A LIGHTWEIGHT OPEN-SOURCE COMMUNICATION FRAMEWORK FOR NATIVE INTEGRATION OF RESOURCE-CONSTRAINED ROBOTICS DEVICES WITH ROS

AI & R Lab
Laboratorio di Intelligenza Artificiale e Robotica del Politecnico di Milano

Relatore: Prof. Andrea BONARINI
Correlatore: Ing. Martino MIGLIAVACCA

Tesi di Laurea di:
Andrea ZOPPI, matricola 765662

Anno Accademico 2011-2012
A chi mi supporta e mi sopporta
# Contents

Abstract XI
Sommario XIII
Ringraziamenti XV

1 Introduction 1

2 State of the art 7

2.1 The need for rapid-prototyping frameworks 7
2.2 Robot Operating System 8
  2.2.1 Architecture 8
  2.2.2 Development process 10
2.3 Open Robot Control Software 10
  2.3.1 Architecture 11
  2.3.2 Development process 12
2.4 Lightweight Communications and Marshaling 12
  2.4.1 Architecture 13
  2.4.2 Development process 14
2.5 Rapid Robot Prototyping 14
  2.5.1 Architecture 15
  2.5.2 Development process 16
2.6 Framework comparison and observations 16

3 Research strategy 19

3.1 Objectives 19
3.2 Strategy 21

4 The ROS communications model 25

4.1 Definitions 25
4.2 Overview 28
4.3 Remote procedure calls 29
  4.3.1 Roles 30
4.3.2 Value types ........................................... 30
4.3.3 Request and response ................................. 30
4.4 Connection patterns .................................... 32
4.5 Data streams ........................................... 33
  4.5.1 Type descriptors .................................. 34
  4.5.2 Message structure ................................ 37
  4.5.3 Connection header ................................ 38

5 The proposed framework — µROSnode 41
  5.1 Implementation choices ............................... 41
    5.1.1 Language observations ........................... 42
    5.1.2 Footprint optimizations ........................... 42
  5.2 Framework description ............................... 43
    5.2.1 Base module ..................................... 44
    5.2.2 Connectivity module .............................. 46
    5.2.3 XML-RPC module ................................ 47
    5.2.4 TCPROS module ................................... 48
    5.2.5 Node module ..................................... 49
    5.2.6 Threading module ................................. 50
    5.2.7 User module ..................................... 50
    5.2.8 Message types module ............................ 51
    5.2.9 Handlers module .................................. 51
  5.3 Development tools .................................... 51
    5.3.1 Code generator ................................... 51
    5.3.2 Static stack analysis ............................. 62
    5.3.3 Demonstration projects ............................ 64
  5.4 Integration with user applications .................. 65

6 Experimental results and evaluation 67
  6.1 Benchmarks ........................................... 67
    6.1.1 Communication setup ............................. 68
    6.1.2 R2P_GW module .................................. 69
    6.1.3 Raspberry Pi ..................................... 77
  6.2 Case study — Triskar2 ............................... 83
    6.2.1 Triskar2 ......................................... 83
    6.2.2 Software architecture ......................... 85
    6.2.3 Observations ..................................... 87

7 Conclusions and future research directions 89

Bibliography 91
A ROS-related API documentation

A.1 ROS Master API .................................. 95
   A.1.1 registerService() ....................... 95
   A.1.2 unregisterService() .................... 96
   A.1.3 registerSubscriber() ................... 96
   A.1.4 unregisterSubscriber() ............... 96
   A.1.5 registerPublisher() ..................... 97
   A.1.6 unregisterPublisher() ................. 97
   A.1.7 lookupNode() ............................ 98
   A.1.8 getPublishedTopics() ................. 98
   A.1.9 getTopicTypes() ......................... 98
   A.1.10 getSystemState() ...................... 99
   A.1.11 getUri() ................................ 99
   A.1.12 lookupService() ....................... 99

A.2 ROS Parameter Server API ....................... 100
   A.2.1 deleteParam() ......................... 100
   A.2.2 setParam() .............................. 100
   A.2.3 getParam() .............................. 101
   A.2.4 searchParam() .......................... 101
   A.2.5 subscribeParam() ...................... 102
   A.2.6 unsubscribeParam() .................... 102
   A.2.7 hasParam() .............................. 102
   A.2.8 getParamNames() ....................... 103

A.3 ROS Slave API .................................. 103
   A.3.1 getBusStats() ......................... 103
   A.3.2 getBusInfo() ............................ 103
   A.3.3 getMasterUri() ......................... 104
   A.3.4 shutdown() .............................. 104
   A.3.5 getPid() ................................ 105
   A.3.6 getSubscriptions() ..................... 105
   A.3.7 getPublications() ...................... 105
   A.3.8 paramUpdate() ......................... 105
   A.3.9 publisherUpdate() ..................... 106
   A.3.10 requestTopic() ....................... 106

A.4 Node API ...................................... 107
   A.4.1 urosNodePublishTopic() ............... 107
   A.4.2 urosNodeUnpublishTopic() ............. 107
   A.4.3 urosNodeSubscribeTopic() ............. 108
   A.4.4 urosNodeUnsubscribeTopic() .......... 108
   A.4.5 urosNodePublishService() ............ 109
   A.4.6 urosNodeUnpublishService() .......... 109
# List of Figures

2.1 ROS logo .............................................. 8
2.2 Conceptual topic flow ................................. 9
2.3 Orocos framework .................................... 11
2.4 Orocos component interface ......................... 12
2.5 lcm-spy screenshot ................................... 14
2.6 TiltOne R2P-based robot ............................. 15
3.1 Legacy ROS versus µROSnode robot designs ........ 22
4.1 Sequence diagram of a `getPid()` XML-RPC call example .... 31
4.2 Topic connection summary ............................. 33
4.3 Service call summary .................................. 34
4.4 Message dump .......................................... 38
4.5 Service call dump ...................................... 38
4.6 Service response dump ................................ 39
4.7 Service error response dump ......................... 39
4.8 Connection header request ........................... 40
4.9 Connection header response ......................... 40
5.1 µROSnode logo ......................................... 42
5.2 Logical view of µROSnode modules .................. 44
5.3 Code generator flowchart .............................. 52
5.4 Static stack analysis flowchart ....................... 63
6.1 R2P_GW prototype board .............................. 69
6.2 Turtlesim R2P_GW firmware footprint .................. 71
6.3 Transmission performance on R2P_GW, on-the-fly .......... 74
6.4 Reception performance on R2P_GW, buffered messages .... 75
6.5 Reception performance on R2P_GW, on-the-fly .......... 76
6.6 Raspberry Pi model B ................................... 78
6.7 Transmission performance on RasPi, on-the-fly .......... 80
6.8 Reception performance on RasPi, buffered messages .... 81
6.9 Reception performance on RasPi, on-the-fly .......... 82
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.10</td>
<td>The Triskar2 robot</td>
<td>84</td>
</tr>
<tr>
<td>6.11</td>
<td>Triskar2 kinematics diagram</td>
<td>84</td>
</tr>
<tr>
<td>6.12</td>
<td>Teleoperator devices and connections</td>
<td>85</td>
</tr>
<tr>
<td>6.13</td>
<td>Triskar2 topics graph</td>
<td>87</td>
</tr>
<tr>
<td>6.14</td>
<td>Triskar2 teleoperator screenshot</td>
<td>88</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Framework feature comparison ............................... 17
4.1 ROS to C type mapping .................................. 37
4.2 Connection header fields ................................. 40
6.1 Turtlesim R2P_GW firmware footprint .................. 70
6.2 Turtlesim stack usages on R2P_GW ...................... 72
6.3 Teleoperator user input bindings ......................... 87
# List of Listings

2.1 LCM message types for path planning .......................... 13

4.1 getPid() call .................................................. 32
4.2 getPid() response .............................................. 32
4.3 Topic message type ............................................ 35
4.4 Clean expanded topic message type ............................ 35
4.5 ROS service descriptor ......................................... 36
4.6 ROS service descriptor ......................................... 36

5.1 Message record .................................................. 53
5.2 Message length computation .................................. 53
5.3 Message initialization ........................................... 54
5.4 Message cleanup ................................................ 54
5.5 Message reception and unmarshaling ......................... 55
5.6 Message marshaling and transmission ......................... 55
5.7 Service records ................................................. 57
5.8 Service message length computations ....................... 57
5.9 Service message initializations ............................... 58
5.10 Service message cleanups .................................... 58
5.11 Service request message reception and unmarshaling ..... 59
5.12 Service response message marshaling and transmission ... 59
5.13 Type registrations .............................................. 59
5.14 Topic publisher handler ...................................... 60
5.15 Topic subscriber handler ..................................... 60
5.16 Service publisher handler ................................... 61
5.17 Service call handler ........................................... 61
5.18 Turtlesim configuration for urosgen.py .................... 65

B.1 Disclaimer ...................................................... 113
B.2 XML-RPC DTD .................................................. 114
B.3 XML-RPC XML Schema ....................................... 114
B.4 TCPROS EBNF syntax ......................................... 116
B.5 triskar/Velcoity message type ................................. 116
Abstract

The ongoing evolution of computing devices, especially of embedded systems, introduces new opportunities to enhance robotics software. The goal of this thesis is to develop a rapid prototyping robotics communication framework which can run on cheap embedded systems, while still having native compatibility with an existing widespread high-level framework, typically run on powerful computers.

We created $\mu$ROSnode, an open source communication framework with small code and memory footprints, able to run on recent advanced microcontrollers, and having native compatibility with the popular ROS robotics framework, which follows the publish/subscribe approach. The developer, often accustomed to ROS nowadays, can still write high-level software for it, with the non-trivial capability to interface directly to hardware modules running $\mu$ROSnode.

The framework is written in ANSI C89, supported by almost all of the compilers targeting embedded systems, and which can be easily integrated with other common programming languages. The codebase is partitioned into purpose-specific modules and extensively documented, to improve ease of use and maintenance. In order to assist the user in the development of $\mu$ROSnode-based software, we developed a code generator and a static stack analysis tool. We performed some simple benchmarks to determine the maximum transmission and reception throughput of a topic with variable-sized messages, running on a Raspberry Pi and a STM32F407 microcontroller (ARM-CM4 core). On both platforms it was possible to reach several thousands of messages per second without too much effort if messages are rather small or processed on-the-fly.

We also measured the usage of both program and working memories by a ROS turtlesim clone, which easily fitted tight resource constraints of the STM32F407. An existing robot, whose hardware/software architecture was built around a CAN bus, was interfaced to ROS through an inexpensive CAN-to-Ethernet gateway running $\mu$ROSnode on a STM32F407.
La continua evoluzione dei dispositivi di calcolo, specialmente dei sistemi embedded, introduce nuove opportunità per migliorare il software per la robotica. Lo scopo della tesi è di sviluppare un framework di comunicazione robotica per la prototipazione rapida, che possa funzionare su sistemi embedded economici, mantenendo compatibilità nativa con uno dei diffusi framework esistenti di alto livello, che tipicamente funzionano su computer avanzati.

Abbiamo creato $\mu$ROSnode, un framework di comunicazione open source con piccola occupazione in memoria, che possa funzionare sugli avanzati microcontrollori recenti e con compatibilità nativa con il popolare framework ROS, che segue l’approccio publish/subscribe. Lo sviluppatore, oggigiorno spesso abituato al sistema ROS, può continuare a sviluppare software per esso, con la non banale possibilità di interfacciarsi direttamente a moduli hardware funzionanti tramite $\mu$ROSnode.

Il framework è scritto in ANSI C89, supportato pressoché dalla totalità dei compilatori per sistemi embedded, il quale può essere facilmente integrato da altri linguaggi di programmazione. Il codice è suddiviso in moduli per compiti specifici e largamente documentato, per migliorare facilità d’uso e manutenzione. Al fine di assistere l’utente nello sviluppo di software basato su $\mu$ROSnode, abbiamo sviluppato un generatore di codice e uno strumento per l’analisi statica degli stack.

Abbiamo compiuto dei semplici benchmark per determinare la massima capacità di trasmissione e ricezione di un singolo topic con messaggi a dimensione variabile, su un Raspberry Pi e un microcontrollore STM32F407 (core ARM-CM4). Per entrambe le piattaforme è stato possibile raggiungere capacità di diverse migliaia di messaggi al secondo senza particolari sforzi in caso di messaggi piccoli o processati al volo.

Abbiamo anche misurato l’occupazione nelle memorie programma e centrale di un clone del turtlesim di ROS, che ha rispettato agevolmente i severi limiti dell’STM32F407.

Un robot esistente, la cui architettura hardware/software fu costruita intorno a un bus CAN, è stato interfacciato a ROS tramite un gateway CAN-a-Ethernet in cui opera $\mu$ROSnode su un STM32F407.
Ringraziamenti

«Dimmi con chi vai e ti dirò chi sei,
   e se so di che cosa ti occupi
   saprò che cosa puoi diventare»
   Johann Wolfgang von Goethe

Questo elaborato di tesi rappresenta solo il culmine della mia carriera scolastica, mi è doveroso ringraziare chi mi ha permesso di poterla portare a termine con successo e dedizione, in particolare in questo percorso magistrale.

Vorrei ringraziare innanzitutto il Professor Bonarini, per avermi permesso di sviluppare il lavoro di tesi in un ambito che mi piace parecchio, ossia quello dei sistemi embedded applicati alla robotica.

A mio favore anche l’eccellente supporto di Martino, grazie al quale ho finalmente sviluppato una piccola piattaforma embedded, dal design della scheda allo sviluppo di un moderno firmware, passando per una miriade di trucchi e aneddoti.

Un ringraziamento anche ai professori del Politecnico di Milano per la nobile missione a cui si dedicano.

La compagnia di studio è stata determinante per confrontarsi, comprendere ciò che sfugge, trovare nuove soluzioni, superare le comuni frustrazioni e compiacersi dei risultati ottenuti.

Un plauso a Sebastiano, compagno inseparabile in tutto ciò, a cui devo parte del mio merito; a Stefano per l’approccio serio e razionale; a Daniele e Andrea, con cui il sorriso e l’entusiasmo non sono mai mancati; a tutti gli altri compagni e compagne con cui ho condiviso fatiche e gioie in questi anni.

Ma gli omaggi non riguardano solo l’ambiente accademico. Ci sono parecchi amici che mi hanno supportato, tra cui i temerari compaesani compagni di avventure: Beppe, Tizio, Nik, Leske, Mile, Marco, Glauco, Mitch, Davide, Lukus, Andre, Gino, Cangio, Nagia, eccetera eccetera, così come Eddy, Gabriele e altri amici sparsi per il Mondo.

Grazie anche ai colleghi di Digital Jockey, con cui condivido la passione per
l’audio e le tecnologie digitali: Bazzooka, Cioce, Spazialex, Attica, Rudy, Maxmora, Animal, Nuwanda, Cybermix, Alessio, Luca, Hackid, MaxB, Gigi, Franco, per citare coloro che in maggior misura hanno contribuito in qualche modo anche alla mia preparazione ingegneristica.

Naturalmente anche alle signorine devo molto, specialmente a: Elenina, Giuly, Ginny, Sandry, Very, Ila, Ele, Vane, Sarah, Lucia. È grazie a donne come voi che posso dimostrare e migliorare il mio essere Uomo, e non solo "ingegnere". La vostra presenza è indispensabile!

Un ringraziamento profondo e dovuto va alla mia famiglia. Senza il costante supporto dei miei genitori non avrei certamente potuto dedicarmi a questa impresa, che è un punto di arrivo nella mia carriera scolastica quanto un potenziale punto di inizio per un’interessante carriera lavorativa. Una riconoscenza anche a Vale, quella bella, che da brava sorella mi dispensa ottimi consigli e so che ci tiene molto.

Ringrazio anche i miei nonni che, avendo vissuto in uno dei secoli più rivoluzionari, col loro immenso bagaglio di esperienze e di sacrifici hanno spianato la strada su cui dovrò "ingegnerizzare" i miei. Vi voglio bene.
Chapter 1

Introduction

Robotics research brings to a high degree of innovation. Many new ideas for future robotics products are conceived every year, targeting a broad range of fields, and requiring multidisciplinary skills. Therefore, researchers need to build and evaluate new ideas quickly, without spending too much time in the development of every subsystem composing the whole robot.

Robots are composed by parts which often belong to a standard set of mechanical, electrical, and software components. While mechatronics parts have been available for a long time, thus providing well established interfaces, the ongoing evolution of computing devices, especially of embedded systems, introduces new opportunities to enhance robotics software.

Several software frameworks were proposed to build robotics communications systems, artificial intelligence, motion control, and so on, assisting the researcher in the rapid development of robotics software.

The goal of this thesis is to develop a rapid prototyping robotics communication framework which can run on cheap embedded systems, while still having native compatibility with an existing widespread high-level framework, typically run on powerful computers. This would allow the development of low-cost hardware modules able to communicate with one of the state-of-the-art, high-level frameworks, to enhance the rapid-prototyping development cycle on the low-level side.

We created $\mu$ROSnode, an open source communication framework with small code and memory footprints, able to run on the latest microcontrollers, and having native compatibility with the popular ROS robotics framework. The framework is written in ANSI C89, supported by almost all of the compilers targeting embedded systems, and which can be easily integrated with other common programming languages. The code is partitioned into purpose-specific modules and extensively documented, to improve ease of use and maintenance.
The choice of a BSD license makes it free software, modifiable by anyone and employable for commercial products. A code generator and a static stack analysis tool assist in the development of the user application, so that repetitive coding is lightened, and the memory usage on resource-constrained platforms is kept under control. A set of benchmarks prove its good performance on low-cost embedded systems, and a demonstration project on a real-life robot was easy to develop.

We first analyzed the requirements for rapid prototyping in the robotics field. The most intuitive way is to assemble some ready-to-use parts together, and apply a high-level behavior to them. While mechatronics parts often share a roughly common interface, robotics software is still under heavy evolution. However, there are some widely accepted software frameworks which help in the rapid development of the robotics software.

ROS [22] is the most popular framework nowadays, providing a huge repository of pre-made software packages, support for both high-performance (C++) and high-level/scripting (Python, Lisp) languages; communication over XML-RPC, TCP (TCPROS), and UDP (UDPROS); an automated compilation toolchain by a strict filesystem structure; compatibility wrappers for many other robotics frameworks.

Orocos [4] follows strict software engineering rules and development cycle, and targets realtime robots; its runtime architecture is CORBA-based.

LCM [12] is a lightweight high-performance system for low-latency communications among processes over UDP.

R2P [2] is a hybrid hardware/software framework to build the robot electronics out of small and cheap hardware modules, which communicate through a robust CAN bus, and has support for hard-realtime applications.

The state-of-the-art offers frameworks which are either very good at the software, communication, or hardware level, but there is no framework which helps in blending of all of them together. Moreover, high-level robotics software is usually run on fairly powerful computers, while the hardware is driven by industrial processors or microcontrollers, which cannot manage a huge computation load.

That being so, we outlined our research strategy to develop a framework which can be used to interface high-level robotics software with the hardware. The framework should have small footprint and simple architecture, so that it can fit into a microcontroller of nowadays. Since most embedded systems do not have a compiler toolchain support as advanced as general-purpose computers or workstations do, the choice for the programming language is limited. There is mature support only for C and C++ targeting embedded system architectures, while other languages are experimental or compatible with only a few platforms.
For maximum compatibility and performance we chose ANSI C (a.k.a ISO C89). Compared to C++ it has some restrictions, but compiler support is still more mature; all of the C89 features are available, while some C++ features (e.g., RTTI and exceptions) must be disabled on resource-constrained platforms. However, it is still possible to write wrappers for C++, and even for other higher level languages (Python, Java, C#, and so on).

Since ROS is the most widely accepted framework for the development of high-level robotics applications, we chose to develop a lightweight communications framework, µROSnode, which has native support for ROS protocols, and can be run even on cheap embedded systems based on recent microcontrollers. The user, often a robotics researcher accustomed to ROS nowadays, can still write software for it, with the non-trivial capability to interface directly to hardware modules connected via the IP network protocol (e.g., via Ethernet).

In order to assist the user in the development of µROSnode-based software, we also developed a code generator and a static stack analysis tool. The code generator creates the desired topic and service handler function stubs, as well as (un)marshaling functions for the involved data types. The stack analyzer is useful to keep the stack usage under control when developing applications for memory-constrained platforms, for instance to size the stacks assigned to topic and service handler thread pools.

We then performed quantitative and qualitative analyses of our work, to prove its effectiveness. The target platforms were an Ethernet board driven by a STM32F407 microcontroller (ARM-CM4 core), which is tightly resource constrained, and a Raspberry Pi model B (ARM-11 core), which is a popular extremely-low-cost embedded computer.

For the quantitative analysis we performed some simple benchmarks to determine the maximum transmission and reception throughput of a variable-sized topic. Both the platforms can reach several thousands of messages per second without too much effort if messages are rather small (less than 500 B, common for simple sensors) or can be processed on-the-fly. When buffering large messages (more than 1 kB) the overall performance decreased dramatically, not because of our framework itself, but because of expected limitations of both the target processors and the 100 Mb/s Ethernet links.

