An Adaptive Robotics Middleware for a Cloud-based bridgeOS

Rafael Afonso Rodrigues

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Manuel Antunes Veiga

Examination Committee

November 2017

Acknowledgements

First and foremost, I acknowledge and express my gratitude to my thesis supervisor and coordinator, Prof. Luís Veiga, from IST and INESC-ID. He oriented me throughout this thesis whenever I requested it, and provided valuable feedback, up until the very last moments. I thank the research lab INESC-ID for providing access to their servers and letting me thoroughly evaluate the work accomplished.

I would also like to convey my thanks to Tiago Costa, from Bridge Robotics, for granting me the opportunity of participating in this project and researching innovative technologies. His added explanations and detailed documentation of bridgeOS were extremely helpful, and provided additional cloud resources for testing the work accomplished.

Finally, I thank my parents for unconditionally supporting me throughout my entire life and enable me to attain this achievement.

17th of October, Lisbon

Rafael Afonso Rodrigues

Dedicated to my parents.

Abstract

Robotic applications and their capabilities have grown exponentially in recent years, but hardware limitations and environment restrictions still lead to unfulfilled requirements. As Cloud Computing matured, however, robotics began taking advantage of its elastic resources by offloading computation and data to the cloud, effectively creating what is now called Cloud Robotics. Although a multitude of frameworks have been proposed over the years, each with its own unique specifications and goals, none has become dominant nor able to provide a standard and generic solution linking both robots, users and the cloud.

An innovative platform, bridgeOS, attempts to take on this role by providing a new solution and framework, integrating recent Services paradigms, using a web-oriented approach and supporting a prominent software for networked robotics, the Robot Operating System (ROS). To accomplish this, we propose a cloud-based extension for the bridgeOS framework, capable of dynamic service deployments for the robots, and add support for adaptive decision making, based on available resources and performance metrics, to optimize in real time, both how those services are distributed and how well they perform.

Overall, the middleware we developed is robust, resilient, versatile and capable of scaling to hundreds of components. Our experimental results show that it is a viable solution, with benefits exceeding the overhead it generates.

Resumo

As aplicações robóticas e as capacidades dos robôs têm crescido exponencialmente nestes últimos anos. No entanto, limitações de hardware e restrições dos meios ambientes continuam a impedir alguns requisitos de serem satisfeitos. Contudo, à medida que a computação em nuvem progrediu, a robótica começou a tirar partido dos seus recursos elásticos, descarregando computação e dados para a nuvem. Isto deu início a um novo campo designado, Cloud Robotics. Múltiplas soluções têm sido propostas, cada uma com as suas particularidades e objetivos, embora nenhuma tenha sido capaz de fornecer uma solução padrão e genérica que vincule robôs, utilizadores e computação em nuvem.

Uma plataforma inovadora, a bridgeOS, tenta assumir esse papel, fornecendo uma nova solução e estrutura, que integra paradigmas de serviços recentes, abordagens orientadas para a web e um software proeminente para redes robóticas, o Robot Operating System (ROS). De modo a alcançar este objetivo, propomos uma extensão de bridgeOS, destinada para a computação em nuvem, capaz de lançamentos dinâmicos de serviços para os robôs e adicione suporte para tomada de decisões adaptativas, com base em recursos disponíveis e métricas de desempenho, para otimizar em tempo real, a forma como estes serviços são distribuídos e qual o desempenho alcançado.

No geral, o middleware desenvolvido é robusto, resistente, versátil e capaz de escalar para centenas de componentes. Os resultados experimentais mostram que é uma solução viável, com benefícios a excederem a sobrecarga gerada.

Keywords

Robotics Middleware

Cloud Robotics

Adaptive Offloading

Dynamic Decision Making

Real-Time QoS Monitoring

Docker Containers

Robotic Operating System (ROS)

Palavras Chave

Mediador Robótico

Nuvem Robótica

Descarregamento Adaptativo

Formulação Dinâmica de Decisões

Monitorização em Tempo Real da Qualidade de Serviço

Contêineres Docker

Robotic Operating System (ROS)

Index

List of Figures

List of Tables

List of Algorithms

1 Introduction

"Today, a group of 20 individuals empowered by the exponential growing technologies of AI and robotics and computers and networks and eventually nanotechnology can do what only nation states could have done before." – *Peter Diamandis (Author, founder of the X-Prize Foundation and co-founder of the Singularity University)*

Cloud Robotics is a fairly recent concept and field in robotics. It was first coined in November 2010 by James Kuffner in Kuffner (2010), who presented through it a novel approach, inspired by the DAvinCI framework (Arumugam et al. 2010), portraying the main advantages of using cloud computing together with robotics. The resulting benefits generally revolve around two topics: having shared databases with general knowledge, skills or behaviors, and migrating the heavier computing tasks to the cloud, as supported by Quintas et al. (2011) and Kehoe et al. (2015). Although, the idea of creating separate databases, often referred to as "remote brain", is not new (Inaba 1994) , and emanated from the Networked Robotics field which appeared in the 90s, derived from advances made in telerobotics and telepresence systems. Back then, researchers increasingly began interconnecting robots, forming peer-to-peer networks allowing for cooperative behaviors between them, and connecting them, mostly through Internet, to external resources, such as servers harboring databases or remote services, or as a swift way to enable remote access (Siciliano & Khatib 2016).

Since then, technology has progressed significantly, and is bound to continue at an exponential rate (Kurzweil 2005), turning Cloud Robotics into the logical next step for connected robots. From one side we have advances in hardware, that increase the computational capacities while reducing costs. On the other hand, they are coupled with new software capable of handling massively distributed and parallelized system, leading to what is known as cloud computing. This easy access to elastic and virtually unlimited resources and storage of huge amounts of information, or related to *Big Data*, opened up a new range of viable applications as it matured. The recent revolution in artificial intelligence and machine learning is a notable example of it. Big Data and massive processing power, together with new methods and algorithms to train and optimize models (LeCun et al. 2015), enabled considerable progress in image processing and classification in general. It is especially visible within the Deep Learning field, where new types of neural networks were able to break efficiency records, held by other types of models, in multiple areas including: character recognition, speech recognition, language translation, object classification and pattern recognition (Schmidhuber 2015); sometimes even surpassing human-level performance (He et al. 2015). Meanwhile, new frameworks, predominantly the Robot Operating System presented in Subsection 2.1.1, appeared with the intention of standardizing not only how robots could communicate and exchange data, but also the drivers provided by the manufacturers for their sensors and actuators.

These advances increased the capabilities of robots and their levels of mobility and autonomy. However those new possibilities also require more resources to operate in real-time, specially when dealing with computer vision and motion planning, which cannot be integrated in their totality, neither on-board nor in an external infrastructure, due to multiple constraints involving costs restrictions, computational resources, energy capacity or even bandwidth limitations. Joining robotics with cloud computing then became a clear necessity in order to profit from its economies of scale, massive infrastructure, adaptive resources and elevated availability. Even though many hurdles still have to be overcome, as demonstrated by the lack of a predominant framework or standards for Cloud Robotics (also discussed in Section 2.1), they will soon be surpassed due to its expected progress. Stimulated, in particular, by service robots, which are only this decade starting to grow into a significant market [1].

Nowadays, an extension to Cloud Robotics is already taking shape under the concept of Robot-as-a-Service, following the trends of ubiquity advocated by the Internet-of-Things, and rendered possible with the "Anything-as-a-Service" paradigm present within the cloud computing model, achievable through further integration, where robots, skills and services can be provided on demand.

1.1 Motivation and Research Proposal

This Master Thesis work emerges from a proposal defined in cooperation with an external company, Bridge Robotics, who is developing the bridgeOS robotics platform (2.1.3). As ad-

[1]International Federation of Robotics: http://www.ifr.org/service-robots/statistics.

dressed in the introduction, robotics is a Future-oriented field with a colossal potential and an exponentially growing market, specially its service robotics branch. However, it still lacks a standardized framework for cloud robotics, which comes in tandem with what this work intended to accomplish. As bridgeOS overcomes current challenges and issues unanswered by the other solutions, with the overall goal of facilitating the development of service robotics. Our work complements Bridge Robotics' offering, by providing a cloud-oriented middleware with low overhead, resource optimization and abstraction of services, that will be beneficial for developing the sector.

1.2 Shortcomings of Current Solutions

The shortcomings of current solutions are addressed by many surveys, Mainaly & Ningombam (2014), Hu et al. (2012) and Chibani et al. (2013). Although not uniformly lacking from each existing solution, it includes:

a) Non-generic or narrow scope of services;

b) Use of proprietary or non web-oriented communication protocols, and custom data formats or drivers, instead of supporting ROS;

c) Static or limited techniques for offloading computation, and no guarantees of service quality nor monitoring capabilities;

d) Lack of data resynchronization mechanisms for handling network failures;

e) Lack of security, privacy and anti-tampering mechanisms for network connections.

