João Pedro Carvalho Rosa

CloudRobotics - Distributed Robotics using Cloud Computing

July 2016

· U C ·

Universidade de Coimbra

FCTUC FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

CloudRobotics – Distributed Robotics using Cloud Computing

João Pedro Carvalho Rosa

Coimbra, July 2016

· U C ·

CloudRobotics – Distributed Robotics using Cloud Computing

Supervisor:
Prof. Doutor Rui Paulo Pinto da Rocha

Jury:
Prof. Doutor Luís Alberto da Silva Cruz (Chair)
Prof. Doutor Manuel Fernando dos Santos Silva
Prof. Doutor Rui Paulo Pinto da Rocha

Dissertation submitted in partial fulfillment for the degree of Master of Science in Electrical and Computer Engineering.

Coimbra, July 2016

Acknowledgements

I would like to use this page to thank you.

If you are Professor Rui Rocha, thank you for all the support, guidance and for always being available to discuss any detail about this dissertation.

If you are Daniel Marcelino, Diogo Amorim, Rui Pires or Daniela Nobre, thank you for sharing your journey with my own.

If you are Gonçalo Martins, thank you for always being available to lend a helping hand.

If you are my family, Matias, Zé Marques, Bruna, Ricardo, Duarte or Girão, thank you for being a part of my everyday life over these past years.

If you are Diana, a big thank you for being my other half.

If you are a reader, thank you for bringing a little more usefulness to this work.

Resumo

A Computação em Nuvem é uma mudança de paradigma que tem ganho força ao longo dos últimos anos, sendo suportada pelo aumento da disponibilidade, omnipresença e fiabilidade das ligações sem fios à Internet. A Computação em Nuvem permite o acesso a recursos computacionais aparentemente ilimitados e localizados num agrupamento de computadores externos (a Nuvem). Em contrapartida, alguns robôs, como por exemplo *drones*, têm requisitos de mobilidade, tais como um tamanho/peso máximo ou uma autonomia mínima, e transportar mais recursos computacionais a bordo significa prejudicar estes requisitos.

Este princípio pode ser importado para o campo de Robótica, dando origem ao nome Robótica em Nuvem. Neste caso, o objetivo é permitir que robôs consigam executar tarefas que não seriam capazes de executar em circunstâncias normais e/ou libertar recursos computacionais a bordo, de modo a que mais tarefas ou tarefas mais complexas possam ser executadas ao mesmo tempo por um robô móvel. Há muitas tarefas robóticas que podem tirar proveito de poder de processamento massivo e armazenamento, tais como mapeamento e localização simultâneos (SLAM), navegação, processamento de imagem, interação humano-robô e aprendizagem. Todas estas tarefas podem esgotar rapidamente os recursos computacionais de um robô, especialmente se algumas delas forem executadas simultâneamente.

No entanto, para estabelecer uma ligação e exportar dados para a Nuvem é necessária alguma largura de banda, tornando assim o sistema num compromisso: por um lado, são libertados carga computacional e espaço de armazenamento, por outro lado é colocada maior pressão sobre o uso da rede sem fios. Esta dissertação tem como objetivo analisar este compromisso, adaptando duas tarefas multi-robô existentes, que operam sobre o Robot Operating System (ROS), e comparar a abordagem baseada em Nuvem com o sistema tradicional.

Para validar as capacidades dos sistemas robóticos baseados na nuvem, foram realizadas tanto simulações como experiências com robôs reais. Os resultados de simulação mostram um claro ganho no tempo de CPU, enquanto que os testes com robôs reais confirmam que os resultados das tarefas permanecem inalterados. Apesar dos sistemas baseados na Nuvem exigirem muito maior largura de banda, um moderno *Wi-Fi router* consegue fornecer o suficiente para suportar qualquer equipa realista de robôs.

Palavras-chave: Computação em Nuvem, Robótica em Nuvem, Tecnologia Sem Fios, ROS, Recursos Computacionais, Largura de Banda.

Abstract

Cloud Computing is a paradigm shift in computation that has been gaining traction over the recent years, which is supported by the increasing availability and ubiquity of a reliable wireless connection to the Internet. Cloud Computing enables the access to seemingly unlimited computer resources that are located on an external computer cluster (the Cloud). In contrast, some robots, *e.g.* drones, have mobility requirements such as maximum size/weight or minimum autonomy, and carrying more onboard computer resources usually means hindering these requirements.

This principle can be brought to the field of Robotics hence the name Cloud Robotics. In this case, the goal is to allow robots to perform tasks they would not be able to under normal circumstances and/or to free onboard resources so that more tasks or more complex tasks can be executed at the same time by a mobile robot. There are many existing robotic tasks that can take advantage of massive processing power and storage, such as simultaneous localization and mapping (SLAM), navigation and trajectory planning, image processing, pattern recognition, human-robot interaction and machine learning to name a few. All of these can quickly drain the robot out of its computer resources, especially if some of these tasks are running at the same time.

However, in order to access and export data to the Cloud some bandwidth is needed, thus making the system a tradeoff: on the one hand, computation load and storage space is being freed, while on the other hand more strain is being put on the wireless network usage. As wireless connection protocols become more and more powerful, a Cloud-based solution becomes more interesting. This dissertation aims to analyse this tradeoff by adapting two existing multi-robot tasks, working on the Robotic Operating System (ROS), and compare the Cloud-based approach to the traditional one.

To validate the capabilities of Cloud-based robotic systems, both simulations and experiments with real robots were conducted. Simulation results show a clear gain in CPU time, while the latter confirms the outcome of the tasks remains the same. Despite the Cloud-based systems, requiring considerably more bandwidth, a modern off-the-shelf Wi-Fi router can provide with enough to support any realistic team of robots.

Keywords: Cloud Computing, Cloud Robotics, Wireless Technology, ROS, Computer Resources, Bandwidth.

Contents

Acknowledgements **ii**

Resumo **iii**

Abstract **iv**

List of Acronyms **viii**

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

1.1 Context and Motivation 2

1.2 Objectives 3

1.3 Outline of the dissertation 4

2 Background and Related Work **5**

2.1 Cloud Robotics 5

2.1.1 What is Cloud computing? 5

2.1.2 Service Models using classic cloud infrastructures 6

2.1.3 Service Models providing robotic services 6

2.1.4 Challenges for Cloud Robotics 8

2.1.5 Cloud Robotics Architectures 8

2.1.6 Comparison between the selected frameworks 15

2.2 SLAM 18

2.2.1 Single robot SLAM 18

2.2.2 Multi-Robot SLAM 19

2.3 Multi-Robot Patrolling 20

2.3.1 Overview of the Multi-Robot patrolling problem 20

2.3.2 Related work 21

2.4 ROS 23

2.5 Summary 25

3 Cloud Robotics using Rapyuta 27

3.1 Setting up Rapyuta's Cloud environment 27

3.2 Summary 31

4 Two Use Cases of Rapyuta 33

4.1 Use Case 1: Multi-Robot SLAM 33

4.1.1 The *mrgs* ROS stack 33

4.1.2 Contribution to the improvement of the *mrgs* ROS stack 35

4.1.3 Adaptation of the *mrgs* ROS stack to Rapyuta Cloud environment . . 37

4.2 Use Case 2: Multi-Robot Patrolling 39

4.2.1 The patrol_isr_demo ROS package 39

4.2.2 Adaptation of the *patrol_isr_demo* ROS package to Rapyuta Cloud environment 41

4.3 Summary 42

5 Results and Discussion 43

5.1 Multi-Robot SLAM Simulation 44

5.2 TSP Patrol Simulation 48

5.3 Experiments with real robots 51

5.4 Summary 54

6 Conclusion 55

6.1 Future work 55

7 Bibliography 57

List of Acronyms

AP	Access Point
AP4ISR	Artificial Perception for Intelligent Systems and Robotics
CPU	Central Processing Unit
HDFS	Hadoop Distributed File System
IP	Internet protocol
ISR	Institute of Systems and Robotics
LAN	Local Area Network
MAS	Multi-Agent Systems
MR-SLAM	Multi-Robot Simultaneous Localization and Mapping
PNG	Portable Network Graphics
ROS	Robot Operating System
SLAM	Simultaneous Localization and Mapping
TCP	Transmission Control Protocol
TSP	Travelling Salesman Problem
VM	Virtual Machine
WAN	Wide Area Network

List of Figures

2.1 Overview of the DAvinCi architecture. Figure reproduced from [20]. 9
2.2 Communications in the Rapyuta framework. Figure reproduced from [44]. . . 11
2.3 Overview of the REALcloud architecture. Figure reproduced from [15]. . . . 12
2.4 Overview of the Robot-Cloud architecture. Figure reproduced from [29]. . . 13
2.5 Overview of the RoCon *multimaster* framework. Figure reproduced from [10]. 14
2.6 Comparison between grid-based map (left) and a topological map (right). Figure reproduced from [58]. 19

3.1 Cloud setup using Rapyuta. The dashed box represents optional task sets. . 28
3.2 Process of connecting to the Cloud. 29
3.3 Comparison between a *single* and *multimaster* systems interacting with Rapyuta's Cloud environment. In both systems two robots are simulated in stage [13] and robot_1 mimics the movemt of robot_0 in reverse. 30

4.1 Interaction between the nodes and topics of the *mrgs* stack. 33
4.2 Comparison between the occupancy grid output by *gmapping* (right) and the one output by *map_dam_node* (left). While both occupancy grids contain the same amount of useful information, the one on the left is trimmed down to reduce the total size. 34
4.3 Representation of the tree-like process of map merging. 35
4.4 Failed merging process of three similar local maps. 36
4.5 ROS environment of a robot that is running the *mrgs* stack on Rapyuta. . . 37
4.6 Comparison between distributed (left) and centralized (right) modes of operation under Rapyuta's Cloud environment. The boxes labeled with LXC (Linux Containers) represent containers running on Cloud machines. 38
4.7 Interaction between the nodes and topics of the *patrol_isr_demo* package. . 40
4.8 The *patrol_isr_demo* package and the navigation stack nodes running on Rapyuta's Cloud environment. The boxes labeled with LXC (Linux Containers) represent containers running on Cloud machines. 41

5.1 Global map generated by the merging process of twenty different local maps. 45
5.2 Box plot of the CPU time used by the Rapyuta-adapted *mrgs* system from the point of view of one robot. 46

5.3 Box plot of the CPU time used by the traditional *mrgs* system from the point of view of one robot. 46
5.4 Box plot of the bandwidth used by the Rapyuta-adapted *mrgs* system from the point of view of one robot. 47
5.5 This figure represents a successful experiment with 20 simulated robots patrolling a TSP generated trajectory. The considered vertexes are placed in a grid-like fashion, where each square is surrounded by four vertexes to a total of one hundred. 49
5.6 Environment used while testing the impact of obstacles in the usage of CPU time. 50
5.7 Box plot of the bandwidth used by the Rapyuta-adapted *patrol_isr_demo* system from the point of view of one robot. 50
5.8 Experiment with real robots running the Rapyuta-adapted *mrgs* system. . . 51
5.9 Experiment of a real robot running the Rapyuta-adapted *patrol_isr_demo* system. 52
5.10 CPU time registered during the experiments of multi-robot SLAM with three real robots. 53
5.11 CPU time registered during the experiments of multi-robot patrol with a real robot. 53

List of Tables

2.1 Pros and Cons of the selected frameworks. 15

5.1 Comparison between the CPU time used by the *mrgs* stack on the traditional and Rapyuta-adapted systems from the point of view of one robot. 45

5.2 Bandwidth required by one robot running the Rapyuta-adapted *mrgs* system. 47

5.3 Size of the topic exchanged among robots running the traditional *mrgs* system. 48

5.4 Comparison between the CPU time used by the *patrol_isr_demo* stack on the traditional and Rapyuta-adapted systems from the point of view of one robot. 49

5.5 Bandwidth required by one robot running the Rapyuta-adapted *patrol_isr_demo* system. 50

1 Introduction

Research and development over the years has sprouted a vast array of tasks able to be executed by robots. Some of these tasks though have to be carried out by a team of robots rather than a single one. A multi-robot system can benefit from several advantages over a single robot one, especially in terms of performance to cost ratio and higher task accomplishment speed, as well as eliminating the single point of failure that is a single robot system [40].

Although the advantages of having a team of robots are trivial, its restraints are only evident when considering practical implementations, namely cost restraints. It becomes obvious that a team of robots has to be composed of multiple cheaper robots, meaning machines with less processing power, but also easier to build, work with, repair and replace. This creates the need for interaction among robots in order to solve complex problems and run computationally heavy algorithms.

There are already example cases, like the Kiva Systems (now called Amazon Robotics) [1], where the sharing of knowledge has been used to create a shared world model for better autonomous navigation; and like the RoboCup [38], where sensor information is shared among robot team members in order to allow for a more reliable dynamic object tracking. Knowledge sharing has also been used for localization services, scene recognition and robotic manipulation despite there being relatively little research on this topic [60].

In order to tackle the problem of communication between the elements of a multi-robot system, two main areas of study have been in research recently: Cloud Computing [37] and Robotic Clusters [40]. This dissertation explores a cloud-based solution to multi-robot systems.

Cloud Computing is a new and emerging paradigm that takes advantage of network connection availability as a resource for parallel processing and data storage and sharing [43]. This Cloud infrastructure can not only provide the means for a robot which needs external data to support its operation but also opens a myriad of ways to interact and cooperate with many other robotic systems connected via Cloud.

Cloud-based systems have already proven useful for robotic applications, as is de case of the cloud-inspired framework named Distributed Agents with Collective Intelligence (DAvinCi) [20], which uses a team of robot as sensing nodes of a main frame that does all the necessary computing [33]. Google's object recognition system is another noteworthy example, which uses a massive cloud database of images and textual labels in order to facilitate object recognition [37].

On the other hand, Robotic Clusters rely on each other, rather than on the Cloud, for parallel processing. Each member of the Cluster shares his available processing power with the remaining, thus enabling a clever usage of resources as a whole. In contrast with Cloud-based architectures, Clusters function as ad hoc networks, thus trading connection to a wide area network with the availability of a local area wireless network. Robotic Clusters have shown to improve performance over traditional architectures as demonstrated in [40] and [33], for the cases of SLAM (Simultaneous Localization and Mapping) and map merging respectively.

1.1 Context and Motivation

The motivation to develop this dissertation emerged from the reduced processing power of a fleet of smaller robots in the Institute of Systems and Robotics (ISR) [18] which still needed to run computationally heavy algorithms. A cloud-based solution was therefore proposed. However, having a mobile robot team being run mostly by a remote server (the Cloud) opens up a vast doorway of opportunities more complex than a mere master-slave relationship.

A robotic system connected to the Internet has available to it large databases and server farms for parallel processing. The use of these external processing units allows for onboard computers to have less power consumption while allowing for a reduced processing delay due to parallelization of tasks. These power savings are more relevant on smaller robots where size, weight and power autonomy are of utmost importance. These savings (size, weight and power) are also valid for data storage, while having more storage space remotely than it would be possible onboard. The access to vast databases offers its own multiple opportunities. Machine learning and artificial intelligence (AI) techniques greatly benefit from huge data repositories and the biggest gain most likely lies on this particular field of robotics [27]. Going even further, robot systems could turn to cloud-based knowledge to expand the capabilities they were initially designed to have.

An always-on connection to the Internet does carry some concerns, namely privacy and security. The potential of remote hacking cannot be overlooked as damage to people or infrastructures is a sensitive matter. On an industrial context, corporate secrecy or production disruption could also be criminally targeted.

Decentralized architectures have proven many advantages, such as fault tolerance and reliability, on distributed computer systems, which would also benefit multi-robot systems. One noteworthy characteristic of such architectures in our context is the high level of abstraction provided by the Cloud: there is no need for the robot to know how the processing is done on the server farms, how data is stored and retrieved or even what software or hardware the Cloud uses; only a solid communication protocol is required. Consequently, on the robot end, there is no need to take into account server maintenance, outages, and software or hardware updates.

Nowadays, embedded wireless LAN adapters are common in small robotic processing boards, thus the strong communication required can be achieved with any of the existing and proven wireless architectures and protocols, such as the ubiquitous Wi-Fi technology. Over the recent years, the general public has seen a large increase of the availability of remote computation on demand, as well as remote storage systems. Open source software has also become widely available and accepted in the Robotics field, being ROS (Robot Operating System) the prime example [53]. ROS is a set of tools and libraries that help to build robot applications and is used by almost all robot developers. It is also run on Linux, an example of open an source operating system. This high level of availability and existing trends in other computer systems provide a favorable setting for the development of Cloud-based solutions for multi-robot systems.

So far the advantages of a Cloud-based solution and why it is the right time to take such approach have been discussed. However, such solution has its pitfalls. The most relevant being the need for communication itself. While remote processing and storage can greatly increase the scalability of the approach, the communication line provides a possible bottleneck and a barrier to such scalability, when wireless communications are required. Traffic created by large teams of robots can choke the wireless network, especially the more dependant the robots are on the remote servers. Communication latency and overhead also pose a problem to tackle, since performance degrades when such overhead is not negligible relatively to computation time. Some tasks are also bound by hard real-time constraints, such tasks cannot be held back by network traffic risking failure. The communication line also presents itself as a critical point of failure, and how the whole system deals with such event is not a trivial matter. These setbacks impose some tasks to be mandatorily run locally rather than remotely, which shortens the range of applicable scenarios.

Overall it is difficult to predict the benefit of a Cloud-based system. Even tasks that require a large amount of data, having more information being fed remotely does not result necessarily into better performance [60]. This coupled with other initiatives that look towards the future, such as the *Internet of Things* [21], serves as additional motivation for this dissertation.

1.2 Objectives

The main objective of this work is to develop an application based on ROS that resorts to cloud computing in order to enable robots with lackluster computational resources to perform computationally heavy tasks. The developed software will be tested in two different multi-robot tasks, in order to validate its usefulness: cooperative simultaneous localization and mapping [42], and cooperative patrolling [49]. After the validation process, the objectives are to study the scalability and network traffic of the application, as well as carry out experiments with real robots.

1.3 Outline of the dissertation

This dissertation is organized into five chapters:

– Chapter 1 gives an introduction to the theme and the motivation behind studying it.

– Chapter 2 presents a study on the existing related work covered by this dissertation.

– Chapter 3 gives a more in-depth look on how to work with the chosen Cloud framework.

– Chapter 4 introduces the systems used as the basis for this dissertation and presents the adaptations to the Cloud environment employed during the development of our work.

– Chapter 5 presents the experiments and their results and analyses and discusses them.

– Chapter 6 gives a reflection on the adaptation to the Cloud and the vision for a possible future complimentary work.

2 Background and Related Work

2.1 Cloud Robotics

2.1.1 What is Cloud computing?

Cloud computing has been defined as being a model for providing remote access, on demand, to a pool of computing resources. It refers to both hardware and software systems delivered or made available primarily over the Internet, being servers, storage systems, applications or services examples of Cloud computing [43].

Associated with this model is the term Cloud, which refers to the data centre that provides such service. We can then segregate this concept into Public Cloud when a Cloud is available to anyone with Internet access, and a Private Cloud when access is restricted to members of one organization or business [19]. This concept can be easily comprehended comparing it with Wide Area Network (WAN) and Local Area Network (LAN). Variations of these deployment methods can be achieved, as are the cases of Community Cloud, where the use of the cloud is restricted to a group of consumers in the former, and Hybrid cloud, where the cloud is composed by two or more data centres bound by standardized or proprietary technology [43].

Cloud computing is characterized for having access on demand, without the need for human interaction with the service provider, coupled with the availability of access on any connection point of the network, despite the client platform used for the access. This means that any platform able to connect to the cloud is able to access and use the same computing resources and services as any other. Such resources can be measured and distributed fairly across the range of users at any given time, while being flexible enough to be allocated or released according to the increase/decrease of the user base at any given time. In other words, one user only uses the resources needed for task completion, while being given the appearance of infinite resources.