On the first board we also measured the usage of both program and working memory by a ROS turtlesim clone. The results proved that a fully-featured µROSnode implementation has a rather low impact on the program memory, in the order of less than 40 kB when compiled with optimizations; XML-RPC and TCPROS thread pools require less than 1300 B for their stacks in the worst case, which is reasonable considering the complexity of such protocols and their management.
The qualitative analysis consisted in the development of a µROSnode-based firmware for a real-life robot, the omnidirectional three-wheeler Triskar2, which is controlled through a CAN-to-Ethernet gateway communicating with some R2P boards. Thanks to the code generator, the well-documented API, and the simple software architecture, the development did not require much effort, indeed the whole firmware was written and tested in less than a week by a single person.

As proven by experimental results, µROSnode fits the missing link between high- and low-level robotics software frameworks. It is lightweight enough to run elementary tasks on modern microcontrollers with good performance and low memory requirements. Its development cycle is eased by the simple software architecture, assistance by development tools, and extensive API documentation. Native compatibility with ROS ensures its effectiveness when building robot prototypes out of fairly inexpensive robotics components running µROSnode.

Future work may include C++ support, by writing wrappers around the C implementation, or as a stand-alone C++ version. UDPROS may be added for low-latency applications and wireless connection efficiency. Existing products may be made ROS-ready by building proprietary-to-ROS hardware adapter running µROSnode, or by writing a custom µROSnode-based firmware if a product already features Ethernet or Wi-Fi. Being employable by inexpensive embedded systems, µROSnode can be used for low-cost educational or recreational products connected to ROS.

The structure of this thesis is explained in the following.

Chapter 2 presents the state-of-the-art robotics frameworks, in terms of software, communication, and hardware features. Through their comparison, we identified a missing link between high- and low-level frameworks.

Chapter 3 describes the objectives and the strategy of the research, in order to develop a communication framework capable of running on cheap embedded systems, with native compatibility to an existing high-level framework, ROS. This will fill the gap between the high-level, number crunching software frameworks, and the cheap hardware platforms.

Chapter 4 illustrates the ROS communication features, adopted by our work to maintain native compatibility with ROS itself.

Chapter 5 shows the detailed architecture of our communication framework, µROSnode, the code generator tool, and the static stack analysis tool.

Chapter 6 contains experimental results, which prove the effectiveness of our framework both in performance (quantitative analysis) through some network benchmarks, and in ease of development (qualitative analysis) by controlling a
real-life robot through a ROS-based application.

Chapter 7 draws conclusions about our work, and outlines some possible future research directions to undertake.

Appendix A lists the API to manage topics, services, and ROS parameters. Appendix B contains some useful listings, like grammar definitions and demonstration scripts.
Chapter 2

State of the art

In the following, the state-of-the-art works related to robotics, communication, and rapid prototyping frameworks are described, highlighting their objectives, architecture, and typical usage scenarios. By analyzing their advantages and disadvantages, we will show that there is still a gap to fill between high-level frameworks which run on powerful computers, and those which are mainly focused on the hardware or the communication sides. These observations will lead to our proposal, illustrated in the next chapters.

2.1 The need for rapid-prototyping frameworks

Robot development is a truly multidisciplinary task, involving mainly mechanical structures, electronics design, software engineering, and artificial intelligence. Each of these disciplines needs its own environment to build a complete robot, requiring time, planning, hardware, and operation. On the contrary, a robotics researcher should build new ideas in a fast, cheap, and easy way. So, it is needed to provide friendly environments, which reduce the resource requirements to build a new idea, not necessarily in its optimized final form, but at least as a meaningful approximation. Also, most parts of a robot, either physical or virtual, share a common design, so a robot prototype could be built without redesigning everything from scratch, but rather by assembling pre-made parts. Indeed, by partitioning the development into purpose-specific tasks, the design, operation, and maintenance of the whole system are much easier than by reinventing the wheel every time; this is a well-known strategy in any fields. Pre-made parts share a well defined interface, and they were already tested by the manufacturer, who can keep the price lower thanks to mass-production.
Throughout our work we will focus mainly on software environments for rapid robot development. Such environments come in the form of frameworks, which are sets of tools and libraries with which the user build complex software out of some small and purpose-specific pieces. For example, to control a robot it is needed to gather data from sensors, reason about their values, and generate commands for the actuators.

An intuitive way is to have a piece of software for each of the sensors, connected to a virtual brain (or even multiple ones), which is then connected to the software components of every actuator. Such a framework typically provides the underlying communication subsystem, to which user modules can communicate through a well-defined API.

The most widespread frameworks also supply some modules commonly used, like those for image elaboration, robotic arm operation, odometry, and so on, as well as some tools to ease development and debugging.

### 2.2 Robot Operating System

The Robot Operating System (ROS [22], logo in **Figure 2.1**) is a meta operating system targeted at the rapid development of robotic software, developed by Willow Garage. It has become a very widespread framework, probably the most active and adopted one in the research field, supplying a large number of pre-made packages. Its source code is released under a revised BSD license, so that it can be freely modified and employed by hobbyists and institutions, but also by commercial companies; by the way, some packages are released under more restrictive licenses, like the common GPL. It natively runs on Ubuntu Linux, but has some degree of support for other Unix-like operating systems. ROS client (core) libraries are written primarily in C++, Python, and Lisp, with experimental support for Java, Lua, and Mono/.NET.

#### 2.2.1 Architecture

The ROS framework is highly modular. Software developed with ROS is composed by a set of Nodes, which are virtual processing elements providing very specific interface and operation. For example, there exist Nodes for
the elaboration of point clouds, those to extract geometrical primitives out of stereo images, others for the path planning, artificial intelligence, motion control, interfaces to common peripherals (e.g., joysticks, Microsoft Kinect, laser scanners, servomotors), and so on.

Nodes by themselves have little purpose, but by connecting them together it is possible to build a very complex robotics application. Nodes communicate through a publish/subscribe technique. Nodes producing some kind of data publish it by a topic of specific type, and those Nodes which need such data receive it by subscribing to that topic; this concept is depicted in Figure 2.2.

ROS supports continuous data streaming through topics, but also supports request/response calls to global services, and shared environment parameters. The setup and control of the Node network (called computation graph) are made through a XML-RPC-based protocol, which ensures maximum portability among any platforms with TCP/IP capabilities; there is a centralized Node, the Master, which supervises the whole network. Instead, topic and service data streams employ a custom low-level protocol, which is typically TCP-based, but also UDP-based for latency-sensitive applications. In-depth information about the ROS communication system can be found in Chapter 4.

Software developed with ROS can be grouped into stacks and packages. A ROS package is a container of software which provides a specific functionality to be reused with ease. For instance, there exist packages for a particular sensor, or for a specific recognition algorithm. A package may contain ROS Nodes, ROS-independent libraries, data sets, configuration files, or anything else that logically constitutes an useful module. A stack is a container of packages which share a common goal; for example, computer vision or motion planning. Stacks can have dependencies on other stacks, in order to be maintained independently. Some rules define the filesystem structure of packages and stacks, so that they can be built through an automated toolchain, based on
the CMake build system [5], keeping the code well organized and maintainable by different people.

### 2.2.2 Development process

Developing robotics software with ROS is straightforward. The user has to know how to navigate inside the ROS meta-filesystem and workspaces (folders which include package source code), how to create a Node, and how to make it communicate with the rest of the computation graph. This knowledge base is well documented inside the ROS Wiki [25], and the set of automated commands of the meta-OS shortens the creation of a bare-bone application.

Then, the user has to outline the software capabilities of the robot. Most of the times, there already exist some ROS packages which provide them, so it is often a matter to just pick the right ones from the official repositories. If some features are missing, specific user Nodes can be written with ease, thanks to the good support for common high level languages; typically, C++ is used for high performance, Python for its simplicity, and Lisp for artificial intelligence.

After writing the user application out of Nodes, the workspace has to be compiled into its binary form, by using the automated package compilation flow defined by ROS. Nodes will then need to be instantiated and connected together through launch files.

If needed, the developer can capture the data being exchanged inside the computation graph thanks to a set of debugging tools. They can inspect the current state of the network, or record streams for further analysis, elaboration, and replay.

### 2.3 Open Robot Control Software

*Open Robot Control Software (Orocos, [4])* is a software framework for the control of realtime robots, mainly the industrial ones. It is written in C++ under the GNU Lesser GPL. Its aim is to provide a highly modular, high-performance, easy-to-use framework for complex robot control and machine learning. There exist some offspring projects for non-realtime control based on Orocos.
2.3.1 Architecture

The framework is split into the following libraries, grouped by Figure 2.3: the Orocos Real-Time Toolkit (RTT), the Orocos Components Library (OCL), the Orocos Kinematics and Dynamics Library (KDL), and the Orocos Bayesian Filtering Library (BFL). The RTT provides the infrastructure and the functionalites to build robotics applications in C++. The emphasis is on realtime, on-line interactive and component-based applications. OCL provides some ready to use control Components. Both Component management and Components for control and hardware access are available. The KDL is a C++ library which allows to calculate kinematic chains in realtime. BFL provides an application independent framework for inference in dynamic bayesian networks, i.e. recursive information processing and estimation algorithms based on Bayes’ rule, such as (extended) Kalman Filters, particle filters (sequential Monte methods), and so on.

Being a modular framework, an Orocos application is made of networked purpose-specific components. A CORBA-based communications system is used for the setup and control of the network, so that applications follow an effective software engineering approach. There are five distinct (and optional) ways in which an Orocos component can be interfaced, as seen in Figure 2.4: through its properties (parameters modifiable at run-time), events (functions executed when a change occurs), methods (immediate-result functions), commands (procedures to reach a goal) and data flow ports (thread-safe un-/buffered data transport channels). Besides defining the above component communication mechanisms, Orocos allows the Component or Application Builder to write hierarchical state machines which use these primitives; this is the Orocos way of defining application specific logic. State machines can be (un)loaded at run-time in any component.
2.3.2 Development process

The Orocos community already provides pre-made components to be used to create the user application; otherwise, one has to write his own components. The application can be setup through XML properties or command/method interfaces.

2.4 Lightweight Communications and Marshaling

The Lightweight Communications and Marshaling framework (LCM, [12]), developed by the Massachusetts Institute of Technology, offers a very high performance communications system over UDP/IP, so that multiple processes can exchange data with high throughput and low latency. It is designed for very complex robotics software, which need optimized communications to satisfy soft-realtime constraints, in a modular fashion, and with the minimum requirements. The most prominent application was for the 2007 DARPA Urban Challenge [12], in which the MIT team governed an autonomous car by running 70 concurrent processes spread across 10 networked workstations, with a very heterogeneous network traffic (almost 100 data types involved). It is used also for aerial, underwater, and land robots.

A set of tools helps in the development of user applications, and thanks to the protocol design they have almost no impact on the network performance. LCM has support for C, C++, Java, Python, MATLAB, and C#, so that it can be natively used with many operating systems and for different purposes (high performance, prototyping, and simulation). It is released under the GNU
2.4.1 Architecture

LCM is a networked communications framework, in which a network Node is typically a single process of the operating system. This guarantees the isolation of memory address spaces for each Node, which helps in keeping the application safe and easy to maintain. Nodes communicate through a publish/subscribe approach, in that Nodes can publish or subscribe to topics. By not requiring centralized control, topics are stateless, reducing protocol and software complexities. LCM employs only UDP multicast messaging, so that a message is sent once for the whole network, scaling very well with the number of Nodes. UDP is a best-effort protocol directly based on IP, so that there is no flow control at all. A dropped packet is not resent, reducing the latency of successive packets, thus making soft-realtime applications possible. If flow control is required, it can be implemented on a higher network layer.

Messages of a topic must be of the same data type. The type descriptor is written in a simple C-like formal language, so that a tool can generate native source code for all of the languages supported by LCM. Each message is headed by the 64-bit hash code of its type descriptor, providing type checking for each message being received. Listing 2.1 displays an example of path planning message types, described with the custom C-like language defined by LCM.

A network debugger, lcm-spy of Figure 2.5, simply subscribes to all of the topics being exchanged across the network, and visualizes message contents (syntax is inferred by type descriptors). Thanks to the multicast protocol, the debugger is almost completely transparent to the network. It can also record the network traffic, which can be analyzed or elaborated for later replay.
2.4.2 Development process

To build an application for LCM, a developer must outline the needed topics and message types. Actual (un)marshaling source code is generated for the chosen programming language through the generator supplied with LCM. The application should then be partitioned into sub-processes, each providing a specific logical functionality. Then, processes are developed and run.

Since there is no centralized build toolchain, nor any particular rules for the source code organization, everything is up to the user’s appeal. LCM is just a communication framework, with no interest in providing pre-packaged modules, nor in defining software engineering rules.

2.5 Rapid Robot Prototyping

*Rapid Robot Prototyping (R2P, [2])* is a hardware/software framework for the robotics research. It has been designed to be low-cost, lightweight, easy to use, and with realtime capabilities. The idea is to shorten development times when researching for a new robot design, by employing small and cheap hardware modules. Sure enough, most of the time is often spent in the development of the mechanics, electronics and their firmware from scratch rather in the development of the actual robot application, even for simple designs [2]. By giving the researcher a set of hardware modules commonly found in robots, realtime communication and operating systems, and a set of development tools, one can make a working physical prototype in a shorter time. R2P is the most promising hybrid hardware/software framework for rapid robot prototyping.
Figure 2.6: TiltOne, a self-balancing robot controlled through R2P

2.5.1 Architecture

R2P has a hybrid hardware/software nature. It employs a publish/subscribe technology just like the other state-of-the-art frameworks, both for communications among physical nodes (boards) and virtual nodes (intra-process communications). The framework is written in C++, and currently supports the ChibiOS/RT RTOS [6].

A R2P hardware module is a tiny board which provides a basic functionality, for example DC motor control, 13-dof IMU (3 acc + 3 gyro + 3 mag + barometer + GPS), ultrasonic or generic Sharp distance sensors, an Ethernet gateway, and so on. Such boards are connected across a CAN bus through Ethernet patch cables, thanks to the high availability and low cost of both CAN controllers and Ethernet cables. The CAN bus allows the implementation of hard-realtime message loops, and is also widely used in harsh environments, such as in the automotive and industrial fields, being a well-established technology. Interconnection cables carry both the differential CAN signals and 5 V power rails for low-power (up to very few amperes) circuits; if high voltage or current are needed (e.g., for DC motors), then auxiliary power sources must be added to the design. Figure 2.6 shows the architecture of a self-balancing robot, TiltOne [20], which uses a R2P network to gather data from sensors and to actuate motors, connected to an external workstation for debugging and user control.

There are three kinds of topic scheduling: hard-realtime, soft-realtime, and non-realtime. Hard-realtime topics exploit the hardware priority packet arbitration defined by the CAN specifications. Their messages are arranged by a time-triggered scheduler, hosted by a centralized node (fairly elected among all of the nodes). A fixed time quantum of the scheduler is declared, and is split into time
slices, each assigned to at most one control loop publisher; this constitutes the (fixed) scheduling policy of hard-realtime control loop messages. Soft-realtime is satisfied by an *Earliest Deadline First* scheduler, applied to the free time slices of the hard-realtime scheduler; this is useful for sporadic or low-latency events with soft-realtime requirements. Any remaining free time slices can be used to exchange non-realtime topic messages.

It is possible to employ a packet capture tool which is completely transparent to the network, thanks to the inherent broadcast nature of the CAN bus. Again, a standard set of tools allows live recording and analysis of the exchanged messages, as well as record and replay features.

### 2.5.2 Development process

A robotic researcher may choose R2P when willing to build a new physical prototype idea. The researcher should first outline the hardware requirements of the structure being developed; for instance, the need for motors, distance sensors, a bridge to Wi-Fi, and so on, suggesting the appropriate pre-made R2P hardware module to be used. Otherwise, designing a custom board is rather simple, requiring only a CAN bus interface, commonly found in almost all of the latest microcontrollers.

After choosing or designing the right hardware modules, the user has to write their firmware application; the simple R2P API makes the writing of custom firmware rather easy. The R2P roadmap proposes some code generators, which will automatically generate a working application template out-of-the-box, and a high-level scripting language for those users who are less knowledgeable about firmware programming; this in turn would help in porting R2P applications to different embedded platforms.

### 2.6 Framework comparison and observations

All of the state-of-the-art frameworks described previously excel in some fields, but they also show lacks elsewhere.

ROS is the best choice when developing robotics research software, thanks to its immense library of ready-to-use algorithms and the automated development process. Being an highly active and successful project, bugs can be fixed quickly, and the huge user base can make the library grow even more. By contrast, the use of high-level languages and protocols make it unsuitable for simple and cheap robots, like those affordable by students or hobbyists, requiring a
moderately powerful computer to be run (no microcontrollers).

Orocos perfectly follows software engineering principles, which helps in making long-term, high-quality realtime robot control. As such, it is more suitable for industrial-grade robots rather than rapid prototyping of new ideas, also because of the relatively complex software architecture (if compared to ROS).

LCM is a very high efficiency communications framework, which can fit in any UDP-capable computer, from workstations to embedded systems. Although simplicity and performance are the key features of this project, it is merely a communications system, lacking any higher level features, such as ready-to-use robotic algorithms, automated toolchains, or even Quality of Service over UDP itself.

R2P is a promising hybrid hardware/software system for the rapid development of new robotic ideas, which is lightweight, realtime, and targeted at cheap embedded systems. Despite of the almost plug-and-play nature, it does not provide any higher level libraries yet, reducing the appeal for a novice.

Overall, there are frameworks which are good either at the software/communications level, or at the lower/hardware levels. We could not find a state-of-the-art framework which can be used for rapid-prototyping among all of these levels. Indeed, the researcher can be fast and assisted only at some stages of the development, while other stages still need to be done alone currently. *Table 2.1* summarizes the features of the state-of-the-art frameworks.

These observations have driven us to outline and develop a framework which interfaces to computation graphs of ROS, to exploit its high-level rapid prototyping features, but can still be run on cheap embedded systems, like the R2P Ethernet gateway module. The proposal illustrated in Chapter 5 can fit the missing link between high- and low-level rapid prototyping.

<table>
<thead>
<tr>
<th>Feature</th>
<th>ROS</th>
<th>Orocos</th>
<th>LCM</th>
<th>R2P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publish/Subscribe</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Scripting languages</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>planned</td>
</tr>
<tr>
<td>Embedded systems support</td>
<td></td>
<td></td>
<td>rosserial</td>
<td>✓</td>
</tr>
<tr>
<td>Lightweight</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Realtime support</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Package repository</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>planned</td>
</tr>
<tr>
<td>Connectivity (transport)</td>
<td>TCP, UDP</td>
<td>TCP, generic</td>
<td>UDP, CAN, generic</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 3

Research strategy

In the previous chapter, state-of-the-art robotics software frameworks were described, each with its benefits and drawbacks. In this chapter, we illustrate our research objectives and strategy, which will lead to the proposed framework described in the next chapters.

3.1 Objectives

It is very important to provide a good framework both in the hardware and software domains, so that it is possible to develop robots without spending too many resources, primarily time and money. Previous works are either very good in the software or in the hardware domains on their own, but there seems to be a missing link between them.

Sometimes, prepackaged algorithms are available only for frameworks which are excessively complex for the actual application. Instead of losing time for the reimplementation of those algorithms for a simpler system, it is common to equip the robot with a more powerful computer. This would be a price and performance overkill, which cannot turn into a future product without re-engineering at some point of the design process. Again, such a powerful computer may not be suitable for the specific robot domain, because of the required size and weight; think about a quadrotor, which cannot equip a desktop-grade computer. For these reasons, we will focus on embedded systems in our research.

Framework design Instead of proposing a brand new software framework, we are going to enhance an existing one towards embedded systems development. As seen in the previous chapter, some works like LCM and R2P are already
available on embedded platforms, but both miss the availability of user level
device libraries. By contrast, ROS has plenty of packages ready to be used,
but ROS itself is not well suitable for embedded systems.
By extending existing frameworks, it is still possible to take advantage of their
benefits, while trying to solve its drawbacks, or at least part of. Frameworks
are designed to be modular, and the purpose of our research is to let simple,
cheap peripheral modules communicate with the central framework modules
(often workstations) in a native way.

Small footprint However, the development of an embedded system always
copes with some resource constraints. An embedded system, with respect to a
desktop computer or a workstation, has often tight constraints about memory
size, CPU speed, and timings. A well endowed microcontroller of nowadays
is equipped with roughly 1 MB of code memory (ROM, EEPROM, Flash),
256 kB RAM, and 200 DMIPS.
While not being able to host a full framework in its entirety, a microcontroller
can still manage the the native communication protocols of the framework itself.
A piece of embedded hardware would still communicate with the framework
out-of-the-box, with the possibility to run prepackaged algorithms, and without
the need of proxies or bridges to handle different protocols.

Portability The proposed framework should also be portable. This is a tricky
feature in the field of embedded systems, because the hardware abstraction layer
is almost incompatible even among devices of the same family. Also, embedded
system compilers often fall behind those for generic-purpose computers; the
most recent programming languages (or their updates) are not available, or
the compiler implementation is not fully standard-compliant.

Handiness Last but not least, the code should be easy to use. A simple
design with simple code, good documentation, and utility tools can make the
development process more efficient and less error-prone. Since the target of
our work is rapid robot prototyping, it must be understood and modified in
the shortest possible time, so that users can focus on their goals rather than
on the software framework library itself.
To summarize, the enhanced system should have the following features:

• support for an existing framework;
• small code/memory footprint;
• portable among different hardware/software architectures;
• simple and clean design.