1.3 Proposed Solution

To address the existing shortcomings and implement the requirements sought by the research proposal, we propose a reliable middleware, distributed between the cloud and robots, that operates privately within a Virtual Private Network, protecting communications and isolating robots. The middleware is divided into a centralized Master Controller, tasked with coordinating the overall system and providing an access point to the current bridgeOS Cloud Platform. And individual Robot Controllers, present in each robot, orchestrating local deployment of Skills. Skills, are dynamic robot functionalities, that are transparently distributed and can adapt in real-time based on current events to optimize performance. Any type of containerized functionality is supported and we provide a backbone for operating ROS networks conjointly.

Furthermore, our proposed solution embraces state of the art technologies and includes mechanisms to mediate disruptions and failures, setting up procedures to adapt accordingly.

1.4 Contributions and Goals

The main goal of this work was to develop a cloud-based extension to bridgeOS, capable of dynamic deployment and management of local and remote services with low overhead, all the while being able to provide adaptive computation offloading based on available resources, general or service-dependent performance metrics, and according to the quality of service or optimization required. We were able to develop such middleware, and implement not only fault-tolerance, connection resiliency and data resynchronization mechanisms, but also, Firewall-friendly communication protocols to ease its deployment within any network and avoid possible traffic restrictions.

Overall, we contributed a powerful, versatile and complete middleware for bridgeOS, that is actually appropriate for robotic applications, as it is scalable enough to accommodate hundreds of concurrent local deployments and support large exchanges throughput.

1.5 Document Roadmap

The remaining of this thesis is organized in a number chapters. The current chapter, introduces the core topics of interest surrounding this work and details the objectives of the developed solution. In Chapter 2, we detail the current state of the art and analyze the related work. Afterwards, in Chapter 3 we present the overall architecture of the solution proposed by this Thesis. While Chapter 4, covers the main considerations, that have to be addressed when implementing the solution, and depicts the relevant development choices undertaken.

Afterwards, Chapter 5 starts by presenting the methodology and metrics by which our solution was assessed, and proceeds to exhibit our evaluation results. To conclude our work, some final remarks, accompanied by a consolidated overview of this thesis, are then provided in Chapter 6. At the beginning of each major chapter, we also outline its structure, and after describing it, summarize the contents and topics presented.

2 Related Work

In this chapter, we address the research work and relevant systems related to our work, its domain and goals. In Section 2.1 we begin by presenting the architectures of the robotics frameworks that our work will focus on, and an overview of their alternatives. Next, on Section 2.2, we detail and compare the available solutions for structuring services into small, portable and independent, blocks of software, and the existing tools for managing them. Finally, Section 2.3 portrays the current methods and strategies dealing with the dynamic relations and adaptive behaviors, between robots and the cloud, that are sought by this work.

2.1 *Architectures for Connecting Robots*

There are many frameworks for interconnecting robots, be it for creating small peer-to-peer networks or establishing links over distributed infrastructures. Some share similar concepts and architectures, however, as we present each, a progression detailing an extension of their capabilities and features will be visible. First, we start by covering the renowned architecture for networking robots, ROS, and proceed to divulge the most significant frameworks. We conclude by presenting the framework implemented by this work and juxtapose it with the alternatives.

2.1.1 Robot Operating System

ROS is a portable open-source framework that provides a structured communication layer for creating heterogeneous networks of robots and other systems interacting with them. ROS natively supports a multitude of robots and other hardware, such as sensors. Its digital ecosystem contains a considerable number of tools, libraries and drivers, some official (from ROS or hardware manufacturers) and others developed by third-parties, that are freely available and regrouped into packages, the main organizational unit of software in ROS, permitting rapid creation and deployment of modular applications. The packages themselves are aggregated into stacks for simplifying code sharing, while providing a collective functionality (e.g. perception, navigation, simulation, etc.) (Koubaa 2016). The popularity and adoption has risen rapidly since its disclosure in 2009, aided by the significant growth of public packages.

The ROS network is composed os ROS nodes, interconnected following a peer-to-peer two-tiered architecture. First a centralized layer which links all nodes to a master node, while the second layer is simply for direct communications between them. This master node functions as a naming service and is responsible for managing the *Topics system*, based on a publisher/subscriber model for exchanging data using topics. Nodes can register themselves to the master, to subscribe, publish or provide a service. The master will in turn advertise each side, so they can open channels without intermediaries. The data exchanged is encapsulated into ROS messages with predefined formats, composed of fields and constants that use a primitive data types, and transmitted via TCP or UDP sockets. Each format contains a header to differentiate and identify each message transmitted. ROS has plenty of default messages and allows full customization whenever necessary. Meanwhile all administrative exchanges are performed using XML-based remote procedure calls made through HTTP.

The modularity of ROS comes from 2 sides: the nodes, which are solely groups of processes that can co-exist in the same machine, and the packages. Packages are used to define the nodes, messages and services to launch or use, and any dependencies they might require to achieve some functionality.

Services are implemented as a mechanism to reduce the overhead caused by the multi-to-multi message broadcasting design of the publisher/subscriber relations. Each service defines a pair of messages and operates in a request/response logic, where a request message is sent first and answered with the corresponding reply message.

2.1.2 Cloud Robotics

Cloud Robotics extend the use of networked robotics by combining them with cloud computing, as a way of removing computational limitations and benefit from its elastic resource and high-availability. Although the term *Cloud Robotics* was coined in late 2010, some projects had already been unveiled by then. In 2009, the project RoboEarth (2.1.2) was awarded funding for its development through the European Union's Programme for Research and Technological Development. Soon after, in 2010, another innovative framework combining the use of cloud computing with robotics, DAvinCi (2.1.2), was disclosed.

Below, we present an evolution of the frameworks that appeared since its conceptualization, and that we deemed most relevant. Each offers novel perspectives or features useful for our work, and are afterwards compared with bridgeOS in Table 2.1.

RoboEarth

As an open-source project, RoboEarth, has the goal of creating a world-wide web for robots using a shared database containing a global world model. This cloud repository is composed by an **Engine**, that generates action plans, and 4 data containers that deal with *positioning*, *mapping*, *recipes* and *environment-dependent knowledge*.

Positioning refers not only to the current positions, but also the characteristics of known robots and objects. This information is associated with mapping data, which includes local (e.g. streets and buildings) and world maps, through a positioning fusion component to indirectly generate a global world model. Instead of creating a new standard for this kind of data, support for many popular formats is included.

RoboEarth uses recipes to describe and specify actions. A Robot can submit requests asking how to perform a task, specifying its capabilities and on-board sensors using the Semantic Robot Description Language. In turn, the RoboEarth Engine dynamically creates a recipe plan for completing the task, specific to each robot, and using environment-dependent knowledge such as dependencies between objects and time constraints (Tenorth et al. 2012). Both the recipes container and knowledge base, have a learning and reasoning component that uses a rule base and data for interpreting the context and creating plans by aggregating recipes, while learning from past results using methods based on reinforcement learning. Specifically, recipes can receive human feedback to avoid unwanted behaviors, resulted from merging recipes. With negative feedback, bad aggregations are automatically detected and adapted, or removed.

To offer a certain level of compatibility with other systems, the ontology used for sharing recipes, environmental knowledge and models, is an extension of KnowRob , a knowledge base that represents information using the Web Ontology Language (OWL) standard (Tenorth et al. 2012). OWL enables representing complex knowledge, about objects and their interactions, with semantic descriptions that can be understood by programs and shared through the web.For robots to access its services, an API using this language is exposed in the cloud, however, additional components are required in each robot. First, a RoboEarth layer used for uploading and downloading recipes, and synchronizing the local and global world models. Each robot generates its local model and enhances it with the respective subset of the world model. Secondly, a Hardware Abstraction layer that converts and interprets both hardware and RoboEarth data.

Another novel feature is that humans can teach robots new recipes locally, which are then uploaded to RoboEarth and become accessible by all other robots. However, no direct user interface is provided, and so developers have to create their own UI, to use either through a robot or build a system emulating one to interact with the API. Of course, shared skills can become an undesirable trait since there is no private or proprietary data without requiring another framework operating beside RoboEarth.

As expected, RoboEarth's semantic reasoning and services limit the scope of applications, excluding any features pertaining to visual perception, motion planning, object position inference, speech and dynamic human interactions (Riazuelo et al. 2015).