2.1.2 Service Models using classic cloud infrastructures

Cloud computing service models can be grouped in three categories, ranging from a low to high level services. These categories are as follows:

– **Infrastructure as a Service (IaaS):** the consumer is provided with minimal software (bare operating system) to operate the hardware resources. In other words, the consumer has no control over the cloud infrastructure, but can choose which operating system/storage system/applications to be run on the cloud. The Amazon EC2 is an example of such cloud service, where the client has control of the cloud like it would on a local machine while benefitting from the flexibility of cloud computing resources [3].

– **Platform as a Service (PaaS):** the cloud provides an operating system, as well as a range of programming languages, libraries, tools and frameworks with which the consumer can develop its owns applications. Despite offering little control over the cloud and restricting the programming environment, this kind of cloud service provides an easier use, since low-level operations are already provided. As examples of PaaS we can point out the Google App Engine [2] or Heroku [4] where applications are developed and run entirely on the cloud.

– **Software as a Service (SaaS):** the user can only access applications already implemented on the cloud, having no control over the infrastructure, servers, storage or individual application capabilities. This is the highest level of cloud structure being the most restrictive one but also the easier to use. The Google Docs is a fine example of a SaaS, where the client has access to applications like a text editor, which is usually installed and run locally on a machine, that are run instead on remote servers.

As we have seen, there are many offers in the market for cloud services, however they do not cater specifically to robots nor present the right tools to develop robotic applications. Lack of compatibility with existing robotics application frameworks and failure to meet robotic applications requirements limit the applicability of these existing cloud computing platforms to robot application scenarios [44].

2.1.3 Service Models providing robotic services

The concept of accessing applications, storage systems, or processing units can be developed into other kinds of services, namely robotic services. In other words, we can use the same paradigm as traditional cloud computing to access, for instance, real-time video or sensor data, among other robotic services [31]. We can then consider three kinds of delivery model:

- **Function as a Service (FaaS) or Equipment as a Service (EaaS):** this low-level model uses the cloud to provide robot resources, such as sensors and cameras or robotic tasks solvers like SLAM frameworks. In [59], the authors use information from a remote camera and cloud processing to control the formation and movement of several robots. These robots (iRobot Create) possess low processing power, so not only do they take advantage from the cloud's better processing capabilities but also extract data from a remote camera as aid to their task. The capability of accessing remote equipment data becomes even more relevant with the recent research advances on the topic of the *Internet of Things* [37] [21].

- **Robot as a Service (RaaS):** in this model the user has access to a robotic platform such as a teleoperated robot. In [32] the authors propose UNR-PF, which consists on an open-source, standards-based platform that enables multi-location daily life support services from a cloud of robots. In a more recent work [23], authors present a system for an autonomous assistive robot working in a smart environment. This system uses the Cloud to improve the ability of the connected robots, by not only providing elastic storage and computational resources but also connecting the robot to caregivers and familiars. On a practical use case, the robot used speech recognition to identify certain keywords and browse the Cloud to retrieve useful information on how to reach the user and perform the requested service.

- **Robotic Service as a Service (RSaaS):** this high-level model intends to offer the user a robotic service without the user specifying the platform that executes such service. For example, while in RaaS the user would ask the framework for a specific robot and then control it to complete a task, in RSaaS the user would ask the framework for a task to be completed, without knowing which robot would be used to complete said task.

We can view these models as internal clouds in a fleet of robots. For example, a small robot with few sensors and low processing power can access a remote cloud to execute algorithms posing a high computing demand, while accessing sensor information from the internal cloud that is composed by all the fleet members of a multi-robot system. Here the term internal cloud has a similar meaning to a robotic cluster, *i.e.* a group of robots provide their sensor data and computational resources to one member of the team forming a similar cloud structure, without resorting to external computer units. Naturally, an external cloud employs computer units foreign to the robotic ecosystem in order to provide with computational resources. With the increasing number of services made available through cloud computing, the term Anything as a Service (XaaS) has also been used to classify cloud computing services [12].

2.1.4 Challenges for Cloud Robotics

So far we have seen how a multi-robot system can use internal and external networking to enhance task solving and speed up algorithm processing. On an internal ad-hoc cloud, sensor and computational resources can be pooled to form an infrastructure for collaborative processing and decision-making, whereas storage and processing resources can be allocated elastically on an external cloud in order to meet given task requirements. These are the wonders of cloud computing which can be exploited to increase efficiency and cleverly reduce costs. However, such architectures present their own set of challenges to be tackled.

In [35], the authors give a systematic view on how cloud computing can be used to overcome some limitations of networked robotics, as well as the implementation challenges faced by this new paradigm.

One of the prime benefits of cloud computing is the ability to offload a computationally heavy task to the cloud, while taking benefit of its parallel processing capabilities to have a relatively low execution time when compared to a machine with good processing power running the same task locally. Hence, the following issues arise: How much data should be offloaded to the cloud? How should the cloud spread the task across its resources? Energy consumption and deadline requirements are proposed as deciding factors when offloading a task to the cloud, thus a robot should not process a task locally if it means consuming more energy or failing a temporal deadline, and vice-versa.

Time required in communication between robot and the cloud, being it an internal or external cloud, must also be taken into consideration, especially when considering mobile robots whose only reasonable source of connection is through some wireless network. As wireless communication systems are prone to delivery failures, either by environmental reasons or channel clutter, a probabilistic approach must be taken into consideration when calculating the time consumed while exchanging messages between client and cloud. Package overhead is also a contributing factor, which coupled with communication failures can make a cloud-based solution to a multi-robot system less desirable, especially when considering real-time scenarios.

Finally we can identify security challenges. For obvious reasons, a remote data centre can be attacked, more so if it interfaces with its users wirelessly. Consequently strong barriers against outside malicious threats must be taken into consideration, with virtualization of resources being the most used form of security. Cloud stored data must also be protected, so confidentiality protection mechanisms are also needed to ensure data integrity and privacy.

2.1.5 Cloud Robotics Architectures

Over recent years some cloud robotics frameworks have been proposed in the effort of exploring this growing computational paradigm. We highlight and explore the most significant of these frameworks in this section.

DAvinCi framework

The motivation for the DAvinCi framework (Fig. 2.1) is the regular employment of computationally intensive tasks in large environments, such as computer vision and mapping [20]. Every robot is assumed to have an embedded controller with Wi-Fi connectivity and basic proprioceptive sensors, *i.e.* a low-cost, single-axis gyro and wheel encoders. Apart from these basic sensors, a range of sensors required to complete the task at hand is distributed among the fleet of robots. Sensor information is uploaded to a central controller, becoming available to every robot on demand and allowing for the completion of a task that requires sensor information from multiple robots.

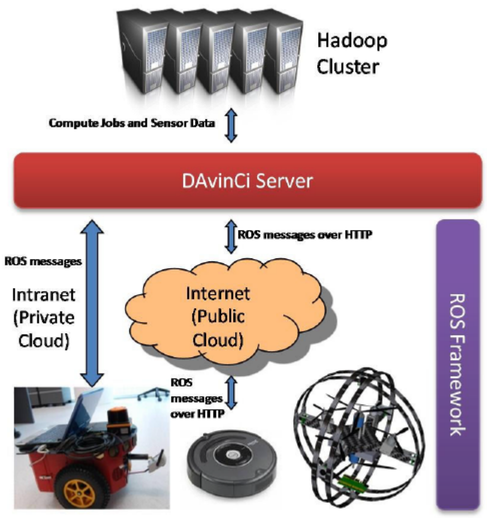

Figure 2.1: Overview of the DAvinCi architecture. Figure reproduced from [20].

The DAvinCi system can be classified as a PaaS, supplying tasks as global map building in the cloud environment. The system is composed of a server that makes the connection between the robot fleet and a Hadoop cluster. The ROS middleware [11] is used for sensor data collection and communication, where the server runs the ROS Master node. ROS messages are wrapped in HTTP requests/responses for robot/server communication. The Hadoop Distributed File System (HDFS) cluster is used for data storage and runs the MapReduce framework for the parallel processing [6]. Note here that parallelizable algorithms are key to the speed up of processing.

Besides managing file storage, the HDFS also splits data files into smaller blocks for easier parallel processing. These blocks are then distributed by the MapReduce framework across the cluster for processing. The MapReduce framework is a programming model for processing large datasets across a cluster [6] [28]. The model operates based on two procedures: *Map* which filters and sorts an input of (in_key, in_value) into an output (out_key, out_value); *Reduce* which performs a summary operation, taking all values from a given key and generating an output value. This technique was developed for search and indexing of large number text files, but nowadays its applications have been extended to other fields, although algorithms need to be written as Map and Reduce procedures.

The ROS messaging system is used, as said, for communication between the DAvinCi server and the robots. On the server side these messages are collected by ROS Subscriber Nodes that function as an HDFS client, which push the information to the HDFS file system to be processed by the Map/Reduce tasks. More information on how ROS implements message communication is described in section 2.4.

The authors present a Grid-Based FastSLAM [57] algorithm adapted to the MapReduce model, and a simulation of the execution of such algorithm using one-, two-, four- and eight-node cluster. The speed increase is notable as the time taken is significantly lower in the four- and eight-node clusters for a high number of particles, proving the parallelization of the algorithm.

Rapyuta framework

Rapyuta [8] [44] is an open-source cloud robotics platform that was closely developed to the RoboEarth [60] knowledge repository, having become also known as the RoboEarth Cloud Engine. This close association comes from the fact that all the processing needed in order to interact with RoboEarth repository can easily be done in the cloud rather than locally. Nevertheless, the main purpose of Rapyuta, in similarity with other cloud robotics frameworks, is to help robots offload tasks to the cloud and utilizing its resources, presenting itself as a Platform as a Service (PaaS) cloud computing model specifically tailored towards multiprocess high-bandwidth robotics applications.

Rapyuta is a ROS-compatible environment whose communication protocols are based on web sockets, thus it allows for ROS robots but also other machines with browser access to connect to the system. Similarly with other cloud computing frameworks, the quality of connection to the network can bottleneck the performance of the system. Internally, Rapyuta dynamically allocates computing environments for the robots, which are used for sharing services and information among them. The system is composed by four main components: the aforementioned computing environments, a set of communication protocols, four core task sets, and a command data structure.

The computing environments are Linux containers which offer a virtualization environment that allows for processes from different robots to be separated from each other in a secure and scalable manner. The containers allow for an easy distribution of memory and processing time, while maintaining application speed as if there was no process isolation. This method presents a clever way of tackling the security and scalability issues of cloud computing. Since the system is based on ROS, all the processes within a single container can easily communicate with each other using the ROS inter-process communication.

The communication protocols are handled by processes named Endpoint (Fig. 2.2), which are composed of Interfaces and Ports. This design was developed to achieve a balance between system complexity and latency in inter-process communication. A low system complexity could be achieved by a bus-based communication at the cost of high latencies, while a low latency system could be achieved by a direct connection of ROS nodes at the cost of high complexity. Interfaces are used to establish communication between a Rapyuta internal process and an external one by the use of the WebSockets protocol; here an external

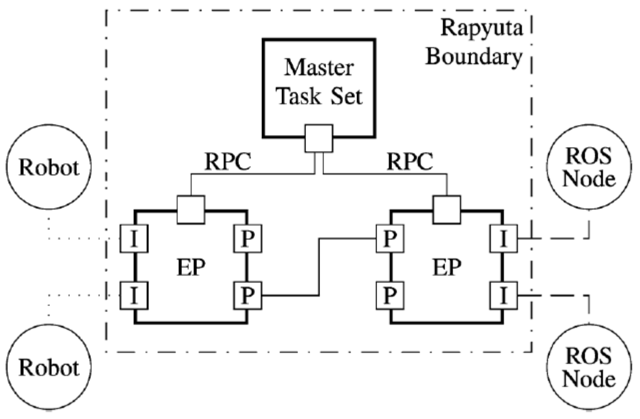

Figure 2.2: Communications in the Rapyuta framework. Figure reproduced from [44].

process can be a ROS node running on the robot or on the cloud environment. Interfaces also convert the data from internal to external formats and vice versa. Ports on the other hand deal only with Rapyuta's internal communication by the use of UNIX sockets.

The core task sets are series of functionalities that are grouped together as processes to administer the system. On a standard case, each of the four core task sets composes a process thus building a PaaS framework. The Master Task Set is the main controller that monitors all the other task sets, organizes the connection between robots and Rapyuta and processes configuration requests from robots. The Robot Task Set makes the connection between robots and the Master Task Set, handling all the communication format conversion necessary. The Environment Task Set deals with the communication between the Rapyuta System and the computing environments, while the Container Task Set deals with starting/stopping computing environments.

Finally, the Rapyuta system is organized in a centralized command data structure divided into four components: Network, User, LoadBalancer and Distributor. The Network component provides a layer of abstraction to be taken advantage of by the other components. The User represents someone who connects his robots to the system. The LoadBalancer manages the machines that run as computing environments. And the Distributor distributes incoming connections to available Endpoint processes.

REALcloud framework

The REALcloud [15] framework follows the Platform as a Service (PaaS) model to provide the REALabs platform [24] developed by the same authors. This platform is accessed over the public Internet and, after a validated access, it provides with robotic services. Robotic applications are then developed on the REALabs servers that also provides with specialized hardware such as powerful GPUs and FPGAs along with a greater processing power than one possessed by a personal computer. In order to provide a wider range of possible robotic applications this platform was integrated on the cloud computing environment REALcloud.

The platform has four main packages that work in tandem with the Cloud environment. The Embedded package contains HTTP micro servers to be run on the mobile robots the platform offers access to. These robots are expected to have limited processing power, so these components only aggregate some basic operations while the processing is done entirely by the servers. The Protocol Handler package handles all the security checking, proxying and network address translation. The Front-end package offers APIs to the user, providing with a high-level interface between the developed robotic applications and the robot itself. Finally the Management package administrates the system and resource access.

Fig. 2.3 shows how the REALcloud framework offers access to the REALabs platform. Here each user is confined to his virtual machine (VM) while accessing the REALcloud environment, which is also run on a virtualized manner. Virtualization (which is done by VirtualBox) not only allows for process separation and conflict avoidance but also introduces a layer of security, since it becomes more difficult for a VM to access the data or resources of another VM. Here it becomes more clear where the aforementioned packages are run: the Front-end packages are run on client side while the Management and Protocol Handler packages are run on server side. The packages run by the REALcloud environment, the VM Management, and the Session Validation, control the process of creation, reconfiguration and destruction of VM, and manage the access of robotic resources by applications, respectively.

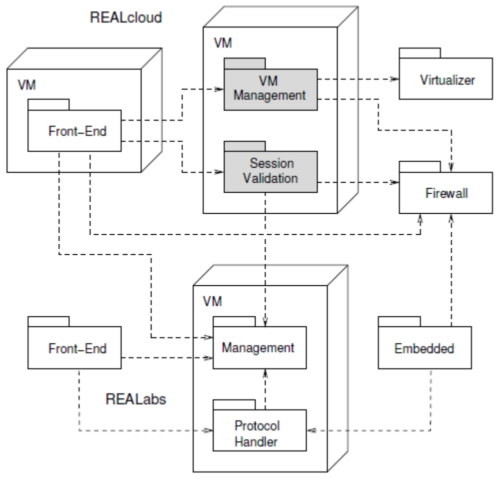

Figure 2.3: Overview of the REALcloud architecture. Figure reproduced from [15].

Robot-Cloud framework

The Robot-Cloud framework [29] was developed to allow low cost robots to offload computationally heavy tasks to the cloud. Fig. 2.4 represents the normal work process of this cloud framework. A central unit (Cloud Controller) is embedded in the ROS Master Node and administers the whole system. This unit forwards all the service requests to the Service Administration Point in order to check for authorization. The Service registration/removal then checks for service availability, which if unavailable puts the request into a queue. If permission is granted, the Cloud Controlled is notified and a reference to the service is passed to the requesting robot. The services provided by the cloud framework include a Map/Reduce computing cluster, storage system, and robotic services such as path planning and map building.

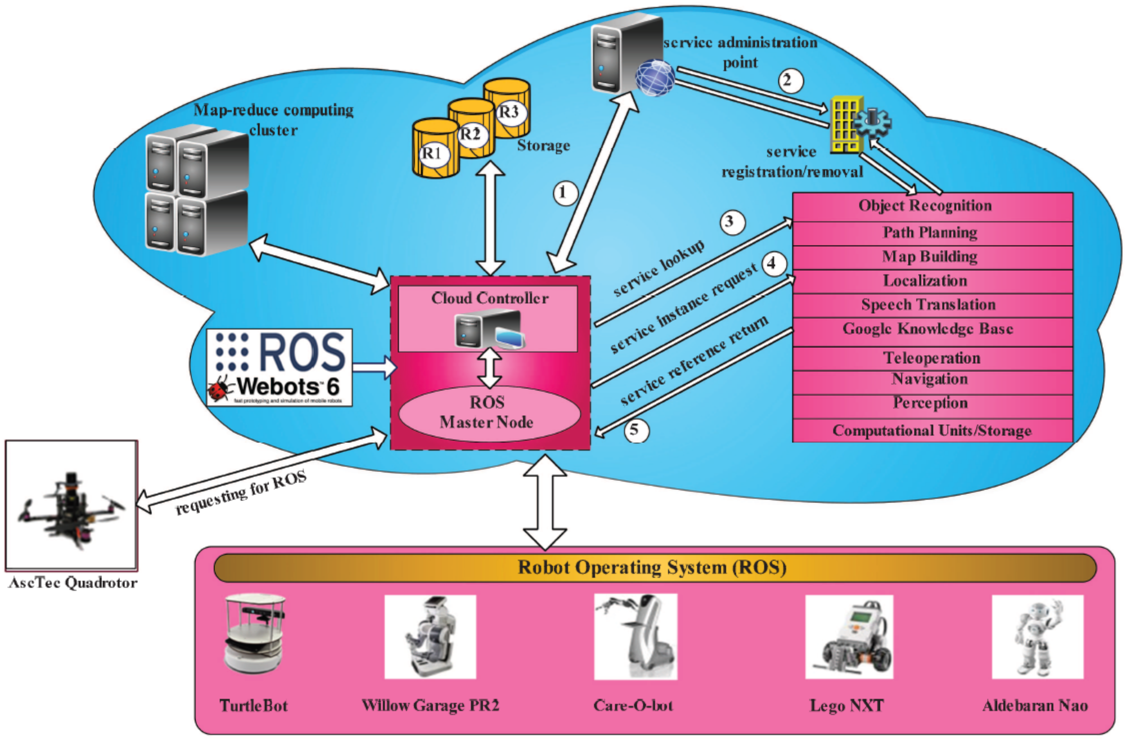

Figure 2.4: Overview of the Robot-Cloud architecture. Figure reproduced from [29].

Robotics In Concert framework

Nowadays ROS is one of the most used toolkits for robot programming, gathering a large amount of drivers, libraries and developer tools [11] (ROS will be further discussed in section 2.4). However, this framework was not conceived for distributed robotic systems, thus the reasoning behind the framework Robotics in Concert (RoCon) [10]. RoCon is a *multimaster* system, based on ROS communication mechanisms, that administrates robot interactions in a centralized architecture.

Although it is not strictly a cloud computing framework, its objectives and architectures are in line with the ones from cloud robotics, making it a framework worth studying. Moreover, the concept of cloud as an external resource for computational power and storage is present on the framework. Additionally, as it will be further explained, the main concept of the RoCon framework is having a central entity that coordinates client (robot or human) interactions and allows for the farm of resources and functionalities. Such entity can be easily projected to the cloud.

It is important to understand the motivation behind the development of this framework as the authors have a much more practical, day-to-day view, contrasting with the more academic approach usually seen on scientific research. The primary objective is to develop a framework that enables application developers to take a more high level approach to robot programming. This high-level programming could enable industrial/commercial robotic solutions to be more marketable since they become more easily developed. On the one hand, making the development of robotic services easier reduces its cost; on the other hand, having a central unit which supplies cheaper robots with the necessary computational resources would mean more viability for robotic solutions, since most of them are dropped as business opportunities on early stages due to high robot cost. With this in mind, the authors highlight four key aspects that could enable marketable robotic solutions:

1. **Cheaper Robots:** the use of external processing resources allows for lower cost robots without losing utility;
2. **Retaskable Robots:** robots should not be restricted to one task, reprogrammability is highly valued;
3. **Interactivity:** if day-to-day markets are to be explored a system must take human interaction into consideration;
4. **New Markets:** devices that integrate human interaction with multi-robot systems should be explored, based on the premise that robots should only be part of the solution rather than the solution itself.