3.2 Strategy

Our work is an extension of an existing framework. Among the ones described in the state-of-the-art presented in Chapter 2, we have found that the most suitable for rapid prototyping purposes is ROS. Its development is the most active, and the availability of hundreds of fully modular packages shortens development times. Conversely, its lack of an embedded systems point of view discourages its use for simple systems, although there are some workarounds (e.g., rosserial).

The framework we are proposing, called µROSnode, is designed to run on embedded systems. Indeed, an user accustomed to ROS can still use its packages on the central computer, and deploy on peripheral modules which can natively communicate with ROS.

A consequence of native ROS support on microcontrollers is that it would be possible to directly connect µROSnode-based devices to a ROS-based network. Take the example of a simple robot, as in Figure 3.1a, made of an iRobot Create [13] moving base controlled via RS232, a Sick LMS100 [27] Ethernet laser scanner, and a custom IMU board yet to be designed. These devices are connected to a host computer running a ROS application which uses them. As they do not communicate with ROS directly, the host must run custom ROS nodes which interface with their low-level drivers to ROS itself.

Instead, take the example in Figure 3.1b. The laser scanner runs a modified µROSnode-based firmware by exploiting the available Ethernet port; the custom IMU board, designed from scratch, is programmed around µROSnode; an inexpensive adapter board interfaces the RS232 mobile robot platform to Ethernet by an inexpensive adapter board. With this design, the user only has to connect them to ROS (e.g., by linking to the host through Ethernet cables) and start developing the application, without worrying about the low-level protocols or device drivers. This design eases prototyping of new ideas, and provides the user a homogeneous API.

Communication The communication stack of ROS (see Chapter 4) was not designed to be used by resource-constrained platforms (unlike LCM and R2P), but rather for easy integration by different programming language libraries. Anyway, it is still possible to keep its implementation small enough to be deployed on a microcontroller, as proven in Chapter 6.

A major flaw of ROS for memory-constrained systems is that it communicates
Language  One of our objectives is software portability. In the embedded systems domain, portability is essentially assured by only one programming language, ANSI C. This language is broadly available for embedded systems of nowadays. Almost all of the embedded-grade C compilers can manage code written in pure ANSI C, thanks to its maturity. This ensures that object code generated by ANSI C compliant compilers will work as it should, while this is not always true for other languages.

Also, software written in ANSI C is still compatible with most of C++ and

Figure 3.1: A simple robot running ROS, the legacy way and the μROSnode way

through point-to-point TCP channels most of the times. This choice has an impact on the bandwidth, RAM allocation, and threads, because each channel needs its own management, while with a broadcast approach (UDP is capable of) they are unified per topic, not per channel. The UDP version of the ROS protocol is still experimental at the moment of writing, so we have to deal with the allocation of one channel per each TCP connection. Despite of being a design flaw, we will prove that the TCP approach is not a major issue with simple systems based on microcontrollers. For example, they can hardly saturate a 100 Mbps Ethernet link, and the user is aware of the fairly small number of simultaneous connections manageable by a microcontroller.
Objective C compilers, so there is still the possibility to integrate it with these higher level languages the native way, or by writing idiomatic wrappers. Incidentally, some features of a full C++ implementation (e.g., RTTI, exceptions) will make the object code explode in size only because they are enabled, thus exceeding the MCU code memory size. C++ without them is simply C with some syntax sugar, a redundancy with the additional risk of non-compliance as stated before.

System abstraction  Software portability for embedded systems is fulfilled also through an appropriate hardware abstraction layer. Our design includes bindings to the low-level features of the hardware, so that the user can write or choose the low-level drivers which best match the target architecture. Our framework needs low-level binding for communication, multithreading, and memory management. Communication libraries talk to a TCP/IP stack in a common fashion. A multithreaded environment is needed because it drastically simplifies the development of a complex firmware, and the overhead introduced by an multithreaded environment is marginal on modern MCUs. Memory management drivers should handle dynamic allocations (e.g., memory heaps). All of these features are often provided by an adequate operating system, and writing of bindings is rather straightforward. The API chosen for the deployment on general-purpose computers is POSIX, which is available in most of the operating systems. In the case of microcontrollers, we have chosen ChibiOS/RT, which supports some of the most advanced microcontrollers available nowadays, and LWIP as its TCP/IP stack.

Coding style and tools  The design of the proposed framework, extensively described in Chapter 5, consists of a few abstract entities, which are managed through a simple and clean software interface. The modular approach is also reflected on the filesystem structure of the package. The code follows mostly the style rules of ChibiOS/RT, which are derived from the canonical K&R (Kernighan and Ritchie) style. There are other coding rules which are aimed at portability (e.g., MISRA rules), but we think that K&R-derived styles are still the most clean and intelligible. The code also features extensive self-documentation for Doxygen, a widespread self-documentation tool. The user is provided with a code generator, which can generate templates suitable for the target application by specifying the needed topics, services, and their respective message types. A static analyzer can help in the development targeted at memory-constrained
platforms, by estimating the total stack usage of a set of functions (entry points of threads).

In conclusion, the enhanced framework of our research has the following main features:

- native compatibility with the ROS communication stack;
- code written in ANSI C;
- preconfigured for POSIX, and ChibiOS/RT + LWIP;
- simple modular design;
- extensively self-documented for Doxygen;
- a code generator for user application message management;
- static stack usage analyzer.
Chapter 4

The ROS communications model

Before going ahead to the detailed description of the research work, a detailed knowledge of the communication protocol is essential. This chapter illustrates the protocol defined by the current implementation of ROS, Groovy. It is the one adopted by our framework, to have native compatibility with ROS itself.

4.1 Definitions

The following are the definitions of terms occurring frequently within the ROS environment, as stated by the ROS documentation. The reader is aware that some terms can have a different meaning outside ROS, and the present document may make an ambiguous use because of the lack of valid synonyms.

**Name** Graph resource names are an important mechanism in ROS for providing encapsulation. Each resource is defined within a namespace, which it may share with many other resources. In general, resources can create resources within their namespace and they can access resources within or above their own namespace. Connections can be made between resources in distinct namespaces, but this is generally done by integration code above both namespaces. This encapsulation isolates different portions of the system from accidentally grabbing the wrong named resource or globally hijacking names.

Names are resolved relatively, so resources do not need to be aware of which namespace they are in. This simplifies programming as nodes that work together can be written as if they were all in the top-level
namespace. When these nodes are integrated into a larger system, they can be pushed down into a namespace that defines their collection of code.

**Computation Graph** The Computation Graph is a peer-to-peer network, which consists of nodes connected together in any *mesh* topologies, exchanging topics or providing services.

**Node** A node is a process that performs computation. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server. These nodes are meant to operate at a fine-grained scale; a robot control system will usually comprise many nodes. For example, one node controls a laser range-finder, one Node controls the robot’s wheel motors, one node performs localization, one node performs path planning, one node provide a graphical view of the system, and so on.

The use of nodes in ROS provides several benefits to the overall system. There is additional fault tolerance as crashes are isolated to individual nodes. Code complexity is reduced in comparison to monolithic systems. Implementation details are also well hidden as the nodes expose a minimal API to the rest of the graph and alternate implementations, even in other programming languages, can easily be substituted.

All running nodes have a graph resource name that uniquely identifies them to the rest of the system.

**Master node** The Master node provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate by peer-to-peer connections. The Master node also provides the Parameter Server.

**Message** Nodes communicate with each other by publishing messages to topics. A message is a simple data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, and so on) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays.

Nodes can also exchange a request and response message as part of a ROS service call.

**Topic** Topics are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the
production of information from its consumption. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the corresponding topic, while nodes that generate data publish to the corresponding topic. There can be multiple publishers and subscribers to a topic.

Topics are intended for unidirectional, streaming communication. Nodes that need to perform remote procedure calls, i.e. receive a response to a request, should use services instead. There is also the Parameter Server for maintaining global parameters.

Each topic is strongly typed by the ROS message type used to publish to it and nodes can only receive messages with a matching type. The Master does not enforce type consistency among the publishers, but subscribers will not establish message transport unless the types match. Furthermore, all ROS clients check to make sure that the MD5 matches. This check ensures that the ROS nodes were compiled from consistent code bases.

**Service** Services are used to execute request/response calls of raw data. A service consists of two messages (the request and the response) exchanged in a way similar to topic messages. If more than one Node advertise the same service, only the last one will be considered when trying to establish a service stream.

**Parameter** A parameter is a globally defined variable, stored by the Parameter Server. Parameters are named using the normal ROS naming convention. This means that ROS parameters have a hierarchy that matches the namespaces used for topics and nodes. This hierarchy is meant to protect parameter names from colliding. The hierarchical scheme also allows parameters to be accessed individually or as a tree.

**Publisher** A publisher node of a specific topic generates and sends TCPROS messages to all of the topic subscribers.

**Subscriber** A subscriber node of a specific topic receives TCPROS messages from all of the topic publishers.

**Parameter server** A parameter server is a shared, multi-variate dictionary that is accessible via network APIs. Nodes use this server to store and retrieve parameters at runtime. As it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters. It is meant to be globally viewable so that tools can easily inspect the configuration state of the system, and modify if necessary.
The Parameter Server is implemented using XML-RPC and runs inside of the ROS Master, which means that its API is accessible via normal XML-RPC calls.

**API** Set of software methods exposed by a process, so that other processes can call it.

**URI** Textual representation of a node API address, in the format:

```plaintext
protocol://host:port
```

where `protocol` is either `http` or `rosrpc`, and `host` is either a host name or an IP address. The `port` is the API port number.

**Remote procedure call** A technique to call a method on a remote machine, and receive its result.

**XML-RPC** Portable XML-based protocol to execute remote procedure calls.

For detailed information, see below.

**TCPROS** TCP-based protocol used by topic/service binary data streams.

For detailed information, see below.

### 4.2 Overview

The communication layer of ROS, part of the `ros_comm stack`, follows the `publish/subscribe` paradigm, as depicted in Figure 2.2. The network, also called `computation graph`, involves several `nodes`, either physical or virtual, which are assigned an unique name. A node is a process which gathers data from its inputs, executes computations, and generates data for its outputs. Each node is specialized to a particular function, and does not rely on the existence of other nodes (purely modular design). A node can publish some `topics`, which other nodes can subscribe to; `services` work a similar way. Topics and services are assigned a known `type`, delivered with their respective ROS package (e.g., `std_msgs/Float`).

For example, think about a simple autonomous rover, equipped with two motors, a set of distance sensors, and the brain computer hosting the ROS framework. A possible ROS network can consist of a node per each sensor or wheel, and a node for the robot brain. Let us suppose that each `sensor_X` node publishes a topic called `sensor_X/distance`, where `X` is the identifier of the sensor itself. The implementation samples a value measured by the actual sensor at a fixed rate, converts it to the representation advertised for the topic, and sends a marshaled message to the topic subscribers. In a similar manner, each `wheel_Y` node subscribes to a topic called `wheel_Y/speed`, with reference
to the speed of the \( Y \)-th wheel. The implementation continuously receives a subscribed topic message, converts its unmarshaled speed value to a voltage, and applies the latter to the wheel DC motor to obtain the communicated speed. Instead, the \texttt{brain} node subscribes to all of the \texttt{sensor\_X/distance} topics, and publishes all of the \texttt{wheel\_Y/speed} topics. Its implementation can follow a suitable logic which determines the wheel speeds by the distances measured by the sensors, for example to move the robot while avoiding obstacles. Its communication flow should be deducible by the reader now.

There can also exist some globally declared \textit{services} (a sort of special request/response topics) and \textit{parameters}. For example, the \texttt{brain} node publishes the service \texttt{sleeping} which replies \textit{true} if the robot is in an idle state, or \textit{false} if the robot is fully operational. This way, \texttt{sensor\_X} and \texttt{wheel\_Y} nodes can call the \texttt{sleeping} service, and switch off their hardware modules if the response is \textit{true}. Again, a global parameter called \texttt{max\_speed} holds the maximum speed value, useful to calibrate the behavior of wheel modules.

A special node, the \textit{Master} node, acts as a centralized dispatcher when connecting together publishers and subscribers. It keeps track of published/subscribed topics/services, and in the current implementation it is also the centralized directory for globally defined parameters (the \textit{parameter server}).

Communication with the Master node is accomplished by the exchange of \textit{XML-RPC} messages, through \textit{remote procedure calls}. The topic and service data streams between peers are encoded in a custom protocol, \textit{TCPROS}. Both these protocols will be extensively explained below. ROS connection addresses are sent in URI format.

### 4.3 Remote procedure calls

ROS nodes are capable of calling \textit{remote procedures}, available through the API exposed by the nodes themselves. Remote procedure calls are executed by using the \textit{XML-RPC} protocol, which is a subset of the XML specification (see Appendix B.1 for its XML Schema and DTD). These calls are used to manage the status of the computation graph and some global settings, and are not supposed for the actual data streams (which use TCPROS or UDPROS instead). XML-RPC was chosen by the designers of ROS because of its large availability of supported languages. The use of XML as its basis makes it simple to debug by a human, and its text-only nature makes it independent by the transport layer encoding format; indeed, it is encapsulated inside HTTP transfers, which prefer text-only contents. Also, XML-RPC calls are \textit{stateless},
a property which simplifies the control logic, since there is no state to keep track of. On the language side, its XML tags follow strict syntax rules, and they lack any attributes. These characteristics make its parsing much easier than XML, while keeping most of its advantages.

4.3.1 Roles

Every ROS node hosts a XML-RPC server. Depending on the role, the set of callable methods is different, and they are defined by the appropriate API. The Master API exposes the methods managed by the unique Master node, for example those for (un)registrations of topics and services. The Slave API declares the methods managed by a generic node (Slave), for instance the request to establish a topic stream, or the shutdown command. The Parameter Server API is used by the unique Parameter Server, which manages the centralized directory of shared global parameters. Detailed information about the APIs is given in Appendix B.1.

4.3.2 Value types

In XML-RPC, method parameters and return values are enclosed by the value entity, and can belong only to a small set of types (value children):

- **string** is an ASCII string. This is the default value type, where not stated otherwise. The only illegal characters are & and <, encoded respectively as &amp; and &lt;.
- **int** or **i4** is a 32-bit signed integer, in decimal representation, prefixed by - if negative.
- **boolean** can be either 0 or 1.
- **double** is a real number, in decimal representation, prefixed by - if negative.
- **dateTime.iso8601** is a date/time value, in ISO-8601 format.
- **base64** is a binary string encoded with the Base64 algorithm.
- **array** is a list of values, globally enclosed by a data entity.
- **struct**, also called **map**, is an associative collection. Each entry is a member, with a name string and a generic value.

4.3.3 Request and response

Remote procedure calls consist of two phases: the **request**, and the **response**. A caller (client) sends the method request to the callee (server), which processes
it. After processing, the callee returns a response, which can be either the call result if successful, or a fault if unsuccessful.

A request is introduced by the methodCall entity, whose method name is specified by the methodName string. Following the method name, a params entity encloses a list of param children, where each one contains a value. The listed params are the method parameters, and depend on the API specification. Usually, the first parameter is the name of the caller node within the computation graph.

A response contains the value returned by the remote method. In ROS, the value is an array with three fields. The first field, the status code, is an integer with one of the following values:

- -1 if there was an error by the caller, for example due to wrong API method name, parameters or types. The method is not executed at all.
- 0 if there was a failure while executing the method call by the remote node.
- 1 if the call was successful.

The second field is for status string, a human readable text which explains the reason of the status code value. Its value is optional (it can be empty).

The third field, the return value, contains the actual XML-RPC value returned by the method call. Its meaning is defined by the specific API.

If the response is a XML-RPC fault, it means that there was a hard error, typically a request non XML-RPC compliant.

An example of XML-RPC call to getPid() is shown by Figure 4.1, with request and response dumps respectively in Listing 4.1 and Listing 4.2.
4.4 Connection patterns

A node on its own has no purpose, and needs to be connected to other nodes some way. In the following, a generic communication pattern is shown, to let a node connect to its peers, and eventually instantiate the required topic data streams. A special case involves service calls.

Upon the introduction of a node to the computation graph, the node has to register its published and/or subscribed topics to the Master node. Each topic publisher is registered through a registerPublisher() remote procedure call, while each topic subscriber is registered through a registerSubscriber() call. The Master is thus omniscient, and can connect together publishers and subscribers of the same topic.

Whenever a node registers its subscribed topics to the Master, a successful response includes the list of publisher URIs already registered. The caller node can then connect to these publishers to receive topic messages. Whenever a new publisher registers to the Master, the latter issues a publisherUpdate()
call to the subscribers, with the updated list of available publisher URIs. Topic message streams are exchanged by the TCPROS protocol.

A symmetric case is the unregistration. Publishers are unregistered through a unregisterPublisher() remote procedure call to the Master node, while subscribers are unregistered through a unregisterSubscriber() call. By the way, it is up to publishers and subscribers to close any established data streams for the unregistered topics.

Services work in a slightly different way. The same service can be published by multiple nodes, but only the last advertisement is taken into consideration. A node willing to call the service first looks up the service URI by the Master, through a lookupService() remote call. Then, it actually calls the service provider via an appropriate request message. The service provider processes the request and generates a response message, if successful; otherwise, a response with an error message is replied. All of these messages are exchanged by the TCPROS protocol, where the response is headed by an additional ok byte (to distinguish success or fault).

### 4.5 Data streams

While XML-RPC provides an easy and clean protocol for remote method calls, its verbosity and text-centric encoding make it unsuitable for high bandwidth and low latency tasks. This is the case of data streams, like the ones sent by sensors or to actuators. ROS defines a custom binary protocol for such streams; this permits the transfer of raw data among nodes, with the minimum required message length (maximizing bandwidth) and almost no processing.
time (minimizing latencies). For detailed information about the topic and service message grammars, please refer to Section 5.3.1.

### 4.5.1 Type descriptors

Data streams are based on the exchange of messages of a particular type, specified by the topic/service at registration time (see previous section). A message is a sequence of values, arranged in the order defined by the type descriptor. In ROS, there exist two kinds, the *topic descriptor*, and the *service descriptor*. We will consider the topic descriptor first, because service types are just a simple extension.

**Topic types** A ROS topic message descriptor is written as a plain text file, where the file name coincides with the type name, plus the `.msg` extension. Every line can define a strongly-typed variable, where the type is either a primitive type, or the name of another message type descriptor, or even an array (fixed or variable) of any of such types. Thus, messages can be nested, and, being strongly typed, their streaming is simplified (no dynamic types to handle). Descriptors can also declare constant names of primitive types.

**Listing 4.3** shows the `rosgraphmsgs/msg/Log.msg` contents, while **Listing 4.4** is its clean expanded version, generated by `rosmsg show`.

**Service types** While topics are uni-directional, thus needing only the definition of one message type, services work on a request/response approach.
### Severity level constants

```plaintext
byte DEBUG=1 #debug level
byte INFO=2 #general level
byte WARN=4 #warning level
byte ERROR=8 #error level
byte FATAL=16 #fatal/critical level
```

### Fields

```plaintext
Header header
byte level
string name # name of the node
string msg # message
string file # file the message came from
string function # function the message came from
uint32 line # line the message came from
string[] topics # topic names that the node publishes
```

**Listing 4.3:** Contents of `rosgraph_msgs/msg/Log.msg`

```plaintext
byte DEBUG=1
byte INFO=2
byte WARN=4
byte ERROR=8
byte FATAL=16
std_msgs/Header header
uint32 seq
time stamp
string frame_id
byte level
string name
string msg
string file
string function
uint32 line
string[] topics
```

**Listing 4.4:** Clean, expanded contents of `rosgraph_msgs/msg/Log.msg`

So, a service involves two types, the one for the request, and the one for the response. Service type descriptors are stored similarly to topic types, where the extension is `.srv` instead. The descriptor is split into two parts by a `--` separator line, where the first part is the request message descriptor, and the second part is the response message descriptor.

**Listing 4.5** is an example of the `dynamic_reconfigure/srv/Reconfigure.srv` contents, while **Listing 4.6** is its clean expanded version, generated by `rossrv show`.

**Hashing** Because descriptor texts may be different among nodes, while their names can be the same, there must be a way to determine if two nodes actually share the same type. For such task, the MD5 sum was chosen for comparing two different type descriptions. The comparison function executes the following
operations on each type descriptor:

1. comments are removed;
2. whitespace is removed;
3. package names of dependencies are removed;
4. constants are moved ahead of variable declarations, keeping their arrangement;
5. for nested types, their hashing text is computed and appended to the current hashing text, in the order they appear;
6. the MD5 sum of the whole hashing text is computed.

If the MD5 sums are the same, then the two descriptors are equal.
Table 4.1: ROS to C type mapping

<table>
<thead>
<tr>
<th>ROS type</th>
<th>C99 type</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8</td>
<td>uint8_t</td>
</tr>
<tr>
<td>uint16</td>
<td>uint16_t</td>
</tr>
<tr>
<td>uint32</td>
<td>uint32_t</td>
</tr>
<tr>
<td>uint64</td>
<td>uint64_t</td>
</tr>
<tr>
<td>int8</td>
<td>int8_t</td>
</tr>
<tr>
<td>int16</td>
<td>int16_t</td>
</tr>
<tr>
<td>int32</td>
<td>int32_t</td>
</tr>
<tr>
<td>int64</td>
<td>int64_t</td>
</tr>
<tr>
<td>time</td>
<td>uros_time_t</td>
</tr>
<tr>
<td>duration</td>
<td>uros_time_t</td>
</tr>
<tr>
<td>string</td>
<td>struct UrosString { size_t length; char *datap; };</td>
</tr>
<tr>
<td>byte[ ]</td>
<td>struct UrosTcpRosArray { uint32_t length; void *entriesp; };</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>uint</td>
<td>uint32_t</td>
</tr>
<tr>
<td>int</td>
<td>int32_t</td>
</tr>
</tbody>
</table>

4.5.2 Message structure

Messages are serialized in little-endian binary form. This makes its processing straightforward by most of the little-endian CPU architectures available on the market, such as those based on x86 or ARM designs. Therefore, a data stream message is much like the serialization of a C struct variable; for example, on x86 architectures a message is a one-to-one dump of the variable itself, because of 8 bit field alignments in memory (by contrast, 32 bit on ARM cores).