DAvinCI

Distributed Agents with Collective Intelligence proposes an approach for sharing data and offloading expensive tasks to the cloud, targeting large scale networks of heterogeneous robots. However, it focuses on a limited set of services, exclusively related to navigation, mapping and planning. Its cloud platform contains a Hadoop cluster and a DAvinCI server, that can be accessible to robots. A novel feature is the integration of ROS for communications, robots can subscribe to topics from the cloud back-end and also from other robots.

Hadoop cluster regroups Apache's Hadoop tools, which are particularly apt for large scale environments. Specifically, DAvinCI uses its two main components: Hadoop Distributed File System (HDFS), for internal storage of the information obtained from all robots, and Hadoop MapReduce framework, for executing the robotic algorithm tasks. The HDFS is a highly scalable and redundant distributed file system suited for large amounts of data, which runs on top of the cluster nodes' local file systems. While MapReduce is used for processing huge data sets efficiently by executing tasks in parallel across all nodes, vastly reducing the overall processing time. Both components are co-deployed into nodes, resulting in lower latency when accessing or returning data.

The DAvinCI server operates as the ROS Master node, acts as an intermediary for exchanges between the cluster and robots, by converting data in and out the ROS format and subscribing and publishing the necessary topics, triggers MapReduce tasks either periodically or after some request from a robot, also provides the services to external entities over the Internet by encapsulating ROS messages through HTTP.

UNR-PF

The *Ubiquitous Network Robot Platform* appeared in 2011 as the first framework to offer generic support for any type of robotic applications and tele-operation, although, it focuses on services for the elderly (Kamei et al. 2012). In 2013, it was accepted by ITU's Telecommunication Standardization Sector [1] under the recommendation **F.747.3** , as a robotic network incorporated into the Open USN Service platform , a standard architecture for ubiquitous sensor networks.

It is organized in a 3-layered architecture: *Service Applications*, regroup users, operators and applications that exploit robot data. The *UNF-PF cloud infrastructure*, composed of a **Global platform** (GPF), used for service management and multi-area integration, and **Local Platforms** (LPFs), for managing robots, with each covering different areas or networks. And thirdly, a *Robot Component* layer representing local networks of robots, running ROS or other frameworks, and devices, containing virtual agents or sensors. All platforms provide a common interface for communicating with the external layers, however only the GPF is linked with the Service Applications, providing a centralized access point to users and operators by all LPFs.

Platforms share basic functionalities, mainly for storing spatial data, detailing local areas or their global relationships, and managing services. Applications can register services, indicating the necessary conditions, at the GPF, who then disseminates to appropriate LPFs. Platforms then use a state manager to monitor the available resources and launch services whenever all conditions are fulfilled. In addition, each LPF runs a resource manager to aggregate local data, user requests and current operators to optimize and assign robots or resources according to the requested services.

To further standardize itself, UNR-PF uses multiple language standards for structuring the data used and exchanged. From the Object Management Group, it uses the Robotic Localization Service specification for describing locations and poses of robots, and the Robotic Interaction Service framework for defining and connecting, through its interfaces, service applications and robotic functional components. From the Open Geospatial Consortium, both CityGML and IndoorGML geographic markup languages for all spatial information retrieved from the robots and stored into the registries (Furrer et al. 2012).

[1]https://www.itu.int/ITU-T/

Rapyuta

Rapyuta began in 2013, derived from the RoboEarth project and built around its knowledge base as an extension to provide a generic platform to offload any type of heavy computation towards the cloud in a secure manner. It creates computational environments, similar to virtual networks, that are secure, dynamically allocated and interconnected through ROS. Connected to these environments, teams of robots can share services and information, and access a RoboEarth repository. The compatibility with ROS goes one step further, as configurations, updates and dependencies of the ROS packages are taken care of. Another novel feature, is the use of Linux Containers (2.2.2), running ROS nodes, that provide isolated processes and dedicated resources, to offload any computation from the robots. And, can be clustered into *NetworkGroups* as a way to enable direct communications and bypass Rapyuta.

To optimize performance, Rapyuta implements diverse communication protocols, leaving ROS only for containers. With internal processes, it uses UNIX sockets using Twisted framework for event-driven asynchronous exchanges when dealing with configurations, involving always the Master process, and custom ROS messages whenever exchanging data (Mohanarajah et al. 2015) . While with robots, a WebSocket (2.3.3) server is integrated, as to provide full-duplex communications and exchange data in JSON format.

The Rapyuta platform contains 4 main components: **Network**, **Users**, **LoadBalancer** and **Distributor**. The *Network*, is centralized around a *Master* node that mainly processes configuration messages, monitors the rest of the network and organizes connections with the robots, in cohesion with the *Distributor*, which distributes incoming connections from robots across the *robot endpoints*. Those robot endpoints expose ports, accessible by other endpoints, and interfaces, for robots, and handle all communications between them, converting data between JSON and ROS formats. Another function includes forwarding configuration messages to the master. Meanwhile, *environment endpoints* manage ROS nodes and their communications with other endpoints. In addition, the *LoadBalancer* manages the machines composing the dedicated cloud infrastructure and can efficiently assign containers to them. Decisions are based on representations provided by the corresponding NetworkGroup and accomplished using a *container process*, capable of launching and stopping computing environments.

Finally, Rapyuta users can register multiple robots and create NetworkGroup, defining computational environments and interfaces that their robots can utilize. As a security mechanism, personal API key are used for authenticating robots and any configuration submitted.

SCMR

Survivable Cloud Multi-Robotics appeared during the same period with the idea of incorporating resiliency and quality guarantees into cloud robotics. Although, the scope and scalability of SCMR is limited, supporting only 1 small team of robots, it does offer some interesting features. It proposes the use of a robot leader that replicates the cloud infrastructure and replaces it whenever the connection fails. And introduces dynamic offloading decisions, performed using an *Optimal Task Execution Policy*, whose only goal is to minimize the energy consumption. All robots are monitored and the metrics obtained are used by this policy to compute an estimation of the energy saved by offloading a task, based on two models: *Robotic Execution Energy Model*, to estimate the energy required to execute that task, based on the CPU workload, memory, cooling system and any other component it might use; and the Cloud Execution Energy Model, which estimates the energy dispensed by the robot for transmitting the offloaded data, receiving the response data and remaining idle during the task execution in the cloud. Anytime the policy's estimation is positive and the cloud is available, the robot offloads the computation.

Based on the offloading decisions, the SCMR platform can also invoke services, implemented as ROS nodes, to execute any task. Frequently requested data, from those services or the cloud database, is stored in a local data buffer, kept synchronized with the robot leader. So that if the connection is severed, offloaded services can partially continue. In practice, a virtual cloud is created collectively by all robots, with the leader, operating as a ROS master, using it to provide critical services: image processing, navigation and path planning.

Exchanges between the cloud and robots are handled by the Twisted framework using the WebSocket protocol. Twisted itself supports various network protocols and implements an event-driven paradigm, defining the logical flow through events and enabling callback triggers, particularly apt for the relation between cloud and robots defined in SCMR. However, when the robots are disconnected from the cloud, the network communications fall back to a gossip protocol, to permit peer-to-peer communications without predefined routing discovery mechanisms, and where packets flow through any node that is in range until reaching their destination (Hu et al. 2012).

RAPP

RAPP started as a project for delivering robotic applications for the health sector, specifically targeting the elderly, and was funded in late 2013, also by the European Union's Programme for Research and Technological Development. However, the architecture proposed and developed upon its completion enables a robotic-oriented ecosystem that can encompass any type of service and usage.

It is the first framework to propose a global repository for robotic services and applications. Developers can create and submit their own *Robotic Applications* (RApps), using multiple programming languages and even ROS nodes, which can be retrieved, through REST web services, and executed by registered robots. Static offloading is supported at application-level, parts of RApps can be deployed automatically to the **RAPP Cloud Ecosystem** as Docker containers (2.2.2), ran by *Cloud agents*.

Its cloud infrastructure also contains a separate platform, that essentially provides a common knowledge pool, a centralized database and inference methodologies, regrouped into the **RAPP Improvement Centre** (RIC). RIC offers basic services, organized around ROS nodes communicating via WebSockets, for robots (e.g. vision, speech, path planning) or centric to the elderly (e.g. cognitive exercise, hazard detection) (Kintsakis et al. 2015). But, for security reasons, they are mapped onto HTTP web services, protected with token-based authentication and the Transport Layer Security protocol. In addition, while the knowledge pool contains data about users and robots, their relations are obscured with special aliases kept by the centralized database, who blocks RApps and external entities from accessing them, to offer privacy.

Finally, from the robots side, each one runs a *Core agent*, responsible for initiating hardware interfaces, to control sensors or actuators and perform actions. On top of that, it manages RApps locally and can execute an instance of one by launching a *Dynamic agent*, however only one can operate at a time.