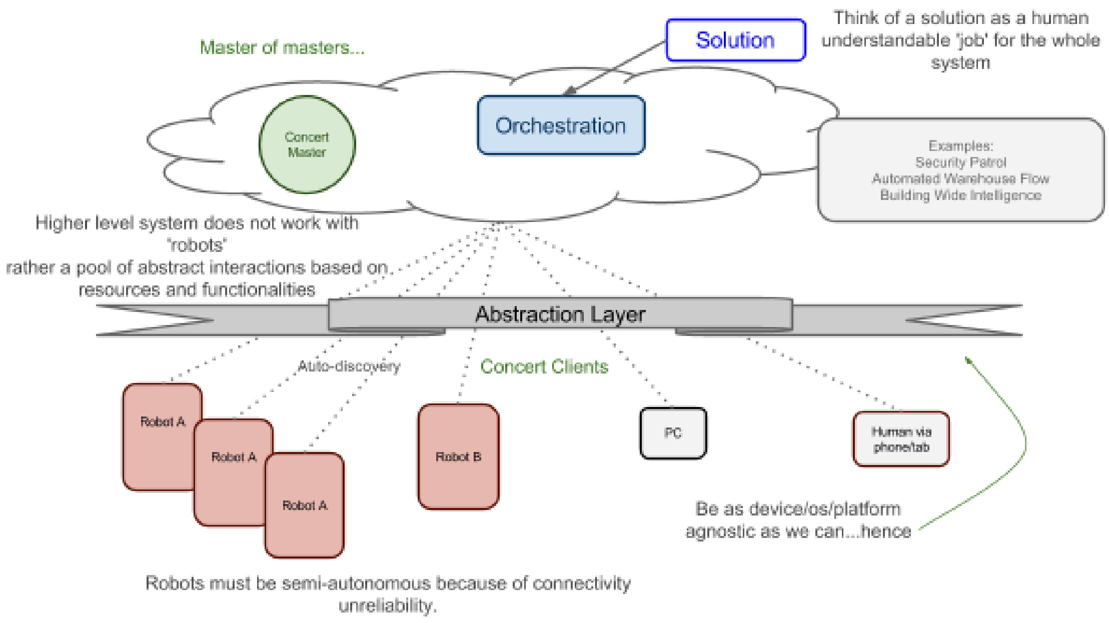
Master of masters...
Solution
Think of a solution as a human understandable 'job' for the whole system
Concert Master
Orchestration
Examples:
Security Patrol
Automated Warehouse Flow
Building Wide Intelligence
Higher level system does not work with 'robots'
rather a pool of abstract interactions based on resources and functionalities
Abstraction Layer
Auto-discovery
Concert Clients
Robot A
Robot A
Robot A
Robot B
PC
Human via phone/tab
Be as device/os/platform agnostic as we can...hence
Robots must be semi-autonomous because of connectivity unreliability.

Figure 2.5: Overview of the RoCon *multimaster* framework. Figure reproduced from [10].

The framework architecture is as presented in Fig. 2.5. There are three main components: a concert, the concert clients and an abstraction layer between them. Concert clients are the robots or humans that interact with the system. The concert is the centralized multi-robot framework that encapsulates the concert master, which is a model that gathers the components necessary in order to establish a concert, and the orchestration.

The orchestration platform presents the tools needed in order to make a solution easier to conceive (the whole motivation behind this framework). Here the term solution represents the high-level concept that wants to be achieved. It is composed of a set of services relevant to the solution such as map building and teleoperation: the Interactions Handler package, which manages the connections to human applications; the Conductor package, which detects, invites and accepts new clients; the Resource Scheduler, which administrates resource distribution among provided services; and the Software Farm packages, which provides with software such as database management and computing modules.

Lastly, in order to provide with a high-level interface for the application developers, the abstraction layer exists to make the connection between high-level concepts (such as resources and functionalities) and the robot hardware.

2.1.6 Comparison between the selected frameworks

Bellow, it is presented a table comparing each of the frameworks mentioned in the last subsection. Each framework was evaluated with the objectives and motivation of our work in mind.

Table 2.1: Pros and Cons of the selected frameworks.

Framework	**Pros & Cons**
DAvinCi [20]	**Pros:** + Communication is based on HTTP which is an established protocol with proven results. Although HTTP requires inspection, real time tasks are not suitable to be run in a Cloud environment + The Hadoop cluster allows for algorithms to be run in parallel automatically **Cons:** - The whole system shares a singles ROS master node run in the server, meaning less robustness and reliability due to the single point of failure on that node - Algorithms must be written in Map Reduce procedures, lowering the flexibility of the system - No process separation or security - It is not publicly available
Rapyuta [44]	**Pros:** + Communication is based on web sockets (http request/response) + Allows for converters to be developed to make robot-cloud communication more efficient + A browser could potentially be used as a client or manager of the cloud environment + Individual virtual environments offer a layer of security + Containers offer an easy way of distributing computer resources (scalability) + Ports and interfaces strike a balance between latency and complexity + Easy access to the RobotEarth knowledge repository

	+ It is open source + Allows for a *multimaster* architecture + Communication mechanisms make the system very flexible **Cons:** - Comparatively to other *multimaster* frameworks it is harder to debug since processes are run on the containers - Centralized architecture, only one machine runs the rce-master node - Parallelization must be done by hand by allocating different ROS nodes to different containers
REALcloud [15]	**Pros:** + Communication is based on HTTP which is an established protocol with proven results. Although HTTP requires inspection, real-time tasks are not suitable to be run in a Cloud environment + Virtual environment and session validation offer a layer of security **Cons:** - It is not publicly available - Any existing onboard computer resources are not available for use
Robot-Cloud [29]	**Pros:** + Integrates a map-reduce computing cluster as well as storage space + The Service Administration Point offers a layer of security + The Service registration/removal offer a resource management mechanism **Cons:** - The whole system shares a singles ROS master node run in the server, meaning less robustness and reliability due to the single point of failure on that node - Algorithms must be written in Map Reduce procedures, lowering the flexibility of the system - No process separation - Robot-Cloud is not publicly available

Robotics in Concert [10]	**Pros:** + Includes human interaction as a client like a robot + Multimaster architecture, each robot has a ROS master node, which increases the overall robustness and reliability of the system + It is open source **Cons:** - Centralized architecture - Not focused on exporting computationally heavy tasks to the cloud. Although it offers the ability to do so, it does not do so in a clear way as the other frameworks

All of the studied Cloud architectures offer different positive characteristics that could be exploited in this work. However, only two of them are publicly available (Rapyuta and RoCon), limiting our choice in the matter. That being said, the DAvinCi and the Robot-Cloud frameworks have a more limited application as they mostly rely on a map-reduce cluster for processing and run only one ROS master node. As the core of this work is the transition of a multi-robot system to the Cloud, these characteristics give them little flexibility.

The REALcloud framework, shares some of these problems as it does not offer, as far as our knowledge goes, the necessary communication mechanisms in order to set up a *multi-master* system. Despite not being a necessary condition to a multi-robot system, we believe allowing for a ROS Master node to be present on each robot is a strategy that should be pushed forward, as it offers advantages in terms of robustness and reliability, which are key when a distributed strategy is in mind. The concept of ROS Master will be further explained in section 2.4.

The possibility of using RoCon during this work was rejected, mainly due to the fact that it does not focus on running tasks in the Cloud. Although there is the possibility of doing so, the main objective of RoCon is to setup a *multimaster* architecture, which overall does not align as well with the objectives of this work as the Rapyuta framework does. Moreover, despite being a project in development, RoCon's communication protocols are stalled waiting for the release of ROS 2.0. It would make more sense to reconsider this framework after significant development is done on this subject.

Finally, the Rapyuta framework was chosen as the basis of this work, as its characteristics offer very high flexibility and easy conversion of existing ROS packages to work inside the Rapyuta's Cloud Environment. Moreover, the creation of *multimaster* architectures comes naturally with the way the communication is handled by the framework.

2.2 SLAM

2.2.1 Single robot SLAM

The problem of a mobile robot building a map of the environment and localizing itself in it is known as simultaneous localization and mapping, or SLAM. The name comes from the fact that both operations occur at the same time. SLAM techniques have been developed mostly for indoor static environments, however the same principles can be applied to outdoor, underwater, and airborne environments [30].

There are usually three types of systems robots can use to gather relevant data for SLAM procedures. These include laser range finders and sonar-based systems, which are fast but depend on noisy sensors like odometer, and vision systems that are very computationally demanding [22]. In order to build a map of the environment, a robot not only needs to process sensor data but also to keep map information on memory. This means that most approaches cannot perform consistent maps for large areas, mainly due to the increase of the computational cost [22], and that cheaper robots (with less resources) will struggle to complete the required tasks. To tackle this problem, solutions that offload processing to remote locations or that resort to resource sharing have been proposed on recent years. One can clearly see how a cloud computing environment could help in such conditions. Apart from environment observation models, the SLAM problem also requires for a motion model, since the control signals sent to the robot's actuators do not necessarily represent how the robot moves, only how it is intended to move [41].

The most successful models of mapping [25] are comprised of dense 2D laser range-finder data (grid-based metric maps) or sparse 2D/3D feature points (topological maps) – see Fig. 2.6. Grid-based maps are learned through Bayesian approaches and the whole environment, including obstacles, is mapped. On these metric maps, each cell represents the probability of being occupied [41], this means that the memory required to maintain such a map greatly increases with resolution and environment size, which prohibits efficient planning and problem solving, reducing the method's scalability [58]. On the other hand, topological maps present a graph with various nodes connected with each other. Each node represents a landmark and each connection a travelling path between landmarks. Since only a fraction of the real environment is mapped, memory is less of a concern, although in this situation we don't have a scaled map of the real environment, only known available pathways through it. Although this kind of map can be used more effectively, they are also more difficult to learn in large environments and have difficulty in differentiating two places that look the same, which can lead to a misidentification of a landmark [58]. Some architectures have employed both methods of mapping in order to merge the strengths of both approaches [46, 58].

Sensors used in SLAM, notably odometric sensors, generate a considerable amount of cumulative noise, thus probabilistic methods are used rather than mathematical, simpler ones. These probabilistic algorithms tackle the problem by explicitly modeling different

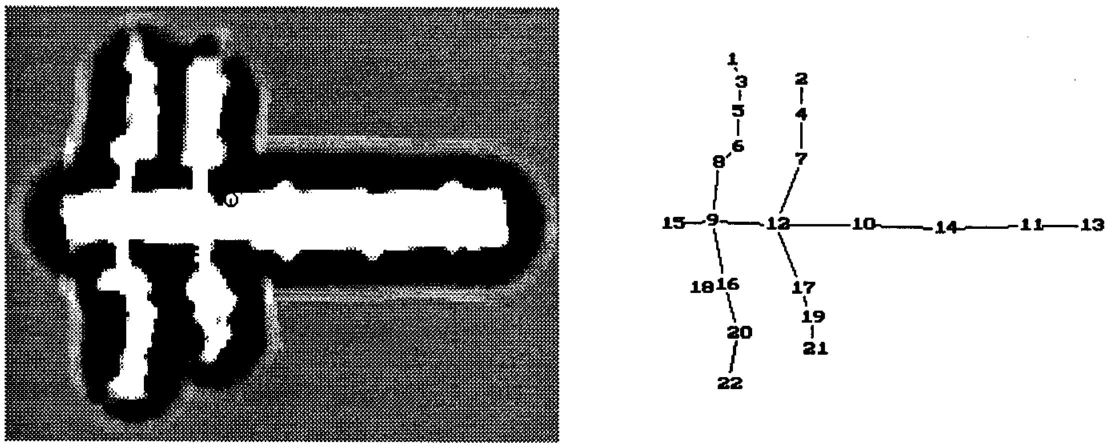

Figure 2.6: Comparison between grid-based map (left) and a topological map (right). Figure reproduced from [58].

sources of noise and their effects on the measurements [22], which leads to a higher demand for processing power. Another problem comes from the necessity for data alignment (*a.k.a.* data association). For every scan the robot makes it needs to check if the current position matches a previous one, by comparing features with past scans. This process allows the closure of a loop in the environment, otherwise a robot would pass through the same position multiple times while considering it a new location, resulting in an faulty map. Accumulated errors along the trajectory (drift), might lead to an inconsistent reconstructed map, *i.e.* the loop of the trajectory is not closed properly [22].

SLAM solutions come in the form of either filtering techniques, which are the most common, or graph-based. Filtering techniques are based on Bayes filters, which are an extension to the Bayes Rule. Many variations of the Bayes filter have been developed over the years and [22] presents a brief overview of the most popular ones.

Graph-based techniques keep all gathered data, including past poses, in the form of a graph (hence the name graph-based). Since all data is kept, and all data is processed on every step, more computation power is required. However, the resulting graphs can be easily shared with collaborating robots [46], which is a mandatory asset in a multi-robot system.

2.2.2 Multi-Robot SLAM

Multi-Robot SLAM (MR-SLAM) is an extension of the original SLAM problem that tries to take advantage of characteristics offered by a multi-robot system such as parallelism and redundancy [41]. Exploring an environment with a team of robots rather than a single one is always going to be more efficient and reliable, however, multi-robot systems present with new challenges, namely unknown robot poses, map fusion and scalability [25]. Networking challenges also apply to this kind of robotic ecosystem since robots need to work together by sharing gathered sensor data and processed information. Having multiple robots is also more expensive, so the elements of a team of robots generally need to be cheaper in order to compensate for the numbers, this means less processing power and storage space. So means of sharing data and computational resources become necessary and the use of remote computing environments becomes a more attractive solution for teams of cheaper robots. Despite the decrease of costs with time of computing resources, scalability is always a concern, thus such solutions are always relevant.

One of the aforementioned problems is that algorithms in MR-SLAM need to know the initial relative positions of the remaining robots. This problem is trivial if all the robots have started in the same position, but such approach is very limiting. Other techniques can be employed, such as in, [25] which presents a solution where it is considered that robots can compute the relative position under special circumstances, *e.g.* meeting or staying in line of sight. Upon contact between two robots, map information is shared, but the algorithm must be able to compute the relative initial positions on a time-reverse order: from the present relative positions, map information is presented as if the robot was moving backwards.

Since in a multi-robot environment each member of a team of robots creates each own map segment of the whole environment, this information must then be shared among the remaining robots so that a complete map can be built. This is known as map fusion. This task in inherent to any approach to MR-SLAM, and here we can see that a cloud environment could benefit the robotic environment by providing with a more trivial trade of information. Moreover, the ability to maintain a complete map on the cloud would allow a new robot to have easy access to it, without having to re-explore the environment.

2.3 Multi-Robot Patrolling

2.3.1 Overview of the Multi-Robot patrolling problem

In a robotic patrolling scenario, a robot/team of robots must sweep the surveilled environment repeatedly. Patrolling algorithms define a metric that is intended to be optimized, such as visit frequency of each location or the distance each robot has to travel to complete its course [36]. The most commonly used metric is the time elapsed between consecutive visits on the same node, usually referred to as idleness.

This problem is usually presented in a patrolling scenario, however the same approach can be used in other circumstances, such as mine clearance or search and rescue operations [48]. Although one could implement a patrolling system with a single agent, the advantages of having multiple ones are rather imposing in this specific scenario, as multiple agents always allow for a more frequent patrol, thus such systems are almost always employed in this manner. However, as we have seen, and considering mobile robots as agents, having a multi-robot system presents with its own problems to be tackled, being one of them cost constraints. Many researches do not take into consideration this kind of real world problem, as it is more common to present results in a simulation environment. However, an algorithm's scalability falls if the costs of maintaining a larger team are too high, thus in a large team robots tend to have to be cheaper to compensate for the numbers. As stated before, this means less computational resources and less memory available. Even if we discard cost limits, size may also be posed as a restraint. Greater computational resources require more power, which in turn requires a larger battery to maintain a reasonable autonomy. We can see here two situations that can take advantage of cloud computing in order to increase scalability in a real world scenario.

In a Multi-Robot Patrolling problem, each robot needs to decide which position to move next while maintaining a global trajectory that not only covers the whole environment but that is also efficient, in other words, the individual robot trajectories should not intercept each other. Most approaches solve trajectory generation with a centralized planner that commands all the robots' paths, while others implement distributed mechanisms that generate trajectories in real time [49], however this is more demanding in terms of computational resources to the robots as there is no a priori processing, nor can they rely on external resources. Moreover, patrolling also involves some task related to perceiving the environment, such as image processing or pattern recognition, in order to detect any anomaly or intrusion. These kind of tasks tend to be costly in terms of computational resources, and coupled with the navigation and obstacle avoidance tasks that run continuously, these resources can quickly become overburdened.

The surveillance method needs to take into account the robot's sensory capabilities: the environment can be modeled by a grid-based map where robots must verify each cell (known as the coverage problem); or the environment can be modeled by a topological map, considering the robots can surveil a single place from just one or a few locations (known as the area patrolling problem) [49], as we have seen before, in this case, each node of the graph represents a landmark to be patrolled and each connection the path to said location. Grid-based and topological maps have been described in section 2.2.

There are three takes on patrolling problems: adversarial patrol, where the robotic team is tasked to find an identified threat, perimeter and area patrol, where the task is to frequently visit important areas of the environment. The difference between perimeter and area patrol is that while in the former robots patrol only the outer rim of the environment, in the latter all areas are surveiled [49]. However, within the same system more than one method can be implemented. As an example, [34] performs area patrol until a threat is detected, in which case it changes to adversarial patrol.

2.3.2 Related work

Pioneer work was developed by [39] where authors defined criteria for evaluating multi-agent systems (MAS): idleness (average number of cycles without a visit), worst idleness (greatest number of cycles without a visit) and exploration time (number of cycles to visit all the nodes). Normalized values of the criteria based on the relation between the number of agents and the number of nodes was also defined in order to evaluate the optimization. In order words, if the agents are in a greater number, the criteria defined above will tend to be lower, however this does not mean that the system is running more efficiently.

In order to test different methods authors defined a few varying parameters: type (reactive/cognitive), communication, next node choice, coordination strategy and monitoring capability. Various methods with different parameter combination were tested in a simulation environment, where the best results were obtained by the following algorithms:

1. **Conscientious Reactive:** since it is a reactive method, it has a limited field of view of adjacent nodes (in this case only the immediate neighboring nodes are perceived); no communication is conducted; the choice of node is computed based on local idleness values without knowledge of other agents' movement; and coordination is done in a decentralized manner.
2. **Conscientious Cognitive:** since it is a cognitive method, it has a field of view of up to the global map; no communication is conducted; the choice of node is computed based on global idleness values without knowledge of other agents' movement; and coordination is done in a decentralized manner.
3. **Idleness Coordinator:** cognitive method; communication is conducted by messages between agents; the choice of node is computed based on global idleness values with knowledge of other agents' movement; and coordination is done in a centralized manner.

While this was pioneer work that set the pace for patrolling research, many considerations/simplifications during simulation were taken that limit the validity of these results in a live scenario [48]. Most of these considerations come from the background of Artificial Intelligence development in a video game, which differs greatly from our case study of robotic patrolling.

Over the years many different approaches have been proposed, each based on different principles [48]. The pioneer work described above was further developed in [16]. Here the author presents Heuristic Agents, which improve on the path finding and decision making process by taking a broader approach that takes into consideration not only the idleness and distance values when planning a path, but also the values carried by the neighboring nodes.

