are printed on the screen, written in the Node log file, and written in `rosout`. `rosout` is useful for debugging: it is possible to analyze messages using `rqt_console` instead of finding the node display window.For each node, a log file is created in `/.ros/log` and can be accessed using:

```
$ roscd log
```

2.3 Executing a ROS node

First make sure that the workspace is correctly overlayed:

```
$ cd ~/ros/workspace/myWorkspace
$ source devel/setup.bash
```

This is mandatory for each opened terminal. Then check if the python scripts are executable (e.g. using `chmod`...).

2.3.1 Simple execution with rosrun

ROS nodes can be executed separately using the `rosrun` command, with the following syntax:

```
$ rosrun <package_name> <executable_name>
```

This is a convienent way to test nodes functionalities without writing a script. For example, the previsouly developed nodes can be executed – provided that the workspace is correctly overlayed and that the `CMakeList.txt` file is correctly filled – by issuinng the following commands:

1. In a first terminal, start the ROS master:

```
$ roscore
```

2. In second terminal, run the first node:

```
$ rosrun package_i testNode
```

3. Run the last node in another terminal:

```
$ rosrun package_i my_scriptX_i.py
```

Note that the executable name (e.g. testNode) is provided in the `CMakelist.txt` file. It has nothing to do with the name of the node on the computational graph, defined within the `init()` function.

2.3.2 Advanced node execution with roslaunch

`roslaunch` is a tool which allows to easily launch multiple ROS nodes locally and/or remotely (via SSH), as well as setting parameters on the ROS Parameter Server (c.f. chapter 3.5). 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, as well as the machines that they should be run on. Many ROS packages come with "launch files", which can be parsed with:

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

These launch files usually bring up a set of nodes for the package that provide some aggregate functionality. It is for example possible to launch several nodes using a single command:

```
<?xml version="1.0"?>
<launch>
  <node name="myCppNode" pkg="package_i" type="testNode"/>
  <node name="myPythonNode" pkg="package_i" type="my_scriptX_i.py"/>
</launch>
```

It is also possible to easily launch several instances of the same node (running in parallel within separated threads) using dedicated namespaces:

```
<?xml version="1.0"?>
<launch>
  <group name="myNodeGroup1">
      <node name="myCppNode" pkg="package_i" type="testNode"/>
  </group>

  <group name="myNodeGroup2">
      <node name="myCppNode" pkg="package_i" type="testNode"/>
  </group>
</launch>
```

In this launch file, two instances of the "`testNode`" executable defined in the CMakeList file of the package (and named `myCppNodeCopy1` and `myCppNodeCopy2` in the ROS computation graph) run in parallel with the python node `myPythonNode`.

IMPORTANT:

- `roslaunch` does not guarantee any particular order to the startup of nodes.
- The detail of launch files syntax is provided in chapter 5.3.

Chapter 3

Mastering the ROS Node Communication Framework

The main components of the ROS communication framework (i.e. **topics**, **services** and **actions**) are introduced in chapter 1.2.1. This chapter goes into more depth and shows how to make the nodes communicate with each other in practice.

3.1 When to use Topics vs Services vs Actions ?

This is indeed the first question to ask yourself when programming a robot application. Here is how things work in practice:

1. **Topics** should be used for continuous data streams (sensor data, robot state, ...).
2. **Services** should be used for remote procedure calls that terminate quickly, e.g. for querying the state of a node or doing a quick calculation such as IK. They should never be used for longer running processes, in particular processes that might be required to preempt if exceptional situations occur and they should never change or depend on state to avoid unwanted side effects for other nodes.
3. **Actions** should be used for everything that moves the robot or that runs for a longer time such as perception routines that are triggered by some node and need a couple of seconds to terminate. The most important property of actions is that they can be preempted and preemption should always be implemented cleanly by action servers. Another nice property of actions is that they can keep state for the lifetime of a goal, i.e. if executing two action goals in parallel on the same server, for each client a separate state instance can be kept since the goal is uniquely identified by its id [1].

3.2 Asynchronous node communication with ROS Topics, Publishers and Subscribers

ROS topics are **asynchronous node communication protocols** used in the ROS network. In practice this means that data production is **decoupled** from its consumption. This is especially useful in robotics when data cannot be obtained "on request", but in stead, comes as a continuous

[1] These explanations were taken from:
`https://answers.ros.org/question/11834/when-should-i-use-topics-vs-services-vs-actionlib-actions-vs-dynamic_reconfigure/`.

stream. A node can either connect with a topic as a "**publisher**" (in order broadcast a data stream), or as a "**subscriber**" (in order to receive data from other topics). **An arbitrary number of publishers and subscribers can be generated with a single topic.**

3.2.1 Defining the Data Format of a Topic (.msg file)

The first step before publishing data in a ROS topic is of course to determine the **type** of the data we want to manipulate. In ROS, this can be specified using dedicated files, with the extension `.msg`, and stored within the `msg` directory of a package (c.f. Fig. 1.4). A `.msg` file is basically a simple text file that describe the fields of a ROS message. Each line contains a field type and field name. The different base types are the following:

- `int8, int16, int32, int64, uint8, uint16, uint32, uint64`
- `float32, float64`
- `bool`
- `string, char`
- `time, duration`
- other `.msg` files
- variable-length `array[]` and fixed-length `array[size]` (useful for pictures)

The first line in a `.msg` is often a `Header`. The header contains a timestamp and coordinate frame information that are commonly used in ROS. The full specification for the message format is available at the Message Description Language page: `https://wiki.ros.org/action/show/msg?action=show&redirect=ROS%2FMessage_Description_Language`.

3.2.1.1 Writing a .msg file and making it visible to ROS

ROS naturally comes with a set of predefined data formats such as `geometry_msgs/WrenchStamped` or `std_msgs/String`. These message types are defined in `/opt/ros/kinetic/share/XXX_msgs/msg/`. The steps for creating a new mesage file and including it within the ROS framework are the following:

1. Create a `.msg` file with the name of the message type we want to define (here `my_message_i`):

```
$ cd myWorkspace/src/package_i/msg   #Switch to the msg subfolder.
$ gedit my_message_i.msg #Edit the file which contains the message
    structure
```

2. Fill the `.msg` filewith the message structure. In the following example, we define a message with two distinct fields: a composite `geometry_msgs/WrenchStamped` (already defined in the ROS system at `/opt/ros/kinetic/share/geometry_msgs/msg/`), and a 32-bit integer:

```
# Message definition
geometry_msgs/WrenchStamped wrench
uint32 score
```

   Each messasge of type "`my_message_i`" published in a ROS topic will contain these two fields.

3. Tell ROS that it must include this message in its conventions:

   - In the `CMakeLists.txt` file, make sure that:
     - `std_msgs` and `message_generation` are specified as arguments of your project (see section 5.1.3 for more details):

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  geometry_msgs
  message_generation
)
```

- It is also necessary to export the dependency to `message_runtime`:

```
catkin_package(
  ...
  CATKIN_DEPENDS message_runtime ...
)
```

- Then it is necessary to indicate which file contains the new message that the system must process:

```
add_message_files(
  FILES
  my_message_i.msg
)
```

- It is then important to add any required packages to the `.msg` files, for example in our case, it is necessary to add `std_msgs` and `geometry_msgs` which gives:

```
generate_messages(
  DEPENDENCIES
  std_msgs
  geometry_msgs
)
```

- Make sure that the manifest `package.xml` has the following lines:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
<depend>std_msgs</depend>
<depend>geometry_msgs</depend>
```

4. Once the new messages have been added, it is mandatory to recompile and install the package:

```
$ roscd package_i
$ cd ../..
$ rm -r build/ devel/
$ catkin_make
$ catkin_make install
$ source /devel/setup.bash # NEVER forget to overlay your workspace
```

5. After successful compilation, any `.msg` file in the msg directory will generate code for use in all supported languages:

- The C++ message header file will be generated in the following folder: `/work_space_folder/devel/include/package_i/`. This message header can then be included in the node source code using: `#include"package_i/my_message.h"`.
- The corresponding Python script will be created in the following folder: `/work_space_folder/devel/lib/python2.7/dist-packages/package_i/msg` and can be included in the node source code using: `from package_i.msg import *`.

3.2.1.2 Debugging ROS messages using rosmsg

It is possible to make sure that ROS can see the new message, using the `rosmsg show [message type]` command. In our case:

```
$ rosmsg show package_i/my_message_i
```

should give the following output:

```
geometry_msgs/WrenchStamped wrench
  std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Wrench wrench
  geometry_msgs/Vector3 force
    float64 x
    float64 y
    float64 z
  geometry_msgs/Vector3 torque
    float64 x
    float64 y
    float64 z
uint32 score
```

3.2.2 Writing a C++ ROS Topic Publisher

3.2.2.1 Commented Code

First create the source file which will contain the code of the ROS publisher and import the corresponding message data types. In this example, we only consider the case of a conventional ROS message of type "`std_msgs/String`" defined in `/opt/ros/kinetic/share/std_msgs/msg/String.msg`:

```
$ cd myWorkspace/src/package_i/src   #Switch to the src subfolder.
$ gedit myPublisher.cpp #Edit the source file which contains the publisher
```

The basic structure of every C++ ROS publisher is the same:

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "package_i/my_message_i.h"

int main(int argc, char **argv)
{
// Initialize a new node with the name "my_publisherNode":
ros::init(argc, argv, "my_publisherNode");
ros::NodeHandle n;

//Creating publisher objects of the node:
ros::Publisher my_firstPublisher = n.advertise<std_msgs::String>("
    firstChatter", 1000);
ros::Publisher my_secondPublisher = n.advertise<package_i::my_message_i>("
    secondChatter", 1000);
ros::Rate loop_rate(10);

int count = 0;
```

```
// Loop on the posting of messages 10 times per second:
while (ros::ok())
{
  // Init messages:
  std_msgs::String my_firstMessage;
  package_i::my_message_i my_secondMessage;
  // Message content:
  my_firstMessage.data = "hello world";
  my_secondMessage.wrench.wrench.force.x = count++;
  my_secondMessage.wrench.wrench.force.y = count++;
  my_secondMessage.wrench.wrench.force.z = count++;
  my_secondMessage.wrench.wrench.torque.x = count++;
  my_secondMessage.wrench.wrench.torque.y = count++;
  my_secondMessage.wrench.wrench.torque.z = count++;
  my_secondMessage.score = count++;
  // Publish messages:
  my_firstPublisher.publish(my_firstMessage);
  my_secondPublisher.publish(my_secondMessage);
  // Update node data and wait:
  ros::spinOnce();
  loop_rate.sleep();
}
return 0;
}
```

- `#include "std_msgs/String.h"` allows to access the C++ declaration of ROS native `std_msgs/String` messages. This header is automatically generated from the file `std_msgs/String.msg` which is located in the `std_msgs` package.
- `#include "package_i/my_message_i.h"` allows to access the C++ declaration of the custom message `my_message_i`. This header is automatically generated from the file `package_i/my_message_i.msg` which is located in the `package_i` package.
- `ros::Publisher` tell the master (instantiated by `rosmaster`) that the node will publish messages of type `std_msgs/String` on the topic "`firstChatter`" and `package_i/my_message_i` on the topic "`secondChatter`". The second argument is the size of the queue in case the subscriber does not process the messages quickly enough. When the queue is full, the oldest data is dropped. The `NodeHandle::advertise()` method returns an object of type `ros::Publisher`. This object has two functions: the first is to provide a `publish()` method to publish data, and the second is to stop the publication of data when the object is no longer in the execution field.
- `my_firstMessage.data = "hello world";` The node sends a message on ROS using a class adapted to messages, generated from a `.msg` file. The "String" type message that has only one member named "data". The wrench message has a more complex structure.
- `my_firstPublisher.publish(my_firstMessage)` requests that the message be sent to all connected nodes.
- The other code lines have already been explained in detail in chapter 2.2.4.1.

3.2.2.2 CMakeLists.txt of a C++ ROS publisher

In addition to the modifications required by the custom message generation (described in 3.2.1.1) it is necessary to add the following lines to the `CMakeList.txt` in order to make ROS aware of the new publisher:

```
add_executable(talker src/myPublisher.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker package_i_generate_messages_cpp)
```

The first two lines are the same as in the example 2.2.4.2. In the third line, note that `package_i_generate_messages_cpp` can be replaced by `package_i_gencpp` with exactly the same result. However the latter will soon be deprecated.

3.2.3 Writing a Python ROS Topic Publisher

3.2.3.1 Commented Code

First create the source file which will contain the code of the ROS publisher and import the corresponding message data types. In this example, we only consider the case of a conventional ROS message of type "`std_msgs/String`" defined in `/opt/ros/kinetic/share/std_msgs/msg/String.msg`:

```
$ cd myWorkspace/src/package_i/scripts   #Switch to the src subfolder.
$ gedit myPublisher.py #Edit the source file which contains the publisher
```

The basic structure of every Python ROS publisher is the same:

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String
from package_i.msg import *

def myPublishFunction():
  my_firstPublisher = rospy.Publisher('firstChatter', String, queue_size
    =1000)
  my_secondPublisher = rospy.Publisher('secondChatter', my_message_i,
    queue_size=1000)
  rospy.init_node('my_publisherNode', anonymous=True)
  rate = rospy.Rate(10)

  while not rospy.is_shutdown():
    my_firstMessage = "hello world"
    my_secondMessage = my_message_i()
    my_secondMessage.wrench.wrench.force.x = rospy.get_time()
    my_secondMessage.wrench.wrench.force.y = rospy.get_time()
    my_secondMessage.wrench.wrench.force.z = rospy.get_time()
    my_secondMessage.wrench.wrench.force.x = rospy.get_time()
    my_secondMessage.wrench.wrench.force.y = rospy.get_time()
    my_secondMessage.wrench.wrench.force.z = rospy.get_time()
    my_secondMessage.score = rospy.get_time()
    my_firstPublisher.publish(my_firstMessage)
    my_secondPublisher.publish(my_secondMessage)
    rate.sleep()

if __name__ == '__main__':
  try:
    myPublishFunction()
  except rospy.ROSInterruptException:
    pass
```

- `from std_msgs.msg import String` allows to access the Python declaration of `std_msgs/String` messages.
- `from package_i.msg import *` allows to access the Python declaration of every custom message in `package_i/msg`.
- `rospy.Publisher('firstChatter', String, queue_size=1000)` tell the master (instantiated by `rosmaster`) that the node will publish messages of type `std_msgs/String` (resp. `package_i/my_message_i`) on the topic `firstChatter` (resp. `secondChatter`). The third argumentation is the size of the queue in case the subscriber does not process the messages quickly enough. When the queue is full, the oldest data is dropped.
- `my_firstPublisher.publish(my_firstMessage)` requests that the message be sent to all connected nodes.
- The other code lines have already been explained in detail in chapter 2.2.4.4.