2.1.3 bridgeOS

BridgeOS was unveiled in 2016 by Bridge Robotics, as a platform to run generic applications for service robots. It provides robots with modular and on-demand functionalities, represented as **Skills**. And allows the deployments of applications, subscribing or processing information related to robots, that expose such data to external, web or mobile based, applications.

The bridgeOS cloud uses a **runtime platform**, responsible for managing and monitoring

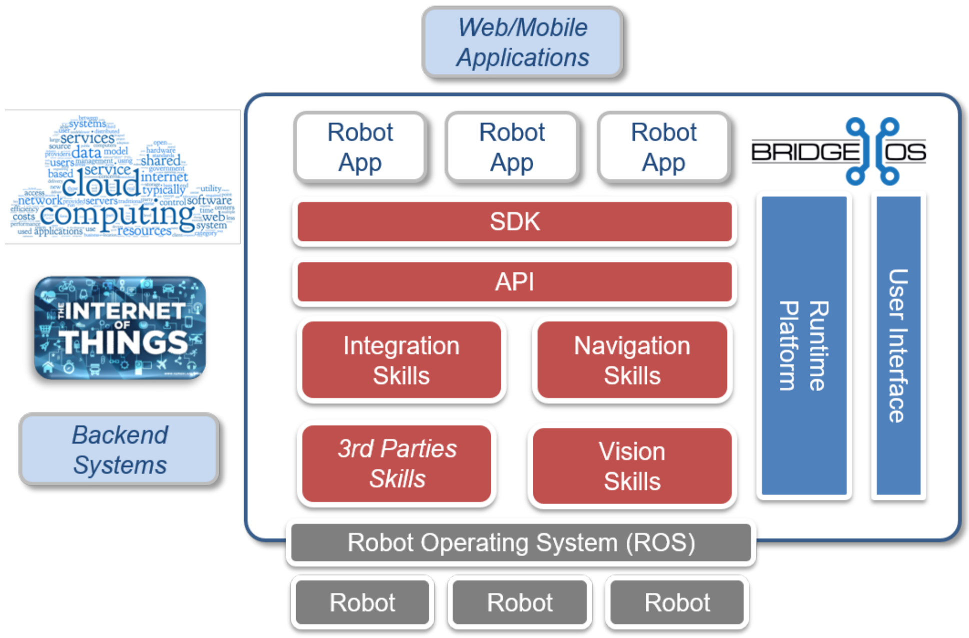


Figure 2.1: bridgeOS Architecture

applications, which in addition, provides an intuitive web *user interface*. Through this UI, end-users can visually monitor their robots and applications, configure them as needed and even upload new ones. Although, bridgeOS supplies basic Skills, users can develop their own or integrate those from third-parties, as stores for both Skills and applications become available. To facilitate development or integration with other platforms, its *development framework* supports diverse programming languages and offers libraries to ease connectivity. Furthermore, the interconnection with robots is performed through ROS for greater compatibility with existing solutions.

Comparison.

All presented frameworks are compared in Table 2.1 using an extensive number of properties, to better reflect their similarities and differences. Those properties are in turn summarized in Appendix A.2.

2.2 Componentization of Services

In this section we present the current state of the art regarding the decomposition of complex applications and services into smaller, modular and scalable, components capable of being deployed transparently. First by showing how to structure and operate those components,

Table 2.1: Comparison of Cloud Robotics Frameworks

Properties	bridgeOS	RoboEarth	DAvinCI	UNR-PF	Rapyuta	SCMR	RAPP
Scope	general	limited	limited	general	general	limited	limited
Scalability	high	medium	medium	high	high	low	high
Redundancy	yes	no	no	yes	no	yes	no
Offloading	dynamic	static	static	no	static	dynamic	static
Modular	yes	no	no	yes	yes	yes	yes
QoS monitoring	yes	no	no	no	no	yes	no
ROS compatible	yes	no	yes	yes	yes	yes	yes
Ontology	custom	KnowRob	ROS	OMG, OGC	KnowRob	ROS	KnowRob
Shared skills	yes	yes	yes	no	yes	yes	yes
On-demand	yes	no	yes	no	yes	yes	yes
Skill Templates	yes	yes	no	no	yes	no	yes
Dynamic apps	yes	no	no	no	yes	no	yes
App store	yes	no	no	no	no	no	yes
User apps	yes	no	yes	no	no	no	yes
Uploadable apps	yes	no	yes	no	no	no	yes
User API	yes	no	yes	yes	no	no	yes
Web dashboard	yes	no	yes	yes	no	no	no
Private data	yes	no	no	yes	yes	no	yes
Built-in security	yes	no	no	no	yes	no	yes

(Note: the characteristics presented here for bridgeOS depict its cloud-based functionalities, extended with the middleware developed for this thesis.)

going through the frameworks that appeared as the technology matured. Starting from distributed application systems (Section 2.2.1) until the more general container-based models (Section 2.2.2). Then, we present the available solutions for running and managing clusters of containers (Section 2.2.3), concluding with a comparison in Table 2.2 .

2.2.1 Traditional Component Frameworks.

In the next paragraphs, an evolution of the frameworks developed for building and operating distributed modular applications based on components, will be portrayed. They tackle some problems centric to mobile environments and each offers different improvements that can gradually lead to a framework capable of dealing with requirements sought by our middleware.

Rover Toolkit

The Rover Toolkit proposed in 1995, is one of the first frameworks for dynamic distributed applications dealing with mobile environments . It was developed solely for the C programming language, presenting a trade-off between performance and code complexity, when compared to other programming languages commonly used in the robotics frameworks presented earlier.

The novelty of this framework, based on a client/server architecture, surrounds two key ideas, **relocatable dynamic objects** (RDOs) and **queued remote procedure calls** (QRPC), that have a primordial role in dealing with the many challenges posed by mobile hosts.

RDOs are the central components of the application. They contain the code itself and encapsulated data, enabling a quick transfer of the its state, a *well-defined* interface, allowing remote use by other objects, and can be loaded in either server or client host. The scope of each object varies greatly, it can be a simple function or an extensive module such as a graphical user interface (Tauber 1996). QRPC is a communication system used to provide non-blocking, asynchronous remote procedure calls between RDOs, that are fault tolerant and recoverable, and even transmit the RDOs themselves. By queuing requests and responses, it ensures no data is lost whenever hosts are disconnected, and automatically restarts their exchanges upon reconnection. Furthermore, it implements split-phase communications, supporting multiple connections and transport protocols between the hosts, with even an optimization perspective where requests and responses can be transmitted through different manners, at any time it seems beneficial.

Its global structure presents a three-layered architecture, shared by both server and client, consisting on an *Application Layer*, where the executed RDOs reside and invocations originate, *System Support Layer* and *Transport Layer*. The System Support layer regroups the core functions for linking the distributed applications, each side uses an *Access Manager* to globally manage the interactions between local applications and their remote counterparts, and control both other modules. It implements the failure recovery functionality for retransmitting data and enables applications to imports or exports objects. An Object Cache that acts as a persistent storage module for safeguarding local copies of the imported objects. It offers multiple options for caching objects, depending on the content variability requested by the applications. And an Operation Log to keep track of the QRPCs and operations executed or requested or events occurred, in order to gradually flush them depending on the current bandwidth and connectivity. Lastly, the Transport layer converts messages between formats of the transport protocols used and those used by the applications. Additionally, this layer contains a Network Scheduler responsible for actually transmitting data kept by the Operation Log, in whichever order it deems most suited, based on multiple criteria involving quality of service, session's consistency and priority requirements.

Although sensitive to mobile constraints, the Rover Toolkit still has some issues with scalability since components are essentially replicated into each host and would become intractable as the objects grow in size and complexity, especially with newer functionalities unknown at his creation.

OSGi

The Open Services Gateway Initiative is a well-known generic specification for component-based modular applications using exclusively the Java platform, defined through a set of standards created by the OSGi Alliance. Although it started in 1999, new versions have been released throughout the years. Currently, the latest general-purpose specification is the OSGi Compendium Release 6, published June 2015, while the mobile-centric (Release 4) is from July 2006. With OSGi, all applications and services are created in Java and solely interact through its ecosystem, operating on top of the JVM (Java Virtual Machine). These specifications provide a myriad of guidelines, standards and protocols, corresponding to topics including service deployment, security, life-cycle, administration, and much more. However, beyond the common core set (minimum requirements for interactions), it is up to the developers to decide, in accordance with their goals, which ones they want to follow and implement.

In OSGi, components are represented by *bundles*, which are essentially JAR (Java ARchive) files. They are modular units composed of multiple services, provided to one another in producer/consumer relations through public interfaces abstracting their inner-workings, and dependencies to packages (sets of classes), that can be imported or exported by them. Meanwhile, its actual architecture is constituted of 4 hierarchical blocks residing above the JVM: *Services*, regrouping all active bundles' services, *Registry*, *Life Cycle* and *Class Loading*.