The solution to the Traveling Salesman Problem (TSP) gives the shortest circular path between a number of locations. As maps are usually represented in a topological fashion, the patrolling problem can be seen as a TSP, and in the case of a multi-agent system, a single map can be divided into smaller sections, giving each agent the task of following the route given by the solution to the TSP associated with each section [17]. In work [26], the author describes single-agent patrolling based on TSP and generalizes the approach to the multi-agent case, showing how multiple agents can take advantage of graph partitioning. Another example of this methodology that involves dividing the global map (graph) into smaller subgraphs can be found in [47].

Other techniques involving negotiation mechanisms and reinforcement learning have also been used to grant agents a more reactive and adaptative behavior. These solutions come from approaches taken in other works related to Multi-Robot systems that revealed promising results [50]. Finally, in [49] authors present two methods based on the Bayes rule with the intent to create a more flexible solution that can adapt to environment changes or agent failures. In this solution no central planner is used, giving the team of robots full autonomy and moving away from pre-computed cyclic routes or partition schemes. This work is further developed in [51] by the same authors, introducing memory to the system by way of maintaining the history of visits of each teammate and adopting concurrent reward-based training in order to reduce the overall complexity of the system.

As one could note, centralized processing is already widely used in multi-robot patrolling platforms, so a cloud framework becomes a natural extension to such systems in order to increase their functionalities, such as parallelism capabilities. Moreover, the concept of using remote processing is not strange to robotic patrolling. In work [55], robots communicate to a central computer system their current state, which does all the computing necessary and instructs the robots one their route. Having said that, distributed platforms can also take advantage of a cloud environment, because smaller, cheaper robots can be introduced in the robotic team and cooperate with other robots by simply offloading computationally heavier tasks to the cloud. Either by having a main role as a route planner or by being just a service of computational resources, the cloud presents promising capabilities to multi-robot patrolling systems.

2.4 ROS

As the name suggests, ROS (Robot Operating System) [53] is an operating system for robots. Albeit not an operating system *per se*, it offers the same services as one, such as hardware abstraction, low-level drivers, process message-passing, *etc.* by providing a communications layer on top of the host computer operating system. Over the years it has become one of the most used toolkits for robot programming, gathering a large amount of drivers, libraries and developer tools.

The main motivation behind ROS is the fact that different robots vary greatly from one another, even if they are built for the same purpose, thus making code reuse rather complicated, which is a major barrier to research and development. In other words, making a robotic application requires deep knowledge on how the robots work, from low-level driver components to high-level application algorithms. While the latter can be ported from one robot to another, the former cannot, making code reuse an arduous task.

With this in mind, ROS was developed to create a framework usable on a wide variety of situations, which has almost become a standard in robotics, thus allowing for a converging of research efforts. Following this characteristic of versatility, ROS supports various programming languages, namely C++, python and LISP, with experimental libraries in Java and Lua [9].

With the code reusability premise in mind, ROS was developed as a framework distributed by various packages. These packages contain runtime processes called nodes, libraries and other relevant files. Packages are a way of grouping files that together offer some sort of functionality, reducing complexity and allowing code to be written without any ROS dependencies, resulting in an easier reusability and a modular fine-grained framework. As an example, it is possible for a robot that uses wheels to move around to use the same code to patrol an environment as a robot that uses tracks, by using different locomotion drivers but the same patrolling package.

As we have seen, a ROS system is composed of many nodes, each providing certain functionality. These nodes communicate with each other by messages, which are data structures like in any programming language. Messages can be passed among nodes by asynchronous (topics) or synchronous (services) mechanisms. Topics are a publish-subscribe method of inter-process (or inter-node) communication. When a node publishes a message to one topic, each node that subscribed to that topic will receive that message. On the other hand, services are a request-reply method of inter-node communication. In this case, a node requests a message to another and waits for the reply before continuing.

As ROS provides this structured communication layer there must be some form of administrating the ongoing communications. Here we can introduce the concept of the Master node. This node is part of the *roscore*, a set of nodes that provide the minimal structure and functionality to a ROS system, and its function is to administrate the system's communication lines by keeping track of all the topics and services, so that nodes can communicate to one another in a peer-to-peer manner. Moreover, the master node provides with naming and registration services. The need for these services comes from the fact that a stack (set of nodes with the same purpose) can require multiple instances of the same node, for example a robot might need two nodes to control each of its tracks. Thus namespaces allow for topics of twin nodes to be addressed without changing any code. Traditional multi-robot architectures exploit the use of namespaces by having a central machine running the master node and multiple robots running nodes communicating to the master under namespaces.

Although the multi-robot situation described above is a clever work-around of ROS inherent limitations, having only one ROS master node in a central machine means that in case a robot loses connection to said machine, it will stop working. This is not of great concern to static robots that can have a wired connection to the central machine with high availability. However, mobile robots require a wireless, and inherently less stable connection, posing a great barrier of reliability to any architecture developed this way. As such, *multi-master* systems have been proposed. On these systems, one *roscore* is run on each robot, which allows for more autonomy and reliability, as robots can continue to operate even if the wireless connection is interrupted.

Key aspects that *multimaster* frameworks must tackle are the development of building blocks and tools, communication layer, and web tools integration [56]. This new approach to multi-robot systems has sprouted a few projects such as RoCon and Multimaster FKIE, as described in section 2.1.5.

2.5 Summary

In this chapter, we presented the key aspects on which our work will be built upon. We started by going over cloud computing (section 2.1) in general and how it can be deployed in practice, as well as robotic approaches to it, which represent a small shift in the paradigm, where robots, sensors or services are provided rather than just computer resources. We also went over the challenges faced by cloud robotics and the most significant frameworks, in the context of our motivation.

Sections 2.2 and 2.3 present an overview of the SLAM and Multi-Robot Patrolling problems, which will be used as test cases for the developed cloud robotics application. Finally, section 2.4 presents with an overview of ROS, which will be the framework used for robotic software development.

Having selected the Rapyuta framework as the basis for our cloud adaptation of multi-robot tasks, section 3 will present how it is used from both the cloud and user sides.

3 Cloud Robotics using Rapyuta

In the previous chapter, we discussed the mechanisms of the Rapyuta framework and the reason it was picked over the others for this work (sections 2.1.5 and 2.1.6). In this chapter, we will go over how the framework is used in practice from both the point of view of a Cloud user and a Cloud manager. We will also take a more in depth look at Rapyuta's communication mechanisms as they are a key component in setting up a functional system.

3.1 Setting up Rapyuta's Cloud environment

To setup a computing cluster – or in this case a Cloud – with Rapyuta there are three key components: the *master task set* (run with *rce-master*), the *robot task set* (run with *rce-robot* and not to be confused with the robots that connect to the Cloud) and the *container task set* (run with *rce-container*). While the master task set is unique to the cloud, *i.e.* only one in execution, the other two run on every machine that belongs to the cluster. In other words, the machine that runs the master task set can optionally run the other two, while every other machine will run both robot and container task sets but not the master one (Fig. 3.1). Setting up a computer cluster with an arbitrary number of machines is made rather easy, since the process of adding/removing machines is decoupled from the master task set, meaning that the size of the cluster can be dynamic since no *a priori* information of which and how many machines compose the cluster is needed. Only the IP address of the machine running the *master task set* needs to be known.

The reason for the robot and container task sets being separate from one another is that the container task set needs to be run under super user privileges in order to create and destroy the Linux containers. Thus, severe security issues could arise if the robot task set were to be one with the container task set, since it makes the connection to an outside robot (user), the users would communicate to the Cloud environment with these super user privileges.

In similarity with a machine joining the computing cluster (or Cloud), a user connecting to Rapyuta's environment (run with *rce-ros*) only needs to know the IP of the machine running the master task set to setup a foreign computing environment for itself. Here a user refers to a robot (the common case) or any machine with a WebSockets protocol compatible browser.

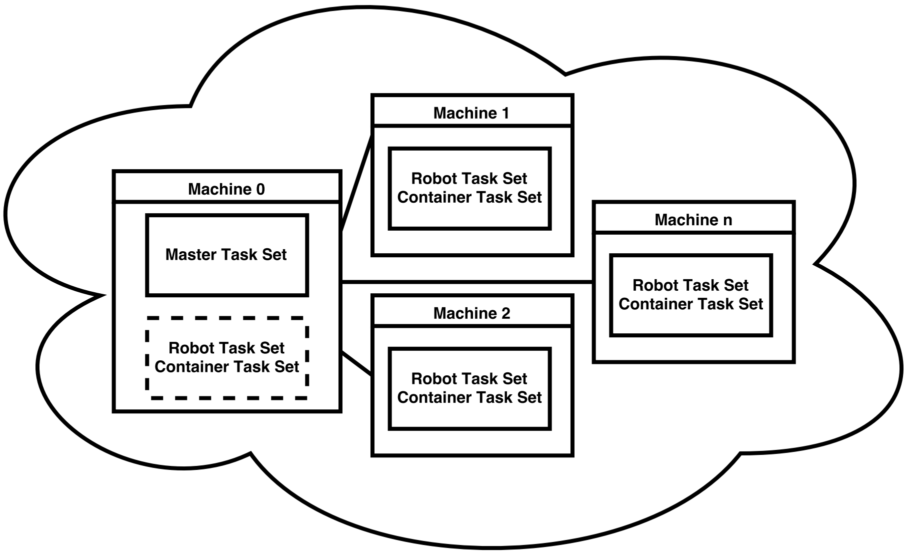

Figure 3.1: Cloud setup using Rapyuta. The dashed box represents optional task sets.

After contacting the master task set the connection is forwarded to one of the machines in the cluster. This machine is assigned by the master that manages these connections so that one machine does not get overloaded while others stay empty. These machines running the robot and container task sets will offer computing environments based on virtualization (discussed in section 2.1.5).

While the users do not control which machine they are assigned to nor how much processing power they are given, they can choose how many virtual environments are created, what runs on each environment and which connections are made to their local machine. To do so, while in contact with the master task set, they must provide a file in JSON (JavaScript Object Notation) format specifying said options. As Rapyuta is targeted towards ROS systems, these JSON files include information on Linux containers to be created, which ROS nodes run on which containers, parameter values assigned to the *roscore*, and which interfaces are to be created.

Binary executable files are not passed to the Cloud during the communication setup, only the referred JSON file (Fig. 3.2). Instead each ROS package or executable must be previously installed on each machine that joins the computing cluster. Each of these packages is then accessible by each container with a path pointer. The virtual environments (containers) have their own file system, with their own root directory and their own version of ROS. While possible, it is cumbersome to install sourced ROS packages in this file system, so the packages can be installed normally if a pointer to them is provided in Rapyuta's configuration file. On some special cases, for example whenever uncommon libraries are used, the packages must be compiled inside the container file system (run with *rce-make*), so that these libraries are found on runtime. This process basically runs a *chroot* command to the container file system's root directory and the package is compiled from there, which also means it will not run outside of a container.

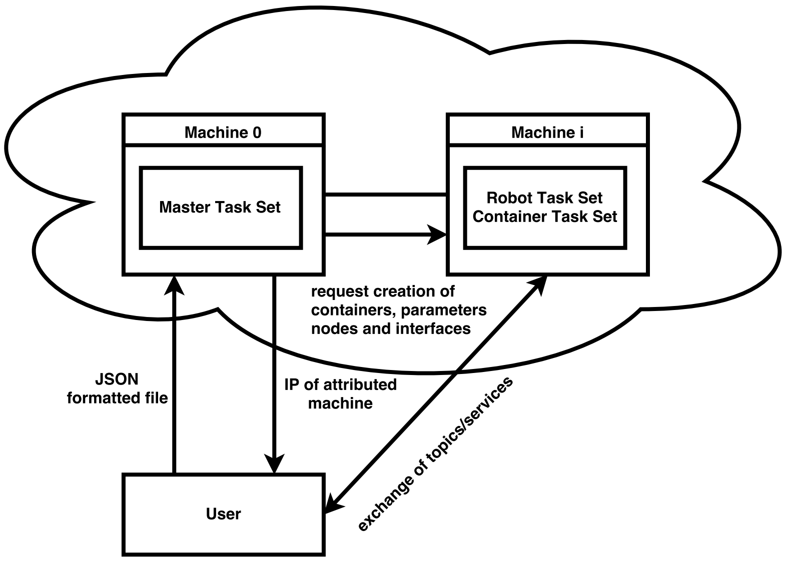

Figure 3.2: Process of connecting to the Cloud.

The interfaces allow for both local machine-to-container and container-to-container communication. At its core, ROS establishes a messaging protocol among running nodes (refer to section 2.4) and since interfaces deal with topics and services transportation they play a crucial role in the setup of a working ROS system. While on a traditional ROS system the communication among nodes is seamless, meaning that the publishing or subscription of a topic comes with no additional work from the user's point of view, if a node running on Rapyuta's Cloud environment subscribes to a topic being published on a local machine or on a different container, this topic must be explicitly propagated via interfaces.

There are two types of interfaces: converters and interfaces. Note that so far we have used interface in the common sense of the word, however, here interface is a type of method of communication that is hold by an endpoint of either a container or a local machine's ROS system (refer to section 2.1.5). While interfaces handle communication from one container to another, *i.e.* carry a published topic or provided service, converters handle communication from a container in the Cloud to a user's local machine. The main reason for the difference is in the communication hardware. While it is safe to assume that all machines in the computing cluster will be connected via Ethernet, it is safe to assume that users (namely robots) will connect wirelessly. That being said, interfaces simply mirror topics as wired connections are fast enough to handle any topic or service independently of its size. On the other hand, converters transform any ROS messages in binary blobs before being sent. This lack of data meaning allows for compression algorithms to be applied to the message. For example, Rapyuta comes predefined with a converter that applies the lossless PNG compression technique to ROS *sensor_msgs/Image*. This mechanism can be exploited to greatly reduce the required bandwidth, though it introduces an overhead of a few milliseconds [44]. Whenever a robot connects to Rapyuta, a ROS node is executed, both locally and inside the containers, that subscribes and publishes the topics defined by the interfaces created.

The fact that connections must be explicitly created brings the advantage that topic names can be modified to suit occasional needs. For instance, consider a case where a simulation is being carried with two robots and there are two containers in the Cloud receiving each analyzing sensor readings from a single robot. These containers can be launched with the same nodes and without topic remapping, since this remapping can be done by giving the correct name to the interfaces: *robot_ 0/scan* local topic will be connected to the *scan* topic on one container while the *robot_ 1/scan* will be connected to the *scan* topic of the second container.

By nature, Rapyuta implements a multimaster system, as in a common case there are at least two master *roscore* running at a given time: one running in the local machine and one in the container. Moreover, Rapyuta is blind to the number of different *roscore* that connect to it, which proves to be an easy tool to setup multimaster systems (Fig. 3.3).

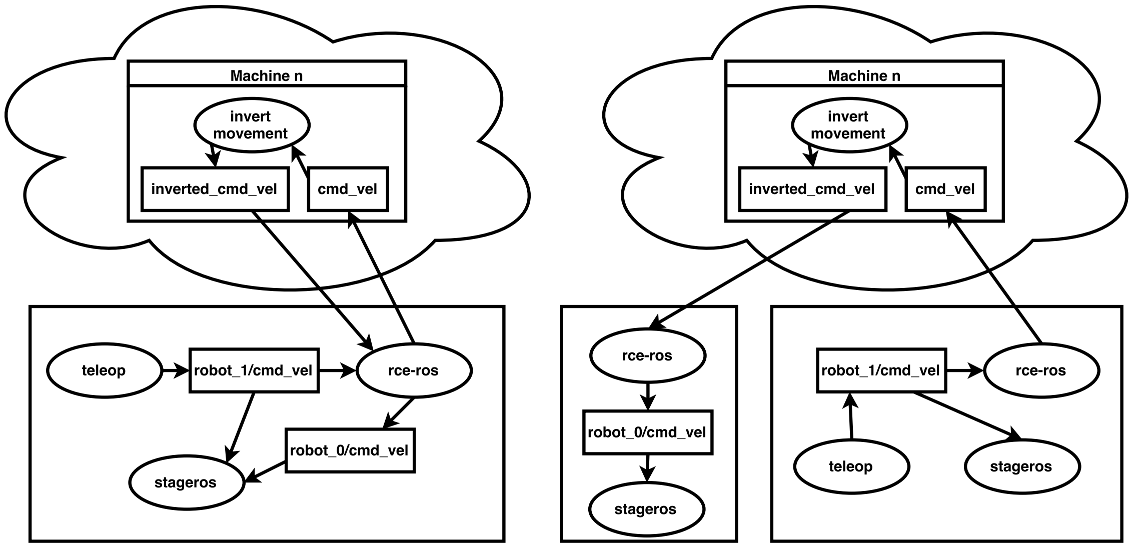

Figure 3.3: Comparison between a *single* and *multimaster* systems interacting with Rapyuta's Cloud environment. In both systems two robots are simulated in stage [13] and robot_1 mimics the movemt of robot_0 in reverse.

On a side note, Rapyuta's messaging mechanisms can be explored to create a multimaster topic and service distributor, since it can be used to pass ROS messages from one machine to another even without the creation of containers or the execution of nodes in the Cloud.

3.2 Summary

This chapter went over Rapyuta's mechanisms to setup and use the Cloud, from both a user the Cloud provider perspectives. Below is a quick overview of the steps to setup and use Rapyuta's Cloud environment.

To setup a Cloud or computer Cluster:

1. Launch the master task set on a single machine. This machine's IP must be known by all other machines, including the users'.
2. Launch the robot and container task sets on any number of machines to form the cluster. The machine that launched the master task set may optionally launch these two task sets as well.
3. The machine referred on 1. will be contacted by users to create computing environments. It does so by forwarding communications to the machines referred in 2., selecting a machine based on free processing resources.

To use Rapyuta as a user:

1. Launch a regular *roscore*.
2. Contact master task set on the Cloud, providing a JSON formatted file containing the desired container, parameter, nodes and interfaces.
3. A node is launched mirroring topics and services to/from the Cloud.

The next chapter will go over the adaptations made to the selected multi-robot tasks so as to work under Rapyuta's cloud environment.

4 Two Use Cases of Rapyuta

4.1 Use Case 1: Multi-Robot SLAM

4.1.1 The *mrgs* ROS stack

Moving the onboard processing to the Cloud has its array of advantages and compromises that were discussed in chapter 2. In order to make a Cloud-based solution more interesting, only minimal efforts ought to be required to make such transition. In other words, significant changes to the traditional system should not be needed. In order to prove this kind of flexibility, a working multi-robot SLAM solution was adapted in order to run under the Rapyuta Cloud environment. The solution used was *mrgs* [41], a ROS stack developed in the AP4ISR Lab. at ISR — University of Coimbra that enables any working single robot SLAM technique to be performed by a team of robots, provided that such technique conform to ROS standards and outputs occupancy grids.

It should be noted that the ROS *fuerte* distribution was used, as Rapyuta was originally developed for this ROS version. Neither the more recent *indigo* distribution of ROS currently used in out lab, nor the version 14.04 of Ubuntu are currently supported by Rapyuta.

A robot running the *mrgs* multi-robot SLAM solution without resorting to a cloud-computing framework has five running nodes [41], as seen on Fig. 4.1 (note that the node *slam_ gmapping* is not a part of the *mrgs* stack).

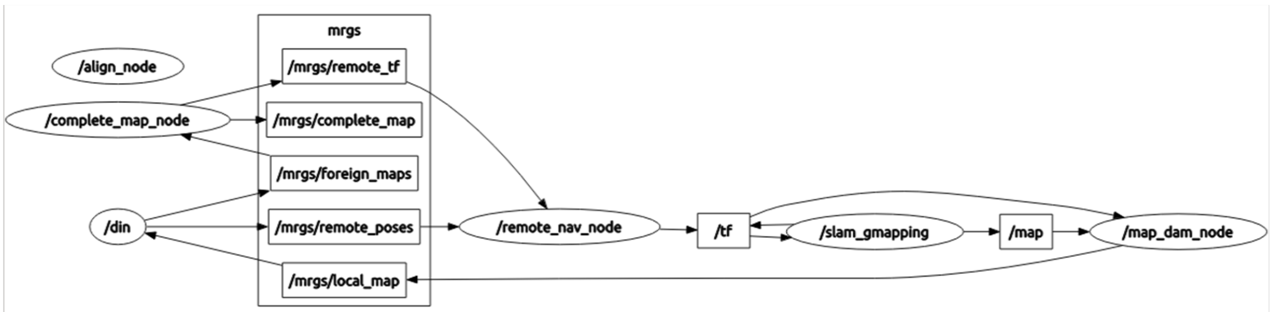

Figure 4.1: Interaction between the nodes and topics of the *mrgs* stack.