3.2.3.2 CMakeLists.txt of a Python ROS publisher

In addition to the modifications required by the custom message generation (described in 3.2.1.1) it is necessary to add the following lines to the `CMakeList.txt` in order to make ROS aware of the new publisher:

```
install(PROGRAMS
  scripts/myPublisher.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

3.2.4 Writing a C++ ROS Topic Subscriber

3.2.4.1 Commented Code

First create the source file which will contain the code of the ROS subscriber and import the corresponding message data types:

```
$ cd myWorkspace/src/package_i/src   #Switch to the src subfolder.
$ gedit mySubscriber.cpp #Edit the source file which contains the subscriber
```

The basic structure of every C++ ROS subscriber is the same:

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "package_i/my_message_i.h"

// Callback function executed each time a new message is received:
void my_firstCallback(const std_msgs::String::ConstPtr& my_message)
{
  ROS_INFO_STREAM("I heard: "<< my_message->data.c_str());
}

void my_secondCallback(const package_i::my_message_i::ConstPtr& my_message)
{
  ROS_INFO_STREAM("I feel the force: "<< my_message->wrench.wrench.force.x);
}

int main(int argc, char **argv)
{
```

```
// Initialize a new node with the name "my_subscriberNode":
ros::init(argc, argv, "my_subscriberNode");
ros::NodeHandle n;

//Subscriber to the ``firstChatter'' and ``secondChatter'' topics:
ros::Subscriber my_firstSubscriber = n.subscribe("firstChatter", 1000,
    my_firstCallback);
ros::Subscriber my_secondSubscriber = n.subscribe("secondChatter", 1000,
    my_secondCallback);

ros::spin();
return 0;
}
```

- `#include "std_msgs/String.h"` allows to access the C++ declaration of ROS native `std_msgs/String` messages. This header is automatically generated from the file `std_msgs/String.msg` which is located in the `std_msgs` package.
- `#include "package_i/my_message_i.h"` allows to access the C++ declaration of the custom message `my_message_i`. This header is automatically generated from the file `package_i/my_message_i.msg` which is located in the `package_i` package.
- `void my_firstCallback(const std_msgs::String::ConstPtr& my_message)` the callback functions will be called when a new message arrives on the topic `firstChatter` (resp. `secondChatter`). Since the message went through in a "`boost::shared_ptr`" it is possible to store it without having to worry about deleting it.
- `ros::Subscriber` subscribes to the topic `firstChatter` (resp. `secondChatter`) from the master. ROS calls the `my_firstCallback()` (resp. `my_secondCallback()`) function at each time a new message arrives. The second argument is the size of the queue. If the messages are not processed fast enough, the system stores up to 1000 messages and then starts to forget the oldest messages. `NodeHandle::subscribe()` returns a `ros::Subscriber` object, that must be maintained during all the time of the subscription. When the object `ros::Subscriber` is destroyed the node is automatically unsubscribed from the topic `firstChatter` (resp. `secondChatter`). There are versions of `NodeHandle::subscribe()` that allow to specify a **method of a class**, or even any **object** that can be called by a `Boost.Function` object.
- `ros::spin()` will not return until the node has been shutdown, either through a call to `ros::shutdown()` or a `Ctrl-C`.
- The other code lines have already been explained in detail in chapter 2.2.4.1.

3.2.4.2 CMakeLists.txt of a C++ ROS subscriber

As previously it is necessary to add the following lines to the `CMakeList.txt` to make ROS aware of the new subscriber:

```
add_executable(listener src/mySubscriber.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_dependencies(listener package_i_generate_messages_cpp)
```

The first two lines are the same as in the example 2.2.4.2. In the third line, note that `package_i_generate_messages_cpp` can be replaced by `package_i_gencpp` with exactly the same result. However the latter will soon be deprecated.

3.2.5 Writing a Python ROS Topic Subscriber

3.2.5.1 Commented Code

First create the source file which will contain the code of the ROS subscriber and import the corresponding message data types:

```
$ cd myWorkspace/src/package_i/scripts   #Switch to the src subfolder.
$ gedit mySubscriber.py #Edit the source file which contains the subscriber
```

The basic structure of every Python ROS subscriber is the same:

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String
from package_i.msg import *

def my_firstCallback(data):
  rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def my_secondCallback(data):
  rospy.loginfo(rospy.get_caller_id() + "I feel the force %d", data.wrench.
    wrench.force.x)

def mySubscribeFunction():
  rospy.init_node('my_subscriberNode', anonymous=True)
  rospy.Subscriber("firstChatter", String, my_firstCallback)
  rospy.Subscriber("secondChatter", my_message_i, my_secondCallback)
  rospy.spin()

if __name__ == '__main__':
  try:
    mySubscribeFunction()
  except rospy.ROSInterruptException:
    pass
```

- `from std_msgs.msg import String` allows to access the Python declaration of `std_msgs/String` messages.
- `from package_i.msg import *` allows to access the Python declaration of every custom message in `package_i/msg`.
- `my_firstCallback(data)` the callback function will be called when a new message arrives on the topic `firstChatter` (resp. `secondChatter`).
- `rospy.Subscriber("firstChatter", String, my_firstCallback)` subscribes to the topic "`firstChatter`" from the master. ROS calls the `my_firstCallback()` function at each time a new message arrives.
- `rospy.spin()` simply keeps python from exiting until this node is stopped. Unlike `roscpp ros::spin()`, `rospy.spin()` does not affect the subscriber callback functions, as those have their own threads.
- The other code lines have already been explained in detail in chapter 2.2.4.4.

3.2.5.2 CMakeLists.txt of a Python ROS subscriber

In addition to the modifications required by the custom message generation (described in 3.2.1.1) it is necessary to add the following lines to the `CMakeList.txt` in order to make ROS aware of the new publisher:

```
install(PROGRAMS
  scripts/mySubscriber.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

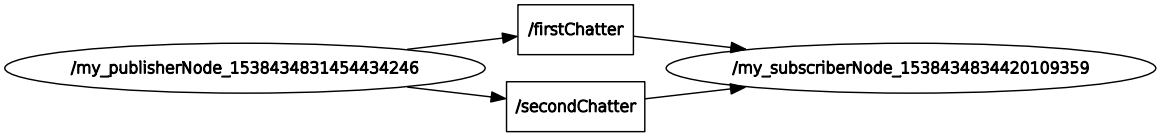


Figure 3.1: The node `my_publisherNode` publishes two topics, namely "`firstChatter`" and "`secondChatter`", with messages of different types. The node `my_subscriberNode` listen to these two topics. This graph was obtained by simply executing `rosrun rqt_graph rqt_graph`.

3.2.6 Debugging topics in command line using rostopic

- `rostopic list`: Displays the list of topics on the local ROS network.
- `rostopic pub`: Publishes data on a topic. For example: `rostopic pub -r 10 /topic_name std_msgs/String hello` publishes "hello" at a frequency of 10 Hz in the topic `/topic_name`.
- `rostopic echo /topic_name`: Displays the messages of the topic `/topic_name`.
- `rostopic info /topic_name`: print information about active topic
- `rostopic bw /topic_name`: Displays the bandwidth of a topic
- `rostopic hz /topic_name`: Displays the update frequency of a topic.
- `rostopic type /topic_name`: Displays the type of a topic (message).
- `rostopic find msg-type`: Displays all the topics of a given type.
- Use auto-complete !

3.3 Synchronous communication with ROS Services, Servers and Clients

Services are the **synchronous alternative to topics**, taking the form of a remote procedure call ("**request/answer**") instead of a "**publish/subscribe**" model. By using a service, a given node will only receive data on request.

3.3.1 Defining the Data Format of a Service (.srv file)

As for ROS topics, the first step before publishing data in a ROS service is to determine the **type** of the data we want to manipulate. In ROS, this can be specified using dedicated files, with the extension `.srv`, and stored within the `srv` directory of a package (c.f. Fig. 1.4). A `.srv` file is basically a simple text file that describe the fields of a ROS service. As previously each line contains a field type and field name. However, unlike `.msg` files, a `.srv` file contains two parts: one for the service **request** and one for the service **response**.

3.3.1.1 Writing a .srv file and making it visible to ROS

The steps for creating a new service file and including it within the ROS framework are the following:

1. Create a `.srv` file with the name of the message type we want to define (here `my_service_i`):

```
$ cd myWorkspace/src/package_i/srv   #Switch to the srv subfolder.
$ gedit my_service_i.srv #Edit the file which contains the service
    structure
```

2. Fill the `.srv` file with the service structure. A service file has two distinct sections, namely the *request* and the *response*, separated by "`---`". In the following example, we define a service with two fields in the request (a `uint8`, and a `float64`) and one single field in the response (a 3×3 matrix of `float64` defined using a `std_msgs/Float64MultiArray`):

```
# Request definition
int8 matrixSize
float64 fillMatrixRequest
---
# Response definition
std_msgs/Float64MultiArray responseMatrix
```

3. Tell ROS that it must include this new service structure in its conventions:

   - In the `CMakeLists.txt` file, make sure that:
     - `std_msgs` and `message_generation` are specified as arguments of your project (see section 5.1.3 for more details):

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
```

     - It is also necessary to export the dependency to `message_runtime`:

```
catkin_package(
  ...
  CATKIN_DEPENDS message_runtime ...
)
```

- Then it is necessary to indicate which file contains the new message that the system must process:

```
add_service_files(
  FILES
  my_service_i.srv
)
```

- It is then important to add any required packages to the `.msg` files, for example in our case, it is necessary to add `std_msgs` and `geometry_msgs` which gives:

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

- Make sure that the manifest `package.xml` has the following lines:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
<depend>std_msgs</depend>
```

4. Once the new messages have been added, it is mandatory to recompile and install the package:

```
$ roscd package_i
$ cd ../..
$ rm -r build/ devel/
$ catkin_make
$ catkin_make install
$ source /devel/setup.bash # NEVER forget to overlay your workspace
```

5. After successful compilation, any `.srv` file in the srv directory will generate code for use in all supported languages:

- The C++ service header file will be generated in the following folder: `/work_space_folder/devel/include/package_i/`. This service header can then be included in the node source code using: `#include"package_i/my_service.h"`.
- The corresponding Python script will be created in the following folder: `/work_space_folder/devel/lib/python2.7/dist-packages/package_i/srv` and can be included in the node source code using: `from package_i.srv import *`.

3.3.1.2 Debugging ROS services using rossrv

It is possible to make sure that ROS can see the new message, using the `rossrv show [service type]` command. In our case:

```
$ rossrv show package_i/my_service_i
```

should give the following output:

```
int8 matrixSize
float64 fillMatrixRequest
---
std_msgs/Float64MultiArray responseMatix
std_msgs/MultiArrayLayout layout
std_msgs/MultiArrayDimension[] dim
string label
uint32 size
uint32 stride
uint32 data_offset
float64[] data
```

3.3.2 Writing a C++ ROS Service Server

3.3.2.1 Commented Code

First create the source file which will contain the code of the ROS service server and import the corresponding message data types:

```
$ cd myWorkspace/src/package_i/src   #Switch to the src subfolder.
$ gedit myServiceServer.cpp #Edit the source file which contains the service
    server
```

The basic structure of every C++ ROS service server is the same:

```
#include "ros/ros.h"
#include "package_i/my_service_i.h"

bool my_Callback(package_i::my_service_i::Request  &req, package_i::
    my_service_i::Response &res)
{
  unsigned int s = req.matrixSize*req.matrixSize;
  for(unsigned int i = 0; i<s ;i++)
  {
    res.responseMatix.data.push_back(req.fillMatrixRequest);
  }
  return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "my_serverNode");
  ros::NodeHandle n;
  ros::ServiceServer my_service = n.advertiseService("my_serviceServer",
    my_Callback);
  ros::spin();
  return 0;
}
```

- `#include "package_i/my_service_i.h"` allows to access the C++ declaration of the custom service `my_service_i`. This header is automatically generated from the file `package_i/my_service_i.srv` which is located in the `package_i` package.
- `bool my_Callback()` provides the service for filling a matrix object of size `matrixSize`×`matrixSize` with the value of `fillMatrixRequest`. It takes in the request and response type defined in

the `my_service_i.srv` file and returns a boolean, set to true when the function is successfully executed.

- `ros::ServiceServer` creates and advertise the server over ROS with the name `my_serviceServer`.
- `ros::spin()` will not return until the node has been shutdown, either through a call to `ros::shutdown()` or a `Ctrl-C`.
- The other code lines have already been explained in detail in chapter 2.2.4.1.

3.3.2.2 CMakeLists.txt of a C++ ROS service server

As previously it is necessary to add the following lines to the `CMakeList.txt` to make ROS aware of the new server:

```
add_executable(serviceServer src/myServiceServer.cpp)
target_link_libraries(serviceServer ${catkin_LIBRARIES})
add_dependencies(serviceServer package_i_generate_messages_cpp)
```

The first two lines are the same as in the example 2.2.4.2. In the third line, note that `package_i_generate_messages_cpp` can be replaced by `package_i_gencpp` with exactly the same result. However the latter will soon be deprecated.