The Service Registry is a central process managing all publishers and subscribers of services, and enabling their interaction through an event-based system, who operates in the same vein as a ROS master node. The registry also shares dependencies and notifies service status and errors. The Life Cycle module controls bundles, manages them and tracks their dependencies during runtime. Despite dependencies being dealt with mostly at deployment, built-in features also allow for dynamic dependencies during runtime.This module can install, remove, resolve, initiate and stop bundles and their services. And, it provides an API with access to those operations, through which bundles can oversee each other, depending on their permissions. Finally, Class Loading handles packages distribution. Each package has a specific version and its role is to carefully transmit one accepted by the importer specifications. Although it tries to minimize the overall number of exports, it also support class spaces, allowing duplicate packages with different versions to operate simultaneously without conflict.

The framework as a whole is designed to offer dynamic services through reusable and adaptive bundles, transparently upgradeable and shared during runtime. All the while providing a versatile execution environment, resilient to failures including disabled or missing services, handled by special module events supported by bundles and applications. Furthermore, the OSGi model also focuses extensively into the security aspect, setting a structure encompassing all levels, ranging from code design, to content exposure or communications, that uses multiple secure mechanisms including access controls, service permissions and protected zones.

Of course, all of those features come at a cost and might bring unneeded complexity to the development of applications. Even more so when, ROS already implements a comparable service discovery and interaction system. In addition, being restricted to Java, can pose compatibility issues, since it is not natively supported by ROS.

R-OSGi

Remoting-OSGi is a proposal for distributing modular applications that builds on the OSGi framework. It focuses on creating an adaptive and general environment with seamless embedding of the OSGi applications and service transparency, all achieved through 4 novel features.

First, dynamic service proxies to abstract the actual location of each service from their clients. Whenever a client subscribes to a new service, a proxy is automatically generated after analyzing the code of the interface sent by the provider. Each proxy then operates locally, forwarding exchanges between both sides. This method is quite useful for providers in constrained environments because they do not retain additional data on their side, while clients can interact with services more spontaneously.

Then a distributed service registry for both registration and location, that completes the centralized system used by the OSGi framework. This new registry is composed of local instances running in individual hosts, which incorporate a service discovery protocol and are responsible for creating and running the local proxies. With this model, bundles register themselves locally, while instances register to the central registry and perform a search whenever new requests or offers appear, while activating the corresponding proxy if a match is found.

R-OSGi also provides a transparent and reliable distribution in which all new errors, either network or remote failures, are mapped to existing events, already supported and handled by the bundles. Finally, it performs Type Injection to enable self-contained and resolvable service proxies. This is accomplished through static code analysis of each bundle when it registers a new service, and all dependencies it might have, to ensure that any non-standard data type, has its definition class added into the an injection list. This list is then simply included within the client proxy at creation.

Generally, as per its features and displayed benchmarks, R-OSGi seems more advantageous than existing solutions also attempting to support distribution within the OSGi model. Additionally, it is lightweight and less intrusive, supporting existing OSGi applications without modifications. But although R-OSGi can performs than other implementations, it does present a scalability problem related to the available services, affecting primarily the host's network bandwidth and memory, caused by requiring an extra proxy per service in each location.

DACIA

Dynamic Adjustment of Component InterActions is a framework for building adaptive distributed applications within a dynamic environment that supports heterogeneous and mobile hosts. Like OSGi, it is implemented using the Java programming language, in part due to its large adoption and for its high portability and cross-platform capability, inherent to the JVM.

In DACIA, applications are modular and flexible, they can be thought of as graphs composed of individual components, whose structure can change during runtime and where different sets and configurations of components can achieve the same result. In turn, components are blocks of software with constrained functionalities corresponding to a limited logical subset of an application. Specifically, they are defined as a **Processing and ROuting Component** (PROC), which receives input data, from one or more sources, and applies a distinct transformation to it. The location of each is itself transparent, they can run locally, close to the end-user, or remotely, and can migrate in real-time depending on the current configuration. Migrations are possible due to how PROCs are structured: they are relocatable data objects who, through Java serialization, can be transmitted to a new instance, while maintaining their operating data, connections state and unprocessed messages. PROCs also implement a simple interface, enabling basic connection and communication functionalities, including asynchronous procedure calls for message transmission.

DACIA also uses two additional modules, indispensable for achieving the adaptive functionality of any application: **Engine** and **Monitor**. The Engine has a general administrative role consisting in managing PROCs and handling all communications. It keeps track of the local PROCs and can migrate them based on the current configurations. It manages connections to other Engines and between PROCs, which are connected in pairs and whose connection order defines the actual data flow of the application. Every host participating in the application runs its own Engine module, this allows them to act as a communications intermediary between hosts and use virtual addresses to map their local components and persist connectivity even during relocations. The second module, the Monitor, supervises the application performance using an event-based monitoring service and generates reconfiguration decisions, based on the implemented reconfiguration policies, specific to both the application and host.

Overall, DACIA tries to operate with a low overhead and a small code footprint. It incorporates an interesting approach for providing dynamic deployments and decomposing applications such that it is possible to optimize their operating locations during runtime. However, the dependence on Java poses the same problems as before.

2.2.2 Container Frameworks

Containers are small blocks of software concatenated to provide a service or application. Operated through a lightweight virtualization technology, they run directly on top of the host OS and have their own isolated processes and resources. This type of virtualization provides portability between a vast number of heterogeneous operating systems and machines, and is language-neutral. Whereas component frameworks are mostly dependent to specific environments.

It also has multiple advantages in comparison to direct virtualization and virtual machines (VMs). Since there is no guest operating system on top of the hypervisor, the boot time is much faster and containers can make direct calls to instructions of the host's CPU with performances near those of native applications, much better than VMs, as proven extensively in multiple performance studies (Kozhirbayev & Sinnott (2016) Seo et al. (2014)). Two frameworks will be presented, Linux Containers and Docker, the popular and powerful newcomer. However, we begin by introducing the concept of microservices, to convey the kind of software containers are inclined to run.

Microservices

Microservices are small services that can operate autonomously and integrate with others to fulfill some requirement, and are in fact a specialized approach to the Service-Oriented Architecture. Whereas with SOA the idea is to functionally decompose complex applications into smaller building blocks, separating user applications, business processes, services and data systems, and to promote reusability. Microservices go one step further in the granularity logic, to attain singular, modular, units which enable meticulous monitoring, allowing to pinpoint bottlenecks and optimize scalability. Of course, a trade-off between such benefits and the associated overhead, be it from memory or bandwidth usage, and increased complexity, of development and coupling, has to be analyzed properly, to ensure favorable and efficient distributions (Namiot & Sneps-Sneppe 2014).

Linux Containers

LXC is a framework specialized in building and operating lightweight linux-based containers within the same host and using a single Linux kernel. The project started in late 2008, however the first stable version was only released recently in 2014. Currently, it can be deployed into any Linux-based host, by patching the kernel and importing the LXC tools. These tools offer all of the required operations for controlling and managing containers, and can be accessed through an API, although written in C, several libraries supporting other programming languages are provided.

In LXC, containers run in virtual environments, each isolated from the system and with specific resources constraints. They are actually created by aggregating two packages already available within the Linux kernel, *cgroups* , for isolating the resource usage of collections of processes referred as the control groups, and *namespaces*, for isolating them using specific namespaces. Overall, *cgroups* enables LXC to run multiple applications within the same container (Kozhirbayev & Sinnott 2016) and includes a set of features that are essential for LXC, such as resource limitation, monitoring and prioritization, and group management. By giving certain control groups larger shares of resources, they become prioritized with relation to the other groups and operate therefore with consistent performance, specially beneficial for crucial containers. With *namespaces*, resources can be isolated and virtualized, thus providing secure environments. Although, the term resources is used in a broader sense, since it includes processes and users IDs, filesystems, hostnames, and more. Therefore, and beyond the straightforward limitations, common to all container frameworks, such as CPU, memory and disk usage (size and performance), LXC can also put restrictions over the network usage (Kozhirbayev & Sinnott 2016).

LXC provides mechanisms for cloning and snapshotting containers, particularly useful during prototyping, and for rapid and scalable deployments. Finally, persistent storage is available for each container, located within the host's filesystem, and to allow customization, different storage back-ends are supported.