The *map_ dam_ node* is a simple auxiliary node that subscribes to the map topic, which is fed by a single-robot SLAM technique such as *gmapping*, and cuts out the excessive space that these techniques usually use (see Fig. 4.2). This map trimming will reduce the overall network load. The result of this operation is then forwarded to the *data_ interface_ node* by a ROS topic.

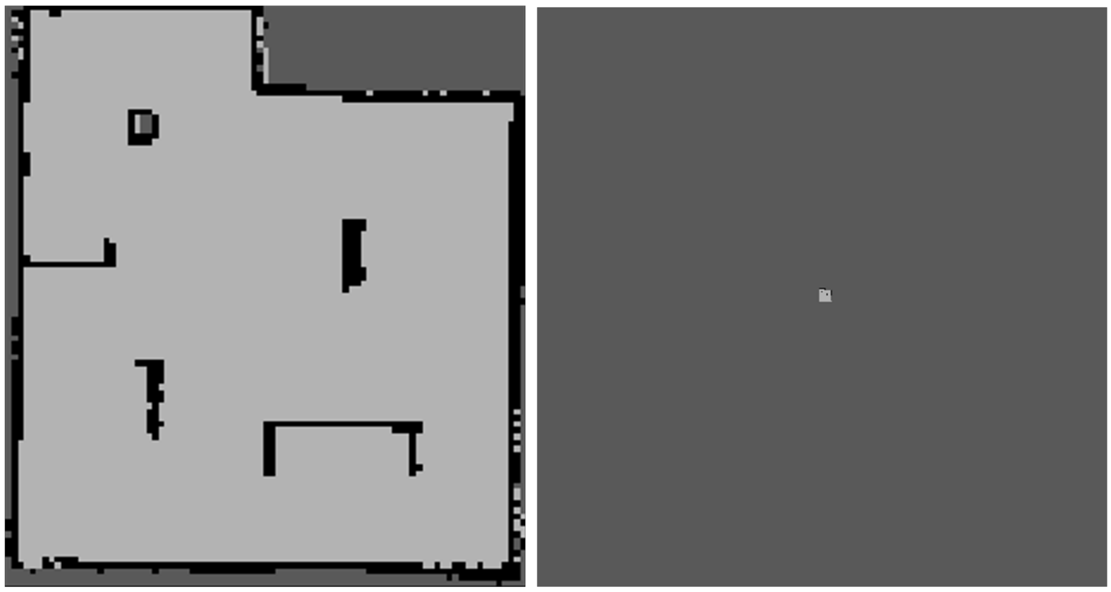
Figure 4.2: Comparison between the occupancy grid output by *gmapping* (right) and the one output by *map_ dam_ node* (left). While both occupancy grids contain the same amount of useful information, the one on the left is trimmed down to reduce the total size.

The *data_ interface_ node* is responsible for broadcasting the robot's local map output by the *map_ dam_ node* to the remaining robotic agents, while receiving their local maps. Note that these nodes implement a compression algorithm (LZ4) in order to reduce the bandwidth necessary during operations [42]. A map vector containing the local maps of all the robotic agents is then forwarded to the *complete_ map_ node*. Overall this node makes the bridge between a single- and multi-robot system, as all interactions flow through it.

The *complete_ map_ node* upon receiving an updated map vector calls a service provided by the *align_ node* building a global map in a hierarchical fashion. The following diagram illustrates how this node minimizes the number of fusion services it needs to call.

As the diagram in Fig. 4.3 shows, maps are coupled and merged. The resulting map from the merging process then moves up in the pyramid, being itself coupled with another map resulting from the same merging process. In the event of an odd number of maps, the last one automatically moves up. This process repeats itself until only one global map is found. Whenever one of the local (bottom) maps is changed only the subsequent maps that depend on it are remerged, thus avoiding costly and unnecessary merging operations.

The *align_ node* provides a service that receives two occupancy grids as input, computes a transformation (a translation coupled with a rotation) between them, and outputs a new occupancy grid representing the result of the merging process, based on the aforementioned transformation. The algorithm is based on computer vision (namely OpenCV libraries [7]),

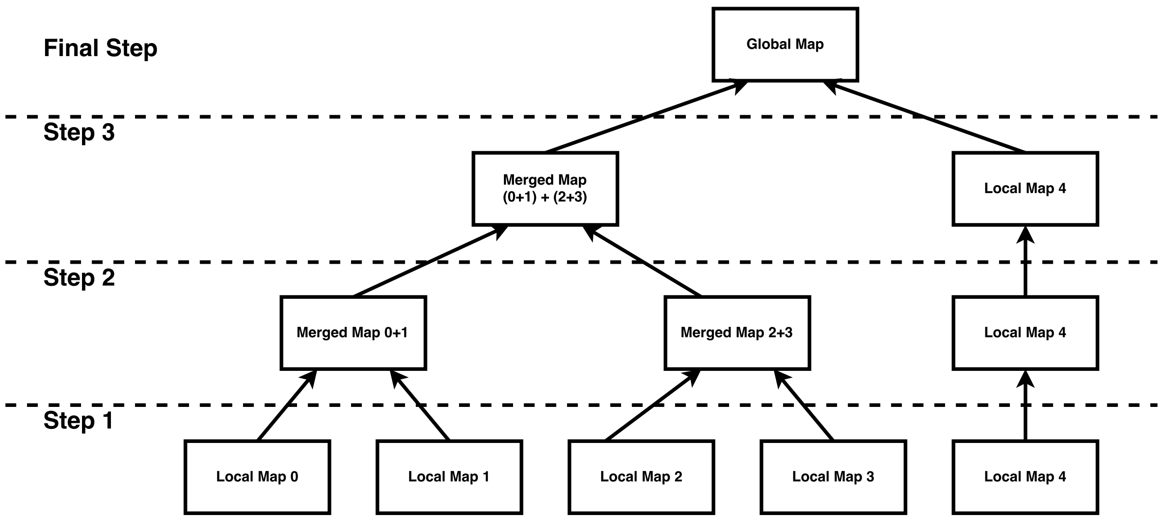

Figure 4.3: Representation of the tree-like process of map merging.

as such, the environment must be endowed with defining features in order to aid *align_node*'s algorithm in finding the correct correspondence between input maps.

Finally, the *remote_nav_node* is another auxiliary node, whose function is to propagate the transform information from the *data_interface_node* and the *complete_map_node* to the *tf* ROS topic. This sort of information is key in the correct representation of information.

These five nodes are modular enough to enable three different configurations [41]. If all of the nodes are running on each robot, the system assumes a distributed form, where each robot receives each other's local map and subsequently computes the global map. This type of configuration is more reliable and robust since there is no single point of failure. On the other hand, if only one robot (or base station) runs the *complete_map_node*, then the system takes the shape of a centralized one, where only that machine will compute the global map. While this kind of configuration introduces a single point of failure, the need for onboard computing power is considerably lower. The third possible configuration is a mixed one, where some of the robots merely explore the environment, while others receive and compute information, thus having access to the global map.

As stated before, all the multi-robot interactions pass through the *data_interface_node*, therefore this is where most of our attention will be focused. As is, in the case of a distributed configuration, the bandwidth requirements of *mrgs* grows linearly with map size but also exponentially with team size, as one local map must be shared with each one of the team's agents.

4.1.2 Contribution to the improvement of the *mrgs* ROS stack

As was stated in the previous chapter, the *align_node* provides a ROS service, which takes in two occupancy grids as input and outputs the result of the merging process. Despite working properly, albeit with the expected frailties, under a team of two robots, this node failed rather notoriously for larger teams. In other words, the merging of two maps that had already been merged themselves was unsuccessful. The figure below shows, the results of a team of three simulated robots.

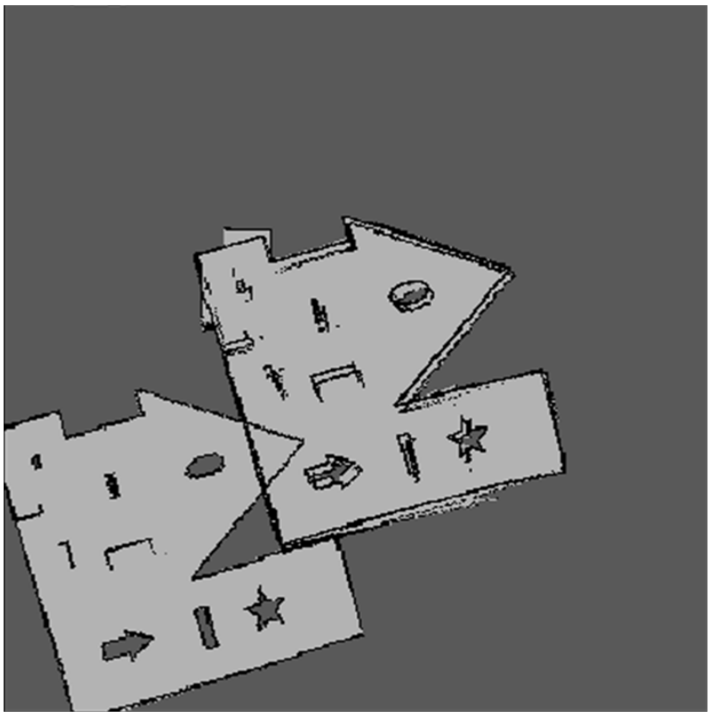

Figure 4.4: Failed merging process of three similar local maps.

In this case, the failure occurred while trying to merge the local map of the third robot with the resulting map of the merging of the first and second robots. Here we can clearly observe the recurring behavior of the system: the first two maps are merged properly with a slight mismatch, while the third one is placed way off the expected position. Note that this represents a redundant test, meaning that each local map had a representation of the global map, thus the *align_ node* ROS node should be able to properly merge all three maps, since every map was similar to one another.

The *mrgs* stack was developed in a modular fashion, so that the alignment node could be easily replaced with a more robust and efficient one. However, most of the existing solutions to this problem assume that the relative initial positions of the agents of a robotic team are known between each other, a condition that might not be true in some robotic situations, and that certainly restricts the possible application of this multi-robot system.

The solution was then to review the code and try to find the bug that was causing subsequent maps to be poorly merged with the first couple. It was found that, while the *align_ node* node assumed the maps it received were trimmed, the end map that was returned as output to the service was not. As a part of the merging process, padding to the maps is used in order to allocate sufficient memory resources, for example if one map continues the bottom part of the other, the final map must have increased size in order to accommodate all of the data in only one occupancy grid. Referring to Fig. 4.4 again, it is possible to see that the map contains unnecessary column and rows on the top and right sides.

Adding a simple function that trims the final result of the merging process, like the one the *map_ dam_ node* node uses (see section 4.1.1), resolves the problem: since the output of the first call to the alignment service is trimmed, when it is used as the input to a further alignment service call, it already obeys to the necessary conditions.

Although with this change the systems function properly for teams of more than two robots, the merging algorithm is still imperfect and lacks robustness. As it is based on computer vision, it greatly depends on map features and common areas in order to produce a reasonable result. The tradeoff of not relying on knowledge of relative initial positions to build the global map is the fact that the zones of the map need to be easily distinguishable.

4.1.3 Adaptation of the *mrgs* ROS stack to Rapyuta Cloud environment

In the traditional system, *mrgs* can be run in a distributed or centralized manner (see section 4.1.1). Adapting this stack to the Rapyuta Cloud environment maintains this ability, due to the flexibility provided by Rapyuta's messaging mechanisms.

Distributed mode

In the traditional system, this mode eliminates the single point of failure that is a base station, while all the tasks are run entirely by the robots. Adapting such architecture to the Cloud does not possess much merit at first glance, because if we can rely on the wireless network to export all the processing to the cloud, then we can also rely on the network to connect to a base station. However, in a centralized architecture if the base station fails, so does the system, while running a distributed mode over the Cloud means that if any of the machines from the Cloud cluster fails the system does not fail entirely, only the robots allocated to said machines will stop functioning as intended.

While in the traditional system a robot needs to run both a SLAM technique and the *mrgs* stack, in the Rapyuta's adaptation a robot only needs to run a node that transmits to the Cloud the necessary sensor readings and transforms (see Fig. 4.5), heavily relieving the robot of the need for local processing resources.

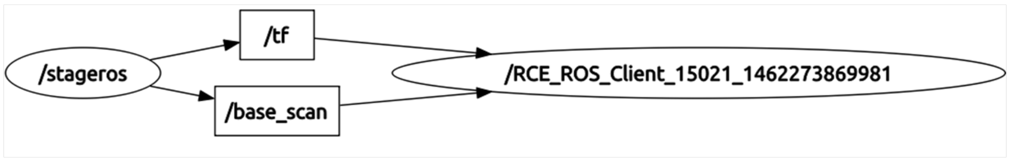

Figure 4.5: ROS environment of a robot that is running the *mrgs* stack on Rapyuta.

Centralized mode

From the robot's perspective nothing changes between this mode of operation or the distributed one, as all processing is done on the Cloud environment (Fig. 4.5). However, the differences between the adapted centralized system and the traditional centralized system diminish: while the bandwidth requirements stay the same, only the SLAM technique used is relieved from the robot with the Cloud adaptation in comparison to the traditional system.

Common to both modes of operation are the bandwidth requirement differences between the adapted and the traditional systems. While in the traditional system each robot sends each other compressed (using the LZ4 compression technique) versions of the local map, in the adapted system only sensor data and transforms are sent over the wireless network. Here the difference is somewhat difficult to compare. On the one hand, occupancy grids are extremely compressible (in [41] the techniques tested presented a compression ratio of at least 10), but their compressed size will vary greatly with map size and complexity. On the other hand, sending sensor readings and transforms guaranties a fixed bandwidth requirement, which can be controlled by the amount of readings done by the sensor. Moreover, in a distributed architecture, the traditional system's bandwidth requirement grows exponentially with the team size as each robot must send its local map to every other agent, while in the adapted system this growth is linear, since each robot only communicates to the Cloud despite team size.

Fig. 4.6 shows a graphical comparison of the aforementioned modes of running the *mrgs* stack.

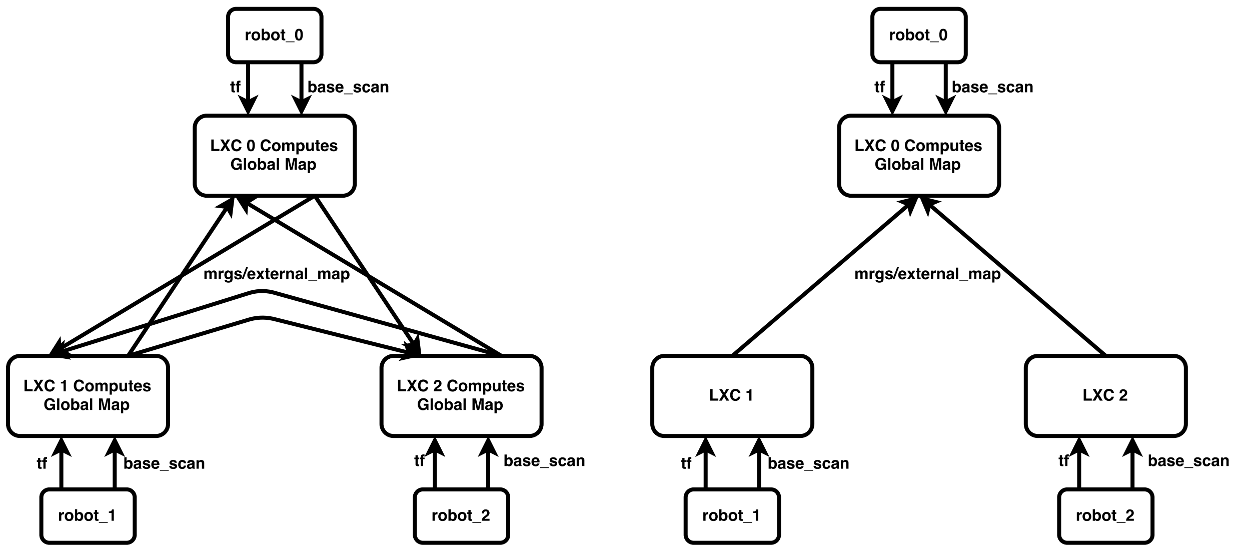

Figure 4.6: Comparison between distributed (left) and centralized (right) modes of operation under Rapyuta's Cloud environment. The boxes labeled with LXC (Linux Containers) represent containers running on Cloud machines.

The process of adapting the traditional system to work under Rapyuta's Cloud environment revolves around the removal of the dependencies on the *wifi_ comm* package [14], so that the system uses Rapyuta's messaging mechanisms instead. This package allows for the creation of a *multimaster* system by mirroring foreign topics on a local *roscore*. The messages that would ordinarily come in and out using these foreign relays are now replaced by two simple ROS topics: one that receives foreign maps and another that sends the robot's local map. This way, in distributed mode every robot has an input topic that is connected to the remaining robots' output topic and one output topic that is connected to the remaining robots' input topic. In the centralized mode, the central computer has only one input topic, while each robot has only one output topic. In order to allow for multiple robots to be simulated under one computer, the MAC address identification system that was originally present [41] was changed to an ID-based system given by the user.

Despite being flexible, Rapyuta's messaging mechanisms come with their setback. As all connections must be explicitly stated, the robotic team size must be known *a priori*, *i.e.* the team can increase in number over time but a new agent must be aware of all remaining active agents in order to make the necessary connections. This knowledge was not required in the original system. It is, however, a minor concern since it is the user that owns the connections and not the robots, thus they do not need to be made during launch time.

4.2 Use Case 2: Multi-Robot Patrolling

4.2.1 The patrol_isr_demo ROS package

The multi-robot patrolling problem was selected as a second use case, since some other task is usually included with it. In other words, while in a SLAM scenario a team of robots can be deployed solely to map a certain area, in this scenario the act of patrolling itself is futile if not coupled with some other task such as intruder detection or object search. Having said that, a Cloud computing architecture could benefit the patrolling act in two ways: offloading the computer resources required to navigate through the environment to the Cloud would allow for more resources to be available to the secondary task; or offloading every task to the cloud, including the task of perceiving abnormal situations (*e.g.* using image processing to detect intruders or unexpected objects), which can be computationally demanding, would mean that cheaper robots with less computational resources would be able to patrol as well as robots with considerable onboard resources.

In order to prove this concept the *patrol_isr_demo*, a ROS package developed in the AP4ISR Lab., was chosen to serve as use case [52]. The basic principles described in section 4.1 were applied here as well, meaning that changes made to the original system should be kept to a minimum in order to prove the Cloud environment's flexibility.

The *patrol_isr_demo* package takes advantage of the navigation stack, namely the *move_base*, *amcl* and *map_server* nodes [5], in order to provide a single robot a way to move through the environment. These three nodes work together so that given a goal pose (meaning position and orientation coordinates) they output the correct velocity commands in order to reach said goal, while avoiding unexpected obstacles. These nodes require not only sensor data and transforms but also odometry information. Note that, once again, despite the concept of stack no longer existing in newer versions of ROS, all work was developed in ROS *fuerte* as Rapyuta has not yet been updated to support a newer ROS version.

The *patrol_isr_demo* implements the Travelling Salesman Problem and the Concurrent Bayesian Learning Strategy (CBLS) methods of patrolling [52]. The TSP technique implemented in the *TSP* ROS node was selected as it is well known but also offers more predictable agents' trajectories. The package also implements a *monitor* node, which only aids in the synchronization of the robotic team's agents and logs all the data, thus the node can be shut down after the patrolling starts.