3.3.3 Writing a Python ROS Service Server

3.3.3.1 Commented Code

First create the source file which will contain the code of the ROS service server and import the corresponding message data types:

```
$ cd myWorkspace/src/package_i/scripts   #Switch to the src subfolder.
$ gedit myServiceServer.py #Edit the source file which contains the service
    server
```

The basic structure of every Python ROS service server is the same:

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from package_i.srv import *
from std_msgs.msg import Float64MultiArray

def my_Callback(req):
  s = req.matrixSize*req.matrixSize;
  X = Float64MultiArray()
  X.data = [0]*s
  for i in range(0, s):
    # Quick and dirty way of handling Float64MultiArray
    X.data[i] = req.fillMatrixRequest
  return my_service_iResponse(X)

def myServerFunction():
  rospy.init_node('my_serverNode', anonymous=True)
  rospy.Service("my_serviceServer", my_service_i, my_Callback)
  rospy.spin()

if __name__ == '__main__':
  try:
```

```
    myServerFunction()
24  except rospy.ROSInterruptException:
    pass
```

- `from package_i.srv import *` allows to access the Python declaration of every custom service in `package_i/srv`.
- `my_Callback(req)` the callback function will be called when a new request arrives.
- `rospy.Service()` declares a new server named `my_serviceServer` with the `my_service_i` service type. All requests are passed to `my_Callback` function. `my_Callback` is called with instances of `my_service_iRequest` and returns instances of `my_service_iResponse`. These objects are created during the `catkin_make install` step and stored in the folder `/work_space_folder/devel/lib/python2.7/dist-packages/package_i/srv`.
- `rospy.spin()` simply keeps python from exiting until this node is stopped. Unlike `roscpp ros::spin()`, `rospy.spin()` does not affect the subscriber callback functions, as those have their own threads.
- The other code lines have already been explained in detail in chapter 2.2.4.4.

Important note: If you have two lines in your response section of the service definition, you will have two fields in the response message. For instance, if the `.srv` file is:

```
1 # Request definition
  int8 someRequestStuff
3 ---
  # Response definition
5 int8 firstThingToReturn
  int8 secondThingToReturn
```

Then the Python ROS service server Callback should include the following:

```
  def my_Callback(req):
2   ...
    resp = servicenameResponse()
4   resp.firstThingToReturn = 1
    resp.secondThingToReturn = 2
6   return resp
```

3.3.3.2 CMakeLists.txt of a Python ROS service server

In addition to the modifications required by the custom message generation (described in 3.2.1.1) it is necessary to add the following lines to the `CMakeList.txt` in order to make ROS aware of the new publisher:

```
install(PROGRAMS
2 scripts/myServiceClient.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
4 )
```

3.3.4 Writing a C++ ROS Service Client

3.3.4.1 Commented Code

First create the source file which will contain the code of the ROS service client and import the corresponding message data types:

```
$ cd myWorkspace/src/package_i/src   #Switch to the src subfolder.
$ gedit myServiceClient.cpp #Edit the source file which contains the service
    client
```

The basic structure of every C++ ROS service client is the same:

```
#include "ros/ros.h"
#include "package_i/my_service_i.h"
#include <cstdlib>

int main(int argc, char **argv)
{
ros::init(argc, argv, "my_clientNode");
if (argc != 3)
{
  ROS_INFO_STREAM("Usage: my_clientNode size number");
  return 1;
}
ros::NodeHandle n;
ros::ServiceClient client = n.serviceClient<package_i::my_service_i>("
    my_serviceClient");

package_i::my_service_i srv;
srv.request.matrixSize = atoll(argv[1]);
srv.request.fillMatrixRequest = atoll(argv[2]);
if (client.call(srv))
{
  unsigned int size = srv.request.matrixSize;
  double k = 0.0;
  for(unsigned int i =0; i<size;i++)
  {
    for(unsigned int j =0; j<size;j++)
    {
      k = srv.response.responseMatix.data.at(i*size+j);
      ROS_INFO_STREAM("Matrix("<<i<<","<<j<<") = "<< k);
    }
  }
}
else
{
  ROS_ERROR_STREAM("Failed to call service my_clientNode");
  return 1;
}
return 0;
}
```

- `#include "package_i/my_service_i.h"` allows to access the C++ declaration of the custom service `my_service_i`. This header is automatically generated from the file `package_i/my_service_i.srv` which is located in the `package_i` package.
- `ros::ServiceClient` creates the client for the `my_serviceServer` server.
- `package_i::my_service_i srv` Here we instantiate an autogenerated service class, and assign values into its request member. A service class contains two members, `request` and `response`. It also contains two class definitions, `Request` and `Response`.

- `if (client.call(srv))` This actually calls the service. Since service calls are blocking, it will return once the call is done. If the service call succeeded, `call()` will return `true` and the value in `srv.response` will be valid. If the call did not succeed, `call()` will return `false` and the value in `srv.response` will be invalid.
- The other code lines have already been explained in detail in chapter 2.2.4.1.

3.3.4.2 CMakeLists.txt of a C++ ROS service client

As previously it is necessary to add the following lines to the `CMakeList.txt` to make ROS aware of the new client:

```
add_executable(serviceClient src/myServiceClient.cpp)
target_link_libraries(serviceClient ${catkin_LIBRARIES})
add_dependencies(serviceClient package_i_generate_messages_cpp)
```

The first two lines are the same as in the example 2.2.4.2. In the third line, note that `package_i_generate_messages_cpp` can be replaced by `package_i_gencpp` with exactly the same result. However the latter will soon be deprecated.

3.3.5 Writing a Python ROS Service Client

3.3.5.1 Commented Code

First create the source file which will contain the code of the ROS service client and import the corresponding message data types:

```
$ cd myWorkspace/src/package_i/scripts   #Switch to the src subfolder.
$ gedit myServiceClient.py #Edit the source file which contains the service
    client
```

The basic structure of every Python ROS service client is the same:

```
#!/usr/bin/env python
# license removed for brevity
import sys
import rospy
from package_i.srv import *

def myClientFunction(matrixSize, fillMatrixRequest):
  rospy.wait_for_service('my_serviceServer')
  try:
    my_serviceServer = rospy.ServiceProxy('my_serviceServer', my_service_i)
    return my_serviceServer(matrixSize, fillMatrixRequest)
  except rospy.ServiceException, e:
    print "Service call failed: %s"%e

def usage():
  return "%my_clientNode size number"%sys.argv[0]

if __name__ == "__main__":
  if len(sys.argv) == 3:
    matrixSize = int(sys.argv[1])
    fillMatrixRequest = float(sys.argv[2])
  else:
```

```
    print usage()
24    sys.exit(1)
  responseMatrix = myClientFunction(matrixSize, fillMatrixRequest)
26 print "Matrix=%s"%(responseMatrix)
```

- `from package_i.srv import *` allows to access the Python declaration of every custom service in `package_i/srv`.
- For clients you don't have to call `init_node()`
- `rospy.wait_for_service('my_serviceServer')` This is a convenience method that blocks until the service named `my_serviceServer` is available.
- `rospy.ServiceProxy('my_serviceServer', my_service_i)` Handle for calling the service. A proxy is an object that will behave as the class it represents. Because we've declared the type of the service to be `my_service_i`, it does the work of generating the `my_service_iRequest` object for you. The return value is an `my_service_iResponse` object. If the call fails, a `rospy.ServiceException` may be thrown, so you should setup the appropriate `try/except` block.
- The other code lines have already been explained in detail in chapter 2.2.4.4.

3.3.5.2 CMakeLists.txt of a Python ROS service client

In addition to the modifications required by the custom message generation (described in 3.2.1.1) it is necessary to add the following lines to the `CMakeList.txt` in order to make ROS aware of the new publisher:

```
install(PROGRAMS
2 scripts/myServiceClient.py
DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
4 )
```

3.3.6 Debugging services in command line using rosservice

- `rosservice list`: Displays the list of active services on the local ROS network.
- `rosservice call /service_name arg1 ... argN`: Call the service with the provided args. For example: `rosservice call /mu_serviceServer 3 3.141592654` will (using the last example) return a 3×3 matrix structure whose elements are equal to π.
- `rosservice args /service_name`: Displays the service arguments.
- `rosservice info /service_name`: print information about service
- `rosservice uri /service_name`: Print service ROSRPC uri.
- `rosservice type /service_name`: Displays the type of a topic (message).
- `rosservice find msg-type`: Displays all the services of a given type.
- Use auto-complete !

3.4 Preemptable services using the ROS actionlib

If a service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing. The `actionlib` package provides tools to create servers that execute long-running goals that can be *preempted*. It also provides a client interface in order to send requests to the server.

3.4.1 Defining the Data Format of an Action (.action file)

As for ROS topics and services, the first step before using a ROS action service is to determine the **type** of the data we want to manipulate. In ROS, this can be specified using dedicated files, with the extension `.action`, and stored within the `action` directory of a package (c.f. Fig. 1.4). A `.action` file is similar to a `.srv` file, except that it has not two but three fields, namely for the **goal**, the **result**, and the **feedback**.

3.4.1.1 Writing a .action file and making it visible to ROS

The steps for creating a new mesage file and including it within the ROS framework are the following:

1. Create a `.msg` file with the name of the message type we want to define (here `my_message_i`):

```
$ cd myWorkspace/src/package_i/action   #Switch to the action subfolder.
$ gedit my_action_i.action #Edit the action message structure
```

2. Fill the `.action` file with the action service structure. An action file has three distinct sections separated by "`---`", namely the **goal**, the **result** and the **feedback**. The following example shows the `Blink.action` file of the ROS-NAO interface package that allows to control the LEDs of the NAO robot:

```
# Goal: colours to use for blinking, plus blinking rate mean and sd
std_msgs/ColorRGBA[] colors
std_msgs/ColorRGBA bg_color
duration blink_duration
float32 blink_rate_mean
float32 blink_rate_sd
---
# Result: "true" if robot is still blinking
bool still_blinking
---
# Feedback: last blinked colour
std_msgs/ColorRGBA last_color
```

3. Tell ROS that it must include this message in its conventions:

   - In the `CMakeLists.txt` file, make sure that:
     - `std_msgs` and `actionlib_msgs` are specified as arguments of your project (see section 5.1.3 for more details). Note that here `message_generation` does not need to be listed explicitly since it is already implicitly referenced by `actionlib_msgs`:

```
find_package(catkin REQUIRED COMPONENTS
roscpp
rospy
std_msgs
actionlib_msgs
)
```

- It is also necessary to export the dependency to actionlib_msgs . Note that here the dependency on message_runtime is happening automatically:

```
catkin_package(
...
CATKIN_DEPENDS actionlib_msgs ...
)
```

- Then it is necessary to indicate which file contains the new message that the system must process:

```
add_action_files(
FILES
my_action_i.action
Blink.action
)
```

- Call the generate_messages macro, not forgetting the dependencies on std_msgs and actionlib_msgs :

```
generate_messages(
DEPENDENCIES
actionlib_msgs
std_msgs
)
```

- Make sure that the manifest package.xml has the following lines:

```
<build_depend>actionlib</build_depend>
<build_depend>actionlib_msgs</build_depend>
<exec_depend>actionlib</exec_depend>
<exec_depend>actionlib_msgs</exec_depend>
<depend>std_msgs</depend>
```

Alternatively format 2 of *package.xml* onward, you can use depend tag:

```
<depend>actionlib</depend>
<depend>actionlib_msgs</depend>
<depend>std_msgs</depend>
```

4. Once the new messages have been added, it is mandatory to recompile the package:

```
$ roscd package_i
$ cd ../..
$ rm -r build/ devel/
$ catkin_make
$ source /devel/setup.bash # NEVER forget to overlay your workspace
```

5. After successful compilation, any .action file in the action directory will generate code for use in all supported languages:

- The C++ action service header file will be generated in the following folder: /work_space_folder/devel/include/package_i/. This service header can then be included in the node source code using: #include"package_i/my_action.h".
- The corresponding Python script will be created in the following folder: /work_space_folder/devel/lib/python2.7/dist-packages/package_i/action and can be included in the node source code using: from package_i.action import *.

3.4.2 Writing a C++ ROS action server

3.4.2.1 Commented Code

First create the source file which will contain the code of the ROS action server and import the corresponding message data types:

```
$ cd myWorkspace/src/package_i/src   #Switch to the src subfolder.
$ gedit myActionServer.cpp #Edit the source file which contains the action
    server
```

The basic structure of every C++ ROS action server is the same [2]:

```
#include "ros/ros.h"
#include "actionlib/server/simple_action_server.h"
#include "package_i/BlinkAction.h"

typedef actionlib::SimpleActionServer<package_i::BlinkAction> actionServer;

void my_Callback(const package_i::BlinkGoalConstPtr& goal, actionServer*
    server)
{
  ros::Rate actionRate(10);
  bool success = true;
  package_i::BlinkFeedback feedback;
  package_i::BlinkResult result;

  // Start executing the action
  for(int i=1; i<=10000*goal->blink_duration.sec; i++)
  {
    // check if preempt has been requested by the client
    if (server->isPreemptRequested() || !ros::ok())
    {
      ROS_INFO_STREAM("Action Preempted !");
      success = false;
      break;
    }
    feedback.last_color.r = rand()/RAND_MAX;
    feedback.last_color.g = rand()/RAND_MAX;
    feedback.last_color.b = rand()/RAND_MAX;
    feedback.last_color.a = rand()/RAND_MAX;
    server->publishFeedback(feedback);
    actionRate.sleep();
  }

  if(success)
  {
    result.still_blinking = false;
    ROS_INFO_STREAM("Action Succeeded !");
  }
  else
  {
    result.still_blinking = true;
  }
```

[2] Take a look at `https://wiki.ros.org/actionlib_tutorials/Tutorials/SimpleActionServer(ExecuteCallbackMethod)` for an object oriented implementation.