Docker

Docker is a recent container framework [2] , released in 2013. It was initially based on LXC, but then developed their own implementation, libcontainer, for using *cgroups* and *namespaces* directly. It contains some differences with relation to LXC, one of them being higher portability, with support for more platforms, including Windows, whereas LXC operates under UNIX systems. Docker is also restricted in terms of constraints and offers no persistent storage within containers, although it is possible to link storage containers or include mount points at startup. The idea is to promote a "single service/application per container" model, synergistic with the microservices paradigm. The argument behind this perspective is to further decompose applications into elementary services, all of which can then communicate or be linked through configurations and dependencies passed on at launch (Jaramillo et al.). This provides additional benefits, services can then be updated individually without necessarily disrupting whole applications and scalability becomes more precise, only increasing the bottlenecked parts.

Here, containers are created from templates known as Docker images and operated by the Docker Engine. Images are stored locally inside a Docker registry and can be easily shared using the global public registry, Docker Hub, or even private registries. Due to its popularity, many software providers have developed their own images for public dissemination, while other open-source software was included by Docker's developers directly. Furthermore, many public cloud platforms already support Docker containers, meaning developers can enjoy the cloud elastic resources without additional configurations. Docker Engine provides a series of tools, besides managing containers and images, and includes an API so that remote or local clients can operate it. Nowadays, it also contains a built-in feature for orchestrating clusters of containers, called Docker Swarm, previously used separately as a middleware on top of the

[2]https://docs.docker.com/

Engine. Docker Swarm is detailed, along with other container managers, in the next Subsection.

2.2.3 Container Managers

Container Managers facilitate the orchestration and scheduling of highly scalable environments with large clusters of containers, through sets of management and supervision tools. While the managers are not exclusive for Docker-based containers, they do provide a powerful extension to it, even if only to display what kind of ecosystem and usages are actually possible by using Docker and how to attain them, since they are all open-source projects. The managers presented in this Subsection were chosen for their individual relevance and to portray, from different perspectives, what it is possible to accomplish besides the basic clustering features. These choices are supported by studies such as Kratzke (2014), although Nomad (2.2.3), a newer framework, was included to provide an indication, with relation to the remaining solutions, of the direction the evolution of this technology has taken.However, Docker Swarm, still serves as the basis since it is the native Docker manager. A comparison overview is then exhibited at the end, surrounding Table 2.2.

Docker Swarm

Docker Swarm [3] is the official manager and comes included inside the Docker Engine since version 1.12. Each Swarm consists of a cluster of nodes, running the *Engine* in *swarm mode*. Two node roles are defined, *manager* and *worker*, which can coexist within the same node. Managers administrate the swarm, schedule services among workers, monitor tasks and provide external access to the swarm API. Usually, a small number of nodes are set as *manager* to provide the cluster with built-in fault-tolerance features, additionally one is randomly elected as leader and focuses solely on orchestrating tasks. Worker nodes only function is to execute containers as per the requests received.

The work performed by the swarm is classified by services, that define a Docker image and the set of tasks required. In turn, a task represents a container, running an instance of that image, and a list of commands needed to be executed by said container. Services can be updated incrementally, with controllable delays between different nodes, while leaving the possibility of roll-backing to a previous version at any time. Two service models are provided, *global services*, in which all workers run a service's task, and *replicated services*, where the manager

[3]https://docs.docker.com/swarm/overview/

redistributes tasks amongst workers depending on the scale desired. Scales can be set dynamically and managers automatically set the appropriate replication by monitoring the current state of workers and their tasks, readjusting whenever necessary, even when dealing with full host failures.

Docker also provides mechanisms for load balancing, be it for internal redistribution of service requests or external access to exposed services. The former is accomplished by using a DNS component, operated by a DNS server embedded within the swarm, for mapping services and containers running them, although primarily used for serviced discovery. The latter is done by automatically assigning a *PublishedPort* to each exposed service, so that it is accessible on any node via this port. Finally, all communications performed inside a swarm are secure by default using the TLS (Transport Layer Security) protocol and mutual authentication between its nodes.

Apache Mesos

Apache Mesos [4] is an open-source cluster manager that appeared before Docker, in 2010. It is aimed at resource management and application deployment in data centers, and is capable of supporting systems regrouping tens of thousands of nodes (Hindman et al. 2011).The Mesos architecture is composed of a Mesos master, interconnecting frameworks with Mesos agents, its workers. The master is the central point, it manages agents, handles communications between all nodes and mediates allocations of resources and tasks. Since it can represent a single point of failure, to provide fault-tolerance, the master is replicated into other nodes that remain in stand-by mode as backups in case of failure, operated by a ZooKeeper module. ZooKeeper (Hunt et al. 2010) is a centralized service, specific for preserving configuration information and providing distributed synchronization, that handles the election of the active master node.

Frameworks are orchestration applications, consisting of a scheduler and executors modules, that can be deployed concurrently and share the resources available. The scheduler component is used for defining jobs and managing tasks while the executors are responsible for launching them using specific environments, depending on the supplied configurations. The built-in executors support binaries, Docker containers and Mesos containers. Although, it is possible to build custom executors, and more generally frameworks and schedulers, using multi-language APIs, accessible via HTTP endpoints present within the cluster.

[4]https://mesos.apache.org/documentation/

Orchestration of tasks follow a bottom-up approach organized around resource offers. Newly registered agents publish their resources (CPUs, GPUs, memory, disks or ports) to the master. In turn, the master broadcasts each resource offer sequentially to all frameworks, along with additional attributes regarding physical information, indicating where the server is located, and system information, such as its operating system and engines version (e.g. JVM, Docker, etc.). Based on their current needs, frameworks can reject these offers, or accept and return a list of tasks with the required individual resources. The tasks' details are then passed on to the corresponding agent, so that requested resources are allocated to the framework's executor, allowing it to launch them in an isolated environment (Hindman et al. 2011).

Mesos also provides a thorough monitoring component, with an extensive set of **observability metrics** supervising the overall system, its nodes, resources, tasks, frameworks, communications, and more. As well as broad logging support (of the same aspects), with customizable modules. All of which are accessible through the API or a web-based user interface.

Currently, there are many frameworks available that work on top of Mesos to complement or augment its functionalities, surrounding areas such as long-running services, batch scheduling, or even Big Data processing and data storage. The most common is Marathon (Estrada & Ruiz 2016), used for orchestrating long-running services, it offers additional availability guarantees and control, through personalized constraints.

Google Kubernetes

Google Kubernetes [5]is a container scheduler specific for Docker, that offers its own original perspective for managing and scaling modular applications. As expected, a master node is used for overseeing the network. It runs a **Scheduler and Controller manager** for scheduling containers and guaranteeing their long term survivability. The master also integrates **Etcd**, a distributed key-value database, for replicating configurations and states, enabling cluster monitoring and service discovery. All interactions with nodes are performed through an API server it contains, that is also responsible for sharing the Docker images used.

On the other hand, worker nodes run a special process controlled by the master, the **Kubelet**, responsible for the local management and monitoring of containers. Additionally, kubelets can pull specifications and images from the API server and perform garbage collection throughout containers' life-cycle to increase efficiency.

[5]https://kubernetes.io/docs/

A novel feature is the organization of containers. In Kubernetes, containers that share resources or fulfill a common functionality are bundled together into a **pod**, isolating them even further. While they can communicate internally, a shared IP is used externally. Furthermore, Kubernetes defines **services** to manage the access of pods and any web services offered through them. They also perform load-balancing and replication, abstracting the location of the actual containers being used.

Lastly, the tools provided for monitoring and controlling resources can be applied at different levels, from single containers to machines. And namespaces allow different specifications of restrictions, however, they remain equal within the same pod.

Nomad

Nomad [6] is a highly functional, general-purpose microservices scheduler, oriented for large scale and geographically distributed infrastructures. Its global infrastructure can regroup multiple regions, each containing jobs, a cluster of Nomad servers and dedicated datacenters with worker nodes. Its high scalability promise of supporting thousands of nodes, relies partly on *Consul*, built by the same company, HashiCorp, a tool capable of handling service discovery and configuration for millions of machines.

By being geographically aware, users can submit jobs region-wise through a local CLI or an API, which also offers monitoring tools. The jobs proposed with Nomad are more complex, they can bind to specific datacenters, follow custom instructions for rolling updates and execute groups of tasks. Meaning, they can launch whole applications at once, as each task group defines a set of tasks to launch and replicate as requested. In turn, task definitions can specify requirements in terms of drivers, images and resources. Nomad is not constrained to Docker, it can also run VMs, jars and pre-installed applications. In addition, multi-level constraints are supported and can regard hardware, software and even business (e.g. compliances) characteristics.

Nomad servers are used for region management, with similar functions and election system as with the previous container managers. However, it uses an optimistically concurrent approach, where all servers participate in making parallel scheduling decisions, while the leader supervises them. Furthermore, each region is managed as an independent cluster, but Nomad servers can communicate with those from different regions to federate and create a single

[6]https://www.nomadproject.io/docs/

system.