Primitive types can be directly mapped to C types as seen in Table 4.1. Variable-length arrays and strings (arrays of chars) are a special case, because they provide the number of entries by a uint32 length, followed by the serialized entries themselves. By contrast, fixed-length arrays are treated as a sequence of entries of the specified type. Any nested types are expanded into their respective primitive types or arrays.

Ahead of the message content just described, a uint32 value anticipates the total length of the content itself.

Topic messages Topic messages simply follow the length+content scheme described above, since a topic involves uni-directional messages of the same type.

An example of "Hello, World!" message of type std_msgs/String is in Figure 4.4. Stream offsets are written as hexadecimal indexes in bold font; values are written as ASCII characters if printable, as hexadecimal numbers otherwise. Comments on the right indicate the specific field being streamed, and its
human-readable value.

**Service messages** Service requests are serialized the same way as topic messages. Instead, the serialization of the response needs to take into account the status of the response, which can be an actual response value, or an error message.

A so-called *ok byte* is sent ahead of the response. If 1, it means that the following streamed data is of the expected service response type. If 0 it means that there was an error while processing the request, and the following streamed value is of **string** type, representing a human-readable error text.

Examples of **rospy.tutorials/AddTwoInts** service request and response are shown by **Figure 4.5** and **Figure 4.6**. Instead, a service response reporting an unknown error can be seen in **Figure 4.7**.

### 4.5.3 Connection header

As soon as the topic/service client and the related server connect, they must agree upon the streaming parameters. They include the topic/service name and type, the possibility of multiple service requests, the use of bandwidth optimizations, and so on. All of these parameters are set up by the *connection header*, which is a special message sent at the very beginning of the whole stream, by both endpoints. At first, a header is sent by the client, which acts as a request. The server will then reply with another header, which acts as a response.

The connection header is made of fields. They are encoded like **strings**, thus a **uint32** telling the string length is sent ahead of the string characters. The value follows the **field=value** format, where the available fields and their semantics
depend on the topic/service request/response headers. The connection header is introduced by a `uint32` holding the whole header length, just like common messages.

Table 4.2 summarizes the fields of a connection header, and their presence within topic, service, or error headers (√), some of which are optional ((✓)) or can be probed by the requesting header (the field value equals to *).

Figure 4.8 and Figure 4.9 show respectively the request and response connection headers for the `/turtle1/command_velocity` of the `/turtlesim` Node, by a `turtlesim_teleop` Node.
### Table 4.2: Connection header fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Request</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>callerid</td>
<td>Sender Node name</td>
<td>✓/✓</td>
<td>✓/✓/✓</td>
</tr>
<tr>
<td>topic</td>
<td>Topic name</td>
<td>✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>service</td>
<td>Service name</td>
<td>✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>md5sum</td>
<td>MD5 sum of the type</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>type</td>
<td>Message type</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>request_type</td>
<td>Service request type</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>response_type</td>
<td>Service response type</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>message definition</td>
<td>.msg/.srv contents</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>persistent</td>
<td>Persistent service</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>latching</td>
<td>Latched values</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>tcp_nodelay</td>
<td>Nagle alg. disabled</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>error</td>
<td>Error text</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
</tbody>
</table>

![Figure 4.8: Dump of a connection header request](image)

![Figure 4.9: Dump of a response to the connection header in Figure 4.8](image)
Chapter 5

The proposed framework – \(\mu\)ROSnodule

According to the objectives of the research, explained in Chapter 3, we designed and implemented a communication framework, \(\mu\)ROSnodule (\texttt{uros} inside the code, logo in Figure 5.1), which has native compatibility with ROS, and is targeted towards embedded systems software. Its software architecture and implementation details highlight its modular approach, along with the design optimized for simplicity and portability. Furthermore, we developed some tools which help the user integrate the application software with the framework, by generating (un)marshaling procedures, and estimating the stack usage for memory-constrained target platforms. A development process is finally proposed, showing an ideal way to develop an application with our framework from scratch, or by integration.

5.1 Implementation choices

In the following, the major features and implementation choices are illustrated, for a better understanding of the work done in the field of embedded systems. Developing for tightly resource-constrained devices, we had to find the right compromise between language portability and features, and between footprint size and code elegance.
5.1.1 Language observations

The framework is written in almost pure ANSI C89 [1], also known as ISO C90 [39], in the corrected version supported by GCC [10]. This ensures the maximum portability across multiple platforms, as extensively illustrated by the research strategy of Section 3.2. The framework widely uses C99 standard integer types (e.g., uint32_t), which can mostly be defined in C89 by hand; actually, any recent compilers support them, by including stdint.h. Conformity to the C89 standard holds if compiling with gcc -ansi -pedantic.

The choice of this language, despite of its portability, made us accept some strong limitations not found in other similar languages, notably its widespread evolution C++. The lack of generic programming features, namespaces and classes, which could have made the code strictly typed and more readable, pushed us down to some common compromises.

For example, objects in lists are weakly addressed by pointers to void, where C++ templates would assign them the right type; the use of typing macros in this case worsens the readability. The absence of namespaces is solved by prefixing function names with ursos, often followed by the module name (.\ld_.+module for low-level drivers); major type names are prefixed by Uros, minor by ursos_. We tried to write a clean codebase, with an user-friendly coding style, in order to cope with the limitations of ANSI C.

As stated in Chapter 3, not all of the C++ features are available on all of the embedded systems, or are an overkill. For example, by just enabling RTTI the compiler generates a much bigger object code, which could not fit into the small code memory of a microcontroller.

Additionally, the useful exception handling is not supported at all by some compilers, target architectures, or operating systems. Moreover, the C++ language has not the same mature support of the plain old ANSI C.

5.1.2 Footprint optimizations

Wherever possible, the code was written to minimize memory usage. This may make the code a little obfuscated, but we tried to minimize the usage of aliased structures (through unions) only where the stack analysis tool showed
a great opportunity to minimize the stack usage. Anyway, such optimizations are typically for internal use, while the user API is clean.

Thanks to the stack analysis tool, we reduced the number of function calls, and minimized the function parameters or local variables. Although declaring more variables with meaningful names and single assignments makes the code more readable, the compiler was not always capable of optimizing the stack usage. So, some functions reuse local variables or parameters to reduce the stack usage, without aggressive reuse to not obfuscate the code.

Objects with a predictable life span are usually placed onto the stack, even the fairly big ones. This way, the stack analysis tool can keep track of such objects, and the programmer is aware of the consequences. Furthermore, stack allocations are faster and managed automatically.

By contrast, heap allocations are slower (even not constant-time) and error-prone; also, negligent allocation into the heap could saturate it (thus failing) unpredictably. This is why we encouraged stack allocations throughout the codebase.

In order to reduce stack usage, we avoided aggressive function nesting wherever possible, while keeping a reasonable degree of factorization.

Object code size is a minor concern in our perspective, since the latest microcontrollers have more than enough space, as proven in Section 3.2. Nevertheless, as common sense suggests, factorization of features, along with a set of preprocessor switches to ignore unused code, can reduce the object code size.

For instance, TCPROS connection header transmission and reception functions, which are rather lengthy, are shared between the client and the server; similarly, XML-RPC client calls rely on a common syntax. Some optimizations made by the compiler can help in reducing the object code footprint too, even those which target speed.

## 5.2 Framework description

The framework is composed by software modules which reflect their domain of expertise. Most of the modules are platform-independent, while some have a platform-dependent counterpart, prefixed by ld (low-level driver) which binds to the low-level libraries provided for the specific platform. There are also some application-dependent modules, which should be provided by the user, defining callback functions. Optionally, some user modules contain serialization functions and handlers for ROS topics and services. Major optimizations for resource-constrained platforms (embedded systems) are highlighted.
Figure 5.2 shows a logical view of software modules of the framework.

5.2.1 Base module

The urosBase module provides the software features used throughout the framework, such as basic types and memory management functions. The low-level counterpart, uros_lld_base, provides bindings for memory management functions.

Basic types The basic types provided by the framework are those defined by the C99 standard, plus some commonly used types. In addition, some useful function types (signatures) are defined, like the predicate, comparison, and thread entry point signatures.

Strings The string type, UrosString, is used for string management, instead of the null-terminated strings used by the C language. They consist of a size_t field which holds the string length, followed by the pointer to the string content. This representation optimizes string management, for example by parsing and serialization functions, thanks to the embedded length counter; this allows to share the same text chunk among different string descriptors, which may employ only parts of it.

The module provides functions for conversion from C strings (both assigned or
cloned), checking functions, and the string comparison function.

Lists The list type, UrosList, is a uni-directional concatenated list of elements of any types (the user is aware of). The list object itself holds the count of entries, and the pointer to the first one. List nodes, UrosListNode, consist of a pointer to the node data, and a pointer to the next node.

The library provides functions for common list operations (addition, removal, existence of nodes), as well as some predicate function types, used for finding entries.

Message types A message type record, UrosMsgType, holds the name, the optional text description, and the MD5 sum of a topic/service message type, usually well known inside the application domain. Thus, a global list, built at start-up, keeps all of the message types used within the application; a set of functions operates on this list.

Service message types are handled the same way, but on a separate global list.

Topic/service descriptors Topics and services share the same descriptor type UrosTopic. It holds the topic/service name, the pointer to its message type, the pointer to the handler function, a set of flags, and a reference counter. The message type is typically contained into the global static register (see above). The flags specify some connection settings, plus the switch between topic and service. The reference counter is used so that the descriptor can be shared by multiple handler instances which refer to it (shared pointer mechanism).

Heap allocators Most of the operations managed by the framework need dynamic allocations; think about the parsing of an incoming XML-RPC request holding a list of URIs. The heap allocator technique is used by the framework. A set of functions permits the dynamic allocation and deletion of memory chunks of any possible size. The library supports multiple heaps, for those platforms with disjointed memory spaces, commonly found in embedded system architectures.

Memory pools A common approach when managing memory blocks of the same size, for example thread pool stacks, is the memory pool. Blocks work as an uni-directional list, where each node is a memory chunk with an additional reserved pointer at the very beginning, which addresses the next node in the list.
A memory pool is defined by the `UrosMemPool` type. It holds the block size (comprehensive of the reserved pointer), the pointer to the first block in the list, and the number of free blocks left, the allocator function and the lock word (for multithreaded access).

If the allocator function is provided, when the pool is full and a new block is requested, then the allocator function will create a new block. Otherwise, the memory pool has fixed size, and the applicant thread is suspended.

The library provides functions for the creation of a memory pool, both dynamically by an allocator function, or from a fixed-size memory chunk (array), and functions for requesting or releasing blocks.

The memory pool mechanism illustrated above is platform independent, but can be overridden for native memory pools, if available.

### 5.2.2 Connectivity module

The `urosConn` module provides the TCP and UDP connectivity features of the framework. The low-level counterpart, `uros_lld_conn`, interfaces to the network stack chosen by the user.

**Addresses** The library provides data types for holding an IP number (`UrosIp`), and a TCP or UDP address (`UrosAddr`). Some helper functions can resolve hostnames or ROS URIs to addresses.

**Connection record** The connection record, `UrosConn`, holds the local and remote addresses, the protocol identifier, and the total number of received and transmitted bytes. Some fields may be added by the chosen low-level driver.

**Comparison with BSD sockets**

The connection life cycle and usage are equivalent to the widespread POSIX sockets [11], with only a few meaningful exceptions, so the reader is aware of consulting the POSIX sockets manual. The differences, explained below, are only in the transmission and reception call behaviors. They were introduced for a higher degree of portability among different network stacks; for example, there was no other clever way to provide bindings for the LWIP stack, while keeping a reasonable implementation for POSIX sockets. Connections can also support timeouts, if available on the target architecture.
Transmission  A call to `urosConnSend()` (or related) sends the specified chunk of data in its entirety, while POSIX sockets can send it partially (even not at all).

Reception  A call to `urosConnRecv()` (or related) fills a buffer provided by the low-level driver, instead of a buffer provided by the user as with POSIX sockets.

5.2.3 XML-RPC module

The `urosRpc` module provides all of the XML-RPC features needed by a ROS-compliant Node, like parsing and streaming functions, along with client calls and the Slave API server.

Parameters  A XML-RPC parameter is described by `UrosRpcParam`. It specifies the value class (parameter type switch), and the polymorphic value itself. See Section 4.3 for more information about XML-RPC parameter types.

Parameter lists  A parameter list, `UrosRpcParamList`, is a double-ended, uni-directional list of `UrosRpcParamNode` entries. The list descriptor holds the number of entries and pointers to the head and tail nodes. A node embeds a `UrosRpcParam` value, and the pointer to the next node.

Such a structure makes parameter management easier while parsing or processing XML-RPC streams. Indeed, nodes are appended at the end (tail), and the list can still be traversed from the oldest node (head) to the newest node (tail); by contrast, a `UrosList` can be managed in constant time and traversed only in reverse order. The embedding of the value inside the node avoids the unneeded indirections and heap fragmentation of a `UrosListNode`.

Parser  Being an XML-based language, one of the many XML parsers supplied by external libraries could be used for XML-RPC parsing. Anyway, most (if not all) were not designed specifically for resource-constrained platforms, as embedded systems are. Thus, this module provides an optimized parser, which can perform a one-pass parsing of the incoming XML-RPC request over HTTP, with small code and stack footprints.

The `UrosRpcParser` record holds the last error code (convenient for avoiding local variables on the stack), the pointer to the connection, and fields for parsing; among these, the pointer to the received HTTP buffer and its length,
and the current cursor status. Moreover, a buffer holds a possible incoming string entity; indeed, the only case where the $LL(k)$ hypotheses [24] do not hold is when a value is of string type without string tags, so a workaround is to prefetch a string of the maximum length possible.

Streamers The library is also capable of streaming XML-RPC messages over HTTP. The streamer object type is UrosRpcStreamer, which is structured similarly to the parser; this makes aliasing possible wherever needed, by reusing a memory chunk for both the parser and the streamer (mutually exclusive), thus optimizing the memory usage.

Calls A set of all of the possible ROS client calls to the Master, Slave, and Parameter server APIs is available. Their footprint in both code and memory usage is kept to a minimum, by exploiting the common syntax and XML-RPC parameters.

Slave The framework is used to create a Slave ROS Node, so it exposes its Slave API server, performed by the Slave server thread. It listens to any incoming XML-RPC request messages, and dispatches them to the appropriate Slave API method handler, which processes them and generates the expected response message.

5.2.4 TCPROS module

The urosTcpRos module provides functions to communicate through the TCPROS protocol.

Connection status The TCPROS connection status is held by the descriptor, UrosTpcRosStatus, which specifies the caller name, the referenced topic or service descriptor and flags, the exit request flag, and other options.

Functions Since the TCPROS protocol is very close to the binary representation in memory, its functions are merely shorthands for raw data transmission and reception. Instead, (un)marshaling routines should be implemented by application-specific code, inside the urosMsgTypes module (see Section 5.2.8). A pair of functions can send and receive TCPROS connection headers, for both publishers and subscribers.
Macros Furthermore, to simplify the body of topic/service handler functions, a set of useful macros provided. They are expanded to (un)marshaling statements, along with meaningful error checking, (un)initialization, and (de)allocation statements. Those macros are extensively used by those handlers created by the code generator (see Section 5.3.1).

Arrays In order to manage TCPROS variable-length arrays, records of type UrosTcpRosArray can be used by (un)marshaling procedures. The array type holds the number of entries, and a pointer to the entries memory chunk (of any type, the user is aware of type checking).

5.2.5 Node module
The urosNode module manages the configuration, the status, and the life of the Node singleton.

Configuration The configuration record, UrosNodeConfig, holds the Node configuration, loaded at boot time. It consists of the node name, the URI and the TCP/IP addresses of the remote Master XML-RPC server, the local XML-RPC Slave server, and the local TCPROS server.

Status The status record, UrosNodeStatus, holds the runtime status of the Node. It contains the publishers and subscribers lists, the active TCPROS connections list, the thread (and memory) pools for XML-RPC and TCPROS handlers, and functions for multithreaded access to all of them.

Registration functions A set of routines is used to manage topic publications/subscriptions, service publications/calls, and parameter subscriptions. These routines perform checks and operations on the Node status lists, as well as automatic management of TCPROS channels to be established or closed.

Node thread The Node thread takes care of the life of the Node. It populates the thread pools, registers topics/services/parameters to the Master node, keeps checking if the latter is always reachable, and performs a graceful shutdown in the reverse order.
5.2.6 Threading module

The **urosThreading** module provides a multithreading environment as needed by the framework, like threads, synchronization primitives, and thread pools. The low-level driver, **uros_lld_threading**, binds to the multithreading API of the target operating system.

**Primitives**  The synchronization primitives supplied by the framework are the canonical ones, such as semaphores, mutexes, and condition variables. Their usage should be straightforward.

**Threads**  As for synchronization primitives, threads follow a canonical approach. A thread can be created with a stack either from the heap or from a memory pool; the thread can be eventually joined.

**Thread pools**  The threading library includes a thread pool mechanism, where thread pools have **UrosThreadPool** type, and they make use of memory pools (**UrosMemPool**) as their stack supplier. The thread pool can be started (threads created) and stopped (threads joined) gracefully.

5.2.7 User module

Some features of the framework cannot work without some kind of management by the user application. Callback functions of the **urosUser** module provide the link between the framework internals and the user application.

**Node configuration persistence**  If deciding not to load the default, hard-coded Node configuration at boot time, a pair of load/save functions access a non-volatile memory device.

**Registrations and unregistrations**  There is a set of functions to manage message type, topic, and service (un)registration callbacks, triggered by the Node thread.

**Shutdown callback**  A callback function must be defined to gracefully close communication channels, and release resources whenever a **shutdown()** remote call is issued to the Node.
Parameter update callback  A further callback function is triggered whenever the Master issues a `paramUpdate()` to this Node, holding the global ROS parameter name, and its value.

5.2.8 Message types module

An optional module, `urosMsgTypes`, should provide the (un)marshaling functions of topic and service message types, along with their registration. This approach is highly suggested, as it keeps the codebase clean and easily maintainable. This module is application-dependent, and it can be automatically generated by the code generator tool. See Section 5.3.1 for in-depth description.

5.2.9 Handlers module

Another optional module, `urosHandlers`, should provide the topic and service handler functions. Like for `urosMsgTypes`, it is highly suggested, and automatically generated (at least its skeleton) by the code generator tool of Section 5.3.1.

5.3 Development tools

In order to facilitate the development of user applications, a code generator, a stack usage analyzer, and demonstration projects are included by the framework package.

5.3.1 Code generator

Writing marshaling procedures and topic/service handler routines by hand can take much time, and it is an error-prone task. The generation of marshaling procedures can be automated by processing the message descriptors delivered with ROS packages. Additionally, handler routines share a common skeleton, which can be exploited and reused. That being so, a generation tool, `urogen` (implemented as a Python 2.7 script, `urogen.py`), was developed.

Compilation flow

The compilation flow, depicted in Figure 5.3, is pretty straightforward. The tool is configured through a file (full example in Listing B.11), which lists the
Figure 5.3: Code generator flowchart

topic/service names and types, along with some options. Once the involved message type names are known, the tool loads their description from the related .msg and .srv files of the installed ROS packages. Finally, the message types are processed, thus producing marshaling and unmarshaling functions. Topic and service handler stubs are generated for those names listed inside the configuration file. The tool also generates detailed self-documentation for Doxygen (not reported in listing examples, as well as most of the comments).

Name mangling

All ROS names have a path-like structure, with / (slash) as separator. C identifiers cannot contain it, so ROS names are mangled by replacing / with __ (double underscore).

Topic and service mangled names always start with __, since they have an implicit / at the beginning.

Messages

A message type is loaded from its .msg file through a call to rosmsg show, which removes comments and has a clean syntax. Its inherent structure mapped to a C struct, where the mangled name is prefixed by msg__ (msg__name). Primitive types are directly mapped, while nested message types are declared separately. In order to conform with the C language, structs are defined in topological order; this can be done because ROS types cannot have circular dependencies.
A message type example, `rosgraph_msgs/Log`, is in Listing 5.1.

The code generator creates functions for the following operations: initialization, cleaning, length computation, marshaling, and unmarshaling.

The initialization function (`init_+msg_name`, Listing 5.3) sets the preliminary, safe values. The cleaning function (`clean_+msg_name`, Listing 5.4) deallocates any allocated fields, reaching the safe initialization state.

The length computation function (`length_+msg_name`, Listing 5.2) computes the length of the serialized message.