```
  server->setSucceeded(result);
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "my_serverNode");
  ros::NodeHandle n;
  actionServer server(n, "my_actionServer", boost::bind(&my_Callback, _1, &
    server), false);
  server.start();
  ros::spin();
  return 0;
}
```

- `#include "actionlib/server/simple_action_server.h"` allows to access the C++ declaration of the action library used from implementing simple actions.
- `#include "package_i/BlinkAction.h"` allows to access the C++ declaration of the custom action `BlinkAction`. Alternatively you may use `#include <naoqi_bridge_msgs/BlinkAction.h>` since `naoqi_bridge_msgs` is an official ROS package which already contains the definition of `BlinkAction`.
- `void my_Callback()` provides the action service.
- The other code lines have already been explained in detail in chapter 2.2.4.1.

3.4.2.2 CMakeLists.txt of a C++ ROS action server

As previously it is necessary to add the following lines to the `CMakeList.txt` to make ROS aware of the new server:

```
add_executable(actionServer src/myActionServer.cpp)
target_link_libraries(actionServer ${catkin_LIBRARIES} ${Boost_LIBRARIES})
add_dependencies(actionServer package_i_generate_messages_cpp)
```

The first two lines are the same as in the example 2.2.4.2. In the third line, note that `package_i_generate_messages_cpp` can be replaced by `package_i_gencpp` with exactly the same result. However the latter will soon be deprecated.

3.4.3 Writing a Python ROS action server

TODO

3.4.4 Writing a C++ ROS action client

3.4.4.1 Commented Code

First create the source file which will contain the code of the ROS action client and import the corresponding message data types:

```
$ cd myWorkspace/src/package_i/src   #Switch to the src subfolder.
$ gedit myActionClient.cpp #Edit the source file which contains the action
    client
```

The basic structure of every C++ ROS action client is the same:

```cpp
#include "ros/ros.h"
#include "actionlib/client/simple_action_client.h"
#include "package_i/BlinkAction.h"

typedef actionlib::SimpleActionClient<package_i::BlinkAction> actionClient;

// Called once when the goal completes
void doneCb(const actionlib::SimpleClientGoalState& state, const package_i::
    BlinkResultConstPtr& result)
{
  ROS_INFO_STREAM("Finished in state: "<< state.toString().c_str());
  ROS_INFO_STREAM("Answer: "<< result->still_blinking);
  ros::shutdown();
}

// Called once when the goal becomes active
void activeCb()
{
  ROS_INFO("Goal just went active");
}


// Called every time feedback is received for the goal
void feedbackCb(const package_i::BlinkFeedbackConstPtr& feedback)
{
  ROS_INFO_STREAM("Got the following Feedback: "<< feedback->last_color);
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "my_clientNode");
  ros::NodeHandle n;
  // create the action client
  // true causes the client to spin its own thread
  actionClient client("my_actionServer", true);
  client.waitForServer();
  package_i::BlinkGoal goal;
  goal.blink_duration.sec = 50;
  client.sendGoal(goal, &doneCb, &activeCb, &feedbackCb);
  ros::spin();
  return 0;
}
```

- `#include "actionlib/server/simple_action_client.h"` allows to access the C++ declaration of the action library used from implementing simple actions.
- `#include "package_i/BlinkAction.h"` allows to access the C++ declaration of the custom action `BlinkAction`. Alternatively you may use `#include <naoqi_bridge_msgs/BlinkAction.h>` since `naoqi_bridge_msgs` is an official ROS package which already contains the definition of `BlinkAction`.
- `void my_Callback()` provides the action client.
- The other code lines have already been explained in detail in chapter 2.2.4.1.

3.4.4.2 CMakeLists.txt of a C++ ROS action client

As previously it is necessary to add the following lines to the `CMakeList.txt` to make ROS aware of the new server:

```
add_executable(actionClient src/myActionClient.cpp)
target_link_libraries(actionClient ${catkin_LIBRARIES} ${Boost_LIBRARIES})
add_dependencies(actionClient package_i_generate_messages_cpp)
```

The first two lines are the same as in the example 2.2.4.2. In the third line, note that `package_i_generate_messages_cpp` can be replaced by `package_i_gencpp` with exactly the same result. However the latter will soon be deprecated.

3.4.5 Writing a Python ROS action client

TODO

3.5 Dominating the ROS Parameter Server

3.5.1 The ROS parameter server

3.5.1.1 What is a parameter server

According to the official definition in `https://wiki.ros.org/Parameter%20Server`: *a parameter server is a shared, multi-variate 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 is implemented using XMLRPC and runs inside of the ROS Master, which means that its API is accessible via normal XMLRPC libraries.

3.5.1.2 Why and how to use the parameter server

First of all, a good question to ask yourself "*why the hell do we need a parameter server on top of all the ROS framework ???*". This is a perfectly legitimate question. To answer this question, suppose that you are coding a controller node (e.g. a PID). Tuning this controller requires that you adjust a set of gains in a trial and error process. If these gains are hard-coded, for example as constant class variables, initialized at each instantiation of your class, you need to recompile your whole source code each time you want to modify them. This is not really practical, especially on a complex real-life robot application, since build time easily reaches several minutes. The solution to this problem lies in the parameter server. Storing the control gains in a "`.yaml`" parameter file allows you to avoid this "recompilation" step. What basically happens is that the value of the parameters stored in the "`.yaml`" file is loaded in your program each time you start it. You can also use a ROS service in order to reload the content of the `.yaml` at any time, without interrupting your program ! Of course you have to give ROS the path of your "`.yaml`" file. This is done in the "`.launch`" file using the `rosparam` tag:

```
<?xml version="1.0"?>
<launch>
  <group name="myNodeGroup1">
    <node name="myCppNode" pkg="package_i" type="testNode"/>
      <rosparam command="load" file="$(find my_package_i)/launch/configs/
    my_yaml_file_1.yaml"/>
    </node>
  </group>

  <group name="myNodeGroup2">
    <node name="myCppNode" pkg="package_i" type="testNode"/>
      <rosparam command="load" file="$(find my_package_i)/launch/configs/
    my_yaml_file_2.yaml"/>
    </node>
  </group>
</launch>
```

In this example, we start two instances of the same node (as explained in 2.3.2) but this time with two diffferent parameter files, namely `my_package_i/launch/configs/my_yaml_file_1.yaml` and `my_package_i/launch/configs/my_yaml_file_1.yaml`.

3.5.1.3 Parameter Types

The ROS Parameter Server can store strings, integers, floats, booleans, lists, iso8601 dates, and base64-encoded data. You can also store dictionaries (i.e. structs) on the Parameter Server, though

they have special meaning. The Parameter Server represents ROS namespaces as dictionaries. For example, imagine you set the following three parameters:

```
/gains/P = 10.0
/gains/I = 1.0
/gains/D = 0.1
```

You can either read them back separately, i.e. retrieving `/gains/P` would return 10.0, or you can retrieving `/gains`, which would return a dictionary: { 'P': 10.0, 'I': 1.0, 'D' : 0.1 }

3.5.1.4 Private Parameters

The ROS naming convention refers to ~`name` as a private name. These private names primarily are for parameters specific to a single Node. The ~ prefix prepends the Node's name to use it as a semi-private namespace – they are still accessible from other parts of the system, but they are generally protected from accidental name collisions. You can use remapping arguments to specify node parameters on the command line by changing the tilde ~ to an underscore _, e.g.:

3.5.2 How to play with parameters in C++

3.5.2.1 Getting parameters

`false` is returned if the parameter does not exist, or is not of the right type. There is also a version that returns a default value:

- `ros::NodeHandle::getParam()`: Parameters retrieved through the `NodeHandle` version are resolved relative to that NodeHandle's namespace. Consider the following code:

```
ros::NodeHandle n;
std::string global_name, relative_name, default_param;

if (n.getParam("/global_name", global_name))
{
  ...
}

if (n.getParam("relative_name", relative_name))
{
  ...
}

// Default value version
n.param<std::string>("default_param", default_param, "default_value");
```

- `ros::param::get()`: Parameters retrieved through the "bare" version are resolved relative to the node's namespace. Consider the following code:

```
std::string global_name, relative_name, default_param;

if (ros::param::get("/global_name", global_name))
{
  ...
}

if (ros::param::get("relative_name", relative_name))
{
```

```
  ...
}

// Default value version
ros::param::param<std::string>("default_param", default_param, "
    default_value");
```

3.5.2.2 Setting parameters

- ros::NodeHandle::setParam(): Parameters defined through the `NodeHandle` version are resolved relative to that NodeHandle's namespace. Consider the following code:

```
ros::NodeHandle n;
n.setParam("/global_param", 5);
n.setParam("relative_param", "my_string");
n.setParam("bool_param", false);
```

- ros::param::set(): Parameters defined through the "bare" version are resolved relative to the node's namespace. Consider the following code:

```
ros::param::set("/global_param", 5);
ros::param::set("relative_param", "my_string");
ros::param::set("bool_param", false);
```

3.5.2.3 Parameter existence

TODO

3.5.2.4 Deleting parameters

TODO

3.5.2.5 Searching for parameter keys

TODO

3.5.2.6 Getting parameter names

TODO

3.5.2.7 Retrieving Lists

TODO

3.5.3 How to play with parameters in Python

3.5.3.1 Getting parameters

TODO

3.5.3.2 Setting parameters

TODO

3.5.3.3 Parameter existence

TODO

3.5.3.4 Deleting parameters

TODO

3.5.3.5 Searching for parameter keys

TODO

3.5.3.6 Getting parameter names

TODO

3.5.4 Debugging the parameter server in command line using rosparam

- `rosparam list`: Displays the list of parameter names on the local ROS network.
- `rosparam load`:
- `rosparam dump`:
- `rosparam get`:
- `rosparam set`:
- `rosparam delete`:

3.6 Passing arguments to a ROS node

3.6.1 Passing arguments to a C++ program

`argv` and `argc` are how command line arguments are passed to `main()` in C and C++. `argc` will be the number of strings pointed to by `argv`. This will (in practice) be 1 plus the number of arguments, as virtually all implementations will prepend the name of the program to the array. The variables are named `argc` (argument count) and `argv` (argument vector) by convention, but they can be given any valid identifier: `int main(int num_args, char** arg_strings)` is equally valid. They can also be omitted entirely, yielding int `main()`, if you do not intend to process command line arguments. `argc` can also be 0, in which case `argv` can be NULL. Try the following program:

```
#include <iostream>

int main(int argc, char** argv)
{
    std::cout << "Have " << argc << " arguments:" << std::endl;
    for (int i = 0; i < argc; ++i)
    {
        std::cout << argv[i] << std::endl;
    }
}
```

Running it with `./test a1 b2 c3` will output:

```
Have 4 arguments:
./test
a1
b2
c3
```

3.6.2 Passing arguments to a C++ ROS node

```
$ rosrun <package_name> <executable_name> <argument1> ... <argumentN>
```

```
<?xml version="1.0"?>
<launch>
  <arg name="ARM" value="right"/>
  <node name="myCppNode" pkg="package_i" type="testNode"/>
    <rosparam command="load" file="$(find my_package_i)/launch/configs/
    my_yaml_file_$(arg ARM).yaml"/>
  </node>
</launch>
```

```
#include "ros/ros.h"
//...
int main(int argc, char **argv)
{
    ros::init(argc, argv, "myNodeName", ros::init_options::AnonymousName);
    ros::NodeHandle n;
    //...
    ros::Rate loop_rate(10);
    //...
    // Check if the number of expected arguments matches that of the progam
    if(argc != 5)
    {
        ROS_ERROR_STREAM("Invalid number of arguments");
        exit(-1);
    }
    while (ros::ok())
    {
        //Do some computations...
        ros::spinOnce(); // NEVER forget that.
        loop_rate.sleep();
    }
    return 0;
}
```

3.6.3 Passing arguments to a Python script

```
#!/usr/bin/env python
# license removed for brevity
import rospy
# ...
def myFunction(myArg1,myArg2):
    rospy.init_node('myNodeNamePython', anonymous=True)
```

```
7       rate = rospy.Rate(10)
        while not rospy.is_shutdown():
9           # Do some computation ...
            # ...
11          rate.sleep()

13  if __name__ == '__main__':
        try:
15          if len(sys.argv) < 3:
                print("usage: my_node.py arg1 arg2")
17          else:
                myFunction(sys.argv[1], sys.argv[2])
19      except rospy.ROSInterruptException:
            pass
```

3.6.4 Passing arguments to a Python ROS node

Chapter 4

Recording and Visualizing data on ROS

4.1 How to record and replay data with ROS

Using ROS, it is possible to *record* and *replay* data streams transmitted by topics. This is done using the **rosbag** command.

4.1.1 Recording data (creating a bag file)

The following command line records all data that passes through the topics from the moment the command is active (and if events or data are generated on the topics):

```
$ rosbag record -a
```

By default the rosbag "**.bag**" file name consists of year, month, day and hour at which data was recorded. It is then possible to save only some topics, and to specify a file name. For example the following command line only records the topics `/turtle1/cmd_vel` and `/turtle1/pose` in the `myBagFile.bag` file:

```
1 $ rosbag record -O myBagFile /turtle1/cmd_vel /turtle1/pose
```

4.1.2 Replay the recorded data:

First of all, it is possible to check the content of a bag file (i.e. recorded topics) by executing the following command from the bagfiles directory:

```
1 $ rosbag info myBagFile.bag
```

The recorded data of a selected bagfile can then be replayed on the ROS network using the **rosbag play** command:

```
1 $ rosbag play myBagFile.bag
```

4.1.3 Export the recorded data to matlab:

Useful to plot the recorded data. Two methods are possible:

1. The first strategy is to convert the data contained in the .bag file into "comma separated value (.csv)" format using the following command:

```
$ rostopic echo -b myBagFile.bag -p /topic > myData.csv
```

   The .csv file can then be imported on matlab using the `csvread()` command.

2. The other strategy only works with recent versions of matlab (i.e. $\geq R2015a$) and consists in using the built-in matlab command `bag = rosbag('myBagFile.bag')`

4.2 How to visualize data on ROS

4.2.1 Publishing a simple marker and visualizing it in rviz

Check the following tutorial `https://wiki.ros.org/rviz/DisplayTypes/Marker`.

4.2.2 Publishing a tf marker and visualizing it in rviz

4.2.2.1 Setting the CMakeLists and the package manifest

1. Make sure that the manifest `package.xml` has the following lines:

```
<build_depend>tf</build_depend>
...
<exec_depend>tf</exec_depend>
```

2. In the `CMakeLists.txt` file, make sure that `tf` is specified as arguments of your project (see section 5.1.3 for more details):

```
find_package(catkin REQUIRED COMPONENTS
roscpp
rospy
tf
...
)
```

4.2.2.2 Publishing a tf marker in C++

The following code is a modified version of the code presented in section 3.2.2. Please refer to this section for more details on the syntax:

```
#include "ros/ros.h"
#include <tf/transform_broadcaster.h>

int main(int argc, char **argv)
{
// Initialize a new node with the name "my_publisherNode":
ros::init(argc, argv, "my_publisherNode");
ros::NodeHandle n;

//Creating publisher objects of the node:
```

```
tf::TransformBroadcaster my_transformBroadcaster;
ros::Rate loop_rate(10);

// Loop on the posting of messages 10 times per second:
while (ros::ok())
{
// Init Message:
tf::Transform my_transform;
tf::Quaternion q;

double x = 0.2;
double y = 0.25;
double z = 1;
double roll = 0.25;
double pitch = 0.1;
double yaw = 0.2;

// Set Translation:
my_transform.setOrigin(tf::Vector3(x, y, z));

// Set Rotation:
q.setRPY(roll, pitch, yaw);
my_transform.setRotation(q);

// Broadcast Transform:
my_transformBroadcaster.sendTransform(tf::StampedTransform(my_transform, ros
    ::Time::now(), "world", my_FrameName));
// Update node data and wait:
ros::spinOnce();
loop_rate.sleep();
}
return 0;
}
```

- `#include <tf/transform_broadcaster.h>` The `tf` package provides an implementation of a `TransformBroadcaster` to help make the task of publishing transforms easier. To use the `TransformBroadcaster`, we need to include the `tf/transform_broadcaster.h` header file.
- `tf::Transform my_transform`: creates a `TransformBroadcaster` object.
- `my_transform.setOrigin()`: defines the translation component of the transform.
- `my_transform.setRotation()` defines the rotation component of the transform. In a `TransformBroadcaster` object, orientation of the child frame with respect to the parent frame is defined using quaternions. You can use `setRPY()` to automatically convert euler angles into normalized quaternions.
- `my_transformBroadcaster.sendTransform()`: publishes the transform on the ROS network. Sending a transform with a `TransformBroadcaster` requires **four** arguments:

  1. the transform object itself.
  2. a timestamp (This is usually `ros::Time::now()`).
  3. the name of the **parent** frame, in this case "`world`"
  4. the name of the **child** frame , in this case "`my_FrameName`".

4.2.2.3 Publishing a tf marker in Python

The following code is a modified version of the code presented in section 3.2.3. Please refer to this section for more details on the syntax:

```
#!/usr/bin/env python
# license removed for brevity
import rospy
import tf

def myPublishFunction():
  my_transformBroadcaster = tf.TransformBroadcaster()

  rospy.init_node('my_publisherNode', anonymous=True)
  rate = rospy.Rate(10)

  x = 0.2;
  y = 0.25;
  z = 1;
  roll = 0.25;
  pitch = 0.1;
  yaw = 0.2;

  while not rospy.is_shutdown():
    my_transformBroadcaster.sendTransform((x, y, z),
                tf.transformations.quaternion_from_euler(roll, pitch, yaw),
                rospy.Time.now(),my_FrameName,"world")
    rate.sleep()

if __name__ == '__main__':
  try:
    myPublishFunction()
  except rospy.ROSInterruptException:
    pass
```

- `import tf` The `tf` package provides an implementation of a `TransformBroadcaster` to help make the task of publishing transforms easier.
- `my_transformBroadcaster=tf.TransformBroadcaster()`: creates a `TransformBroadcaster` object.
- `my_transformBroadcaster.sendTransform()`: publishes the transform on the ROS network. Sending a transform with a `TransformBroadcaster` requires **five** arguments:
  1. the translation component of the transform
  2. the rotation component of the transform
  3. a timestamp (This is usually `ros::Time::now()`).
  4. the name of the **child** frame , in this case "`my_FrameName`".
  5. the name of the **parent** frame, in this case "`world`"

4.2.3 Visualizing data using rqt_plot

Check the following tutorial `https://wiki.ros.org/rqt_plot`.

Chapter 5

Mastering ROS CMakeLists, Package Manifests and Launchfiles

5.1 The dark magic behind CMakeList files

The file "**CMakeLists.txt**" is the input of the CMake build system, allowing to compile software packages[1]. Any CMake-compliant package contains one or more CMakeLists.txt file that describe how to build the code and where to install it. A CMakeLists.txt file **MUST** follow the following format:

1. Required CMake Version (`cmake_minimum_required`)
2. Package Name (`project()`)
3. Find other CMake/Catkin packages needed for build (`find_package()`)
4. Enable Python module support (`catkin_python_setup()`)
5. Message/Service/Action Generators (`add_message_files()`, `add_service_files()`, `add_action_files()`)
6. Invoke message/service/action generation (`generate_messages()`)
7. Specify package build info export (`catkin_package()`)
8. Libraries/Executables to build (`add_library()/add_executable()/target_link_libraries()`)
9. Tests to build (`catkin_add_gtest()`)
10. Install rules (`install()`)

Important note: The step 4-6 and 9-10 are optional.

5.1.1 CMake Version

Every catkin CMakeLists.txt file must start with the required version of CMake needed. Catkin requires version 2.8.3 or higher:

```
1 cmake_minimum_required(VERSION 2.8.3)
```

[1]The content of this section is mostly inspired from the following wiki: `https://wiki.ros.org/catkin/CMakeLists.txt`

5.1.2 Package name

The next step is to set the name of the package you wan to build (here `package_i`, c.f. Fig.1.4):

```
project(package_i)
```

You can reference the project name anywhere later in the CMake script by using the variable `${PROJECT_NAME}` wherever needed.

5.1.3 Finding required CMake packages with "find_package()"

It is then necessary to specify which other CMake packages have to be found in order to build our project. This is achieved using the CMake `find_package` function. There is always at least one dependency on catkin:

```
find_package(catkin REQUIRED)
```

If your project depends on other packages, they are automatically turned into components (in terms of CMake) of catkin. Instead of using `find_package` on those packages, if you specify them as components, it will make life easier. For example, if you use the packages "`urdf`, `roscpp`, and `std_msgs`":

```
find_package(catkin REQUIRED COMPONENTS urdf
                                        roscpp
                                        std_msgs)
```

5.1.3.1 What does find_package() actually do?

If a package is found by CMake through `find_package()`, it results in the creation of several CMake **environment variables** that will give information about the found package. These environment variables can be utilized later in the CMake script. The environment variables describe where the packages exported header files are, where source files are, what libraries the package depends on, and the paths of those libraries. The names always follow the convention of `<PACKAGE NAME>_<PROPERTY>`:

- `<NAME>_FOUND` - Set to true if the library is found, otherwise false
- `<NAME>_INCLUDE_DIRS` or `<NAME>_INCLUDES` - The include paths exported by the package
- `<NAME>_LIBRARIES` or `<NAME>_LIBS` - The libraries exported by the package

5.1.3.2 Boost

If using C++ and Boost, you need to invoke `find_package()` on Boost and specify which aspects of Boost you are using as components. For example, if you wanted to use Boost threads, you would say:

```
find_package(Boost REQUIRED COMPONENTS thread)
```

5.1.4 Declaring the package attributes with "catkin_package()"

`catkin_package()` is a catkin-provided CMake macro. This is required to specify catkin-specific information to the build system which in turn is used to generate pkg-config and CMake files. This function must be called before declaring any targets with `add_library()` or `add_executable()`. The function has 5 optional arguments:

1. INCLUDE_DIRS - The exported include paths (i.e. cflags) for the package
2. LIBRARIES - The exported libraries from the project
3. CATKIN_DEPENDS - Other catkin projects that this project depends on
4. DEPENDS - Non-catkin CMake projects that this project depends on. For a better understanding, see this explanation.
5. CFG_EXTRAS - Additional configuration options

Full macro documentation can be found at the following address: `https://docs.ros.org/kinetic/api/catkin/html/dev_guide/generated_cmake_api.html#catkin-package` As an example:

```
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES ${PROJECT_NAME}
  CATKIN_DEPENDS roscpp urdf std_msgs
  DEPENDS eigen opencv)
```

- the folder "`include`" contains the package headers
- the environment variable `${PROJECT_NAME}` corresponds to what was passed to the `project()` function in 5.1.2 (in this case "`package_i`")
- The packages "`roscpp`", "`urdf`" and "`std_msgs`" are catkin dependencies required to build/run the considered package
- "`eigen`" and "`opencv`" are system dependencies that need to be present to build/run the considered package

5.1.5 Specifying build targets

Build targets can take many forms, but usually they represent one of two possibilities:

1. **Executable Target** - programs we can run
2. **Library Target** - libraries that can be used by executable targets at build and/or runtime

5.1.5.1 Include paths and library paths with "include_directories()" and "link_directories()":

Prior to specifying targets, it is necessary to specify where resources can be found for said targets, specifically header files and libraries:

- **Include Paths** - Where can header files be found for the code (most common in C/C++) being built
- **Library Paths** - Where are libraries located that executable target build against?

The argument to `include_directories(<dir1>, <dir2>, ..., <dirN>)` should be the `*_INCLUDE_DIRS` variables generated by the `find_package()` calls and any additional directories that need to be included:

```
include_directories(include ${Boost_INCLUDE_DIRS} ${catkin_INCLUDE_DIRS})
```

The first argument "include" indicates that the include directory within the package is also part of the path. The CMake `link_directories(<dir1>, <dir2>, ..., <dirN>)` function can be used to add additional library paths, however, this is **not recommended**. All catkin and CMake packages automatically have their link information added when they are "`find_packaged`". Simply link against the libraries in `target_link_libraries()` as explained in section 5.1.5.4.

5.1.5.2 Add executable targets with "add_executable()":

To specify an executable target that must be built, we must use the `add_executable()` CMake function.

```
add_executable(myProgram src/my_node1_i.cpp ... src/my_nodeX_i.cpp)
```

This will build a target executable called `myProgram` which is built from X different source files: `src/my_node1_i.cpp`,$\cdots$, `src/my_nodeX_i.cpp`.

5.1.5.3 Add library targets with "add_library()":

To specify libraries to build, we must use the `add_library()`. By default catkin builds shared libraries:

```
add_library(${PROJECT_NAME} ${${PROJECT_NAME}_SRCS})
```

5.1.5.4 Specify which librarie executable targets links against with "target_link_libraries()":

Use the `target_link_libraries()` function to specify which libraries an executable target links against. This is done typically after an *add_executable()* call.

```
target_link_libraries(<executableTargetName>, <lib1>, <lib2>, ... <libN>)
```

Example:

```
add_executable(foo src/my_node1_i.cpp)
add_library(moo src/my_node1_2.cpp)
target_link_libraries(foo moo ${catkin_LIBRARIES})
```

This code links **foo** against **libmoo.so**. Note that there is no need to use `link_directories()` in most use cases as that information is automatically pulled in via `find_package()`. In case you want to add a custom library e.g. libpouetpouet.so) to your project, you can proceed in the following manner:

```
FIND_LIBRARY(POUETPOUET_LIBRARY pouetpouet /absPath/libpouetpouet)
TARGET_LINK_LIBRARIES(testExecutable ${POUETPOUET_LIBRARY})
```

5.1.6 Messages, Services, and Action Targets

Messages (`.msg`), services (`.srv`), and actions (`.action`) files in ROS require a special preprocessor build step before being built and used by ROS packages. The point of these macros is to generate programming language-specific files so that one can utilize messages, services, and actions in their programming language of choice. The build system will generate bindings using all available generators (e.g. gencpp, genpy, genlisp, etc). There are three macros provided to handle messages, services, and actions respectively:

1. `add_message_files()`
2. `add_service_files()`
3. `add_action_files()`

These macros must then be followed by a call to the macro that invokes generation " `generate_messages()`".

5.1.6.1 Important Prerequisites/Constraints

- these macros must come BEFORE the `catkin_package()` macro in order for generation to work correctly.:

```
find_package(catkin REQUIRED COMPONENTS ...)
add_message_files(...)
add_service_files(...)
add_action_files(...)
generate_messages(...)
catkin_package(...)
...
```

- the *catkin_package()* macro must have a `CATKIN_DEPENDS` dependency on `message_runtime`.