The patrolling task is achieved by the cooperation between the *TSP* and the previously mentioned navigation stack nodes (Fig. 4.7). The *TSP* node computes a trajectory based on the algorithm with the same name and informs the remaining agents of its intentions. Secondly, it provides a goal (or target pose) to the navigation stack, which represents the next vertex to be visited. The navigation stack then takes care of moving through the environment as well as avoiding obstacles.

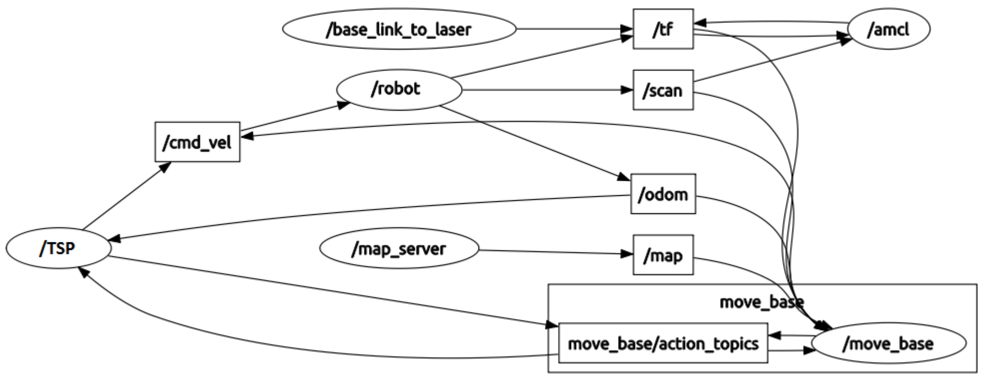

Figure 4.7: Interaction between the nodes and topics of the *patrol_ isr_ demo* package.

In order for the *TSP* node to compute a valid route and to send valid goals, two files containing detailed map information are needed. As was discussed in section 2, it is common for topological information from a map to be used as a way of determining the regions of interest (or vertexes) in the Multi-Robot Patrolling problem. With this purpose, a simple text file containing all the necessary information to make a web of vertexes is used to represent a map. This information includes coordinates, neighbours, direction of each neighbour and the cost to move to each one. Since the optimal TSP route (in case one exists) does not change over time, another file containing this route is fed to the node so that it does not need to be computed every time. Having established the general TSP route, the node has to determine afterwards where to start (since the route is cyclical). To do so, it consults a coded list of initial positions that, based on the robot's ID, provides a correct starting position. However, this assumes the robot with a given ID always starts on the same position.

Upon starting execution, communication among agents is required to compute a degree of proximity and to adjust trajectories accordingly, so as to avoid potential collisions. Finally, each goal provided to the navigation stack nodes are based on the coordinates retrieved from the file containing the map's topological information.

Throughout the whole execution of the navigation stack nodes, a considerable amount of computer resources are consumed, even if the robot is held to a stop, since the nodes are continuously scanning for obstacles and computing/sending velocity commands.

In similarity with the use case of section 4.1.1, the *patrol_ isr_ demo* uses the *wifi_ comm* package in order to establish communication among the robotic team's agents, thus this was the main focus point in the adaptation of this package to the use on Rapyuta's cloud environment.

4.2.2 Adaptation of the *patrol_ isr_ demo* ROS package to Rapyuta Cloud environment

There are three main components that need to be launched to start patrolling: a single monitor to synchronize all agents, navigation stack nodes running on all agents and a node that implements a patrolling algorithm also running on all agents. Inter-agents communication is done via a single topic that connects the monitor and all algorithms implementing nodes.

This single topic was spread over the *multimaster* setup by the *wifi_ comm* package that mirrors foreign topics. In this way, the dependency on this package was removed and replaced by simple publishers and subscribers, similarly to the distributed case of *mrgs* described on section 4.1.3. The package had traces on the code that indicated it had been tested under a single master setup, in other words, the system had a hidden ability to function with many robots working under a single *roscore* (in such case, there is no usage of the *wifi_ comm* package). This meant that, with the flexibility Rapyuta's messaging mechanisms offer, adapting the traditional system was a matter of having the algorithm implementing nodes believe they were running on a single master environment while using Rapyuta's abilities to distribute messages over a *multimaster* one (Fig. 4.8).

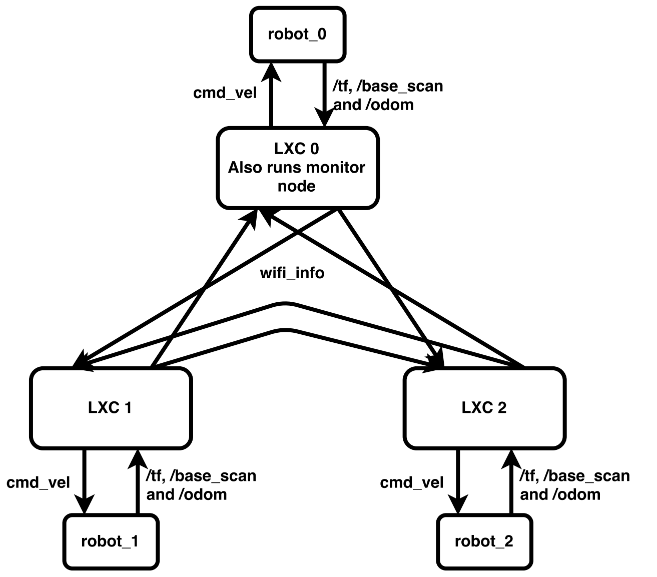
robot_0
cmd_vel
/tf, /base_scan and /odom
LXC 0 Also runs monitor node
wifi_info
LXC 1
LXC 2
cmd_vel
/tf, /base_scan and /odom
cmd_vel
/tf, /base_scan and /odom
robot_1
robot_2

Figure 4.8: The *patrol_ isr_ demo* package and the navigation stack nodes running on Rapyuta's Cloud environment. The boxes labeled with LXC (Linux Containers) represent containers running on Cloud machines.

Again, similarly to section 4.1.3's distributed case, the bandwidth requirements for this package grow exponentially with team size, since all agent communication is a broadcast to the remaining. However, messages consist of four *int8* integers, making it realistically always lower than the bandwidth needed to run the system on Rapyuta, which grows linearly with team size.

Most of the workload comes from the navigation stack nodes, due to the constant execution path planning and obstacle avoidance tasks. This means that to take advantage of a Cloud environment all of the processing must be offloaded, thus bandwidth requirement is being tradeoff with CPU time.

Another consideration with the adaptation of the traditional system is how the navigation stack's parameters are launched, as Rapyuta does not provide the possibility of loading *yaml* files that are commonly used in ROS to configure complex nodes in a flexible way. However it does provide a way of launching parameters one by one, thus all *yaml* files must be incorporated in the configuration file used to launch a container or a node in the Cloud environment.

Moreover, the value the parameters used in traditional system should be reconsidered when adapting to the cloud, since wireless connections take a considerable amount of time. This means that the agents' maximum velocity should be lowered in order to allow more time to react to an unexpected obstacle. In the same train of thought, the trajectory generation should be comparatively more conservative, *i.e.* agents ought to corner further from the wall to give enough time in case of a miscalculated route or trajectory slip.

4.3 Summary

This chapter described the use cases chosen to work as proof of concept of a Cloud-based solution to a multi-robot task. A detailed comparison between the traditional and the adapted systems was also presented, as well as what changes were made in the adaptation process and what tradeoffs are incurred upon offloading the chosen tasks to the Cloud.

These theoretical assumptions are further explored in the next chapter, where live simulations are conducted to demonstrate empirically the expected results.

5 Results and Discussion

As discussed previously, the main characteristic that a Cloud-based solution provides to a robotic system is the scalability it provides in terms of computer resources. Not only these resources can be made seemingly infinite to the robots, but also forming a large robotic team is made much cheaper. As such, we believe experiments involving such a team should be conducted as a proof of concept of this highly desirable characteristic. While other works have already taken advantage of the Cloud for external processing [54] [45], only experiments with two robots were conducted. Thus, we carried out experiments with twenty robots in a simulation environment using the *stage* simulator available in ROS. We believe simulated robots are only justifiable if the order of magnitude of the size of the team of simulated robots is greater than the one achievable with the existing live robots. On the other hand, we only had available three desktop machines with Intel® Core™2 Quad Processor Q6600, Intel® Core™2 Quad Processor Q9300 and Intel® Core™2 Quad Processor Q9400, which limited the Cloud's resources and the size of the robotic team to the specified twenty.

Apart from providing with a proof of concept, it was also our objective to evaluate the tradeoff between CPU time and bandwidth that comes with a Cloud-based approach, from the point of view of a single robot. Despite our beliefs that a Cloud-based solution is more attractive when considering larger scales, this characteristic comes solely from the fact that each individual in a robotic team can be considered weak in terms of computer resources, so the point of view of the robot is also a key aspect. With this in mind, experiments were conducted that used the *ps* Linux command in order to register the CPU time spent by the considered robotic tasks in both the traditional and the Cloud-adapted systems. For bandwidth measurements we used both the command rostopic and the program nethogs available in a Linux system with ROS.

The use of simulation time and clock mismatches in the Cloud and local machines can cause timing mismatch errors, especially on the *tf* ROS topic. Moreover, delay in the transmission over the network of these time sensitive topics can further increase the occurrence of timing errors. However, most of these errors, and in our use cases all of them, are artificial errors, meaning that they are caused not by a fault in the system but because of lack of clock synchronization and the fact that the Cloud machines receive topics with a transmission delay. In order to solve this, we implemented a ROS node that runs on every container and subscribes to every time sensitive topic and republishes them with the machine's local current time. Due to the topic naming that Rapyuta's interfaces provide, these topics do not overlap. Still on this topic, it is important to note that these timing considerations can have a negative impact on the robotic task or even make it unreliable to run in the Cloud. This is specially true for tasks with strict hard real-time constraints. While clock desynchronization between machines can be easily overcome, delays caused by network transmission can not.

5.1 Multi-Robot SLAM Simulation

As stated above, two types of simulation experiments were conducted. On this first experiment, as a proof of concept, we launched twenty robots on the *stage* simulator [13] in a multimaster setting, meaning that there were twenty *roscore*'s and twenty instances of the *stage* simulator. This configuration resembles the most that of an actual team of twenty independent (with its own *roscore*) robots. On the side of the Cloud, there was one container running the merging and global map building nodes and twenty other containers (one for each simulated robot) running the local map building nodes.

Each simulated robot feeds the laser scan data and its transforms to the *rce-ros* node, which propagates them to a container in the Cloud under a different topic name due to the aforementioned timing constraints. These topics are then republished under their normal name and with a new timestamp to be read by the *gmapping* node. When a new local map is generated, the *map_ dam_ node* processes it and the *data_ interface_ node* propagates it to the merging container.

Fig. 5.1 represents a successful experiment of this setup. Each of the simulated robots discovered the central area and one of the twenty outside branches. The end result was a global map similar to the original. Predictably, the aligning process is not perfect, which results in an accumulative error evidenced by the thick lines and the cloudy effect on the common areas.

A second set of experiments centered around the point of view of the robot, rather than the results of the system as a whole, was conducted in order to evaluate and better comprehend the tradeoff between CPU time and bandwidth imposed by this Cloud solution. On the previous experiment there was one different trajectory for each of the robots, so now we repeated the same trajectories but this time we registered the CPU time demand in both the traditional and the Rapyuta-adapted systems.

For the case of the Rapyuta-adapted system, only one node (*rce-ros*) is running at all times, while on the traditional system, under a centralized mode, there are three: *gmapping*, *map_ dam_ node* and the *data_ interface_ node*. However, the *data_ interface_ node* CPU time demand was not registered because it was negligible.

As Tab. 5.1, Fig. 5.2 and Fig. 5.3 show, the difference in terms of computational power required to run the system on both approaches is stark. Not only do you save up a lot of CPU time on the Cloud approach, but you also gain a more predictable CPU consumption.

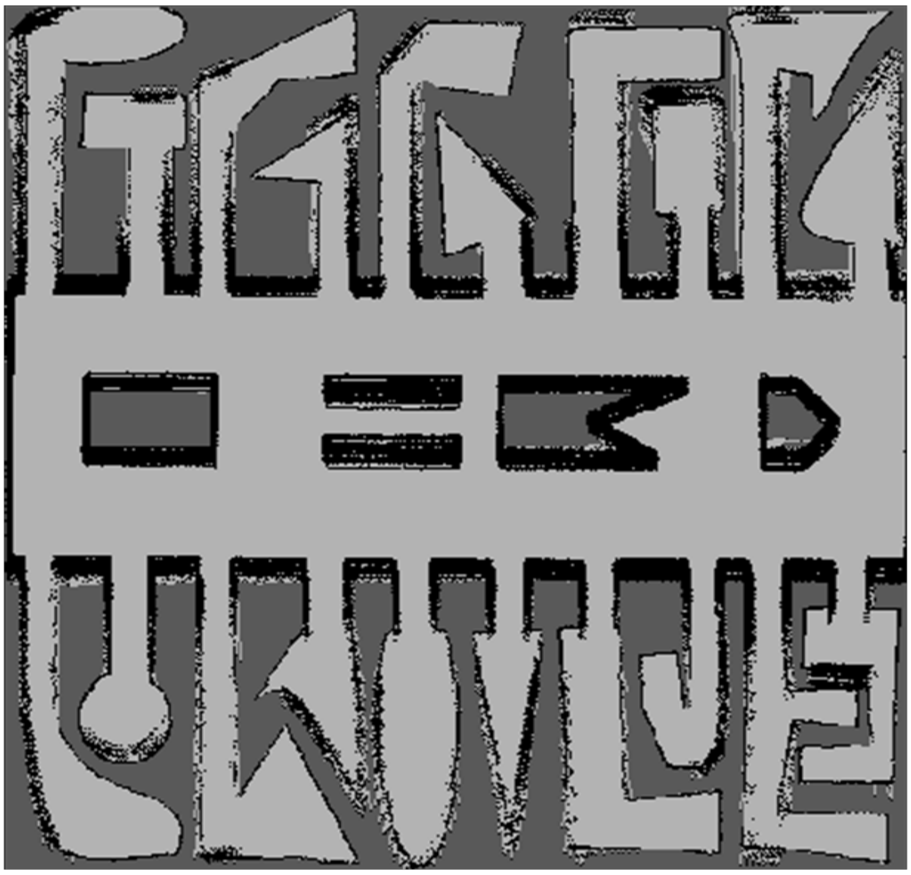
Figure 5.1: Global map generated by the merging process of twenty different local maps.

Table 5.1: Comparison between the CPU time used by the *mrgs* stack on the traditional and Rapyuta-adapted systems from the point of view of one robot.

	Mean (%)	**Max (%)**	**Min (%)**
Rapyuta-adapted system	2.4	2.5	2.3
Traditional system	(41.5+3.5)=45.0	(42.5+3.7)=46.1	(40.2+3.4)=43.7

While the *rce-ros* node requires the same amount from the CPU, the nodes run under the traditional system might require more or less depending on environment complexity or the robot speed through the environment. This is evidenced by the fact that the first, second and third quartiles are stacked on each other for the Cloud-based system (Fig. 5.2).

Although the differences in CPU time are clear and significant, the bandwidth aspect is trickier to evaluate. The Cloud approach requires a steady amount of bandwidth as it is moving local topics that are published at a predefined rate (10 Hz in this case) to the Cloud. On the other hand the traditional system does not possess this characteristic, making it hard to register non-faulty values of bandwidth consumption. Moreover, in this case, the required amount of bandwidth depends a lot on the environment. As the maps are compressed, a larger amount of features in the environment will lead to a worse compression rate, while a blank featureless one will lead to a higher one. The size of the environment will also take a considerable impact on the final size of the maps to be sent to the network. Maps are also not sent at a steady rate but only when a significant change to the local map is detected, thus the speed of the robot is another variable that impacts the traditional system's bandwidth

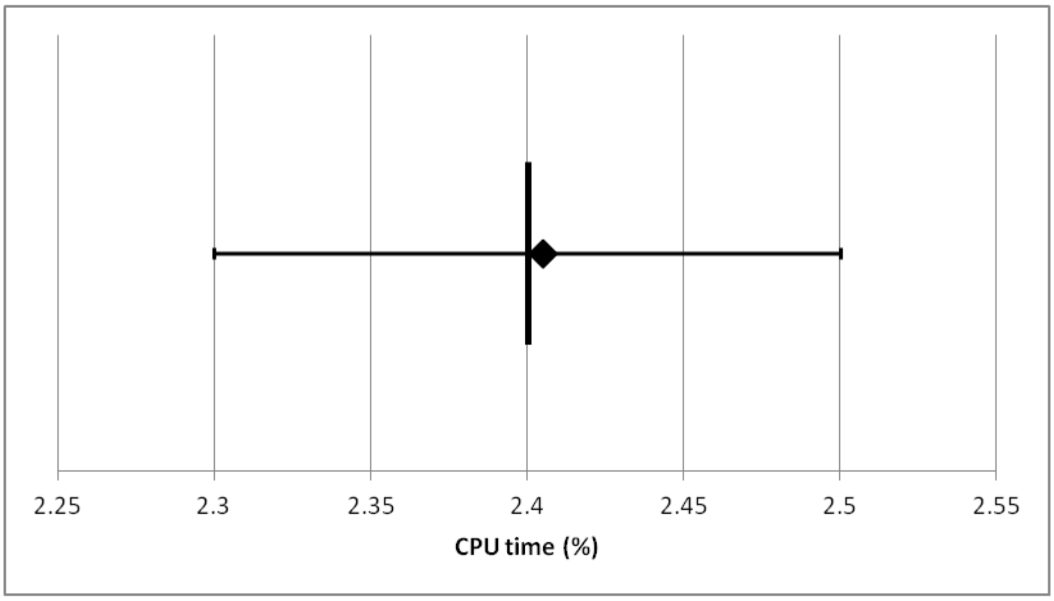

Figure 5.2: Box plot of the CPU time used by the Rapyuta-adapted *mrgs* system from the point of view of one robot.

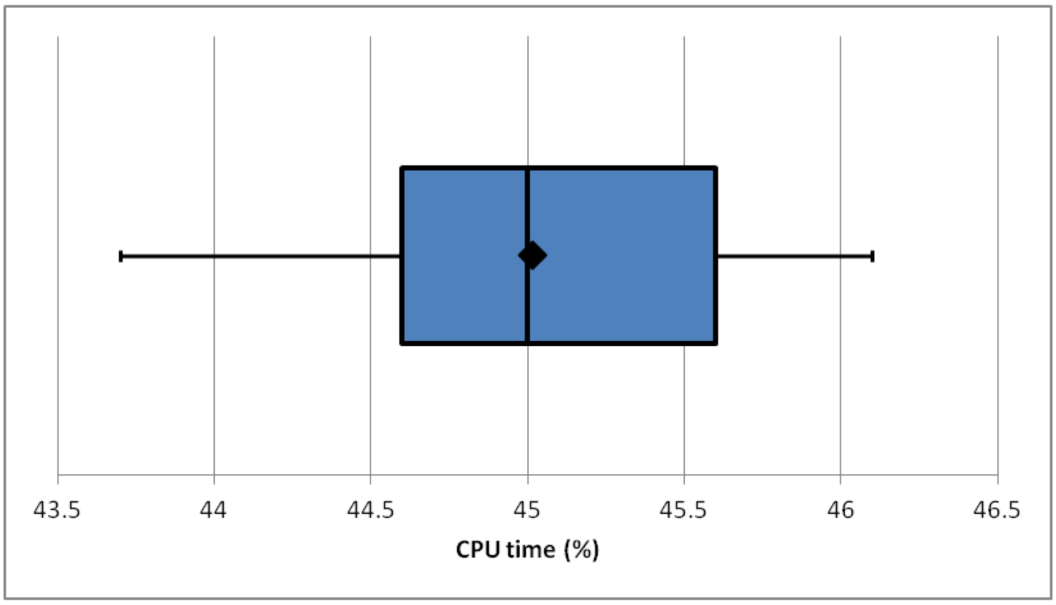

Figure 5.3: Box plot of the CPU time used by the traditional *mrgs* system from the point of view of one robot.

requirements.