The marshaling function (`send_+msg_name`, Listing 5.6) serializes the message contents and sends them over the outgoing TCPROS stream. Instead, the unmarshaling function (`recv_+msg_name`, Listing 5.5) unserializes the received message from the incoming TCPROS stream.
```c
void init_msg__rosgraph_msgs__Log(
    struct msg__rosgraph_msgs__Log *objp
) {
    uint32_t i;
    urosAssert(objp != NULL);
    init_msg__std_msgs__Header(&objp->header);
    uroStringObjectInit(&objp->name);
    uroStringObjectInit(&objp->msg);
    uroStringObjectInit(&objp->file);
    uroStringObjectInit(&objp->function);
    uroTcpRosArrayObjectInit((UrosTcpRosArray *)&objp->topics);
    for (i = 0; i < objp->topics.length; ++i) {
        uroStringObjectInit(&objp->topics.entriesp[i]);
    }
}
```

Listing 5.3: Initialization of a `rosgraph_msgs/Log` descriptor

```c
void clean_msg__rosgraph_msgs__Log(
    struct msg__rosgraph_msgs__Log *objp
) {
    uint32_t i;
    if (objp == NULL) { return; }
    clean_msg__std_msgs__Header(&objp->header);
    uroStringClean(&objp->name);
    uroStringClean(&objp->msg);
    uroStringClean(&objp->file);
    uroStringClean(&objp->function);
    for (i = 0; i < objp->topics.length; ++i) {
        uroStringClean(&objp->topics.entriesp[i]);
    }
    uroTcpRosArrayClean((UrosTcpRosArray *)&objp->topics);
}
```

Listing 5.4: Cleaning function of a `rosgraph_msgs/Log` descriptor
uros_err_t recv_msg__rosgraph_msgs__Log{
    UrosTcpRosStatus *tcpstp,
    struct msg__rosgraph_msgs__Log *objp
} {
    uint32_t i;
    urosAssert(tcpstp != NULL);
    urosAssert(urosConnIsValid(tcpstp->csp));
    #define _CHKOK { if (tcpstp->err != UROS_OK) { goto _error; } }
    recv_msg__std_msgs__Header(tcpstp, &objp->header); _CHKOK
    urosTcpRosRecvRaw(tcpstp, &objp->level); _CHKOK
    urosTcpRosRecvString(tcpstp, &objp->msg); _CHKOK
    urosTcpRosRecvString(tcpstp, &objp->file); _CHKOK
    urosTcpRosRecvString(tcpstp, &objp->function); _CHKOK
    urosTcpRosRecvRaw(tcpstp, objp->line); _CHKOK
    urosTcpRosArrayObjectInit((UrosTcpRosArray *)&objp->topics, topics.length, _CHKOK
    objp->topics.entriesp = urosArrayNew(objp->topics.length, _CHKOK
        (UrosString);
    if (objp->topics.entriesp == NULL) { tcpstp->err = UROS_ERR_NOMEM; goto _error; }
    for (i = 0; i < objp->topics.length; ++i) {
        urosTcpRosRecvString(tcpstp, &objp->topics.entriesp[i]); _CHKOK
    }
    return tcpstp->err;
    #undef _CHKOK
}

uros_err_t send_msg__rosgraph_msgs__Log{
    UrosTcpRosStatus *tcpstp,
    struct msg__rosgraph_msgs__Log *objp
} {
    uint32_t i;
    urosAssert(tcpstp != NULL);
    urosAssert(urosConnIsValid(tcpstp->csp));
    #define _CHKOK { if (tcpstp->err != UROS_OK) { return tcpstp->err; } }
    send_msg__std_msgs__Header(tcpstp, &objp->header); _CHKOK
    urosTcpRosSendRaw(tcpstp, objp->level); _CHKOK
    urosTcpRosSendString(tcpstp, &objp->name); _CHKOK
    urosTcpRosSendString(tcpstp, &objp->msg); _CHKOK
    urosTcpRosSendString(tcpstp, &objp->file); _CHKOK
    urosTcpRosSendString(tcpstp, &objp->function); _CHKOK
    urosTcpRosSendRaw(tcpstp, objp->line); _CHKOK
    for (i = 0; i < objp->topics.length; ++i) {
        urosTcpRosSendString(tcpstp, &objp->topics.entriesp[i]); _CHKOK
    }
    return tcpstp->err = UROS_OK;
    #undef _CHKOK
}

Listing 5.5: Reception and unmarshaling of a rosgraph_msgs/Log message

Listing 5.6: Marshaling and transmission of a rosgraph_msgs/Log message
Service messages

Service messages are a pair of plain in/out messages, thus being split into an input struct (in.srv_. + mangled name, in.name) and an output struct (out.srv_. + mangled name, out.name), shown in Listing 5.7. They are loaded from the relative .srv file through a call to rossrv show, similarly to plain messages.

Each of the sub-messages have dedicated initialization (Listing 5.9), cleaning (Listing 5.10), and length computation (Listing 5.8) functions, with the assignments and prefixes seen above.

Only the input message has an unmarshaling function (recv_.+in.name, Listing 5.11), while only the output message has a marshaling function (send_.+out.name, Listing 5.12).

Type registrations

The urosMsgTypesReqStaticTypes() procedure registers all message types to their respective static registers (see Section 5.2.1), as reported by Listing 5.13.

Handler routines

Handler routines, except those for service calls, are supposed to work inside their own thread. The user can decide whether to place local message variables on the stack or in the heap, to optimize and harmonize memory management; as a rule of thumb, small message descriptors are put on the stack, big message descriptors on the heap.

Topic publishers

The topic publisher routine, pub.tpc + mangled name, generates and sends plain messages to the subscriber, until disconnection.

A generated template can be seen in Listing 5.14, where the message descriptor is allocated into the heap. The user is only required to remove the dummy loop statements, and fill in the msgp descriptor fields.

Topic subscribers

On the other hand, the topic subscriber routine, sub.tpc + mangled name, continuously receives and processes messages until disconnection.
As seen in Listing 5.15, the user is only required to process the contents of the received message, msgp.

Service publishers The service publisher, pub_srv + mangled name, receives the request message (in_name), processes it, then sends the response message (out_name) if successful, or the error string if unsuccessful. If persistent, these operations are repeated until disconnection.

Listing 5.16 is the generated template of a service publisher. It is an union of message reception and transmission, so the user must provide code for both the request processing and the response generation.

Service calls The service call routine, call_srv + mangled name, is executed by clients. It sends a request to the publisher, then receives the response or the error string. As opposed to the other handler routines, which generate or process messages within their body, the service call routine is used only for
void init_in_srv__turtlesim__Spawn(
struct in_srv__turtlesim__Spawn *objp
) {
urosAssert(objp != NULL);
urosStringObjectInit(&objp->name);
}

void init_out_srv__turtlesim__Spawn(
struct out_srv__turtlesim__Spawn *objp
) {
urosAssert(objp != NULL);
urosStringObjectInit(&objp->name);
}

Listing 5.9: Initialization of turtlesim/Spawn descriptors

void clean_in_srv__turtlesim__Spawn(
struct in_srv__turtlesim__Spawn *objp
) {
urosAssert(objp != NULL);
urosStringClean(&objp->name);
}

void clean_out_srv__turtlesim__Spawn(
struct out_srv__turtlesim__Spawn *objp
) {
urosAssert(objp != NULL);
urosStringClean(&objp->name);
}

Listing 5.10: Cleaning function of turtlesim/Spawn descriptors

An example of generated handler is shown in Listing 5.17. The user is not required to do anything, but it is suggested to deallocate the request descriptor before receiving the response, in order to reduce the memory usage.
uros_err_t recv_in_srv__turtlesim__Spawn(UrosTcpRosStatus *tcpstp, struct in_srv__turtlesim__Spawn *objp) {
    urosAssert(tcpstp != NULL);
    urosAssert(urosConnIsValid(tcpstp->csp));
    #define _CHKOK { if (tcpstp->err) { goto _error; } }
    urosTcpRosRecvRaw(tcpstp, objp->x); _CHKOK
    urosTcpRosRecvRaw(tcpstp, objp->y); _CHKOK
    urosTcpRosRecvString(tcpstp, &objp->name); _CHKOK
    tcpstp->err = UROS_OK;
    _error:
    clean_in_srv__turtlesim__Spawn(objp);
    return tcpstp->err;
    #undef _CHKOK
}

Listing 5.11: Reception and unmarshaling of a turtlesim/Spawn request message

uros_err_t send_out_srv__turtlesim__Spawn(UrosTcpRosStatus *tcpstp, struct out_srv__turtlesim__Spawn *objp) {
    urosAssert(tcpstp != NULL);
    urosAssert(urosConnIsValid(tcpstp->csp));
    #define _CHKOK { if (tcpstp->err) { return tcpstp->err; } }
    urosTcpRosSendString(tcpstp, &objp->name); _CHKOK
    tcpstp->err = UROS_OK;
    #undef _CHKOK
}

Listing 5.12: Marshaling and transmission of a turtlesim/Spawn response message

void urosMsgTypesRegStaticTypes(void) {
    urosRegisterStaticMsgTypeSZ("rosgraph_msgs/Log", NULL, "acff30c3b66d93c61c177c93fb");
    urosRegisterStaticMsgTypeSZ("std_msgs/Header", NULL, "2176dcaeebce78abc396ef049fbed");
    urosRegisterStaticSrvTypeSZ("turtlesim/Spawn", NULL, "0b2d2e762a8e2887d5ed626f2fbf2c561");
}

Listing 5.13: Registration of the static types used in the examples above
uros_err_t pub_tpc_rosout(UrosTcpRosStatus *tcpstp) {
    UROS_TPC_INIT_H(msg__rosgraph_msgs__Log);
    while (!urosTcpRosStatusCheckExit(tcpstp)) {
        UROS_MSG_SEND_LENGTH(msgp, msg__rosgraph_msgs__Log);
        UROS_MSG_SEND_BODY(msgp, msg__rosgraph_msgs__Log);
        clean_msg__rosgraph_msgs__Log(msgp);
    }
    tcpstp->err = UROS_OK;
    _finally:
    UROS_TPC_UNINIT_H(msg__rosgraph_msgs__Log);
    return tcpstp->err;
}

Listing 5.14: Generated handler template for a common /rosout publisher

uros_err_t sub_tpc_rosout(UrosTcpRosStatus *tcpstp) {
    UROS_TPC_INIT_H(msg__rosgraph_msgs__Log);
    while (!urosTcpRosStatusCheckExit(tcpstp)) {
        UROS_MSG_RECV_LENGTH();
        UROS_MSG_RECV_BODY(msgp, msg__rosgraph_msgs__Log);
        clean_msg__rosgraph_msgs__Log(msgp);
    }
    tcpstp->err = UROS_OK;
    _finally:
    UROS_TPC_UNINIT_H(msg__rosgraph_msgs__Log);
    return tcpstp->err;
}

Listing 5.15: Generated handler template for a /rosout subscriber
uros_err_t pub.srv...reconfigure(UrosTcpRosStatus *tcpstp) {
  UROS_SRVCALL_INIT_HISO(in_srv__dynamic_reconfigure__Reconfigure,
                          out_srv__dynamic_reconfigure__Reconfigure);

  do {
    UROS_MSG_RECV_LENGTH();
    UROS_MSG_RECV_BODY(inmsgp, in_srv__dynamic_reconfigure__Reconfigure);
    /* TODO: Process the request message. */
    tcpstp->err = UROS_OK;
    urosStringClean(&tcpstp->errstr);
    okByte = 1;
    clean_in_srv__dynamic_reconfigure__Reconfigure(inmsgp);
    /* TODO: Generate the contents of the response message. */
    UROS_SRV_RECV_OKBYTE_ERRSTR();
    UROS_MSG_RECV_LENGTH();
    UROS_MSG_RECV_BODY(outmsgp, out_srv__dynamic_reconfigure__Reconfigure);
    clean_out_srv__dynamic_reconfigure__Reconfigure(outmsgp);
  } while (tcpstp->topicp->flags.persistent &&
           !urosTcpRosStatusCheckExit(tcpstp));
  tcpstp->err = UROS_OK;
  _finally:
  return tcpstp->err;
}

Listing 5.16: Generated handler template for a service publisher

uros_err_t call.srv...reconfigure(
    UrosTcpRosStatus *tcpstp,
    struct in_srv__dynamic_reconfigure__Reconfigure *inmsgp,
    struct out_srv__dynamic_reconfigure__Reconfigure *outmsgp
) {
  UROS_SRVCALL_INIT_HISO(in_srv__dynamic_reconfigure__Reconfigure,
                          out_srv__dynamic_reconfigure__Reconfigure);
  UROS_MSG_SEND_LENGTH(inmsgp, in_srv__dynamic_reconfigure__Reconfigure);
  UROS_MSG_SEND_BODY(inmsgp, in_srv__dynamic_reconfigure__Reconfigure);
  /* TODO: Dispose the contents of the request message. */
  UROS_SRV_RECV_OKBYTE();
  UROS_MSG_RECV_LENGTH();
  UROS_MSG_RECV_BODY(outmsgp, out_srv__dynamic_reconfigure__Reconfigure);
  tcpstp->err = UROS_OK;
  _finally:
  return tcpstp->err;
}

Listing 5.17: Generated handler template for a client service call
5.3.2 Static stack analysis

When dealing with tightly resource-constrained platforms running multiple threads, it is important to keep the stack usage at a minimum. This is even more important when exploiting thread pools with homogeneous stack size, because this size depends on the maximum stack depth reached by any of its worker threads. A static stack analysis tool come handy to estimate, better if precisely, the stack usage of a set of functions of interest. The stack analysis tool provided with µROSnode is urosstan, implemented by the urosstan.py Python script.

Analysis flow

A diagram of the compilation flow is shown in Figure 5.4. The user supplies the configuration file (actual example in Listing B.12) and the static analysis outputs to the tool. For each entry point, it builds the call graph, and it traces the paths with maximum stack usage. The tool then outputs the list of unresolved symbols, a report for every entry point, and a summary.

Configuration

The user must provide a configuration file, which specifies the analysis options, and the sets of entry points, custom terminator symbols, and source file mappings.

Entry points An analysis is performed for each entry point (global function name) listed in the entry point set. Each entry point is assigned a value which indicates the stack depth before entering it.

Terminators The user may want to cut away some call graph branches, or to manually define the stack usage of unresolved symbols. For instance, external library functions or assembly calls cannot be analyzed, but the user can still provide the maximum number of bytes allocated by those call triggers. This way, the call graph analysis is stopped when reaching any of the terminator symbols provided by the user.

File mappings The stack analyzer relies upon the static analysis generated by the GCC toolchain. The mapping set maps each source file to the .gkd,
Figure 5.4: Static stack analysis flowchart

.*su, and *.nm outputs; respectively, the GCC RTL dump, the GCC stack usage report, and the GNU nm object symbols.

Observations about the effectiveness

Despite of its usefulness, there are often serious obstacles which reduce the effectiveness of the static analysis. Primarily, it analyzes the output files of the GCC toolchain only for C/C++ source files. External libraries and assembly code cannot be tracked by the GCC static analysis, thus being impossible to build the complete call graph, or to exploit some stack usage reports.

Moreover, indirect functions assigned at runtime cannot be tracked by static analysis tools. This is why the user should be aware of providing a meaningful list of entry points.

Nevertheless, recursion is an unremovable obstacle for static analysis. The tool can spot cycles within the call graphs, so the user is encouraged to profile their impact on the stack usage at runtime.

Anyway, the tool is still very useful to discover paths with unexpectedly deep allocation on stacks, or to have a better idea of the overall stack usage of the analyzed code. This way, the user can optimize the allocation of local variables, or harmonize stack allocation wherever possible.

The tool shows its best usefulness when there is full access to the source code, written entirely in C/C++. By the way, the maximum precision can still be
achieved if the usage of all of the assembler and library functions are specified inside the configuration file. If some of the information is missing but known to be bounded, for example through their full-coverage runtime profiling data, then the analysis can report at least conservative results.

It is important to point out that urosstan currently supports only the GCC toolchain for a C/C++ codebase, because this is the toolchain we used throughout the development of our projects. If the user is interested only of the overall stack allocation, then a pure C/C++ codebase can be compiled with GCC and no optimizations, still providing a rough conservative analysis. If the target toolchain is different, the equivalence of the object code is coarse, but still effective for preliminary analysis.

5.3.3 Demonstration projects

The package provides a turtlesim demo, which is almost equivalent to the turtlesim_node [7] official ROS node. It supports all of its topics, services and parameters, exploiting almost all of the features needed by a real-life Node. The maximum number of turtles is bounded by the MAX_TURTLES constant. The turtle pose is updated at 1 kHz, and streamed at 100 Hz.

Message types and handlers were first generated with the configuration file shown by Listing 5.18. Since the names of turtleX/* topics and services change with the turtle, their creation and deletion were moved to the application (app) module. This module handles the turtle spawn and kill operations, as well as the generation of /rosout messages.

The static stack analysis configuration file is shown in Listing B.12.

The demo comes into two ports: one developed with only POSIX API, and a second one which runs under ChibiOS/RT with the LWIP network stack. Eclipse project files are included.

Project: turtlesim-posix demo

This demo follows the POSIX standard for all of its low-level features. It was tested with Linux Mint 14 Nadia on a standard laptop, and Raspbian Wheezy on a Raspberry Pi model B with 256 MB of RAM (see Section 6.1.3).

When the /turtlesim node is shut down, the application will exit.
# urosgen.py configuration file for turtlesim

[Options]
author = Andrea Zoppi <texzk@email.it>
licenseFile = ../../../COPYING
includeDir = ../include
sourceDir = ../src
nodeName = turtlesim
fieldComments = false

[PubTopics]
rosout = rosgraph_msgs/Log
turtleX/pose = turtlesim/Pose
turtleX/color_sensor = turtlesim/Color

[SubTopics]
turtleX/command_velocity = turtlesim/Velocity

[PubServices]
clear = std_srvs/Empty
kill = turtlesim/Kill
spawn = turtlesim/Spawn
turtleX/set_pen = turtlesim/SetPen
turtleX/teleport_absolute = turtlesim/TeleportAbsolute
turtleX/teleport_relative = turtlesim/TeleportRelative

[CallServices]
# none

Listing 5.18: Turtlesim configuration file for urosgen.py

---

Project: turtlesim-chibios+lwip

This demo uses a combination of the ChibiOS/RT RTOS and the LWIP network stack, and can run on an ARM Cortex M4 core. Specifically, it was tested with ChibiOS/RT 2.5.2 and LWIP 1.4.1. The processor is a STM32F407 on a custom board with Ethernet capabilities (see Section 6.1), by using a DP83848 PHY in RMII mode. The USB port is used as a serial port terminal/shell emulator.

When the /turtlesim node is shut down, only the Node stops running, while the other features of the board keep running.

5.4 Integration with user applications

µROSnode was developed to be integrated into the user application with low effort. In the following, generic integration requirements are illustrated.

Makefile scripts The µROSnode package supplies a set of Makefile scripts to be included into the root Makefile; they can be found in the mk folder. The main script, uros.mk, defines the lists of core source and header files, which are required.

In addition, the user has to include scripts for the target platform, which define
the lists of low-level-driver source and header files. If a subsystem of the target platform is not yet supported, it can be developed by the template files found in the template/src/lld folder; comments labeled with TODO give instructions for a correct implementation.

**Configuration** µROSnode needs to be configured for the user application. A shared header file, urosconf.h, must be included by the application, and provides settings for the µROSnode subsystems. A template can be found in the template/include folder of the µROSnode package. Comments assist the user in tuning the required settings.

**Callbacks** The User module (see Section 5.2.7) requires some callback functions to be defined inside the urosUser.c application source file. A template file in template/src assists in the development of such callback functions.

**Handler functions** The user should create the Message types and the Handlers modules (see Section 5.2.8 and Section 5.2.9), so that µROSnode can properly process message streams of topics and services. This task is assisted by the code generator tool extensively illustrated in Section 5.3.1.

**Initialization** The µROSnode framework must be initialized by calling urosInit(), which initializes the global state. This should be done when the platform is completely initialized.

The Node is brought to life as soon as the actual user application is run. This is accomplished by a call to urosNodeCreateThread() to create the Node thread, which keeps track of the Node life cycle; topics, services, and parameters are registered and unregistered by the Node thread. This thread exits when the Node is shut down.
Chapter 6

Experimental results and evaluation

In order to prove the effectiveness of µROSnode, both quantitative and qualitative analyses were accomplished. The quantitative analysis consists of a set of benchmarks which show the performance of some basic yet meaningful applications, suggesting any possible bottlenecks. Instead, the qualitative analysis focuses more on the ease of software development on a real-life robot, to spot implementation difficulties of a µROSnode-based application.

6.1 Benchmarks

The performance of µROSnode was evaluated through a suite of benchmarks. Designing benchmarks is not a trivial task, because it is almost impossible to approximate real-life applications, especially in the robotics research field. Some basic benchmarks measured the memory occupation and speed performance of a µROSnode-based firmware, giving an idea of the overhead introduced by µROSnode on simple embedded systems based on recent microcontrollers.

Although these benchmarks are rather simple, even slightly more complex ones would rely more on the multithreading and networking subsystems than on µROSnode itself.

For instance, the CPU impact with many concurrent topics is mainly caused by the RTOS thread synchronization performance, while other effects are caused by the networking subsystem configuration choices (buffering, TCP segmentation, timeouts, and so on).