```
catkin_package(
...
CATKIN_DEPENDS message_runtime ...
...)
```

- You must use `find_package()` for the package `message_generation`, either alone or as a component of catkin:

```
find_package(catkin REQUIRED COMPONENTS message_generation)
```

- The `package.xml` file must contain a build dependency on `message_generation` and a runtime dependency on `message_runtime`:

```
<build_depend>message_generation</build_depend>
...
<run_depend>message_runtime</run_depend>
```

- If a target or package depends on some other target that needs messages/services/actions to be built, it is necessary to add an explicit dependency on `catkin_EXPORTED_TARGETS` and `${PROJECT_NAME}_EXPORTED_TARGETS`, so that they are built in the correct order:

```
add_dependencies(some_target ${${PROJECT_NAME}_EXPORTED_TARGETS} ${
    catkin_EXPORTED_TARGETS})
```

5.1.6.2 Example

If your package has two messages in a directory called "msg" named "`MyMessage1.msg`" and "`MyMessage2.msg`" and these messages depend on `std_msgs` and `sensor_msgs`, a service in a directory called "srv" named "*MyService.srv*", defines executable `message_program` that uses these messages and service, and executable `does_not_use_local_messages_program`, which uses some parts of ROS, but not the `messages/service` defined in this package, then you will need the following in your CMakeLists.txt. If, additionally, you want to build `actionlib` actions, and have an action specification file called "`MyAction.action`" in the "action" directory, you must add `actionlib_msgs` to the list of components which are `find_packaged` with catkin and add the following call before the call to `generate_messages`. Furthermore the package must have a build dependency on `actionlib_msgs`:

```
# Get the information about this package's buildtime dependencies
find_package(catkin REQUIRED COMPONENTS message_generation std_msgs
    sensor_msgs)

# Declare the message files to be built
add_message_files(FILES
                  MyMessage1.msg
                  MyMessage2.msg
)

# Declare the service files to be built
add_service_files(FILES
                  MyService.srv
)

# Declare the action files to be built
add_action_files(FILES
                 MyAction.action
)

# Actually generate the language-specific message and service files
generate_messages(DEPENDENCIES std_msgs sensor_msgs)

# Declare that this catkin package's runtime dependencies
catkin_package(CATKIN_DEPENDS message_runtime std_msgs sensor_msgs)

# define executable using MyMessage1 etc.
add_executable(message_program src/main.cpp)
add_dependencies(message_program ${${PROJECT_NAME}_EXPORTED_TARGETS} ${
    catkin_EXPORTED_TARGETS})

# define executable not using any messages/services provided by this package
add_executable(does_not_use_local_messages_program src/main.cpp)
add_dependencies(does_not_use_local_messages_program ${
    catkin_EXPORTED_TARGETS})
```

5.1.7 Enabling Python module support

If your ROS package provides some Python modules, you should create a setup.py file and call:

```
catkin_python_setup()
```

before the call to *generate_messages()* and *catkin_package()*.

5.1.8 Optional Step: Specifying Installable Targets

After build time, targets are placed into the *devel* space of the catkin workspace. However, often we want to install targets to the system. In other words, if you want to be able to do a "make install" of your code, you need to specify where targets should end up. This is done using the CMake `install()` function which takes as arguments:

1. `TARGETS` - which targets to install
2. `ARCHIVE DESTINATION` - Static libraries and DLL (Windows) .lib stubs

3. `LIBRARY DESTINATION` - Non-DLL shared libraries and modules
4. `RUNTIME DESTINATION` - Executable targets and DLL (Windows) style shared libraries

5.1.8.1 Installing Python Executable Scripts

The first step is to make the python script executable:

```
chmod a+x scripts/my_scriptX_i.py
```

To mark the executable script "`my_scriptX_i.py`" for installation, just add the following lines at the end of the `CMakelist.txt` file:

```
install(PROGRAMS
  scripts/my_ScriptX_i.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

5.1.8.2 Installing header files

5.1.8.3 Installing roslaunch Files or Other Resources

5.2 Writing proper package manifests

5.2.1 Basic structure of the package.xml manifest

Each package.xml file has the `<package>` tag as the root tag in the document. Two different formats can be specified in the `<package>` field. Although the format 1 is set by default, the format 2 is recommended for new packages (see the discussion in `https://wiki.ros.org/Manifest`):

```
<?xml version="1.0"?>
<package format="2">
  ...
</package>
```

5.2.1.1 Required Tags

A set of 5 fields is needed for the package manifest to be complete:

1. `<name>`: name of the package.
2. `<version>`: the version number of the package.
3. `<description>`: description of the package contents
4. `<maintainer>`: name of the package maintainer
5. `<license>`: license under which the code is published. The most frequently used license is BSD because it allows the code to be exploited for commercial applications. It is important to pay attention to the context in which the code may be licensed, especially in a professional context.

```
<?xml version="1.0"?>
<package format="2">
  <name>package_i</name>
  <version>0.0.0</version>
  <description>Description of the package... blablabla... </description>
```

```
  <!-- One maintainer tag required, multiple allowed, one person per tag -->
  <maintainer email="donald.trump@tum.de">Donald Trump</maintainer>
  <!-- One lic. tag required, multiple allowed, one license per tag -->
  <!-- Commonly used lic. strings: -->
  <!-- BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
  <license>BSD</license>
  <!-- Author tags are optional, mutiple are allowed, one per tag -->
  <!-- Authors do not have to be maintianers, but could be -->
  <author email="donald.trump@tum.de">Donald Trump</author>
  <!-- Url tags are optional, but mutiple are allowed, one per tag -->
  <!-- Optional attribute can be: website, bugtracker, or repository -->
  <url type="website">http://ics.ei.tum.de</url>
  ...
</package>
```

5.2.1.2 Dependencies

Packages can have 6 types of dependencies:

1. **build dependencies**: specify which packages are needed to build this package. This is the case when any file from these packages is required at build time. The corresponding tag is: `<depend>`. **Note that this is replacing the `<build_depend>` and `<run_depend>` tags of the package format 1.** If you only use some particular dependency for building your package, and not at execution time, you can use the `<build_depend>` tag. A more detailed explanation can be found at the following address: `https://docs.ros.org/indigo/api/catkin/html/howto/format2/migrating_from_format_1.html#migrating-from-format1-to-format2`
2. **build tool dependencies**: specify build system tools which this package needs to build itself. Typically the only build tool needed is catkin. The corresponding tag is: `<build_tool_depend>`
3. **build export dependencies**: specify which packages are needed to build libraries against this package. The corresponding tag is: `<build_export_depend>`
4. **execution dependencies**: specify which packages are needed to run code in this package. The corresponding tag is: `<exec_depend>`
5. **test dependencies**: specify only additional dependencies for unit tests. They should never duplicate any dependencies already mentioned as build or run dependencies. The corresponding tag is: `<test_depend>`
6. **doc dependencies**: specify documentation tools which this package needs to generate documentation. The corresponding tag is: `<doc_depend>`

```
<?xml version="1.0"?>
<package format="2">
  ...
  <buildtool_depend>catkin</buildtool_depend>

  <depend>roscpp</depend>
  <depend>std_msgs</depend>
  <build_depend>message_generation</build_depend>
  <exec_depend>message_runtime</exec_depend>
  <exec_depend>rospy</exec_depend>
  <test_depend>python-mock</test_depend>
  <doc_depend>doxygen</doc_depend>
</package>
```

5.2.2 Metapackages

It is often convenient to group multiple packages as a single logical package. This can be accomplished through metapackages. A metapackage is a normal package with the following export tag in the `package.xml`:

```
<?xml version="1.0"?>
<package format="2">
  ...
   <export>
     <metapackage />
   </export>
</package>
```

Other than a required `<buildtool_depends>` dependency on catkin, metapackages can only have execution dependencies on packages of which they group. Additionally a metapackage has a required, boilerplate `CMakeLists.txt` file:

```
cmake_minimum_required(VERSION 2.8.3)
project(package_i)
find_package(catkin REQUIRED)
catkin_metapackage()
```

5.3 Writing proper launch files

5.3.1 Specifying the ROS nodes to execute

TODO

5.3.2 Specifying the parameters

TODO

5.3.3 Specifying the arguments

TODO IMPORTANT: roslaunch does not guarantee any particular order to the startup of nodes.

5.3.4 Detail of the launch file tags

5.3.4.1 <launch> tag:

The <launch> tag is the root element of any `roslaunch` file. Its sole purpose is to act as a container for the other elements.

5.3.4.2 <node> tag:

- `pkg="package_i"`: ROS package of the considered node.
- `type="message_program"`: Node type. There must be a corresponding executable with the same name defined in the CMakelist.txt file. Here, `"message_program"` corresponds to `src/main.cpp` in CMakelist.txt.
- `name="nodename"` Node name in the rosgraph. Overwrite the name defined in the `ros::init()` function.
- `args="arg1 arg2 ... argX"`(optional): Pass arguments to node.

- `machine="machine-name"`(optional) Launch node on designated machine.
- `respawn="true"`(optional, default: False): Restart the node automatically if it quits.
- `respawn_delay="30"` (optional, default 0): If respawn is true, wait `respawn_delay` seconds after the node failure is detected before attempting restart.
- `required="true"` (optional): If node dies, kill entire `roslaunch`.
- `ns="foo"` (optional): Start the node in the "`foo`" namespace.
- `clear_params="true/false"` (optional): Delete all parameters in the node's private namespace before launch.
- `output="log/screen"` (optional): If 'screen', stdout/stderr from the node will be sent to the screen. If 'log', the stdout/stderr output will be sent to a log file in `$ROS_HOME/log`, and `stderr` will continue to be sent to screen. The default is `'log'`.
- `cwd="ROS_HOME/node"` (optional): If 'node', the working directory of the node will be set to the same directory as the node's executable. In C Turtle, the default is '`ROS_HOME`'.
- `launch-prefix="prefix arguments"` (optional): Command/arguments to prepend to node's launch arguments. This is a powerful feature that enables you to enable gdb, valgrind, xterm, nice, or other handy tools.

5.3.4.3 <param> tag:

The <param>tag defines a parameter to be set on the Parameter Server. Instead of value, you can specify a textfile, binfile or command attribute to set the value of a parameter. The <param> tag can be put inside of a <node> tag, in which case the parameter is treated like a private parameter.

Part II

ROS: Advanced Features

Chapter 6

Bridging ROS and OpenCV for Computer Vision Applications

6.1 Subscribing to a video stream

TODO

6.2 Aruco Marker Detection

TODO

Chapter 7

Trajectory planning using ROS and MoveIt

TODO

Chapter 8

Kinematic and Dynamic Modeling of a Robot using ROS

8.1 Understanding robot modeling using URDF

ROS naturally comes with a wide variety of powerful tools, allowing to generate three-dimensional animated representations of robots. These representations are usually coded into so called "URDF" files. The Unified Robot Description Format (URDF) is a specific XML convention, describing robots in a way that is both both human- and machine-readable. A URDF file basically contains all the informations necessary to generate the *geometric*, *kinematic*, *dynamic*, *collision*, and *sensory* models of a specific robot. These informations are organized using a set of dedicated *tags*: since robots are defined as a set of links, joints and sensors, the corresponding tags allow to precisely parameterize each of these elements:

- **Link**: The link tag allows to model a robot link and its properties. The syntax is as follows:

```
<link name="<name_of_the_link>">
<inertial>.....</inertial>        #Optional
<visual> .....</visual>           #Optional
<collision>.....</collision>      #Optional
</link>
```

  As indicated by its name, the "`visual`" section describes the link visual properties (e.g. size, geometry or color). It is even possible to import a 3D mesh in order to represent the robot link in a more realistic way. The "`collision`" section describes the link collision model. This model usually encapsulates the real link in order to detect collision before it actually happens. Finally the "`inertial`" section defines the link dynamic parameters (e.g. mass, inertia). Figure 8.2a shows a representation of a single link as it is defined in the URDF convention. More details can be found at the following address: `http://wiki.ros.org/urdf/XML/link`.

- **Joint**: The joint tag is used in order to describe robot joints and their properties. The syntax is as follows:

```
<joint name="name_of_the_joint" type="type_of_the_joint">
<origin xyz="z y z" rpy="r p y"/>              #Optional
<parent link="link1"/>                         #Required
<child link="link2"/>                          #Required
<axis .... />                                  #Optional
<calibration rising, falling />                #Optional
```

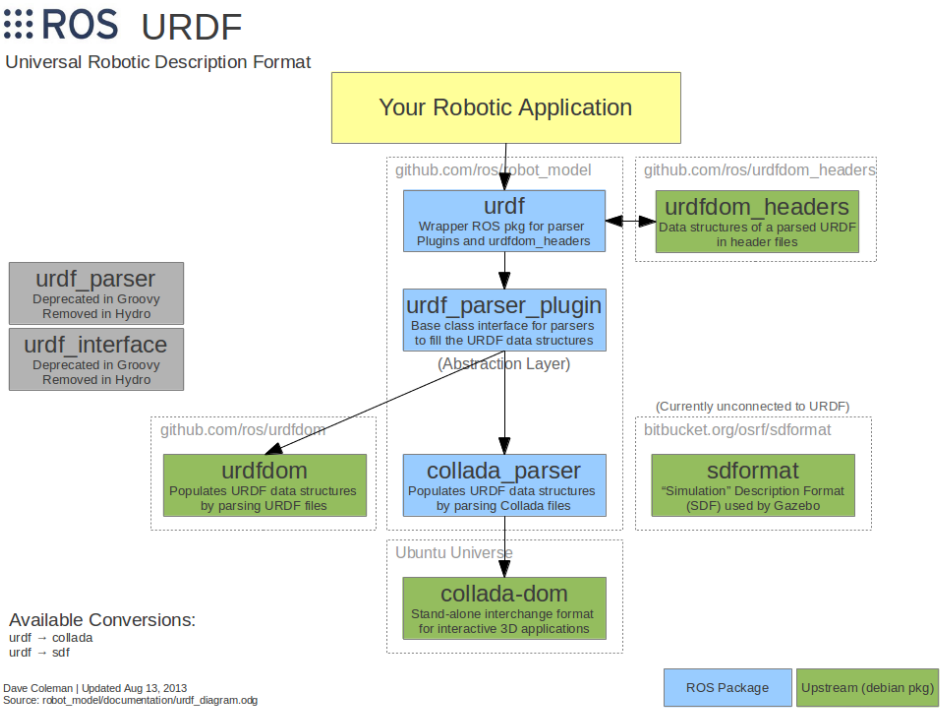


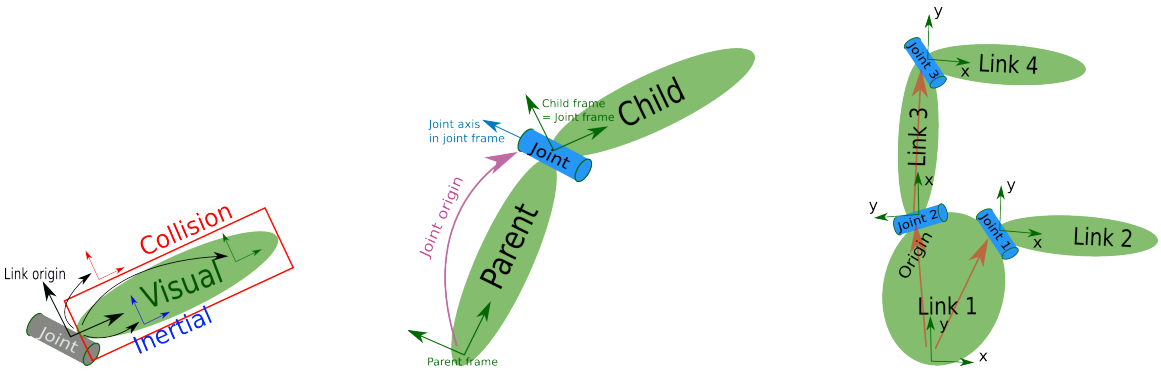
Figure 8.1: A number of different packages and components make up urdf. The following diagram attempts to explain the relationship between these components

```
  <dynamics damping, friction/>                    #Optional
  <limit lower, upper, effort, velovity />         #Optional
  </joint>
```

A URDF joint is always defined with respect to a `parent` and a `child` link. The joint tag supports the following types of joints: prismatic, revolute, continuous, fixed, floating, and planar. Optionally, joints can be provided with dynamics properties, such as damping or friction. A set of limits can even be defined in order to better fit to the real robot. Figure 8.2b provides an illustration of a robot joint with its different links. More details can be found at the following address: `http://wiki.ros.org/urdf/XML/joint`.