Having said that, we registered the bandwidth used for the Rapyuta-adapted system and the amount of data propagated through the network by the topics of the traditional system. The latter will give an idea to which degree the map size changes even on our simple and small sized (25x25 meters) map. This will not result in a direct comparison between both, but rather in an analysis on how many robots could be viably deployed under current Wi-Fi standards.

Although the amount of data sent to the Cloud is steady (see Tab. 5.2 and Fig. 5.4), there is still some variation, due to the fact that TCP requires acknowledge packages to be sent, as well as error checking that might lead to some being resent. Nevertheless, the bandwidth required by each robot is somewhat predictable. Additionally, laser scan length and sample rate are variables that we control and that have great impact on bandwidth, which gives the system an appealing flexibility.

Table 5.2: Bandwidth required by one robot running the Rapyuta-adapted *mrgs* system.

Rapyuta-adapted System	**KB/s**
Mean	31.0
Max	35.5
Min	25.9

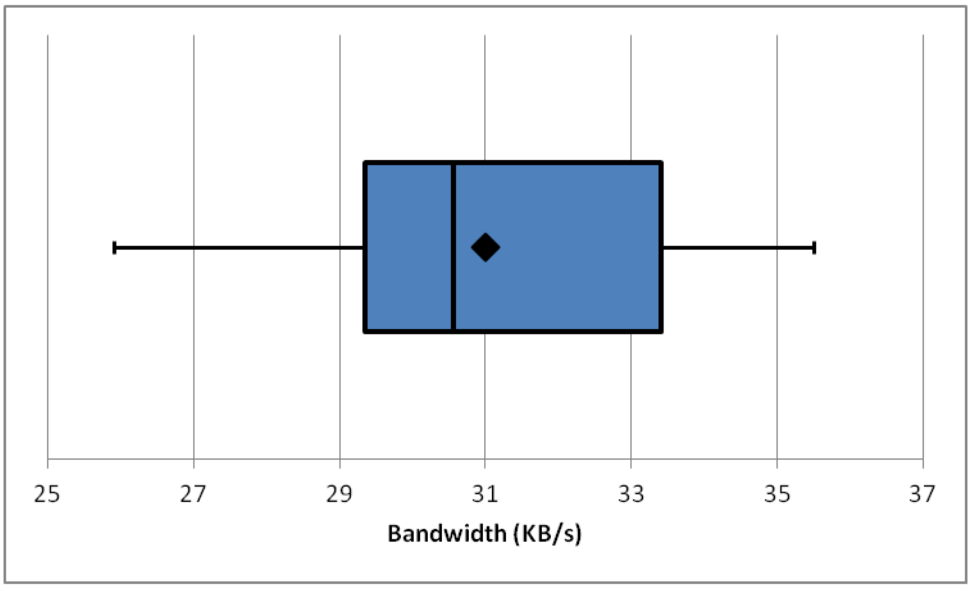

Figure 5.4: Box plot of the bandwidth used by the Rapyuta-adapted *mrgs* system from the point of view of one robot.

A modern and off the shelf 802.11n compatible router can provide a wireless speed of 65 Mb/s (or 8125 MB/s), which can serve, in theory, a total 228 robots, if we consider they require the maximum value of 35.5 KB/s. While theoretical values might be far from practical ones, the resulting number is high enough to comfortably say that a dedicated network could serve any realistic team of robots.

Furthermore, with the right setup, the 802.11n protocol is reported to reach speeds of 600 Mb/s and the newest 802.11ac over 1Gb/s. Again, despite being only theoretical values, they give a sense on how advanced wireless networks are nowadays, and that they could easily provide a good enough connection to a large robotic team.

In the case of the traditional system, instead of simply measuring the consumed bandwidth by the *data_interface_node* (node that sends/receives local maps), we registered the size of the topics used to propagate local maps through the robotic team (Tab. 5.3). Due to the unsteady way maps are generated, *i.e.* after a map is sent there is a considerable period of time where there is no bandwidth usage, reading the bandwidth would result in improper values. Thus we present the growth in size of the topic over the course of the experiments.

Table 5.3: Size of the topic exchanged among robots running the traditional *mrgs* system.

rostopic	**Mean**	**Max**	**Min**
Mean (KB)	26.4	27.7	25.0
Min (KB)	5.3	5.6	5.3
Max (KB)	35.8	38.0	34.2
Average rate (KB/s)	4.9	5.2	4.5

This gives us a rough estimate of the volume of data transmitted over the network in the traditional system. Note that the actual values would be a bit higher due to TCP related issues already mentioned.

The first point to note is that the average rate is negligible due to the fact that local maps are only sent once significant changes have occurred. During the experiments each robot only sent around 22 local maps, over the course of 2 to 3 minutes. However, while in the Rapyuta-adapted system, the volume of information is steady and predictable, in the traditional system, network load comes in the form of bursts, which is not an appealing feature.

Secondly, the table perfectly shows clearly the high variability on map size growing from a minimum of 5.3 KB to a maximum of 38.0 KB. This tells us that we can never predict the total bandwidth necessary of the whole system, as the size and rate of messages depend on uncontrollable variables, and transmission over the network in a burst fashion.

5.2 TSP Patrol Simulation

Similarly to the previous use case, we conducted two experiments with two different focuses for the multi-robot patrolling use case. On the first experiment, a *stage* simulator with 20 robots fed data to twenty containers on the Cloud (see Fig. 5.5). Note that in this case, we did not run the experiment in a multimaster setting, solely for visualization purposes. Although we could have run the system with twenty different *roscore*'s, this would have made it difficult to show the results of the patrolling of the whole environment in action. The containers in the Cloud ran every node associated with the patrolling package, including the node needed to republish topics under a valid timestamp, and except for the monitor node that was only run in one of the containers.

In order to evaluate the differences in CPU time required by each of the systems, we conducted a second set of experiments. In these experiments, only a single robot was deployed in the *stage* simulator, in order to measure this time from the robot's point of view. We identified the *move_ base*, *amcl* and *TSP* nodes on the traditional system as the ones that required a relevant amount of CPU time. As Tab. 5.4 shows, CPU time is one order of magnitude lower in the Rapyuta-adapted system, when compared to the traditional one, which is a considerable gain.

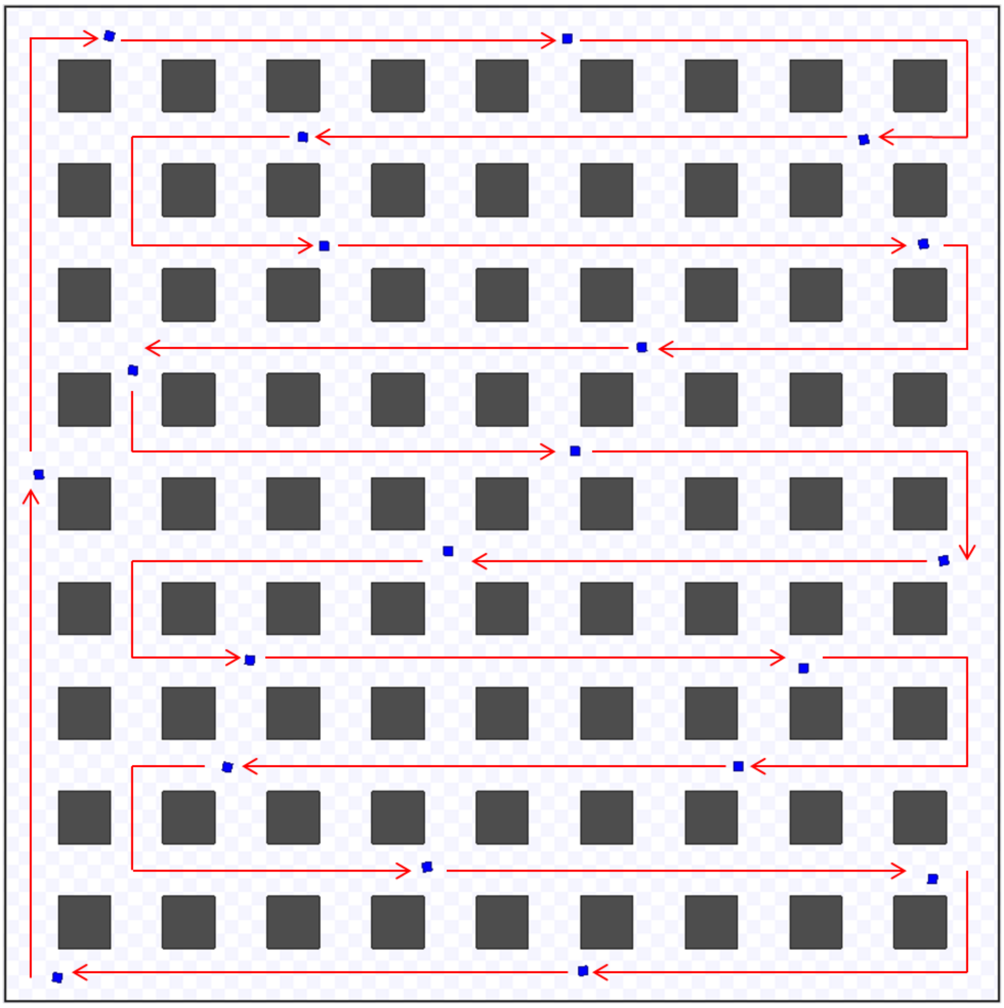

Figure 5.5: This figure represents a successful experiment with 20 simulated robots patrolling a TSP generated trajectory. The considered vertexes are placed in a grid-like fashion, where each square is surrounded by four vertexes to a total of one hundred.

Table 5.4: Comparison between the CPU time used by the *patrol_isr_demo* stack on the traditional and Rapyuta-adapted systems from the point of view of one robot.

	CPU time (%)
Rapyuta-adapted System	3.8
Traditional system (move_base+amcl+TSP)	(34.6+3.4+1.1)=39.1

Another experiment was conducted to check if obstacles increased or not the CPU time required by the *move_base* node. Obstacles were introduced in the *stage* simulator as shown by Fig. 5.6. However no significant changes were observed on the usage of CPU time, thus further tests were set aside.

In this case, as robots only exchange vectors containing four elements of type *int8* (total of four bytes) among each other, there is little reason to compare bandwidth requirements between the two systems as the Rapyuta-adapted one will always require much more than the traditional one. Thus, this is a rather extreme case of the tradeoff present when moving a system to the Cloud: although it is possible to reduce CPU time by a considerable amount, the bandwidth required to do so can be much larger. However, as stated before, wireless

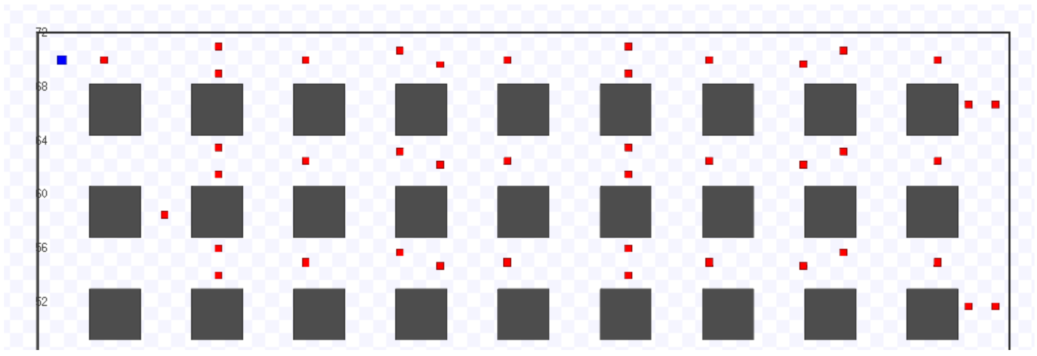

Figure 5.6: Environment used while testing the impact of obstacles in the usage of CPU time.

technology is increasingly more powerful and a dedicated network for robots can provide with more than enough bandwidth for any realistic robotic team.

Tab. 5.5 and Fig. 5.7 present the results of consumed bandwidth registered in the course of twenty experiments. On each experiment, one robot patrolled an environment similar to Fig. 5.5 for around six minutes. Despite the theoretical value of required bandwidth to transport the necessary local topics to the Cloud being constant, there are external variables related to the traffic of the network that can increase this value in practice, thus the need for more than one experiment.

Table 5.5: Bandwidth required by one robot running the Rapyuta-adapted *patrol_ isr_ demo* system.

	KB/s
Average	47.1
Max	53.5
Min	39.4

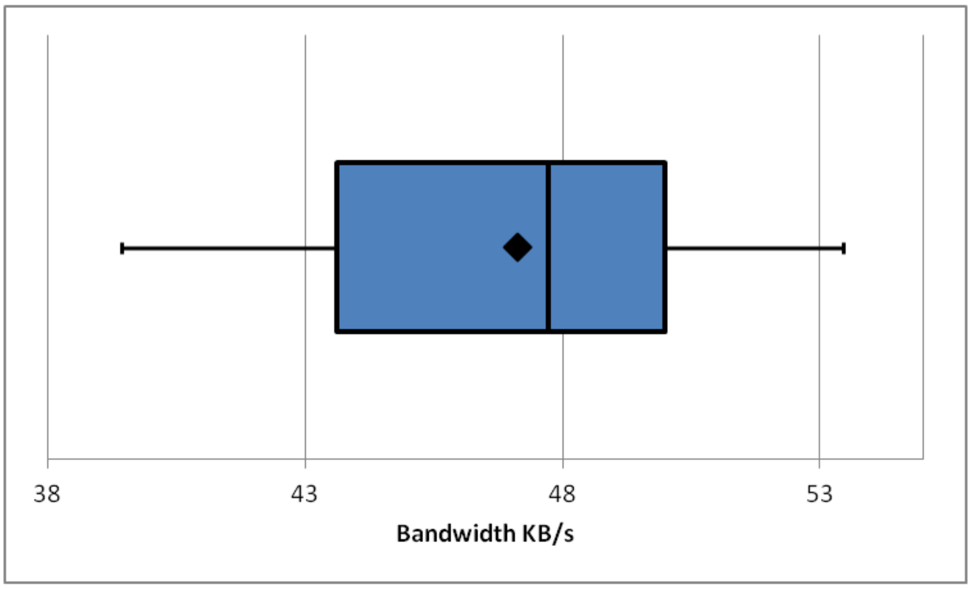

Figure 5.7: Box plot of the bandwidth used by the Rapyuta-adapted *patrol_ isr_ demo* system from the point of view of one robot.

Once again, despite the Rapyuta-adapted system requiring a lot more bandwidth than the traditional one, a common off-the-shelf home 802.11n router that provides a speed 65 Mb/s could theoretically support 165 robots. Keep in mind that this protocol can reach speeds of up to 600 Mb/s and the more recent 802.11ac can reach over 1 Gb/s. Nowadays' wireless Internet speeds give enough bandwidth room to consider a Cloud-based solution for multi-robot systems.

5.3 Experiments with real robots

In order to further prove the concept, experiments with real robots were conducted [1]. These experiments had the objective of proving the applicability of a Cloud-based solution on real world scenarios. Although simulation provides a close approximation to reality under ideal conditions, real experiments are subject to erroneous sensor readings, robot slips during movement and a not ideal wireless Internet connection. All of these impact the final outcome, and it is important for the system to be robust enough to handle such imperfections if the system is to be applicable outside of the laboratory.

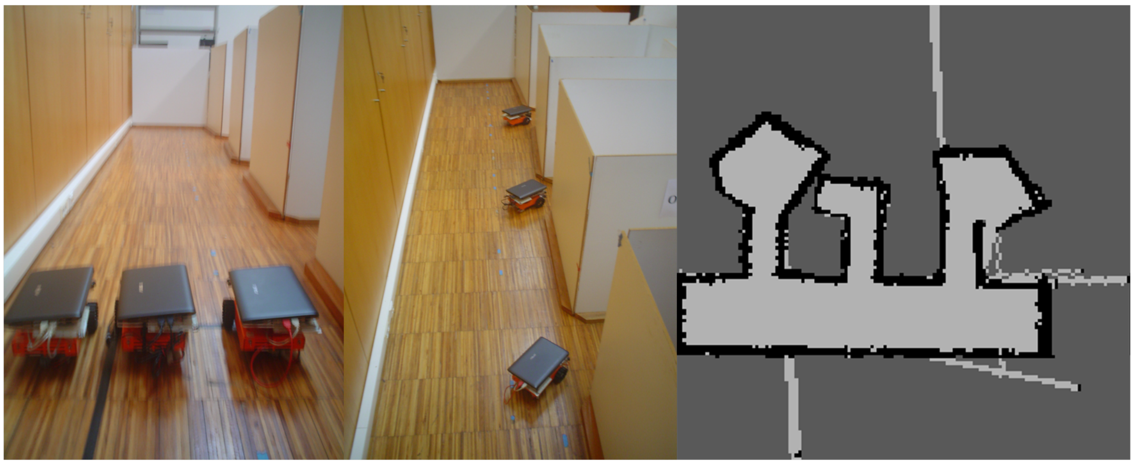

Figure 5.8: Experiment with real robots running the Rapyuta-adapted *mrgs* system.

Fig. 5.8 represents a successful multi-robot SLAM experiment where all processing was done on the Cloud, which was made of a single desktop for this case. Similarly to the map used in simulation in section 5.1, there was a common corridor open to all the robots and three side branches to be explored by only one. Despite the few holes on the environment picked up by the laser range finders and a few erroneous readings, the global map built by the system is usable and a close approximation to reality, thus providing a result equivalent to the one obtained by a traditional multi-robot system not resorting to the Cloud.

The first idea for the following experiment was to patrol the whole floor where our laboratory is located. This was not possible because the Wi-Fi access points (AP) are located inside the laboratories, which causes the Internet connection to degrade quickly. This is due to the long corridors causing the thickness of the wall between robot and AP to

[1]Demonstrative video of our work `https://goo.gl/1GVfrK`

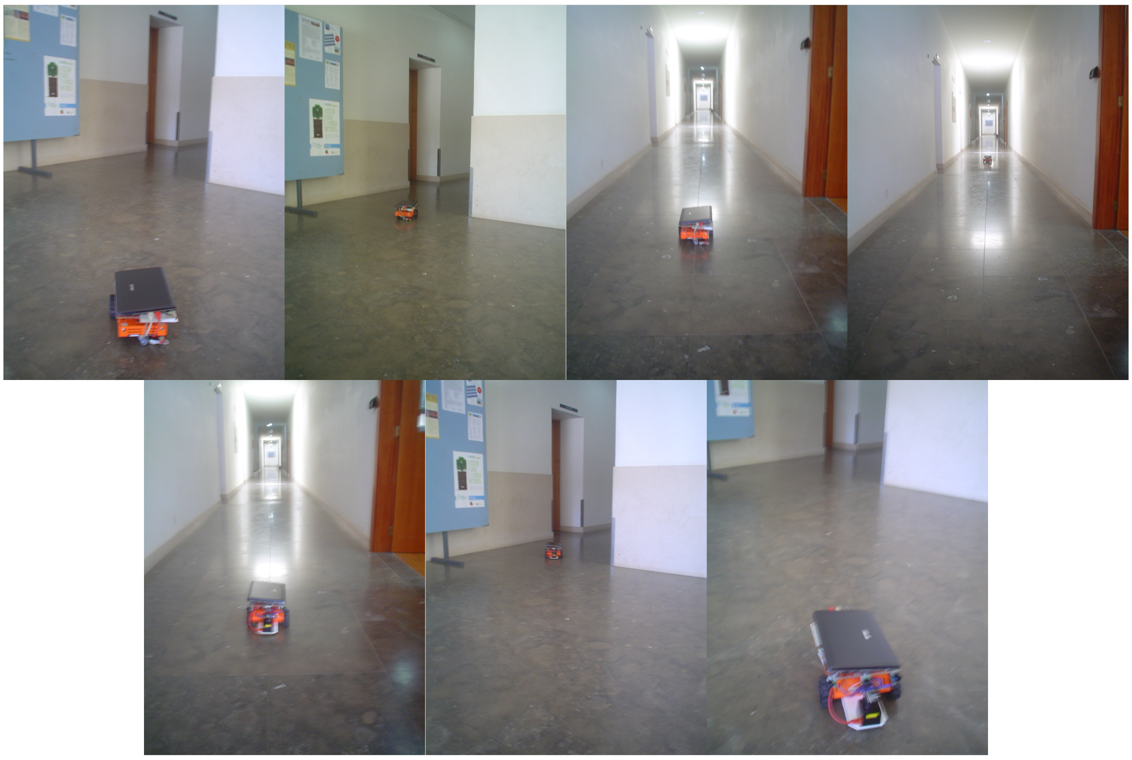
Figure 5.9: Experiment of a real robot running the Rapyuta-adapted *patrol_ isr_ demo* system.

increase greatly. Thus we restricted the patrolling area to the vertexes with good Internet connection, which are being patrolled as shown in Fig. 5.9. This solidifies the idea that a solid Internet access coverage is required for any Cloud-based system. Only one robot was used for these experiments for two main reasons: firstly there was not enough space with proper Internet access to justify adding a second robot; secondly the behaviour of the acting robot would not change by adding a team mate, unless their trajectories intersected each other.