Nevertheless, every real-life application needs the integration of multiple subsys-
tems running concurrently, on different operating systems, different hardware, and implementing complex communication patterns.

6.1.1 Communication setup

The communication performance was measured by connecting the R2P_GW board (see Section 6.1.2) to a standard notebook host, mounting an Intel T6500 processor (dual core, 2.1 GHz), through a 100 Mb/s Ethernet connection. The firmware was compiled with GCC at the maximum default optimization level (-O3), disabling both assertions and error messages.

The communication performed by our tests consist of a single TCPROS topic being continuously streamed. The topic is of std_msgs/String type, so that it is possible to easily assign contents with fixed length to each test. The global parameter /benchmark_size controls the string length, while the global parameter /benchmark_rate controls the message rate. The /benchmark_rate parameter was set to 0 (no delays between messages) for both benchmarks, to reach the maximum possible throughput.

The transmission benchmark involves a target platform which publishes the /benchmark/output topic, subscribed by a rostopic node on the host. The rostopic subscriber on the host machine can receive up to ≈20000 msg/s. This benchmark evaluates the marshaling performance of µROSnode on the target platform, by measuring the CPU usage and the rate of packets being actually streamed.

Instead, the reception benchmark involves a /benchmark/input topic published by a rostopic process on the host, generating messages at the maximum possible rate, and the target platform subscribed to that topic. The throughput limit of a rostopic publisher on the host machine is of ≈10000 msg/s. This benchmark evaluates the unmarshaling performance, again by CPU usage and packet rate. The topic subscriber handler consumes the incoming message stream on-the-fly.

Result plots show the impact of the message size on both the CPU usage and the throughput. Log-lin plots can be useful to estimate the maximum number of concurrent threads, because usage and throughput are shown in linear scale. Log-log plots can suggest linear relationships between some series, as well as deriving exponential relationships. Indeed, a log-log line in the form $y = mx + b$ represents a lin-lin power function in the form $y = 10^b x^m$. So, two parallel lines in a log-log plot represent a linear relationship of two power functions of the same degree; more generally, two functions with the same distance over the same domain are tied by a linear relationship.
6.1.2 R2P_GW module

The Rapid Robot Prototyping [2] Gateway module [29] (R2P_GW, prototype board in Figure 6.1) interfaces the R2P CAN network to an Ethernet network; a prototype can be seen in Figure 6.1. It mounts a STM32F407 MCU by STMicroelectronics, with 192 KiB RAM (112 KiB shared globally), clocked at 168 MHz, and featuring an ARM Cortex-M4 core. The platform exposes a UART, a CAN, an USB-FS micro-B, and a 100 Mb/s Ethernet ports; the latter controlled by a DP83848 PHY by Texas Instruments. The default operating system is the ChibiOS/RT RTOS, combined with the LWIP network stack.

Although this board was not designed with ROS in mind, especially memory-wise, it is possible to turn it into a native ROS node, by integrating µROSnode into the firmware. Its RAM size limits the maximum number of communication threads instantiated (for both XML-RPC and TCPROS). Anyway, even by using only the 112 KiB of shared memory provided by the microcontroller, it is still possible to handle a few tens of communication threads, each allocating a 2 KiB stack; this is more than enough for a simple robotics device.

Program memory and stack footprints

A benchmark computed the size of the turtlesim firmware, supplied with the turtlesim-chibios+lwip demo of the µROSnode package, which emulates the turtlesim demo of ROS [7]. The code was compiled with GCC by applying the -00, -02, and -03 optimizations levels. A batch had all of the µROSnode and ChibiOS/RT assertions and checks enabled, while another batch had all of them disabled. LWIP checks and error messages were always disabled, while
both \(\mu\)ROSnode and ChibiOS/RT had all of the remaining features enabled in their configuration files.

This benchmark was made by compiling a plain firmware with ChibiOS/RT and some related features (USB, shell emulator, self-tests, and initializations), then incrementally including LWIP, \(\mu\)ROSnode, and the turtlesim user application. Experimental results are summarized by Table 6.1 and Figure 6.2. \(\mu\)ROSnode is only slightly bigger than the operating system; this is reasonable, since some features like XML-RPC and multithreading stuff are not that lightweight, but they are needed. Anyway, it is small enough to leave more than 70\% (\(\approx730\) kB) of the program memory unused in the worst case, and more than 85\% (\(\approx890\) kB) in the best case.

Since \(\mu\)ROSnode heavily relies on multithreading, maximum stack usage for each thread was estimated. Table 6.2 summarizes the maximum stack depth reached by the principal threads of the turtlesim demo, identified by either thread entry point or thread pool worker functions, all of which already take into account the space needed to be instantiated. All of the threads involved by XML-RPC and TCPROS thread pools do not show much diversity in memory occupation, and they never exceed 1280 B (5 \(\times\) 256 B). Indeed, if threads in a thread pool have a rather homogeneous stack usage, then the associated memory pool is exploited properly.

### Transmission performance

The transmission benchmark was run so that the board could achieve the maximum possible speed, by setting the \texttt{/benchmark\_rate} parameter to 0 (no delays between messages). Result plots are depicted in Figure 6.3. Message contents are streamed directly from a single static message (i.e. no multiple messages are allocated). \texttt{TcpRosServer} is the \(\mu\)ROSnode topic handler function, \texttt{lwp\_thread} the LWIP event dispatcher thread, and \texttt{lwp\_process} the LWIP packet processing thread.

When sending messages smaller than 100 B, the topic can reach a throughput
Figure 6.2: Footprint of the turtlesim demo in the R2P_GW program memory

of \( \approx 20000 \) msg/s. This limit is actually imposed by the performance of a rostopic process on the host. The CPU is still idle for a small amount of time, while most of the usage is due to the LWIP packet processor. The \( \mu \)ROSnode topic handler has an overhead of 30%, caused by the frequent execution of its message transmission loop.

It must be noticed that the step in the handler impact between 8 B and 16 B is caused by the allocation of the string contents; at 8 B the data string of the message is empty, thus not calling the allocator.

Around 100 B per message, the CPU usage becomes saturated. By increasing the message size, the LWIP processing thread increases its contribution to the CPU saturation almost linearly with the logarithm of the message size, reaching a peak of \( \approx 90\% \) at 2000 B. At this point, the topic handler impact is as low as 6%. The board can never reach the 100 Mb/s limit of the Ethernet connection, but it gets closer as the message size is increased.

Beyond 2000 B per message, the 100 Mb/s limit is reached. This is confirmed by the small growth of CPU idle time.

The log-log plot suggests a linear relationship between the throughput and the overhead caused by the message transmission loop of the topic handler.

Reception performance

The reception benchmarks were run by sending messages to the board at the maximum rate the host can achieve. The limit of rostopic on such machine was set to \( \approx 10000 \) msg/s. Thanks to this limit it is possible to trace graceful CPU utilization plots of the major threads involved.
Thread/pool entry point functions | Maximum stack depth [B]
---------------------------------|-------------------
lwip_thread                      | 952               |
main                             | 264               |
pub_srv__clear                   | 716               |
pub_srv__kill                    | 1100              |
pub_srv__spawn                   | 1244              |
pub_srv__turtleX__set_pen        | 740               |
pub_srv__turtleX__teleport.absolute | 764           |
pub_srv__turtleX__teleport.relative | 764            |
pub_tpc__rosout                  | 708               |
pub_tpc__turtleX__color_sensor   | 660               |
pub_tpc__turtleX__pose           | 660               |
sub.tpc__turtleX__command.velocity | 724           |
uroscNodeThread                  | 1188              |
urosRpcSlaveListenerThread       | 660               |
urosRpcSlaveServerThread        | 1004              |
urosTcpRosListenerThread         | 652               |
uosThreadPoolWorkerThread        | 240               |

Table 6.2: Stack usages of the turtlesim demo entry points on the R2P_GW working memory

TcpRosClient is the µROSnode topic handler function, lwipthread the LWIP event dispatcher thread, and lwip_process the LWIP packet processing thread.

A first benchmark buffers the entire received message upon reception, which is the default behavior of the generated topic handlers, and also the most intuitive way to process messages. This benchmark is affected by the tight memory limits of the target platform, which cannot handle a high number of buffered messages.

As shown by Figure 6.4, the board cannot handle more than 1000 B per message at 10000 msg/s. At 1000 B/s, the allocator was not able to buffer messages anymore after ≈120 seconds, because of heap fragmentation. At 2000 B/s, the allocator was able to store only 5 messages before failing. This suggests not to buffer large messages on a tightly constrained platform, but rather to process them on-the-fly whenever possible.

The board can receive a 10000 msg/s stream up to 500 B per message, still having some idle CPU time left. Because of buffered message allocations, the topic handler requires an increasing amount of CPU time, from less than 10% for small messages, to over 50% at 500 B per message.

A second benchmark was then developed so that it excludes any buffering or user processing code, by skipping the entire message contents being streamed by the host. This benchmark simulates the on-the-fly processing mentioned above. Result plots are shown in Figure 6.5.

72
The board can receive messages at 10000 msg/s up to 500 B with an idle time higher than 50%, and an handler load less than 20%. There is a linear relationship between the LWIP processor and messaging threads, while the handler thread has a slight quadratic behavior due to the fragmented reception overhead.

Between 1000 B and 2000 B the LWIP threads require \( \approx 70\% \) of the CPU time, while the topic handler stays below 30%. The CPU is at full load.

Beyond 2000 B/s the 100 Mb/s is reached, indeed the CPU idle time grows a little.
Figure 6.3: Transmission performance on R2P_PW, on-the-fly
Figure 6.4: Reception performance on R2P_GW, buffered messages
Figure 6.5: Reception performance on R2P_GW, on-the-fly
6.1.3 Raspberry Pi

The Raspberry Pi [8] (model B, shown in Figure 6.6, abbreviated RasPi) is a low-cost computer designed for educational purposes, developed by the Raspberry Pi Foundation charity. Its main component is a BCM2835 system-on-chip by Broadcom, which features an ARM1176JZF-S processor running at 700 Mhz (1 GHz boosts), and a VideoCore 4 GPU, capable of high-definition video resolutions and support for OpenGL ES2.0. It also mounts a LAN9512 PHY by SMSC, with 100 Mb/s Ethernet capabilities.

The board provides the Ethernet RJ-45 socket, two USB-HS type A ports, HDMI and composite video outputs, stereo Line headphone socket, and a SD-HC card slot. Most of the BCM2835 signals (GPIO, UART, I2C, SPI, PWM, display, camera, and so on), are exposed by a set of pin headers and Camera Interface connectors. The model used for the benchmarks mounts 256 MiB of RAM.

The operating system chosen for our experiments is Raspbian, a Debian-based Linux distribution with specific support for the Raspberry Pi. The platform is controlled through a SSH connection, which makes negligible impact on the performance. No other user software or services are running, except the SSH connection and the benchmark executables.

Instead as single threads, like for the R2P_GW benchmarks, CPU usages were collected as aggregate values from /proc/stat. The benchmark application and µROSnode run at the user level, while the network stack runs at the system and interrupt levels.

Transmission performance

Similarly to R2P_GW, the RasPi generates /benchmark/output messages at the maximum speed achievable, by not introducing forced delays. Timeouts are disabled, and a single message actually resides in memory, being streamed by the message loop. The transmission results are similar to those of R2P_GW too, as seen in Figure 6.7; as expected, the curves are biased to a higher throughput.

The platform can saturate the host rostopic receiver at ≈20000 msg/s when the message size is not greater than ≈200 B. The CPU is used less than 50%, mainly by system processes (around 25%) and the topic handler (around 15%). As the message size increases from 8 B to 200 B, the impact of (software) interrupt requests grows, but stays below 10%.

Between 200 B and 500 B per message there is a sudden increase of the
CPU usage, saturated by interrupts and system processes, which limits the throughput to \( \approx 13000 \) msg/s. The topic handler usage stays around 15\%, which means that the Linux network stack has a substantial effect in these circumstances.

With messages larger than 500 B there are no considerable changes in the CPU usage. At 10000 msg/s interrupts have a share of \( \approx 40\% \) and system calls of \( \approx 55\% \), while the topic handler uses the CPU for less than 5\%. The bandwidth gets close to 100 Mb/s, but it is still not reached at 10000 msg/s; indeed, the idle time stays around 1\% without growing.

**Reception performance**

The reception performance was measured by streaming messages published by `rostopic` at \( \approx 14000 \) msg/s, the maximum achievable by the host computer. As for R2P\_GW, the reception was first evaluated by buffering each new incoming message, and then by processing the incoming message stream by skipping its contents.

The results with message buffering are depicted by Figure 6.8. Up to 100 B per message, the platform can receive all of the messages with low effort. The CPU is idle for more than 40\% of the time, with the topic handler using less than 20\% of the CPU time, and the system calls less than 40\%. There is a strange decrease in the effect of system calls at a message size of 50 B, probably
caused by some kernel optimization. The throughput stays at the maximum.

Between 100 B and 500 B, the CPU usage of interrupts increases over 40%, and the CPU becomes saturated. The topic handler and system calls do not show significant changes in their impact. After 200 B per message, the throughput starts decreasing, but is still above 13000 msg/s.

With a message size beyond 500 B, the bandwidth is completely used. Software interrupts use the CPU at ≈10%, while the effect of system calls keeps around 35%, and that of the topic handler decreases as low as ≈10%. The idle time goes back to almost 50%.

The performance results of on-the-fly reception are shown in Figure 6.9. Below 100 B per message, the platform can receive all of the messages with low effort. The CPU is idle for ≈60% of the time, primarily used by system calls for less than 30%, and the topic handler for ≈10%, the rest by (software) interrupts. Again, there is a strange decrease in usage by system calls at a size of 50 B.

Between 100 B and 500 B per message, where the CPU usage of system calls and interrupts increases up to ≈45% and ≈35% respectively, while the topic handler stays slightly above 10%. Here the plot shows the minimum peak of the idle time, around 10%.

With a message size greater than 500 B, the bandwidth reaches the 100 Mb/s limit, and the CPU load decreases. Software interrupts are steadily below 10% as well as the topic handler, which keeps decreasing. System calls go down to ≈30%, and the idle time almost reaches 60% again.
Figure 6.7: Transmission performance on RasPi, on-the-fly
Figure 6.8: Reception performance on RasPi, buffered messages
Figure 6.9: Reception performance on RasPi, on-the-fly
6.2 Case study – Triskar2

\(\mu\)ROSnode was used to teleoperate a robot, the Triskar2 [21], by a remote ROS application. A \(\mu\)ROSnode-based board acted as a gateway from a realtime \(R2P\) [2] CAN network, connecting sensors and actuators, to an Ethernet network, which is then linked to the remote host via Wi-Fi.

The gateway translates specific messages gathered from the CAN bus to native ROS messages, and vice versa. Thanks to the code generator tool supplied with \(\mu\)ROSnode, the development of topic handlers was straightforward, needing only manual addition of few more lines in the handlers stubs, to republish messages between CAN and ROS.

6.2.1 Triskar2

Triskar2 is a highly modular, omnidirectional robot base developed by AIRLab PoliMi. Triskar2 is modular from the mechanical, the electronic, and the software points of view. Its small dimensions are suitable for indoor or cluttered environments. The robot base and the fully equipped robot are shown by Figure 6.10.

**Mechanics**

Triskar2 is a three-wheeled robot, with wheels spread at 120° on a circle, having their axes crossed at the center of the body frame. Its wheels, spun by motor modules (motor, transmission, encoder), are of omni type, to achieve three degrees of freedom.

Its kinematics diagram is depicted in Figure 6.11. The body frame, which has origin in the middle of the robot, is aligned so that the \(y\) axis points forwards, the \(x\) axis rightwards, and the \(z\) axis upwards. The angle between the \(x\) axes of body and world frames, with respect to the \(z\) axis, is \(\gamma\). Wheel angular positions are \(\theta_1\), \(\theta_2\), and \(\theta_3\). The only meaningful parameters, from the kinematics point of view, are the wheel radius, \(R\), and the distance of the wheel origin from the body frame origin, \(L\).

Direct kinematics:

\[
\begin{align*}
\dot{x} &= R \cos \left(\frac{2\pi}{6}\right) \theta_1 + R \cos \left(\frac{4\pi}{6}\right) \theta_2 - R \theta_3 \\
\dot{y} &= -R \cos \left(\frac{1\pi}{6}\right) \theta_1 + R \cos \left(\frac{5\pi}{6}\right) \theta_2 \\
L \dot{\gamma} &= -R \theta_1 - R \theta_2 - R \theta_3
\end{align*}
\]

(6.1)
Inverse kinematics:

\[
\begin{aligned}
R \theta_1 &= \cos\left(\frac{2}{3}\pi\right) \dot{x} - \cos\left(\frac{1}{3}\pi\right) \dot{y} - L \gamma \\
R \theta_2 &= \cos\left(\frac{2}{3}\pi\right) \dot{x} + \cos\left(\frac{1}{3}\pi\right) \dot{y} - L \gamma \\
R \theta_3 &= -1 \dot{x} - L \gamma
\end{aligned}
\]  \hspace{1cm} (6.2)

Electronics

Triskar2 hosts the Rapid Robot Prototyping Power [17], DC motor [15], IR [16], and Gateway [29] modules. They are connected by a CAN bus, and can communicate with ROS through the µROSnode-based firmware running on the Gateway module, which in turn is linked to the on-board computer via Ether-
Two webcams, plugged to the on-board computer, provide vision capabilities in the forwards and upwards directions. The robot optionally mounts a safety switch to shut down motors in case of emergency, and an USB-CAN debugger. Figure 6.12 displays the devices and connections for the teleoperator application.

Computing

Triskar2 is fitted with a small-form-factor computer by Zotac, based on an Intel Core i3 2330M (dual core, 2.2 GHz), equipped with 6 GB RAM and Wi-Fi connectivity. The operating system is Ubuntu 12.04, hosting a full installation of ROS Groovy.

6.2.2 Software architecture

Our goal was to develop a first-person teleoperator application, to drive the robot from remote using ROS. We exploited the R2P Gateway module to act as a bridge between the underlying CAN-based network, involving sensors and actuators, and the IP-based network, managed by ROS.
Topics

Triskar2 spans two communication domains, R2P and ROS, both working in a publish/subscribe fashion. Figure 6.13 summarizes the topics involved in the teleoperator application, highlighting their nature (R2P or ROS) and physical location (on the robot or on the remote host).

R2P  Topics on the R2P side are exchanged among board modules through a CAN bus, which resides on the robot. The teleoperator application needs the speed setpoints to be sent from the Gateway module to the motor boards via the /speed123 topic, which groups the speed setpoints of the three motors. The proximity sensor board sends the value of the four proximity sensors, one for each cardinal direction with respect to the body frame, to the gateway, via the /irraw topic.

ROS  Topics on the ROS side are exchanged locally via 100 Mb/s Ethernet or shared-memory nodelets, and remotely via Wi-Fi. The camera, mounted on the robot, streams frames to the remote host via the /image_raw topic, which uses the compressed format to save bandwidth. The uncompressed frames, republished by a local ROS node, are gathered by the teleoperator node via the /triskar/front_camera topic. Velocity setpoints, changed by remote keyboard and mouse user interactions, are sent to the Gateway module via the /triskar/velocity topic. On the other hand, the proximity values are received by the remote host from the Gateway module via the /triskar/proximity topic.

Gateway software

The R2P Gateway application translates R2P topic messages to ROS topic messages, and vice versa, performing the required mathematical transformations. Detailed listings can be found in Appendix B.4.

Teleoperator software

We created a ROS package, triskar, which provides the teleop_node ROS Node. It is written in Python, employing the pygame library [19] to draw the user interface and to manage user inputs. The teleoperator node only requires around 200 lines of code; its source code can be found in Listing B.7. This
node subscribes to the `/triskar/front_camera` topic, which streams frames captured from the front camera of Triskar2.

User inputs work like in *first person* games, which exploit a keyboard and a mouse to control the velocity of the *alter ego*; bindings are listed in Table 6.3. A minimalistic user interface shows the current *forward* (vertical green line), *strafe* (horizontal red line), and *angular* (blue arc) setpoints, as displayed in Figure 6.14. Setpoints are sent to Triskar2 through the `/triskar/velocity` topic.

Movements of Triskar2 are inhibited in those directions where proximity sensors data, read from the `/triskar/proximities` topic, reveal the presence of obstacles.

### 6.2.3 Observations

The development of the Triskar2 gateway firmware did not require much effort. Since the target platform was already supported by µROSnode, we simply included the related makefile scripts into the Makefile.
The codebase derived from the benchmark-chibios+lwip demo, supplied with µROSnode, which was integrated with the R2P middleware to control the R2P boards over a CAN bus.

The urosMsgTypes and urosHandlers software modules were regenerated by urosgen, provided with an appropriate configuration file. This task was accomplished in a few minutes.

As seen in Section 6.2.2, the development of topic handler routines was straightforward, and only required to translate messages between R2P and µROSnode (thus, ROS).

The whole firmware was completed in a few hours, including testing.
Chapter 7

Conclusions and future research directions

We were looking for a robotics software framework to program hardware modules for rapid prototyping of new robot designs. Since the state-of-the-art frameworks cannot easily blend high-level software with hardware platforms, we developed µROSnode, a software communication framework capable of integrating embedded systems with a ROS network. This way a robotics researcher, often accustomed to ROS for its simplicity and high availability of pre-made software packages, can natively connect to hardware modules running µROSnode.