- **Robot**: This tag describes the root element of the URDF: it must *encapsulate the entire robot model*. Inside the robot tag, we can define the name of the robot as well as its different links and joints. The syntax is as follows:

```
  <robot name="<name_of_the_robot>"
  <link> ..... </link>
  <link> ..... </link>
  <joint> ..... </joint>
  <joint> ..... </joint>
  </robot>
```



(a) Visualization of a URDF link

(b) Visualization of a URDF joint

(c) Robot URDF model

Figure 8.2: URDF tags

As showed in Figure 8.2c, the robot model consists of a set of *links*, interconnected by *joints*, into a *kinematic chain*.

- More URDF tags can be found at `http://wiki.ros.org/urdf/XML`.

In ROS, the `urdf` package provides a powerful C++ parser, allowing to recursively generate the entire mathematical model of a robot based on its URDF file. Although highly versatile, the URDF is however limited to the description of rigid robots with tree-like[1] kinematic chains.

8.2 Understanding robot modeling using XACRO

8.2.1 Motivation:

The use of URDF may become problematic in the case of complex robotic systems, since its non-modular nature unnecessarily increases the code complexity as the number of degrees of freedom of the considered robot get bigger. In fact, some of the main features that URDF is missing are the simplicity, reusability, modularity, and programmability. Modularity, in particular, refers to the possibility of including several URDF files as subparts of a main robot description file, thereby resulting in enhanced code readability. Programmability here refers to the possibility of defining variables, constants, mathematical expressions, or conditional statement, in the description language, in order to make it more user friendly. XACRO (Xml-mACROS) can be considered as an updated version of URDF, created with these problems in mind. Capable of generating – or importing – reusable macros within a given robot description, the XACRO language moreover support simple programming statements in its description. The possibility of using variables, constants, mathematical expressions or conditional statements makes the robot description more intelligent and efficient. XACRO files can be automatically converted to URDF whenever it is necessary, using dedicated ROS tools.

8.2.2 Using properties:

Using XACRO, we can declare constants or properties which can be used anywhere in the code. The main use of these constant definitions are, instead of giving hard coded values on links and joints, we can keep constants like this and it will be easier to change these values rather than finding the

[1]Parallel robots cannot be described using URDF

hard coded values and replacing them. An example of using properties are given here. We declare the base link and pan link's length and radius. So, it will be easy to change the dimension here rather than changing values in each one:

```
<xacro:property name="base_link_length" value="0.01" />
<xacro:property name="base_link_radius" value="0.2" />
<xacro:property name="pan_link_length" value="0.4" />
<xacro:property name="pan_link_radius" value="0.04" />
```

We can use the value of the variable by replacing the hard coded value by the following definition as given here:

```
<cylinder length="${pan_link_length}"
radius="${pan_link_radius}"/>
```

Here, the old value "0.4" is replaced with "{pan_link_radius}" , and "0.04" is replaced with "{pan_link_radius}" .

8.2.3 Using the math expression:

We can build mathematical expressions inside ${} using the basic operations such as + , - , * , / , unary minus, and parenthesis. Exponentiation and modulus are not supported yet. The following is a simple math expression used inside the code:

```
<cylinder length="${pan_link_length}"
radius="${pan_link_radius+0.02}"/>
```

8.2.4 Using macros:

One of the main features of xacro is that it supports macros. We can reduce the length complex definition using xacro to a great extent. Here is a xacro definition we used in our code for inertial:

```
<xacro:macro name="inertial_matrix" params="mass">
<inertial>
<mass value="${mass}" />
<inertia ixx="0.5" ixy="0.0" ixz="0.0"
iyy="0.5" iyz="0.0" izz="0.5" />
</inertial>
</xacro:macro>
```

Here, the macro is named `inertial_matrix` , and its parameter is mass. The mass parameter can be used inside the inertial definition using ${mass} . We can replace each inertial code with a single line as given here:

```
<xacro:inertial_matrix mass="1"/>
```

The xacro definition improved the code readability and reduced the number of lines compared to urdf. Next, we can see how to convert xacro to the urdf file.

8.2.4.1 Conversion of XACRO to URDF:

After designing the xacro file, we can use the following command to convert it into a UDRF file:

```
$ rosrun xacro xacro.py pan_tilt.xacro > pan_tilt_generated.urdf
```

We can use the following line in the ROS launch file for converting xacro to UDRF and use it as a `robot_description` parameter:

```
<param name="robot_description" command="$(find xacro)/xacro.py
$(find mastering_ros_robot_description_pkg)/urdf/pan_tilt.xacro"/>
```

8.3 Building a URDF robot model

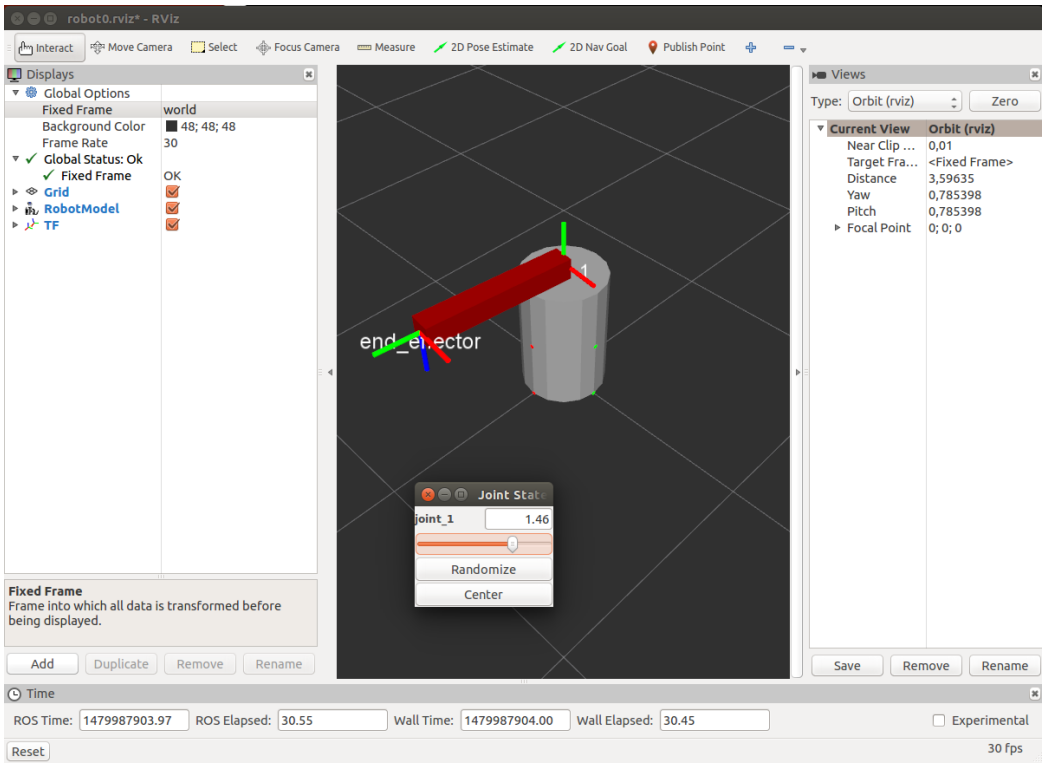


Figure 8.3: Robot visualization on Rviz

1. Create a folder for your model inside your catkin workspace:

```
$ mkdir urdf_tutorial
```

2. Create a ros package for the robot description:

```
$ cd urdf_tutorial
$ catkin_create_pkg robot0_description urdf
```

3. Create a folder called urdf:

```
$ cd robot0_description
$ mkdir urdf
```

4. Create a file named `robot0.urdf`:

```
$ cd urdf
$ gedit robot0.urdf
```

5. Start the file as a standard xml 1.0. Then start a robot element:

```
<?xml version="1.0"?>
<robot name="robot0">
</robot>
```

6. Define the first link as world. This will be an empty link and is used to state the world frame. Just after this link add a joint element as fixed.

```
<link name="world" />

<joint name="joint_0" type="fixed">
<parent link="world"/>
<child link="link_0"/>
</joint>
```

7. After this, we can define the further kinematic chain with as many links and joints as needed:

```
<?xml version="1.0"?>
<robot name="robot0">

<link name="world" />

<joint name="joint_0" type="fixed">
<parent link="world"/>
<child link="link_0"/>
</joint>

<link name="link_0">
<visual>
<geometry>
<cylinder length="0.6" radius="0.2"/>
</geometry>
<material name="gray">
<color rgba="0.5 0.5 0.5 1"/>
</material>
</visual>
</link>

<joint name="joint_1" type="revolute">
<origin xyz="0 0 0.35" rpy="1.57079632679 0 0"/>
<parent link="link_0"/>
<child link="link_1"/>
<limit effort="30" velocity="1.0" lower="-3.1415926535897931" upper
  ="3.1415926535897931" />
<axis xyz="0 1 0"/>
```

```
  </joint>

  <link name="link_1">
  <visual>
  <origin xyz="0 0 0.35" rpy="0 0 0"/>
  <geometry>
  <box size="0.1 0.1 0.7" />
  </geometry>
  <material name="red">
  <color rgba="0.5 0.0 0.0 1"/>
  </material>
  </visual>
  </link>

  <joint name="end_efector_joint" type="fixed">
  <origin xyz="0 0 0.7" rpy="1.57079632679 0 0"/>
  <parent link="link_1"/>
  <child link="end_effector"/>
  </joint>

  <link name="end_effector" />

  </robot>
```

8. To test the descriptor, create another package inside the `urdf_tutorial` folder.

```
$ catkin_create_pkg robot0_bringup robot_state_publisher
  robot0_description
```

9. Create a launch folder and a launch file inside this package:

```
$ cd robot0_bringup
$ mkdir launch
$ cd launch
$ gedit robot0_bringup.launch
```

10. Copy this code inside the launch file:

```
<?xml version="1.0"?>

<launch>

<arg name="gui" default="false" />

<param name="robot_description" command="cat $(find robot0_description
  )/urdf/robot0.urdf" />
<param name="use_gui" value="$(arg gui)"/>

<node name="joint_state_publisher" pkg="joint_state_publisher" type="
  joint_state_publisher" />

<node name="robot_state_publisher" pkg="robot_state_publisher" type="
  state_publisher" />
<node name="rviz" pkg="rviz" type="rviz" args="-d $(find
  robot0_bringup)/launch/rviz_config/robot0.rviz" required="true" />
```

```
15  </launch>
```

11. Compile the catkin workspace.
12. Run in one terminal the next command to test the model with a joint command gui:

```
$ roslaunch robot0_bringup robot0_bringup.launch gui:=true
```

13. The rviz window will look empty. Here set the fixed frame to world and import the robot model and the TF tree.
14. Save the rviz config file into the launch file in a folder named `rviz_config` as `robot0.rviz`

8.4 Building a XACRO model using macros

1. Create a ros package for the robot description:

```
$ cd urdf_tutorial
2 $ catkin_create_pkg r1_robot_description urdf xacro
```

2. Create a folder called urdf:

```
1 $ cd r1_robot_description
  $ mkdir urdf
```

3. Create a file named `r1_robot.xacro`:

```
1 $ cd urdf
  $ gedit r1_robot.xacro
```

4. Start the file as a standard xml 1.0. Then start a robot element stating the xacro specification for the parser. In this case we will not specify the robot name:

```
1 <?xml version="1.0"?>
  <robot xmlns:xacro="http://www.ros.org/wiki/xacro" >
3
  </robot>
```

5. Inside the robot element, we can define constants as properties:

```
1 <property name="M_PI" value="3.1415926535897931" />
  <property name="DEG2RAD" value="0.01745329251994329577" />
```

6. After this, we can define the xacro macro specifying the arguments. Note that the arguments are used inside the macro using the ${ } tags:

```
1 <?xml version="1.0"?>
  <robot xmlns:xacro="http://www.ros.org/wiki/xacro">
3
  <property name="M_PI" value="3.1415926535897931" />
5 <property name="DEG2RAD" value="0.01745329251994329577" />
```

```
  <xacro:macro name="single_joint_robot" params="name xposition
   yposition">

  <joint name="${name}_joint_0" type="fixed">
  <parent link="world"/>
  <child link="${name}_link_0"/>
  <origin rpy="0 0 0" xyz="${xposition} ${yposition} 0" />
  </joint>

  <link name="${name}_link_0">
  <visual>
  <geometry>
  <cylinder length="0.6" radius="0.2"/>
  </geometry>
  <material name="gray">
  <color rgba="0.5 0.5 0.5 1"/>
  </material>
  </visual>
  </link>

  <joint name="${name}_joint_1" type="revolute">
  <origin xyz="0 0 0.35" rpy="${M_PI/2} 0 0"/>
  <parent link="${name}_link_0"/>
  <child link="${name}_link_1"/>
  <limit effort="30" velocity="1.0" lower="-${M_PI}" upper="${M_PI}" />
  <axis xyz="0 1 0"/>
  </joint>

  <link name="${name}_link_1">
  <visual>
  <origin xyz="0 0 0.35" rpy="0 0 0"/>
  <geometry>
  <box size="0.1 0.1 0.7" />
  </geometry>
  <material name="red">
  <color rgba="0.5 0.0 0.0 1"/>
  </material>
  </visual>
  </link>

  <joint name="${name}_end_efector_joint" type="fixed">
  <origin xyz="0 0 0.7" rpy="${M_PI/2} 0 0"/>
  <parent link="${name}_link_1"/>
  <child link="${name}_end_effector"/>
  </joint>

  <link name="${name}_end_effector" />

  </xacro:macro>

  </robot>
```