Another feature of Nomad, is its dynamic scheduling process, triggered by events emanating from users and nodes. It is composed of an evaluation phase, where the region leader evaluates the vacant resources, constraints and availability requirements, and generates a schedule plan, ending with an allocation phase. The main objective being maximizing resource utilization by optimizing tasks allocation within clients.

Comparison

Table 2.2 resumes the properties of each manager, with short descriptions provided in Appendix A.1. Given the recent nature of container managing systems, there are no performance benchmarks available. Meaning comparisons are at most qualitative, and since they all share similar concepts and characteristics, case in point being the master-worker relation and election system. Therefore, each has its own advantages and the better positioned changes with relation to the infrastructure and objectives ambitioned, and their evolution, an argument pointed out and reinforced by Kratzke (2014).

Another consideration to have, refers to the cloud infrastructure used. As it is becoming commonplace for Cloud Providers (i.e. AWS, Microsoft Azure, Google Cloud Platform, IBM Bluemix) to offer container services, already with their own container management and orchestration solutions, some of which are adapted versions of the manager here compared. Consequently, this choice might be dependent on whether a private ou public cloud infrastructure is selected.

2.3 Operational Decomposition: Local vs Cloud Processing

This last section will address the current mechanisms for offloading processing and other requests for computational resources over to the cloud, means for monitoring and enabling such exchanges. And finally, how to ensure that such communication channels can properly operate in strict networks, while at the same time, reduce deployment barriers.

2.3.1 Cloud Offloading Techniques.

Offloading computation or data to the cloud is used mostly for optimizing resource usage and the quality of the application or service provided. The decisions and policies used, vary de-

Table 2.2: Comparison of Container Managers.

Properties	Swarm	Mesos	Kubernetes	Nomad
Scalability	medium	high	medium	high
Overhead	low	variable	medium	medium
Monitoring	yes	yes	yes	yes
Redundancy	yes	yes	yes	yes
Service Discovery	yes	yes	yes	yes
Service Replication	yes	no	yes	yes
Image Sharing	yes	yes	yes	yes
General Constraints	no	no	no	yes
Node constraints	no	yes	no	yes
Non-standard restrictions	no	no	custom	yes
Multi-level restrictions	no	no	yes	yes
Docker only	yes	no	yes	no
Load-balancing	yes	no	no	yes
Single system	no	no	no	yes
Multi-cluster	no	yes	yes	yes
Secure communications	yes	yes	yes	no

(Note: Mesos characteristics relate solely to a clean distribution.)

pending on the desired requirements, characteristics and security, and are based on a myriad of metrics such as performance, latency, energy usage, or even cost. However, any resulting utility depends always on a trade-off between the overheads, caused by the heuristics and algorithms employed, and the optimization gains. The most common criteria used for mobile environments, including robots, involve: minimizing execution time, which includes the cloud computation time and time taken to transmit the data, and minimizing energy consumption, from computing locally and transmitting data (Fernando et al. 2013).

The survey Khan (2015), does a great job in analyzing the current state of the art pertaining to mobile cloud offloading. In it, the techniques are categorized into 2 approaches, *static* and *dynamic*, that can be applied at multiple scopes, be it into components, whole applications, threads or even single methods, that are computationally expensive. Static offloading utilizes performance prediction models or offline profiling to estimate the performance of the different services and tasks. Based on the estimations obtained, complex computations are separated and migrated unto the servers. While predictive models usually generate and minimize cost graphs, offline profiling has a more manual directive, that consists in creating an initial profile, testing it and adapting until agreeable. Each profile defines the different properties that should be respected to meet the quality of service, performance or cost targets. Meanwhile, with dynamic offloading, a static analysis of the code is first performed, to obtain an initial configuration, and then online profiling is applied during execution, to continuously optimize the profiles and deal with real-time constraints. Many complex and high-performance algorithms, such as fuzzy models, genetic programming and Monte Carlo, have been used for this task as presented by the survey.

Some interesting implementations, containing features useful for our work, have been developed so far. This is the case of Park et al. (2014), which proposes a **Platform-independent Mobile Offloading System** that uses annotations, easily accessible as web pages, to define the offloading properties between clients and servers. With those annotations, a local proxy module can determine when and how to offload the execution of resource intensive functions of a particular application, based of the current quality of service and battery usage. Another example is Zhang et al. (2010), where an **Elastic Application Model** is presented. Similarly to paradigms addressed before, it proposes to partition applications into smaller, platform independent, components called *weblets*, to be executed transparently location-wise. With the objective of augmenting the mobile capabilities, through dynamic deployments. Decisions are determined by a cost model, optimized through naive Bayesian learning, based on common resource availability metrics and user preferences, such as power consumption, monetary cost, performance, security and privacy.

Offloading Policies.

When it comes to offloading, different robots and different functionalities can have distinct requirements and thus, divergent goals. Usually, these goals will be addressed by a set of guidelines, designed to dictate how to attain them. Those sets are often characterized as policies. For offloading in mobile environments, policies can target power consumption, network usage, general resource usage, monetary cost, or more importantly, performance and quality of service. Policies are versatile and can be employed individually, or in conjunction to reach higher-level optimizations.

Obviously, these are simply abstractions that require, beforehand, to be formulated by cost functions mapping metrics into values. Enabling state comparison and optimization based on monitored data and real-time statistics collected from robots, services or processes. The policies then signal how the computed values should be used and the direction of the optimization (i.e. minimization versus maximization). However, higher-level abstractions may need complex algorithms. In that regard, we can feed those values directly into the learning models, which also take into account the consequences of committing to certain policies, and by extension their potential rewards.

Neural Networks.

Neural Networks are an information processing paradigm, inspired by the biological neural networks, able to approximate highly complex functions, often unknown, through the hyper-connection of neurons, each performing a simplistic computation. Although, the first neural network models have been presented during the 1940s, their application as powerful learning models, whether they are used for classification or regression, have been vastly proven in the last decade. Where, thanks to recent advances in the fields of artificial intelligence and reinforcement learning, they have shattered innumerable efficiency and performance records, surpassing virtually every other type of model, and sometimes, even humans. Those advances, championed by the architectural complexity paradigm that is deep learning, were enabled by modern breakthroughs during the 1980s-1990s period, with *backpropagation* being the most notable, and exponential gains in computational processing capacities.

Neural networks are composed of multiple hierarchical layers of neurons, interlinked through weighted connections and feeding values along the chain. The first layer, referred as input layer, serves the sole purpose transmitting the external data it receives. While the last layer, output layer, combines the internal inputs of the previous layer to compute and emit the model's decision or final value, depending on the application sought. Although, depending on the architecture implemented, this structure can vary. For instance, the recursivity introduced by more advanced architectures (e.g. Recurrent Neural Networks), can blur the separations between layers in high-dimensional setups. Also, while neural networks are normally employed for processing data, sophisticated architectures, such as Long Short-Term Memory, provide storage capabilities, albeit limited.

The models learn by adjusting the connection weights with a gradient descent algorithm. For instance, the backpropagation method propagates backwards the errors, computed for each neuron by a loss function, and in turn, applies a stochastic gradient descent to optimize those weights. A particularity, is that networks can be optimized at any moment. It is merely a question of applying the error propagation step after generating an output.

Compared to other learning models, neural networks present some advantages besides online learning. Since each neuron represents a simple function performing some basic operation, the networks can be massively parallelized and are therefore suitable for large scale applications. Another interesting characteristic, is that they can be repurposed and combined to perform new tasks. For example, a neural network classifying objects can be channeled into a neural network trained to describe scenes.

The downsides of using neural networks relate to their internal complexity, which reduces interpretability, and black box aspect, in the sense that their network structures give no insight on what exactly is being learnt (i.e. the functions approximated) by the model. Another quite notorious issue, is the need for large quantities of data in order to optimize the networks and attain sufficiently good approximations to become efficient for a given application.

2.3.2 Resource Monitoring Strategies.

The offloading techniques presented give a vast overview of the metrics that can, and should, be monitored. Although, they depend largely on the desired effect. However, the strategies for obtaining and transmitting those metrics are quite simpler, and can essentially be grouped into two options, with a third, the hybrid version, combining both.

The first approach is based on periodic data transmissions, a rudimentary example being a connection keep alive packet. The advantage lies in its straightforwardness, though, it can quickly become inefficient and cause significant waste and overhead. Since, regardless of state changes, transmissions always happen. A variant includes using dynamic intervals, regulated by the current activity, as to avoid using crucial resources during peak usage. However, repetitive data remains a problem.