Despite the robot patrolling just a portion of the whole building it does so repeatedly proving the correct behaviour of the Cloud-based system. This tells us that if a proper Internet access coverage was available, the robot would have been able to patrol the entire floor as tried in a first approach.

Figs. 5.10 and 5.11 show the values of CPU time registered during the experiments. Each robot had a netbook with an Intel® Atom™ Processor N2800, thus having limited computation power when compared with a common laptop computer. Similarly to simulation results, the differences are stark between the local and Cloud-based systems. The initial downwards slope evidenced by the local system's curves is created during the moments between launching the nodes and running the task. It is also noteworthy that in the patrolling scenario, one of the CPU cores [2] is completely used by the task, while the Cloud-based system requires only 14%. During these experiments, the consumed bandwidth values were 39.4 KB/s and 42.6 KB/s for the multi-robot SLAM and patrol tasks. The increased bandwidth

[2]The processor is dual core.

observed during the multi-robot SLAM experiment comparatively to simulation values can be explained by the increased volume of data provided by the laser range finder.

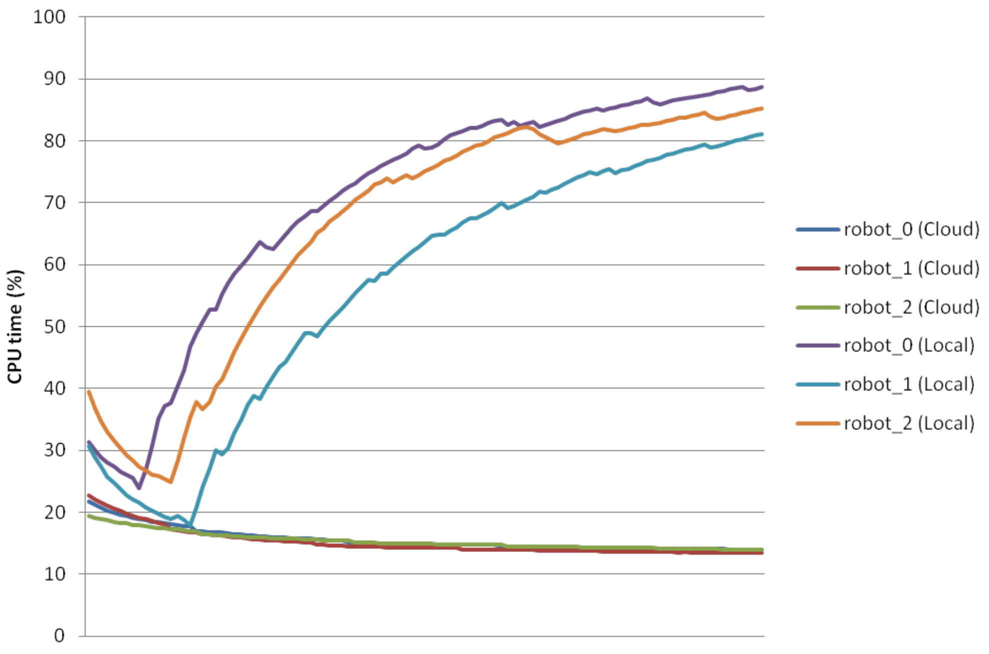

Figure 5.10: CPU time registered during the experiments of multi-robot SLAM with three real robots.

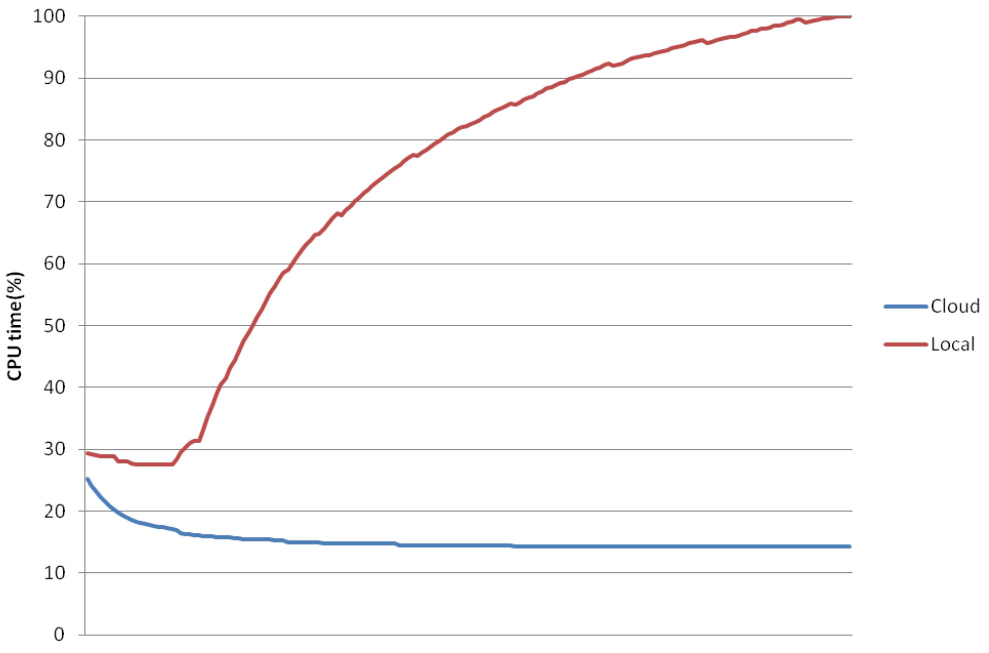

Figure 5.11: CPU time registered during the experiments of multi-robot patrol with a real robot.

5.4 Summary

In this chapter, we presented the experiment results of both multi-robot SLAM and patrolling systems, evidencing the expected tradeoff between CPU load and bandwidth. The gain in freed CPU time is considerable on a Cloud-based system and while the need for bandwidth is higher, we believe modern wireless routers can provide with more than enough to support any team of real robots with an arbitrary team size.

Experiments with real robots were also conducted in order to further prove the concept of Cloud-based multi-robot systems.

6 Conclusion

A Cloud-based solution always presented itself as a tradeoff between computer resources (namely CPU time and storage) and network bandwidth. We have successfully proven that tradeoff favourably, as high bandwidth wireless internet connection is highly available nowadays with off-the-shelf components. However, it should be noted that the tasks run on the Cloud can not degrade with time delays, *i.e.*, there should be no hard-real time deadlines.

The Cloud-based solution was also shown as a scalable one as long as the computer cluster that forms the Cloud is powerful enough and the tasks running in it enable such scaling. During the adaptation process of existing robotic tasks to the Cloud not many changes to the original systems were needed, thus this Cloud-based ecosystem can coexist with traditional systems. This becomes a key aspect when developing new robotic tasks, as debugging a program running on a foreign machine is much more troublesome. We also believe the *multimaster* nature of the presented Cloud environment is a big plus, as it allows for robots to be more independent, multi-robot system to be more modular and if the robot control is done locally, a robot is not lost in the case of network failure.

That being said, the solution is far from perfect, as only large teams or robots with little computer resources take the full benefit from it. Moreover a permanent and reliable Internet connection is mandatory, which might not be available is some scenarios.

Overall we believe to have been successful in demonstrating the utility of Cloud environments in the field of Robotics.

At the time this dissertation is being written and submitted, the author is preparing together with his supervisor a scientific paper presenting the main conclusions of the work developed in this dissertation, to be submitted on September 2016 to the 32nd ACM SIGAPP Symposium On Applied Computing (SAC 2017).

6.1 Future work

Here we present a more complete vision of how the Cloud could be used in Robotics, either on a laboratory/classroom or a corporate robotic service provider context; in both cases keeping costs low and ensuring a long product life are key aspects.

An always-on infrastructure should interface with any web browser to make the connection to the Cloud in a more intuitive way. This infrastructure would provide with existing installed robotic tasks on the Cloud and allow the users to choose which of these tasks were to be run. This way a robot connecting to this service would have access to any robotic task without the need to go through the process of installing all of them (only requiring ROS topics and services declaration to be installed), relieving the robot of some storage space. The JSON files used to communicate with the Cloud would be automatically generated and ready to use.

On a laboratory/classroom context a researcher could easily set up a team of robots to do some robotic task on the Cloud, while local computer resources would be used for the developing task. After development, the new robotic task could be added to the list of tasks provided by the Cloud infrastructure. On a corporate context, large teams of robots could be used in many different scenarios always having access to the latest version of robotic tasks software, without ever needing to update them locally or replace the non-faulty robots due to a lack of computer resources.

7 Bibliography

[1] Amazon robotics. https://www.amazonrobotics.com/. Accessed: 15-Jul-2016.

[2] App engine A powerful platform to build web and mobile apps that scale automatically. https://cloud.google.com/appengine/. Accessed: 26-Jan-2016.

[3] EC2 - Amazon Web Services. http://aws.amazon.com/ec2/. Accessed: 26-Jan-2016.

[4] Heroku: Cloud Application Platform. https://www.heroku.com/. Accessed: 26-Jan-2016.

[5] Navigation stack. http://wiki.ros.org/navigation. Accessed: 14-Jul-2016.

[6] Open-source software for reliable, scalable, distributed computing. http://hadoop.apache.org/. Accessed: 25-Jan-2016.

[7] OpenCV library. http://opencv.org. Accessed: 14-Jul-2016.

[8] Rapyuta A Cloud Robotics Platform. http://rapyuta.org/. Accessed: 27-Jan-2016.

[9] Robot Operating System. http://wiki.ros.org/. Accessed: 9-Feb-2016.

[10] Robotics in Concert. http://www.robotconcert.org. Accessed: 3-Feb-2016.

[11] ROS: powering the world's robots. http://www.ros.org/. Accessed: 3-Feb-2016.

[12] SearchCloudComputing: Cloud computing information, news and tips. http://searchcloudcomputing.techtarget.com/. Accessed: 4-Feb-2016.

[13] Stage simulator. http://wiki.ros.org/stage. Accessed: 10-Jun-2016.

[14] Wifi_comm package. http://wiki.ros.org/wifi_comm. Accessed: 14-Jul-2016.

[15] L. Agostinho, L. Olivi, G. Feliciano, F. Paolieri, D. Rodrigues, E. Cardozo, and E. Guimaraes. A cloud computing environment for supporting networked robotics applications. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pages 1110–1116. IEEE, 2011.

[16] A. Almeida. Patrulhamento multiagente em grafos com pesos. 2003. Master's thesis, Dissertação (Mestrado em Ciência da Computação), Universidade Federal de Pernambuco, Centro de Informática, Recife, 2003.

[17] A. Almeida, G. Ramalho, H. Santana, P. Tedesco, T. Menezes, V. Corruble, and Y. Chevaleyre. Recent advances on multi-agent patrolling. In *Brazilian Symposium on Artificial Intelligence*, pages 474–483. Springer, 2004.

[18] A. Araújo, D. Portugal, M. S. Couceiro, and R. P. Rocha. Integrating Arduino-based educational mobile robots in ROS. *Journal of Intelligent & Robotic Systems*, 77(2):281–298, 2015.

[19] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[20] R. Arumugam, V. R. Enti, L. Bingbing, W. Xiaojun, K. Baskaran, F. F. Kong, A. S. Kumar, K. D. Meng, and G. W. Kit. DAvinCi: A cloud computing framework for service robots. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 3084–3089. IEEE, 2010.

[21] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[22] J. Aulinas, Y. R. Petillot, J. Salvi, and X. Lladó. The SLAM problem: a survey. In *CCIA*, pages 363–371. Citeseer, 2008.

[23] M. Bonaccorsi, L. Fiorini, F. Cavallo, A. Saffiotti, and P. Dario. A Cloud Robotics Solution to Improve Social Assistive Robots for Active and Healthy Aging. *International Journal of Social Robotics*, pages 1–16, 2016.

[24] E. Cardozo, E. Guimaraes, L. Rocha, R. Souza, F. Paolieri, and F. Pinho. A platform for networked robotics. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 1000–1005. IEEE, 2010.

[25] H. J. Chang, C. S. G. Lee, Y. C. Hu, and Y.-H. Lu. Multi-robot SLAM with topological/metric maps. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1467–1472. IEEE, 2007.

[26] Y. Chevaleyre. Theoretical analysis of the multi-agent patrolling problem. In *Intelligent Agent Technology, 2004.(IAT 2004). Proceedings. IEEE/WIC/ACM International Conference on*, pages 302–308. IEEE, 2004.

[27] J. Civera, M. Ciocarlie, A. Aydemir, K. Bekris, and S. Sarma. Guest Editorial: Special Issue on Cloud Robotics and Automation. *Automation Science and Engineering, IEEE Transactions on*, 12(2):396–397, 2015.

[28] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[29] R. Doriya, P. Chakraborty, and G. C. Nandi. Robot-Cloud: A framework to assist heterogeneous low cost robots. In *Communication, Information & Computing Technology (ICCICT), 2012 International Conference on*, pages 1–5. IEEE, 2012.

[30] H. Durrant-Whyte and T. Bailey. Simultaneous Localization and Mapping: Part I. *IEEE robotics & automation magazine*, 13(2):99–110, 2006.

[31] A. A. Dyumin, L. A. Puzikov, M. M. Rovnyagin, G. A. Urvanov, and I. V. Chugunkov. Cloud computing architectures for mobile robotics. In *Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW), 2015 IEEE NW Russia*, pages 65–70. IEEE, 2015.

[32] J. Furrer, K. Kamei, C. Sharma, T. Miyashita, and N. Hagita. UNR-PF: An open-source platform for cloud networked robotic services. In *System Integration (SII), 2012 IEEE/SICE International Symposium on*, pages 945–950. IEEE, 2012.

[33] B. D. Gouveia, D. Portugal, D. C. Silva, and L. Marques. Computation sharing in distributed robotic systems: a case study on SLAM. *Automation Science and Engineering, IEEE Transactions on*, 12(2):410–422, 2015.

[34] Y. Guo, L. E. Parker, and R. Madhavan. Collaborative robots for infrastructure security applications. In *Mobile Robots: The Evolutionary Approach*, pages 185–200. Springer, 2007.

[35] G. Hu, W. P. Tay, and Y. Wen. Cloud robotics: architecture, challenges and applications. *IEEE Network*, 26(3):21–28, 2012.

[36] L. Iocchi, L. Marchetti, and D. Nardi. Multi-robot patrolling with coordinated behaviours in realistic environments. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2796–2801. IEEE, 2011.

[37] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg. A survey of research on cloud robotics and automation. *Automation Science and Engineering, IEEE Transactions on*, 12(2):398–409, 2015.

[38] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. Robocup: The robot world cup initiative. In *Proceedings of the first international conference on Autonomous agents*, pages 340–347. ACM, 1997.

[39] A. Machado, G. Ramalho, J.-D. Zucker, and A. Drogoul. Multi-agent patrolling: An empirical analysis of alternative architectures. In *International Workshop on Multi-Agent Systems and Agent-Based Simulation*, pages 155–170. Springer, 2002.

[40] A. Marjovi, S. Choobdar, and L. Marques. Robotic clusters: Multi-robot systems as computer clusters: A topological map merging demonstration. *Robotics and Autonomous Systems*, 60(9):1191–1204, 2012.

[41] G. S. Martins. A cooperative slam framework with efficient information sharing over mobile ad hoc networks. Master's thesis, University of Coimbra, 2014.

[42] G. S. Martins, D. Portugal, and R. P. Rocha. A Comparison of General-Purpose FOSS Compression Techniques for Efficient Communication in Cooperative Multi-Robot Tasks. In *ICINCO (2)*, pages 136–147, 2014.

[43] P. Mell and T. Grance. The NIST definition of cloud computing. 2011.

[44] G. Mohanarajah, D. Hunziker, R. D'Andrea, and M. Waibel. Rapyuta: A cloud robotics platform. *IEEE Transactions on Automation Science and Engineering*, 12(2):481–493, 2015.

[45] G. Mohanarajah, V. Usenko, M. Singh, R. D'Andrea, and M. Waibel. Cloud-based collaborative 3D mapping in real-time with low-cost robots. *IEEE Transactions on Automation Science and Engineering*, 12(2):423–431, 2015.

[46] M. Pfingsthorn, B. Slamet, and A. Visser. A scalable hybrid multi-robot SLAM method for highly detailed maps. In *Robot Soccer World Cup*, pages 457–464. Springer, 2007.

[47] D. Portugal and R. P. Rocha. MSP algorithm: multi-robot patrolling based on territory allocation using balanced graph partitioning. In *Proceedings of the 2010 ACM symposium on applied computing*, pages 1271–1276. ACM, 2010.

[48] D. Portugal and R. P. Rocha. A survey on multi-robot patrolling algorithms. In *Doctoral Conference on Computing, Electrical and Industrial Systems*, pages 139–146. Springer, 2011.

[49] D. Portugal and R. P. Rocha. Distributed multi-robot patrol: A scalable and fault-tolerant framework. *Robotics and Autonomous Systems*, 61(12):1572–1587, 2013.

[50] D. Portugal and R. P. Rocha. Multi-robot patrolling algorithms: examining performance and scalability. *Advanced Robotics*, 27(5):325–336, 2013.

[51] D. Portugal and R. P. Rocha. Cooperative multi-robot patrol with Bayesian learning. *Autonomous Robots*, 40(5):929–953, 2016.

[52] D. B. S. Portugal. *Effective Cooperation and Scalability in Multi-Robot Teams for Automatic Patrolling of Infrastructures.* PhD thesis, Universidade de Coimbra, 2013.

[53] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[54] L. Riazuelo, J. Civera, and J. M. M. Montiel. C2TAM: A cloud framework for cooperative tracking and mapping. *Robotics and Autonomous Systems*, 62(4):401–413, 2014.

[55] F. Sempé and A. Drogoul. Adaptive patrol for a group of robots. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, pages 2865–2869. IEEE, 2003.

[56] D. Stonier. Multimaster and Beyond. ROSCon", Stuttgart, Germany, 2013.

[57] S. Thrun. Probabilistic robotics. *Communications of the ACM*, 45(3):52–57, 2002.

[58] S. Thrun and A. Bücken. Learning Maps for Indoor Mobile Robot Navigation. Technical report, DTIC Document, 1996.

[59] L. Turnbull and B. Samanta. Cloud robotics: Formation control of a multi robot system utilizing cloud infrastructure. In *Southeastcon, 2013 Proceedings of IEEE*, pages 1–4. IEEE, 2013.

[60] M. Waibel, M. Beetz, J. Civera, R. d'Andrea, J. Elfring, D. Galvez-Lopez, K. Haussermann, R. Janssen, J.M.M. Montiel, A. Perzylo, et al. A World Wide Web for Robots. *IEEE Robotics & Automation Magazine*, 18(2):69–82, 2011.