Software development with µROSnode has proven to be easy, thanks to its simple software architecture, the well documented API, and the easy-to-use code generator.

Support for generic operating systems and IP network stacks, available through a lightweight abstraction layer, permits the use of µROSnode on multiple platforms, while sharing the same codebase and semantics.

The µROSnode codebase can be compiled for almost all of the embedded system platforms, by exploiting mature ANSI C89 support for such platforms. The small size of µROSnode in both program and working memories allows the development of firmware for advanced modern microcontrollers. The static stack analysis tool, supplied with µROSnode, assists the developer in optimization and control of stack usage.

Experimental results showed good performance even for a very simple platform, based on a recent microcontroller hosting an ARM-CM4 core. A platform of this type may be enough for simple specific-purpose hardware modules, where a few concurrent ROS topics are needed and message size is small. This allows the development of inexpensive robotics hardware modules, with the non-trivial
advantage of being natively connected to a ROS computation graph by an IP network. Companies making robotics devices may take advantage of µROSnode to design ROS-ready components. µROSnode may be used to create a bridge to existing hardware, or directly inside the firmware of those devices already exposing Ethernet or Wi-Fi interfaces. µROSnode may also run on devices for educational or recreational applications working with ROS.

In order to provide a more flexible, higher-level codebase, µROSnode may support the C++ language in the future, by the means of wrapper classes around the C implementation, or even by a dedicated C++ version. Wrappers can be done fast, as C++ was designed on top of C, but an entire C++ version would provide a more idiomatic codebase. The C++ version may also provide an API closer to that of roscpp, which is already known by most of the ROS users.

While TCPROS is implemented by all of the ROS nodes (it is the fallback protocol), µROSnode currently lacks UDPROS support, which better suits low-latency communication and those applications where packet drops may be tolerated. Furthermore, UDP has higher efficiency than TCP over wireless communication.
Bibliography


http://www.cmake.org/

http://www.chibios.org/

http://www.ros.org/wiki/turtlesim


[9] GNU. *GNU Compiler Collection.*
http://gcc.gnu.org/
[10] GNU. *Language Standards Supported by GCC.*
http://gcc.gnu.org/onlinedocs/gcc/Standards.html

http://beej.us/guide/bgnet/


http://github.com/openrobots-dev/R2P_DCM_module

http://github.com/openrobots-dev/R2P_IR_module

http://github.com/openrobots-dev/R2P_PS_module

http://github.com/openrobots-dev/R2P_Sonar_module

http://www.pygame.org/

http://airwiki.elet.polimi.it/index.php/TiltOne

http://airwiki.elet.polimi.it/index.php/Triskar2


http://www.ros.org/wiki/Events/CoTeSys-ROS-School

92


Appendix A

ROS-related API documentation

This appendix describes the µROSnode API, close to that of ROS, used to communicate with the Master Node, the Parameter Server, and the common Slave Nodes, via XML-RPC.

Furthermore, the µROSnode Node singleton API manages (un)registration of message types, topics, and services.

A.1 ROS Master API

A.1.1 registerService()

 Registers the caller as a provider of the specified service.

```c
uros_err_t urosRpcCallRegisterService(  
    const UrosAddr *addrp,  
    const UrosString *caller_id,  
    const UrosString *service,  
    const UrosString *service_api,  
    const UrosString *caller_api,  
    UrosRpcResponse *resp
);  
```

- **addrp** [in] Pointer to the Master API connection address.
- **caller_id** [in] ROS caller ID. Non-empty string pointer.
- **service** [in] Fully-qualified name of service. Non-empty string pointer.
- **service_api** [in] ROSRPC Service URI. Non-empty string pointer.
- **resp** [out] Pointer to the response object. Format:

  ```
  [ int code, str statusMessage, any ignore ]
  ```
Appendix A. ROS-related API documentation

A.1.2 unregisterService()

Unregisters the caller as a provider of the specified service.

```c
uros_err_t urosRpcCallUnregisterService(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    const UrosString *service,
    const UrosString *service_api,
    UrosRpcResponse *resp
);
```

- **addrp** [in] Pointer to the Master API connection address.
- **caller_id** [in] ROS caller ID. Non-empty string pointer.
- **service** [in] Fully-qualified name of service. Non-empty string pointer.
- **service_api** [in] API URI of service to unregister. Unregistration will only occur if current registration matches. Non-empty string pointer.
- **resp** [out] Pointer to the response object. Format:
  ```
  [ int code, str statusMessage, int numUnregistered ]
  ```
  Number of unregistrations (either 0 or 1). If this is zero it means that the caller was not registered as a service provider. The call still succeeds as the intended final state is reached.

A.1.3 registerSubscriber()

Subscribes the caller to the specified topic. In addition to receiving a list of current publishers, the subscriber will also receive notifications of new publishers via the publisherUpdate() API.

```c
uros_err_t urosRpcCallRegisterSubscriber(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    const UrosString *topic,
    const UrosString *topic_type,
    const UrosString *caller_api,
    UrosRpcResponse *resp
);
```

- **addrp** [in] Pointer to the Master API connection address.
- **caller_id** [in] ROS caller ID. Non-empty string pointer.
- **topic_type** [in] Datatype for topic. Must be a package-resource name, i.e. the .msg name. Non-empty string pointer.
- **caller_api** [in] API URI of subscriber to register. Will be used for new publisher notifications. Non-empty string pointer.
- **resp** [out] Pointer to the response object. Format:
  ```
  [ int code, str statusMessage, [str publisherN] ]
  ```

A.1.4 unregisterSubscriber()

Unregisters the caller as a publisher of the topic.
A.1. ROS Master API

uros_err_t urosRpcCallUnregisterSubscriber(
  const UrosAddr *addrp,
  const UrosString *caller_id,
  const UrosString *topic,
  const UrosString *caller_api,
  UrosRpcResponse *resp
);

addrp [in] Pointer to the Master API connection address.
caller_id [in] ROS caller ID. Non-empty string pointer.
caller_api [in] API URI of service to unregister. Unregistration will only occur if current registration matches. Non-empty string pointer.
resp [out] Pointer to the response object. Format:
[ int code, str statusMessage, int numUnsubscribed ]
If numUnsubscribed is zero it means that the caller was not registered as a subscriber. The call still succeeds as the intended final state is reached.

A.1.5 registerPublisher()

Registers the caller as a publisher the topic.

uros_err_t urosRpcCallRegisterPublisher(
  const UrosAddr *addrp,
  const UrosString *caller_id,
  const UrosString *topic,
  const UrosString *topic_type,
  const UrosString *caller_api,
  UrosRpcResponse *resp
);

addrp [in] Pointer to the Master API connection address.
caller_id [in] ROS caller ID. Non-empty string pointer.
topic_type [in] Datatype for topic. Must be a package-resource name, i.e. the .msg name. Non-empty string pointer.
resp [out] Pointer to the response object. Format:
[ int code, str statusMessage, [str subscriberApiN]]

A.1.6 unregisterPublisher()

Unregisters the caller as a publisher of the topic.

uros_err_t urosRpcCallUnregisterPublisher(
  const UrosAddr *addrp,
  const UrosString *caller_id,
  const UrosString *topic,
  const UrosString *caller_api,
  UrosRpcResponse *resp
);

addrp [in] Pointer to the Master API connection address.
caller_id [in] ROS caller ID. Non-empty string pointer.
call_api [in]  API URI of publisher to unregister. Unregistration will only occur if current registration matches. Non-empty string pointer.

resp [out]  Pointer to the response object. Format:

[ int code, str statusMessage, int numUnregistered ]

If numUnregistered is zero it means that the caller was not registered as a publisher. The call still succeeds as the intended final state is reached.

A.1.7 lookupNode()

Gets the XML-RPC URI of the node with the associated name. This API is for looking information about publishers and subscribers. Use lookupService() instead to lookup ROS-RPC URIs.

uros_err_t urosRpcCallLookupNode(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    const UrosString *node,
    UrosRpcResponse *resp
);

addrp [in]  Pointer to the Master API connection address.
caller_id [in]  ROS caller ID. Non-empty string pointer.
node [in]  Name of node to lookup. Non-empty string pointer.
resp [out]  Pointer to the response object. Format:

[ int code, str statusMessage, str URI ]

A.1.8 getPublishedTopics()

Gets the list of topics that can be subscribed to. This does not return topics that have no publishers.

uros_err_t urosRpcCallGetPublishedTopics(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    const UrosString *subgraph,
    UrosRpcResponse *resp
);

addrp [in]  Pointer to the Master API connection address.
caller_id [in]  ROS caller ID. Non-empty string pointer.
subgraph [in]  Restrict topic names to match within the specified subgraph. Subgraph namespace is resolved relative to the caller’s namespace. Use empty string to specify all names. Valid string pointer.
resp [out]  Pointer to the response object. Format:

[ int code, str statusMessage,[[str topicN, str typeN]]]

A.1.9 getTopicTypes()

Retrieves list topic names and their types.
A.1. ROS Master API

uros_err_t urosRpcCallGetTopicTypes(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    UrosRpcResponse *resp
);

addrp[ in] Pointer to the Master API connection address.
caller_id[ in] ROS caller ID. Non-empty string pointer.
resp[ out] Pointer to the response object. Format:
    [ int code, str statusMessage, [[str topicN, str typeN]]]

A.1.10 getSystemState()

Retrieves list representation of system state.

uros_err_t urosRpcCallGetSystemState(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    UrosRpcResponse *resp
);

addrp[ in] Pointer to the Master API connection address.
caller_id[ in] ROS caller ID. Non-empty string pointer.
resp[ out] Pointer to the response object. Format:
    [ int code, str statusMessage, 
      [ str topicN, [str topicN_publisherM] ],
      [ str topicN, [str topicN_subscriberM] ],
      [ str serviceN, [str serviceN_providerM] ] ]

A.1.11 getUri()

Gets the URI of the the Master.

uros_err_t urosRpcCallGetUri(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    UrosRpcResponse *resp
);

addrp[ in] Pointer to the Master API connection address.
caller_id[ in] ROS caller ID. Non-empty string pointer.
resp[ out] Pointer to the response object. Format:
    [ int code, str statusMessage, str masterURI ]

A.1.12 lookupService()

Looks up up all the providers of a particular service.
uros_err_t urosRpcCallLookupService(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    const UrosString *service,
    UrosRpcResponse *resp
);

addrp [in] Pointer to the Master API connection address.
caller_id [in] ROS caller ID. Non-empty string pointer.
resp [out] Pointer to the response object. Format:
    [ int code, str statusMessage, str serviceURI ]

A.2 ROS Parameter Server API

A.2.1 deleteParam()

Deletes a parameter.

uros_err_t urosRpcCallDeleteParam(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    const UrosString *key,
    UrosRpcResponse *resp
);

addrp [in] Pointer to the Master API connection address.
caller_id [in] ROS caller ID. Non-empty string pointer.
resp [out] Pointer to the response object. Format:
    [ int code, str statusMessage, any ignore ]

A.2.2 setParam()

Sets a parameter. If value is a dictionary it will be treated as a parameter tree, where key is the parameter namespace.

uros_err_t urosRpcCallSetParam(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    const UrosString *key,
    const UrosRpcParam *value,
    UrosRpcResponse *resp
);

addrp [in] Pointer to the Master API connection address.
caller_id [in] ROS caller ID. Non-empty string pointer.
value [in] Parameter value.
resp [out] Pointer to the response object. Format:
    [ int code, str statusMessage, any ignore ]
A.2. ROS Parameter Server API

A.2.3  getParam()

Retrieve a parameter value from server.

```c
uros_err_t urosRpcCallGetParam(
    const UrosAddr * addrp,
    const UrosString * caller_id,
    const UrosString * key,
    UrosRpcResponse * resp
);
```

`addrp` [in] Pointer to the Master API connection address.
`caller_id` [in] ROS caller ID. Non-empty string pointer.
`key` [in] Parameter name. If key is a namespace, it will return a parameter tree. Non-empty string pointer.
`resp` [out] Pointer to the response object. Format:

```
[ int code, str statusMessage, any parameterValue ]
```

A.2.4  searchParam()

Searches for a parameter key on the server. Search starts in caller’s namespace and proceeds upwards through parent namespaces until Parameter Server finds a matching key.

Its behavior is to search for the first partial match. For example, imagine that there are two `robot_description` parameters:

```
/robot_description

/arm

/base

/pr2
    /robot_description

/base
```

If starting in the namespace `/pr2/foo` and search for `robot_description`, searchParam() will match `/pr2/robot_description`.

If searching for `robot_description/arm` it will return `/pr2/robot_description/arm`, even though that parameter does not exist (yet).

```c
uros_err_t urosRpcCallSearchParam(
    const UrosAddr * addrp,
    const UrosString * caller_id,
    const UrosString * key,
    UrosRpcResponse * resp
);
```

`addrp` [in] Pointer to the Master API connection address.
`caller_id` [in] ROS caller ID. Non-empty string pointer.
`key` [in] Parameter name to search for. Non-empty string pointer.
`resp` [out] Pointer to the response object. Format:

```
[ int code, str statusMessage, str foundKey ]
```
A.2.5 subscribeParam()

Retrieves a parameter value from the server, and subscribes to updates to that parameter.

```c
uros_err_t urosRpcCallSubscribeParam(  
const UrosAddr *addrp,  
const UrosString *caller_id,  
const UrosString *caller_api,  
const UrosString *key,  
UrosRpcResponse *resp
);
```

- `addrp [in]` Pointer to the Master API connection address.
- `caller_id [in]` ROS caller ID. Non-empty string pointer.
- `resp [out]` Pointer to the response object. Format:
  ```c
  [ int code, str statusMessage, any parameterValue ]
  ```
  If `code` is not 1, `parameterValue` should be ignored. `parameterValue` is an empty dictionary if the parameter has not been set yet.

A.2.6 unsubscribeParam()

Retrieves a parameter value from server and subscribe to updates to that parameter.

```c
uros_err_t urosRpcCallUnsubscribeParam(  
const UrosAddr *addrp,  
const UrosString *caller_id,  
const UrosString *caller_api,  
const UrosString *key,  
UrosRpcResponse *resp
);
```

- `addrp [in]` Pointer to the Master API connection address.
- `caller_id [in]` ROS caller ID. Non-empty string pointer.
- `resp [out]` Pointer to the response object. Format:
  ```c
  [ int code, str statusMessage, int numUnsubscribed ]
  ```
  If `numUnsubscribed` is zero, it means that the caller was not subscribed to the parameter.

A.2.7 hasParam()

Checks if a parameter is stored on the server.

```c
uros_err_t urosRpcCallHasParam(  
const UrosAddr *addrp,  
const UrosString *caller_id,  
const UrosString *key,  
UrosRpcResponse *resp
);
```

- `addrp [in]` Pointer to the Master API connection address.
A.3. ROS Slave API

A.3.1 getBusStats()

Retrieves transport/topic statistics.

```c
uros_err_t urosRpcCallGetBusStats(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    UrosRpcResponse *resp);
```

- `addrp` [in] Pointer to the Master API connection address.
- `caller_id` [in] ROS caller ID. Non-empty string pointer.
- `resp` [out] Pointer to the response object. Format:
  ```c
  [ int code, str statusMessage, 
    publishStats, subscribeStats, serviceStats ]
  publishStats: [ [ topicName, messageDataSent, pubConnectionData ]... ]
  subscribeStats: [ [ topicName, subConnectionData ]... ]
  serviceStats: [ numRequests, bytesReceived, bytesSent ] (proposed)
  pubConnectionData: [ connectionID, bytesSent, numSent, connected ]*
  subConnectionData: [ connectionID, bytesReceived, dropEstimate, connected ]*
  dropEstimate is -1 for no estimate.
```

A.3.2 getBusInfo()

Retrieves transport/topic connection information.
uros_err_t urosRpcCallGetBusInfo(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    UrosRpcResponse *resp
);

addrp [in]  Pointer to the Master API connection address.
caller_id [in]  ROS caller ID. Non-empty string pointer.
resp [out]  Pointer to the response object. Format:
    [ int code, str statusMessage, busInfo ]
    [ [ connectionIdN, destinationIdN, directionN, transportN, topicN, connectedN ] ]

connectionId is defined by the node and is opaque
destinationId is the XMLRPC URI of the destination
direction is one of ‘i’, ‘o’, or ‘b’ (in, out, both)
transport is the transport type (e.g. ‘TCPROS’)
topic is the topic name
connected indicates connection status

A.3.3 getMasterUri()

Gets the URI of the Master node.

uros_err_t urosRpcCallGetMasterUri(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    UrosRpcResponse *resp
);

addrp [in]  Pointer to the Master API connection address.
caller_id [in]  ROS caller ID. Non-empty string pointer.
resp [out]  Pointer to the response object. Format:
    [ int code, str statusMessage, masterURI ]

A.3.4 shutdown()

 Stops the Slave server.

uros_err_t urosRpcCallShutdown(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    const UrosString *msg,
    UrosRpcResponse *resp
);

addrp [in]  Pointer to the Master API connection address.
caller_id [in]  ROS caller ID. Non-empty string pointer.
msg [in]  A message describing why the node is being shutdown. Valid string pointer.
resp [out]  Pointer to the response object. Format:
    [ int code, str statusMessage, any ignore ]
A.3.5 getPid()

Get the PID of the Slave server.

```c
uros_err_t urosRpcCallGetPid(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    UrosRpcResponse *resp
);
```

- `addrp` [in] Pointer to the Master API connection address.
- `caller_id` [in] ROS caller ID. Non-empty string pointer.
- `resp` [out] Pointer to the response object. Format:
  ```
  [ int code, str statusMessage, int serverProcessPID ]
  ```

A.3.6 getSubscriptions()

Retrieves a list of topics that this node subscribes to.

```c
uros_err_t urosRpcCallGetSubscriptions(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    UrosRpcResponse *resp
);
```

- `addrp` [in] Pointer to the Master API connection address.
- `caller_id` [in] ROS caller ID. Non-empty string pointer.
- `resp` [out] Pointer to the response object. Format:
  ```
  [ int code, str statusMessage, [[str topicN, str topicTypeN]] ]
  ```

A.3.7 getPublications()

Retrieves a list of topics that this node publishes.

```c
uros_err_t urosRpcCallGetPublications(
    const UrosAddr *addrp,
    const UrosString *caller_id,
    UrosRpcResponse *resp
);
```

- `addrp` [in] Pointer to the Master API connection address.
- `caller_id` [in] ROS caller ID. Non-empty string pointer.
- `resp` [out] Pointer to the response object. Format:
  ```
  [ int code, str statusMessage, [[str topicN, str topicTypeN]] ]
  ```

A.3.8 paramUpdate()

Callback from Master with updated value of subscribed parameter.
uros_err_t urosRpcCallParamUpdate{
    const UrosAddr *addrp,
    const UrosString *caller_id,
    const UrosString *parameter_key,
    const UrosRpcParam *parameter_value,
    UrosRpcResponse *resp
};

addrp [in]  Pointer to the Master API connection address.
caller_id [in]  ROS caller ID. Non-empty string pointer.
parameter_value [in]  New parameter value.
resp [out] Pointer to the response object. Format:
    [ int code, str statusMessage, any ignore ]

A.3.9 publisherUpdate()

Callback from Master of current publisher list for the specified topic.

uros_err_t urosRpcCallPublisherUpdate{
    const UrosAddr *addrp,
    const UrosString *caller_id,
    const UrosString *topic,
    const UrosRpcParamList *publishers,
    UrosRpcResponse *resp
};

addrp [in]  Pointer to the Master API connection address.
caller_id [in]  ROS caller ID. Non-empty string pointer.
publishers [in]  List of current publishers for topic in the form of XMLRPC URIs.
resp [out] Pointer to the response object. Format:
    [ int code, str statusMessage, any ignore ]

A.3.10 requestTopic()

Publisher node API method called by a subscriber node. This requests that
source allocate a channel for communication. Subscriber provides a list of
desired protocols for communication. Publisher returns the selected protocol
along with any additional params required for establishing connection. For
example, for a TCP/IP-based connection, the source node may return a port
number of TCP/IP server.

uros_err_t urosRpcCallRequestTopic{
    const UrosAddr *addrp,
    const UrosString *caller_id,
    const UrosString *topic,
    const UrosRpcParamList *protocols,
    UrosRpcResponse *resp
};

addrp [in]  Pointer to the Master API connection address.
caller_id [in]  ROS caller ID. Non-empty string pointer.
A.4. Node API

The following API are used to manage the current state of the Node singleton, by (un)registering topics, services, and parameters, as well as resolving the address of a specific topic or service publisher.

A.4.1 urosNodePublishTopic()

Issues a publishTopic() call to the XML-RPC Master.

**Warning:** The access to the topic registry is thread-safe, but delays of the XML-RPC communication will also any other threads trying to publish/unpublish any topics.

**Preconditions:**
- the topic is not published;
- the TPCORS service flag must be clear.

```c
uros_err_t urosNodePublishTopic(
    const UrosString *namep,
    const UrosString *typep,
    uros_proc_f procf,
    uros_topicflags_t flags
);
```

**Parameters:**
- **namep** [in] Pointer to the topic name string.
- **typep** [in] Pointer to the topic message type name string.
- **procf** [in] Topic handler function.
- **flags** [in] Topic flags.

A.4.2 urosNodeUnpublishTopic()

Issues an unpublishTopic() call to the XML-RPC Master.