7. Now create a folder named `robots` inside the `r1_robot_description` package

```
  $ cd r1_robot_description
  $ mkdir robots
```

8. Create a file named `2robot.xacro`:

```
$ cd robots
$ gedit 2robot.xacro
```

9. In this file we will start a robot model specifying a name and the xacro parser. Then include the `r1_robot.xacro`. Finally, create a *world* link and use the `single_joint_robot` macro twice to create two ramifications from the world link.

```
<?xml version="1.0"?>

<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="r1_robot">

<xacro:include filename="$(find r1_robot_description)/urdf/r1_robot.
  xacro" />

<link name="world" />

<xacro:single_joint_robot name= "robot1" xposition="0.0" yposition
  ="0.0" />

<xacro:single_joint_robot name= "robot2" xposition="1.0" yposition
  ="0.0" />

</robot>
```

10. To test the descriptor, create another package inside the `urdf_tutorial` folder.

```
$ catkin_create_pkg r1_robot_bringup robot_state_publisher
  r1_robot_description
```

11. Create a launch folder and a launch file inside this package:

```
$ cd r1_robot_bringup
$ mkdir launch
$ cd launch
$ gedit two_robots_bringup.launch
```

12. Copy this code inside the launch file:

```
<?xml version="1.0"?>

<launch>

<arg name="gui" default="false" />

<param name="robot_description" command="$(find xacro)/xacro.py '$(
  find r1_robot_description)/robots/2robot.xacro'" />
<param name="use_gui" value="$(arg gui)"/>

<node name="joint_state_publisher" pkg="joint_state_publisher" type="
  joint_state_publisher" />

```

```
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="
    state_publisher" />
13  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
    r1_robot_bringup)/launch/rviz_config/r1_robot.rviz" required="true"
    />

15  </launch>
```

Note that this time, the `robot_description` is loaded using the xacro parser.

13. Compile the catkin workspace.
14. Run in one terminal the next command to test the model with a joint command gui:

```
$ roslaunch r1_robot_bringup r1_robot_bringup.launch gui:=true
```

15. The rviz window will look empty. Here set the fixed frame to world and import the robot model and the TF tree.
16. Save the rviz config file into the launch file in a folder named `rviz_config` as `r1_robot.rviz`

Chapter 9

Generating Real-Time Controllers using ROS Control

9.1 Introduction

9.1.1 What is ROS Control

ROS control aims at simplifying the process of interfacing controllers to robot-hardware within the ROS environment. The `ros_control` framework provides the capability to implement and manage robot controllers with a focus on both **real-time performance** and sharing of controllers in a "**robot-agnostic**"[1] manner. The primary motivation for a separate robot-control framework is the lack of real-time-safe communication layer in ROS (see 1.3.2). Furthermore, the framework implements solutions for controller-lifecycle and hardware resource management as well as abstractions on hardware interfaces with minimal assumptions on hardware or operating system. The `ros_control` packages are a robot-agnostic rewrite of the former `pr2_controller_manager`[2] packages. Its development started in late 2012 by *hiDOF* in collaboration with the *Willow Garage* and *PAL Robotics*.

9.1.2 How does ROS Control works

The backbone of the ROS Control framework is the robot "**HardWare Abstraction**" layer (HWA) which communicates with the robot's hardware (or simulation) and organizes the gathered data in a coherent manner. The `hardware_interface` package provides "**Resources**" (e.g. joints) that can be accessed by controllers[3]. Similar resources are regrouped in "**Hardware Interfaces**" (e.g. `VelocityJointInterface` on Fig. 9.1). Hence the definition of a "**Robot**" in the ROS Control framework, as a set of hardware interfaces.

The "**Controller Manager**" is responsible for managing the *lifecycle*[4] of controllers, and hardware resources through the interfaces and handling resource conflicts between controllers. The lifecycle of controllers is not static. It can be queried and modified at runtime through standard ROS services provided by the `controller_manager` package. Such services allow to start, stop and configure controllers at runtime.

[1] It can be used on whatever robot platform, not just on a specific robot.

[2] **pr2_controller_manager** was developed in 2009 by Willow Garage. As indicated by its name, it is specific to the PR2 robot.

[3] The controllers don't talk directly to the robot hardware and require resources from the HWA. This prevent from directly communicating with the robot memory and therefore having to deal with timing and conflict issues.

[4] The *lifecycle* of a controller here refers to the process of initializing, starting, running, and stopping a controller.

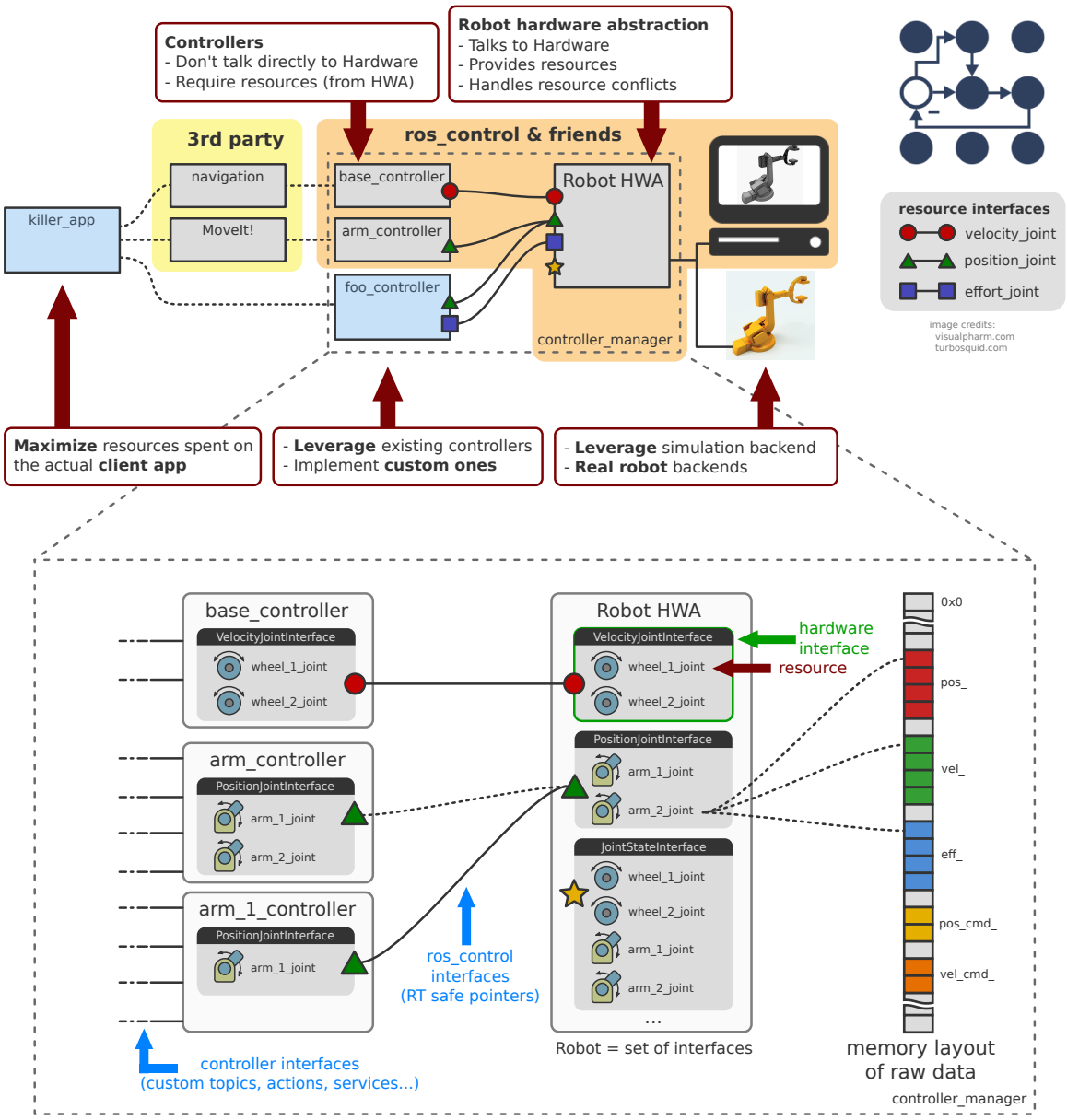


Figure 9.1: ROS control main concepts. The **3rd party** block refers to the non real-time (or soft real-time) user application in the ROS space (e.g. navigation, SLAM, path planning and so on). These 3rd party apps are connected to ROS control via a set of **controller interfaces** which can for instance be custom ROS topics, services or actions (c.f. 3). Note that there can be multiple controllers trying to access the same hardware interface (e.g. `arm_controller` and `arm_1_controller` in Fig. 9.1). In this case, `ros_control` automatically handles conflicts using an **exclusive ownership** policy: only one controller is effectively running and accessing resources, but not several at the same time. The **memory layout of the raw data** depicts the memory regions where you read/write to inform hardware. A resource is nothing less than a pointer to the raw data that add some semantics (i.e. identify data as position, velocity and so on...).

9.2 Getting into ROS Control

9.2.1 Robot hardware abstraction layer

The robot hardware abstraction layer is provided by the hardware_interface::RobotHW class. Any specific robot implementation have to inherit from this class. Instances of this class model hardware resources provided by the robot such as electric and hydraulic actuators and low-level sensors such as encoders and force/torque sensors. It also allows for integrating heterogeneous hardware or swapping out components transparently whether it is a real or simulated robot. There is a possibility for composing already implemented `RobotHW` instances which is ideal for constructing control systems for robots where parts come from different suppliers, each supplying their own specific `RobotHW` instance. The rest of the `hardware_interface` package defines read-only or read-write typed joint and actuator interfaces for abstracting hardware away, e.g. state, position, velocity and effort interfaces.

```
class MyRobot: public hardware_interface::RobotHW
{
  public:
    MyRobot(); // Setup Robot

    // Talk to HW:
    void read(); // Acquire state (using e.g. ROS industrial or custom)
    void write(); // Send commands via Ethernet, Ethercat, CAN bus...

    // ONLY if exclusive ressource ownership is undesired:
    virtual bool checkForConflict(...) const;
};
```

- hardware_interface::RobotHW is the robot hardware abstraction (i.e. the software representation of the considered robot)

In short, the hardware_interface package:

- Provides hardware abstraction (i.e. allows to access real robot hardware data without caring about conflicts or memory low level stuffs). Information is organized in:
  - **Resources:** actuators, joints, sensors
  - **Interfaces:** set of similar resources
  - **Robots:** set of interfaces
- Handles resource conflicts automatically (**exclusive ownership** by default)
- Defines **read-only** or **read-write** typed joint and actuator interfaces:
  - **Read-only:**
    - Joint state
    - IMU
    - Force-torque sensor
  - **Read-write:**
    - Position joint
    - Velocity joint
    - Effort joint

9.2.2 Controller manager

The *ros_control* packages takes as input the joint state data from your robot's actuator's encoders and an input set point. It uses a generic control loop feedback mechanism, typically a PID controller, to control the output, typically effort, sent to your actuators. *ros_control* gets more complicated for physical mechanisms that do not have one-to-one mappings of joint positions, efforts, etc but these scenarios are accounted for using transmissions.

The `ros_controllers` git repository (`https://github.com/ros-controls/ros_controls`) provides collection of ready-to-use tools, messages and action definitions that are useful in the context of control and also **real-time ready**:

- `control_msgs` (message containers useful for controller implementation)
- `realtime_tools` (contains a set of tools that can be used from a hard realtime thread, without breaking the realtime behavior.)
- `control_toolbox` (contains several C++ classes useful in writing controllers)
- `ros_control` (core framework)
- `ros_controllers` (robot agnostic controllers compatible with
- `ros2_control` (proof of concept for ROS2 integration)
- `urdf_geometry_parser` (extract geometry value of a vehicle from urdf)
- `gazebo_ros_control` (integrates the `ros_control` controller architecture with the `Gazebo` simulator)

Excellent Slides: `https://roscon.ros.org/2014/wp-content/uploads/2014/07/ros_control_an_overview.pdf` Corresponding video: `https://vimeo.com/107507546`

Chapter 10

Control a Simulated Robot using ROS Control and Gazebo

TODO

Appendix A

ROS in a Nutshell

A.1 ROS Node Communication

A.1.1 Debugging topics in command line using rostopic

- `rostopic list`: Displays the list of topics on the local ROS network.
- `rostopic pub`: Publishes data on a topic. For example: `rostopic pub -r 10 /topic_name std_msgs/String hello` publishes "hello" at a frequency of 10 Hz in the topic `/topic_name`.
- `rostopic echo /topic_name`: Displays the messages of the topic `/topic_name`.
- `rostopic info /topic_name`: print information about active topic
- `rostopic bw /topic_name`: Displays the bandwidth of a topic
- `rostopic hz /topic_name`: Displays the update frequency of a topic.
- `rostopic type /topic_name`: Displays the type of a topic (message).
- `rostopic find msg-type`: Displays all the topics of a given type.
- Use auto-complete !

A.1.2 Debugging services in command line using rosservice

- `rosservice list`: Displays the list of active services on the local ROS network.
- `rosservice call /service_name arg1 ... argN`: Call the service with the provided args. For example: `rosservice call /mu_serviceServer 3 3.141592654` will (using the last example) return a 3×3 matrix structure whose elements are equal to π.
- `rosservice args /service_name`: Displays the service arguments.
- `rosservice info /service_name`: print information about service
- `rosservice uri /service_name`: Print service ROSRPC uri.
- `rosservice type /service_name`: Displays the type of a topic (message).
- `rosservice find msg-type`: Displays all the services of a given type.
- Use auto-complete !