**Warning:** The access to the topic registry is thread-safe, but delays of the XML-RPC communication will also any other threads trying to publish/unpublish any topics.

**Preconditions:**
- the topic is published.

**Postconditions:** If successful, the topic descriptor is dereferenced by the topic registry, and will be freed:
- by this function, if there are no publishing TCPROS threads; or
- by the last publishing TCPROS thread which references the topic.
uros_err_t urosNodeUnpublishTopic(
    const UrosString *namep
);

namep  [in]  Pointer to a string which names the topic.

A.4.3 urosNodeSubscribeTopic()

Issues a registerSubscriber() call to the XML-RPC Master, and connects to known publishers.

Warning: The access to the topic registry is thread-safe, but delays of the XML-RPC communication will delay also any other threads trying to subscribe/unsubscribe to any topics.

Preconditions:
- the topic is not subscribed;
- the TCPROS service flag must be clear.

Postconditions:
- connects to known publishers listed by a successful response.

uros_err_t urosNodeSubscribeTopic(
    const UrosString *namep,
    const UrosString *typep,
    uros_proc_f procf,
    uros_topicflags_t flags
);

namep  [in]  Pointer to the topic name string.

typep  [in]  Pointer to the topic message type name string.


A.4.4 urosNodeUnsubscribeTopic()

Issues an unregisterSubscriber() call to the XML-RPC Master.

Warning: The access to the topic registry is thread-safe, but delays of the XML-RPC communication will delay also any other threads trying to subscribe/unsubscribe to any topics.

Preconditions:
- the topic is published;
- the TCPROS service flag must be clear.

Postconditions: If successful, the topic descriptor is dereferenced by the topic registry, and will be freed:
- by this function, if there are no publishing TCPROS threads; or
- by the last publishing TCPROS thread which references the topic.

uros_err_t urosNodeUnsubscribeTopic(
    const UrosString *namep
);

namep  [in]  Pointer to a string which names the topic.
A.4.5  urosNodePublishService()

Issues a registerService() call to the XML-RPC Master.

**Warning:** The access to the service registry is thread-safe, but delays of the XML-RPC communication will delay also any other threads trying to subscribe/unsubscribe to any services.

**Preconditions:**
- the service is published;
- the TCPROS service flag must be clear.

**Postconditions:** If successful, the topic descriptor is dereferenced by the topic registry, and will be freed:
- by this function, if there are no publishing TCPROS threads; or
- by the last publishing TCPROS thread which references the topic.

```c
uros_err_t urosNodePublishService(
    const UrosString *namep,
    const UrosString *typep,
    uros_proc_f procf,
    uros_topicflags_t flags
);
```

- namep  [in]  Pointer to the service name string.
- typep  [in]  Pointer to the service type name string.

A.4.6  urosNodeUnpublishService()

Issues a unregisterService() call to the XML-RPC Master.

**Warning:** The access to the service registry is thread-safe, but delays of the XML-RPC communication will delay also any other threads trying to publish/unpublish any services.

**Preconditions:**
- the service is published.

**Postconditions:** If successful, the service descriptor is dereferenced by the topic registry, and will be freed:
- by this function, if there are no publishing TCPROS threads; or
- by the last publishing TCPROS thread which references the service.

```c
uros_err_t urosNodeUnpublishService(
    const UrosString *namep
);
```

- namep  [in]  Pointer to a string which names the service.

A.4.7  urosNodeCallService()

Gets the service URI from the Master node. If found, it executes the service call once, and the result is returned.
Preconditions:
- the TCPROS service flag must be set, persistent clear.

```c
uros_err_t urosNodeCallService(
    const UrosString *namep,
    const UrosString *typep,
    uros_tcpsrvcall_t callf,
    uros_topicflags_t flags,
    void *resobjp
);
```

- `namep` [in] Pointer to the service name string.
- `typep` [in] Pointer to the service type name string.
- `resobjp` [out] Pointer to the allocated response object. The service result will be written there only if the call is successful.

### A.4.8 urosNodeSubscribeParam()

Issues a `subscribeParam()` call to the XML-RPC Master, and connects to known publishers.

**Warning:** The access to the parameter registry is thread-safe, but delays of the XML-RPC communication will delay also any other threads trying to subscribe/unsubscribe to any parameters.

Preconditions:
- the parameter has not been registered yet.

```c
uros_err_t urosNodeSubscribeParam(
    const UrosString *namep
);
```

- `namep` [in] Pointer to the parameter name string.

### A.4.9 urosNodeUnsubscribeParam()

Issues an `unsubscribeParam()` call to the XML-RPC Master, and connects to known publishers.

Preconditions:
- the parameter has been registered.

Postconditions:
- if successful, the parameter descriptor is unreferenced and deleted by the parameter registry.

```c
uros_err_t urosNodeUnsubscribeParam(
    const UrosString *namep
);
```

- `namep` [in] Pointer to a string which names the parameter to be unregistered.
A.4.10 urosNodeResolveTopicPublisher()

Requests the TCPROS URI of a topic published by a node.

```c
uros_err_t urosNodeResolveTopicPublisher(
    const UrosAddr *apiaddrp,
    const UrosString *namep,
    UrosAddr *tcprosaddrp
);
```

- `apiaddrp` [in] XML-RPC API address of the target node.
- `namep` [in] Pointer to the topic name string.
- `tcprosaddrp` [in] Pointer to an allocated `UrosAddr` descriptor, which will hold the TCPROS API address of the requested topic provider.

A.4.11 urosNodeResolveServicePublisher()

Requests the TCPROS URI of a service published by a node.

```c
uros_err_t urosNodeResolveServicePublisher(
    const UrosString *namep,
    UrosAddr *pubaddrp
);
```

- `namep` [in] Pointer to the topic name string.
- `pubaddrp` [in] Pointer to an allocated `UrosAddr` descriptor, which will hold the TCPROS API address of the requested service provider.
Appendix B

Useful listings

This appendix contains miscellaneous listings, for a better understanding of XML-RPC and TCPROS protocols, as well as the µROSnode license, and some demonstration configuration files for the code generator and the static stack analysis tool.

B.1 Disclaimer

The following is the µROSnode source code disclaimer, based on a 2-clause BSD license.

Copyright (c) 2012-2013, Politecnico di Milano. All rights reserved.
Andrea Zoppi <texzk@email.it>
Martino Migliavacca <martino.migliavacca@gmail.com>
http://airlab.elet.polimi.it/
http://www.openrobots.com/
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Listing B.1: Source code disclaimer of µROSnode
Appendix B. Useful listings

B.2 XML-RPC grammar

No official grammar specification exists, but a Document Type Definition (DTD) and an XML Schema (XSD) can be outlined, although none of them can synthesize all of the XML-RPC grammar rules.

B.2.1 Document Type Definition

```xml
<!DOCTYPE methodCall PUBLIC "-//Sun Microsystems, Inc.//DTD XML-RPC//EN"
    "http://java.sun.com/xml/rpc.dtd">

<!ELEMENT methodCall (methodName, params)>
<!ELEMENT methodName (#PCDATA)>
<!ELEMENT params (param)>
<!ELEMENT param (value)>
<!ELEMENT value (i4 | int | string | double | dateTime.iso8601 | base64 | struct | array)>

<!ELEMENT i4 (#PCDATA)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT dateTime.iso8601 (#PCDATA)>
<!ELEMENT double (#PCDATA)>
<!ELEMENT base64 (#PCDATA)>
<!ELEMENT array (data)>
<!ELEMENT data (value)*>
<!ELEMENT struct (member)+>
<!ELEMENT member (name, value)>
<!ELEMENT name (#PCDATA)>

<!ELEMENT methodResponse (params | fault)>
<!ELEMENT fault (value)>
```

Listing B.2: Document Type Definition of XML-RPC (unofficial)

B.2.2 XML Schema

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="methodCall">
    <xs:complexType>
      <xs:all>
        <xs:element name="methodName" type="string"/>
        <xs:element name="params" type="ParamType" minOccurs="0" maxOccurs="unbounded"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="methodResponse">
    <xs:complexType>
      <xs:all>
        <xs:element name="params" type="ParamType" minOccurs="0" maxOccurs="1"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing B.2: XML Schema of XML-RPC (unofficial)
150 as "&" and "<", respectively.

- strings can contain binary data, except for('&' and '<', which are encoded
- a value element can contain either an ASCII string or a type element such
- one of the two members in a fault struct must have the name faultCode and
- does not have mixed content.

I need to figure out how to say that this is either a simple xsd:string

Listing B.3: XML Schema of XML-RPC (unofficial)
B.3 TCPROS syntax

The following is an unofficial EBNF syntax [14] of TCPROS streams.

```
(* Unofficial TCPROS EBNF syntax *)
length = ? little-endian uint32_t ?;
text = { ? printable characters ? };
string = length, text;
header = length, { eq_field };
eq_field = length, text, '='; text;
stream = tpc_stream | srv_stream; (* root symbol *)
(* Topic stream *)
tpc_stream = header, { message };
message = length, contents;
contents = ? defined by the message descriptor ?;
(* Service stream *)
srv_stream = header, { transaction };
transaction = request, ok_byte, response;
request = message;
ok_byte = ? uint8_t ?;
response = message | string;
```

Listing B.4: Unofficial EBNF syntax of TCPROS streams

B.4 Case study

This section contains some useful listings extracted from the Triskar2 teleoperator application.

B.4.1 Triskar2 message types

The following are the types for velocity and proximity messages of Triskar2.

```
float32 strafe  # Strafe speed (X axis, m/s)
float32 forward # Forward speed (Y axis, m/s)
float32 angular # Angular speed (Z axis, rad/s)
```

Listing B.5: Contents of triskar/msg/Velocity.msg

```
uint32 NUM_SENSORS = 4  # Number of proximity sensors
uint32 EAST = 0  # East index
uint32 NORTH = 1  # North index
uint32 WEST = 2  # West index
uint32 SOUTH = 3  # South index
float32[4] proximities  # Normalized proximities (1.0 far, 0.0 near)
```

Listing B.6: Contents of triskar/msg/Proximity.msg
B.4. Case study

B.4.2 Triskar2 teleoperator node

The following is the complete source code of the teleoperator ROS Node, used to control Triskar2 from remote the first person way.
Listing B.7: Source code of the Triskar2 teleoperator node (triskar/teleop_node)

```python
101  mod = pygame.key.get_mods();
102  gear = int(bool(mods & KMOD_SHIFT))
103  if keys[K_d]: sp_IMPULSE_SPEED[gear] = 1
104  if keys[K_a]: sp_IMPULSE_SPEED[gear] = -1
105  if keys[K_w]: sp_IMPULSE_SPEED[gear] = 1
106  if keys[K_s]: sp_IMPULSE_SPEED[gear] = -1
107  if keys[K_l]: sp_IMPULSE_SPEED[gear] = 1
108  if keys[K_j]: sp_IMPULSE_SPEED[gear] = -1
109  if keys[K_k]: sp_IMPULSE_SPEED[gear] = 1
110  if keys[K_i]: sp_IMPULSE_SPEED[gear] = -1
111  # Apply mouse movements (either accumulated motion or impulses)
112  for i in range(3): sp_accum[i] = sp_temp[i]
113  else:
114      for i in range(3): sp_impulse[i] = sp_temp[i]
115      # Block movements towards constrained directions
116      if proximity[WEST] > PROXIMITY_THRESHOLD[WEST]:
117          sp_accum[0] = max(0.0, sp_accum[0])
118          sp_impulse[0] = max(0.0, sp_impulse[0])
119      if proximity[EURTH] > PROXIMITY_THRESHOLD[EURTH]:
120          sp_accum[1] = max(0.0, sp_accum[1])
121          sp_impulse[1] = max(0.0, sp_impulse[1])
122      if proximity[SOUTH] > PROXIMITY_THRESHOLD[SOUTH]:
123          sp_accum[2] = max(0.0, sp_accum[2])
124          sp_impulse[2] = max(0.0, sp_impulse[2])
125  sp_temp = sp_accum + sp_impulse
126  sp_temp[0] = max(0.0, sp_temp[0])
127  sp_temp[1] = max(0.0, sp_temp[1])
128  sp_temp[2] = max(0.0, sp_temp[2])
129
130  def draw_circle_arc(surface, color, pos, radius, start, stop, width=6):  # Draws a centered arc
131      c = screen.get_rect()
132      r = c[0] - radius, c[1] - radius, (c[0] + radius, c[1] + radius)
133      if start > stop: start_, stop = stop, start
134      pygame.draw.arc(surface, 0x00FF00, r, start, stop, width)
135
136  def render_scenescreen, setpos):  # Renders the current screen frame
137      screen.fill([0x000000])
138      screen.blit(camera_frame, (0, 0))
139      c = screen.get_size()
140      r = (c[0] / 2, c[1] / 2)
141      for i in range(4):
142          pygame.draw.circle(screen, 0x00FF00, (r[0] + i * 16, r[1]), 3)
143          pygame.draw.circle(screen, 0xFF0000, (r[0] + i * 16, r[1]), 2)
144  draw_circle_arc(screen, 0x00FF00, r, math.pi / 2, math.pi / 2 + setpos[2], 2)
145  draw_circle_arc(screen, 0xFF0000, r, math.pi / 2 + setpos[2], 2)
146
147  def main():
148      # Initialize pygame stuff
149      global screen, camera_frame
150      pygame.init()
151      camera_frame = pygame.display.set_mode((480, 360))
152      camera_frame.fill((32, 32, 32))
153      pygame.display.flip()
154      # Initialize ROS stuff
155      rospy.init_node('teleop_node')
156      pubVelocity = rospy.Publisher('Triskar/velocity', Velocity)
157      pubVelocity.publish(Velocity(0.0, 0.0, 0.0))
158      rospy.Subscriber('triskar/front/camera', Image, camera_frame_cb)
159      rospy.Subscriber('triskar/proximity', Proximity, proximity_cb)
160      # Initialize setpoints
161      sp_impulse = [0.0, 0.0, 0.0]
162      sp_accum = [0.0, 0.0, 0.0]
163      sp_smooth = SmoothedVec3(EuFilter(1.0 / 60.0, 1.0, 0.000001), sp_accum)
164      # Process inputs
165      handle_inputs(sp_accum, sp_impulse)
166      if not rospy.is_shutdown(): break
167      sp = sp_smooth.process([sp_impulse[i] + sp_accum[i] for i in range(3)])
168      pubVelocity.publish(Velocity(sp[0], sp[1], sp[2]))
169  main()
```

118 Appendix B. Useful listings
B.4. Case study

B.4.3 Triskar2 urosgen configuration

The following is the urosgen configuration for the Triskar2 µROSnode-based firmware.

```python
# urosgen.py configuration file for Triskar2

[Options]
author = Andrea Zoppi <texzk@email.it>
licenseFile = ../../../COPYING
includeDir = ../include
sourceDir = ../src
nodeName = triskar_node
fieldComments = false
msgOnStack = true
inOnStack = true
outOnStack = true

[PubTopics]
/triskar/proximity = triskar/Proximity

[SubTopics]
/triskar/velocity = triskar/Velocity

[PubServices]

[CallServices]
```

Listing B.8: urosgen configuration file for the Triskar2 µROSnode-based firmware

B.4.4 Triskar2 gateway handler routines

The following are the topic handlers for the Triskar2 teleoperator application, which translate R2P messages to ROS messages, and vice versa.

```c
uros_err_t sub_tpc__triskar__velocity(UrosTcpRosStatus *tcpstp) {
    UROS_TPC_INIT_S(msg__triskar__Velocity);
    if ((tcpstp = &msg__triskar__Velocity) != NULL) {
        /* Message allocation and initialization.*/
        UROS_MSG_RECV_LENGTH();
        UROS_MSG_RECV_BODY(&msg, msg__triskar__Velocity);
        /* Republish the received ROS message to R2P DC motor modules.*/
        r2p_msg = lpubSpeeds.alloc();
        if (r2p_msg != NULL) {
            velocity_to_setpoints(msg, *r2p_msg);
            lpubSpeeds.broadcast(r2p_msg);
            palTogglePad(GPIOC, GPIOC_LED2);
        }
        /* Dispose the contents of the message.*/
        clean_msg__triskar__Velocity(&msg);
    }
    tcpstp->err = UROS_OK;

    finally:
    /* Message deinitialization and deallocation.*/
    UROS_TPC_UNINIT_S(msg__triskar__Velocity);

    return tcpstp->err;
}
```

Listing B.9: Publication handler function for the /triskar/velocity ROS topic
uros_err_t pub_tpc__triskar__proximity(UrosTcpRosStatus *tcpstp) {
    /* Message allocation and initialization.*/
    UROS_TPC_INIT_S(msg__triskar__Proximity);
    IRRaw *r2p_msg;
    Subscriber<IRRaw, 8> lsubProximity("IRRaw");
    Node mwNode("mwNode");
    Middleware::instance().newNode(&mwNode);
    mwNode.subscribe(&lsubProximity);

    /* Published messages loop.*/
    while (!urosTcpRosStatusCheckExit(tcpstp)) {
        mwNode.spin(_MS2ST(1000));
        r2p.msg = lsubProximity.get();
        if (r2p.msg != NULL) {
            irraw_to_proximities(*r2p.msg, msg);
            lsubProximity.release(r2p.msg);
            /* Send the message.*/
            UROS_MSG_SEND_LENGTH(&msg, msg__triskar__Proximity);
            UROS_MSG_SEND_BODY(&msg, msg__triskar__Proximity);
            /* Dispose the contents of the message.*/
            clean_msg__triskar__Proximity(&msg);
        }
        tcpstp->err = UROS_OK;
    }
    _finally:
        /* Message deinitialization and deallocation.*/
        UROS_TPC_UNINIT_S(msg__triskar__Proximity);
        return tcpstp->err;
}

Listing B.10: Subscription handler function for the /triskar/proximity ROS topic
B.5 Code generator configuration demo

The following is a demonstration configuration file for the code generator, urosgen.py. All of the possible options are listed and commented. The chosen data types involve all of the code generator features.

```python
# urosgen.py configuration file, for demonstration purposes.
# This is the options section. All the options are assigned below. An undefined
# option is assigned its default value. Run urosgen.py without arguments to see
# their default values.

[Options]

# Author of the generated files
author = Your Name <your.email@example.com>

# Optional license text file to comment at the beginning of generated files
licenseFile = ../../../COPYING

# Name of the generated node
nodeName = uros_demo_node

# Paths of the generated files, relative to this configuration file
includeDir =.
sourceDir =.

# File names for "<filename>.h" and "<filename>.c" generation
msgTypesFilename = urosMsgTypes
handlersFilename = urosHandlers

# Generation switches
genMsgTypesHeader = true
genMsgTypesSource = true
genHandlersHeader = true
genHandlersSource = true

# Allocates the message/service values on the stack instead of into the heap
msgOnStack = false
inOnStack = false
outOnStack = true

# Base name of handler variables. Depending on the stack/heap options, they
# will be declared as: [*namep = NULL] if in heap, [name] if on stack.
msgVarBaseName = msg
inVarBaseName = inmsg
outVarBaseName = outmsg

# Generates comments above the declaration of each structure field
fieldComments = true

# Generated function names
regTypesFuncName = urosMsgTypesRegStaticTypes
regPubTopicsFuncName = urosHandlersPublishTopics
unregPubTopicsFuncName = urosHandlersUnpublishTopics
regSubTopicsFuncName = urosHandlersSubscribeTopics
unregSubTopicsFuncName = urosHandlersUnsubscribeTopics
regPubServicesFuncName = urosHandlersPublishServices
unregPubServicesFuncName = urosHandlersUnpublishServices

[PubTopics]
output = stereo_msgs/DisparityImage # 3 levels deep type
rosout = rosgraph_msgs/Log # A common type

[SubTopics]
input = stereo_msgs/DisparityImage # Just to check
constants = bond/Constants # This type defines only constants
rosin = rosgraph_msgs/Log # Just to check

[PubServices]
reconfigure = dynamic_reconfigure/Reconfigure # 3 levels deep type

[CallServices]
reconfigure = dynamic_reconfigure/Reconfigure # Just to check
```

Listing B.11: Documented demo configuration file for urosgen.py, with complex types
### Appendix B. Useful listings

#### B.6 Stack usage analyzer configuration demo

The following is the configuration file for the static stack analysis tool, urosstan.py, used to analyze the turtlesim [7] demonstration project for the R2P_GW [29] project.

```
| # urosstan.py demo configuration file for turtlesim
| # List of options for urosstan.py

[Options]
# List of source units to be analyzed, in the form:
# <codefile> = <usage>
# where <codefile> is unmangled (see GKD/SU) and globally defined

[SourceUnits]
# where paths are relative to the Makefile

[EntryPoints]
# List of starting functions, in the form: <function> = <usage_bias>

[Terminators]
# Known graph leafs (terminators), in the form: <function> = <usage>
```

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

import os

# Output directory (must exist), relative to this configuration file
outDir = '../build/obj

# Reference Makefile location, relative to this configuration file
makefileDir = './build/obj

# Project source directories

# List of starting functions, in the form: <function> = <usage_bias>

# Known graph leafs (terminators), in the form: <function> = <usage>
```
Listing B.12: Documented demo configuration file for urosstan.py, turtlesim running on ChibiOS/RT

```

ChibiOS/RT

Listing B.12: Documented demo configuration file for urosstan.py, turtlesim running on ChibiOS/RT

```

124 Appendix B. Useful listings