The second strategy consists in using an event-driven approach, where events trigger the transmission of metrics. Therefore, exchanges only occur when something actually happened, minimizing resource usage and overhead. It presents a significant inconvenient however, failures and errors can lead to lack of events, resulting in missing information. For this reason, a mix of both is recommended, to take advantage of the best in each case: event-driven system with conditional periodic exchanges (Ji et al. 2016).

2.3.3 Protecting Networks and Circumventing Restrictions.

Most distributed environments, interconnected through many networks, can be faced with security and privacy issues, or network traffic restrictions, due to firewalls or regional blocks. Since it is critical, for the robots and services they interact with, to communicate freely, mechanisms resolving those issues should be implemented. In this sense, we present below two separate approaches, that can be combined for better results.

Virtual Private Networks

VPNs are classified as extensions of private networks, that permit spanning across public networks, mainly Internet, securely. VPNs use encryption and tunneling to transmit data confidentially through otherwise insecure networks. There are multiple protocols available and can operate in either network or data-link layers. Regardless, the scheme consists in exchanging packets, while encapsulating the encrypted content from the layers above and include it along a new packet header, which identifies the recipient. They also use diverse cryptography systems, for both encrypting data and authenticating peers, primarily secret and public keys schemes.

Besides confidentiality and privacy, VPNs have another benefit: since they encapsulate the transport layer, the protocols used are unknown to external entities. So, even protocols and ports blocked by firewalls can be used, if tunneled through a VPN. This characteristic is particularly interesting for the interoperability of services required by Cloud Robotics, which can use services bound to a myriad of ports.

An important open-source software for implementing and managing VPNs is OpenVPN (Feilner 2006). It is deemed capable enough to operate within firewall-enabled networks and traverse Network Address Translation mechanisms. It also integrates tools for creating Public Key Infrastructures, and features newer and robust security protocols, primarily the Transport Layer Security (TLS).

WebSocket Protocol

WS is a TCP-based protocol, originally directed at browser interactions between client and server, that offers full-duplex connections, permitting more exchanges than the request/response model of HTTP. It shares some similarities to HTTP, including connection establishment and ports, that enable it to operate within any network that already supports HTTP, and TLS tunneling, for securing communications. Albeit, the extended bi-directionality renders it more suited for real-time communications, such as those needed for long-running web services and robotic environments. A choice cemented by its small overhead and higher performance, as demonstrated in Pimentel & Nickerson (2012) and Liu & Sun (2012).

An implementation confirming its utility is the **Rosbridge** package (Lee 2012), which provides a library for using WS as the transport protocol, tools for creating a WS server and interacting with a ROS network. Additionally, a main objective for the next version of ROS, **ROS 2.0**, is to add native support of the WS protocol.

Summary

In this Chapter, we presented the relevant related work found in the literature for the themes addressed in the thesis. We analyzed the current technological state of the art across varied fields, focusing mainly on cloud robotics, container systems and cloud offloading. Presenting the core theoretical concepts, alongside possible development perspectives, and completing those analyzes with comparative tables and short descriptions on the individual advantages found for each main platform, framework or paradigm.

3 Architecture

This chapter explains the design choices undertaken during the development of the proposed middleware. First of all, in Section 3.1, we exhibit a general overview of the architecture for the entire bridgeOS ecosystem. And, in the following sections of this chapter, continue with more in-depth regards to reflect the architecture and implemented functionalities of our middleware in terms of its software modules. Concluding with the exposition of the network architecture and communications protocols instated into our modules.

3.1 Architecture Overview

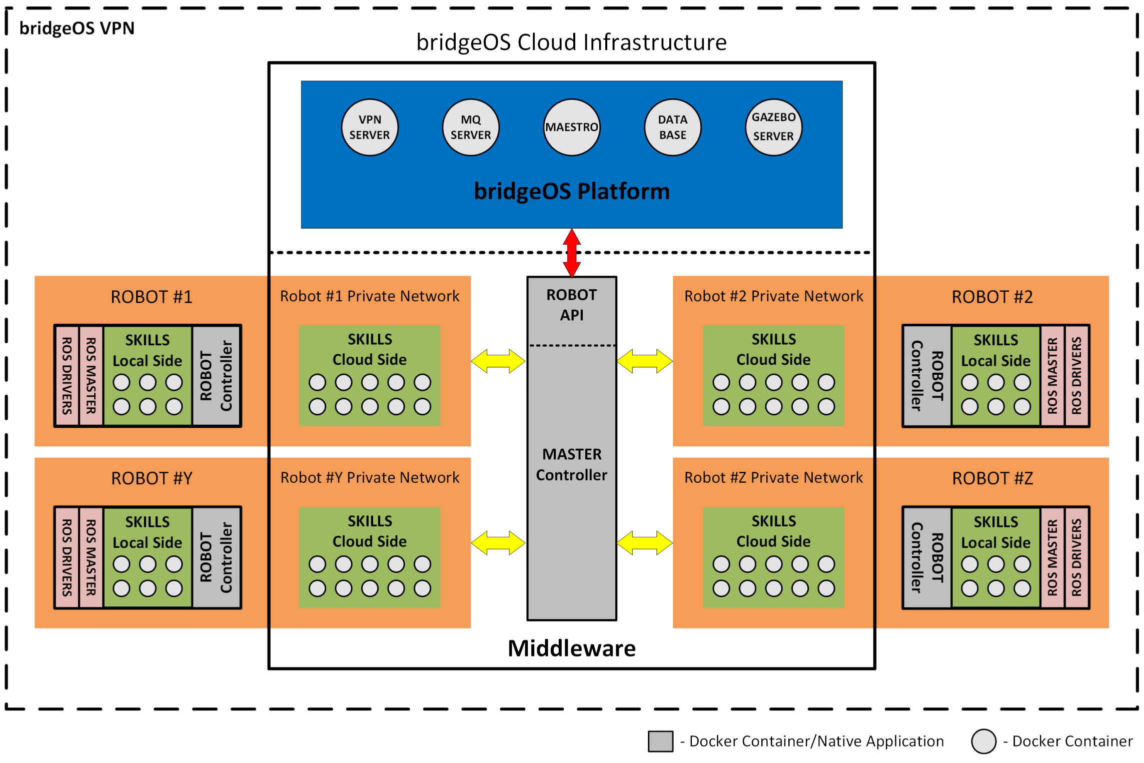


Figure 3.1: Overview of the extended bridgeOS architecture

The system-wide architecture for the cloud-based bridgeOS implementation, augmented by our proposed middleware, is depicted in Figure 3.1. With it, the extensions undertaken by this work for the bridgeOS platform are apparent. To be noted however, the separation between the cloud-part of the middleware is merely for illustration purposes, as they are fully integrated and, from an outside perspective, represent the same cloud infrastructure.

By reason of security considerations nowadays necessary, the whole system is enclosed by a bridgeOS Virtual Private Network, acting as a general protection mechanism providing security features sought for all established communication channels and robots using bridgeOS. While a VPN creates a barrier blocking external parties, our security precautions go one step further, adding another layer of protection inside the system, to fend off possible vectors of attack coming from within, due to compromised or rogue internal elements. Each robot has its own private network, isolating restricted data or skills, regardless of their location, and only exposing the pertinent parts through the Controllers API to the bridgeOS platform, and thus other robots and users. Further explanations are provided later in this chapter, in Section 3.5.

That sections also details our development choices regarding communications between modules and presents our event-based protocol (Table 3.3) for dealing with monitoring and middleware management, which was specifically developed for communications through WebSockets. As for other design choices, each middleware module, contains a web interface implementing said protocol, to permit interactions using WebSockets. All modules, with the exception of *Skill Routers* (Section 3.4.2) were developed using Node.JS (Tilkov & Vinoski 2010), a high-performance and scalable JavaScript environment based on asynchronous events, with clear advantages over Python with regards to this web-oriented approach (Lei et al. 2014). Since Skill Routers have to interact with ROS networks, their module was built in Python, taking advantage of ROS native python library, *rospy*. Rospy is used in most of ROS core components due to its ease of integration and performance.

The proposed middleware supports groups of robots. They can access the bridgeOS cloud infrastructure by means of their local Robot Controller, which establishes the VPN tunnels and links them with the Master Controller. Figure 3.2 offers a more in-depth look about how robots interact with and use our middleware, and subsequently, the bridgeOS cloud platform. Logically, some robots might already possess core functionalities and will turn to bridgeOS as an option to enhance them. It is then plausible, that some robots will have native ROS drivers not instantiated by bridgeOS, such as those controlling local actuators or sensors, and likely a ROS master to manage them. Therefore, we have to anticipate the occurrence of a local ROS master, while also planning for the opposite possibility. The middleware is permits both scenarios, it can adapt by managing network routes and enable packet forwarding, and if no ROS master is detected locally, a cloud container will provide a dedicated ROS master instance to such robot.

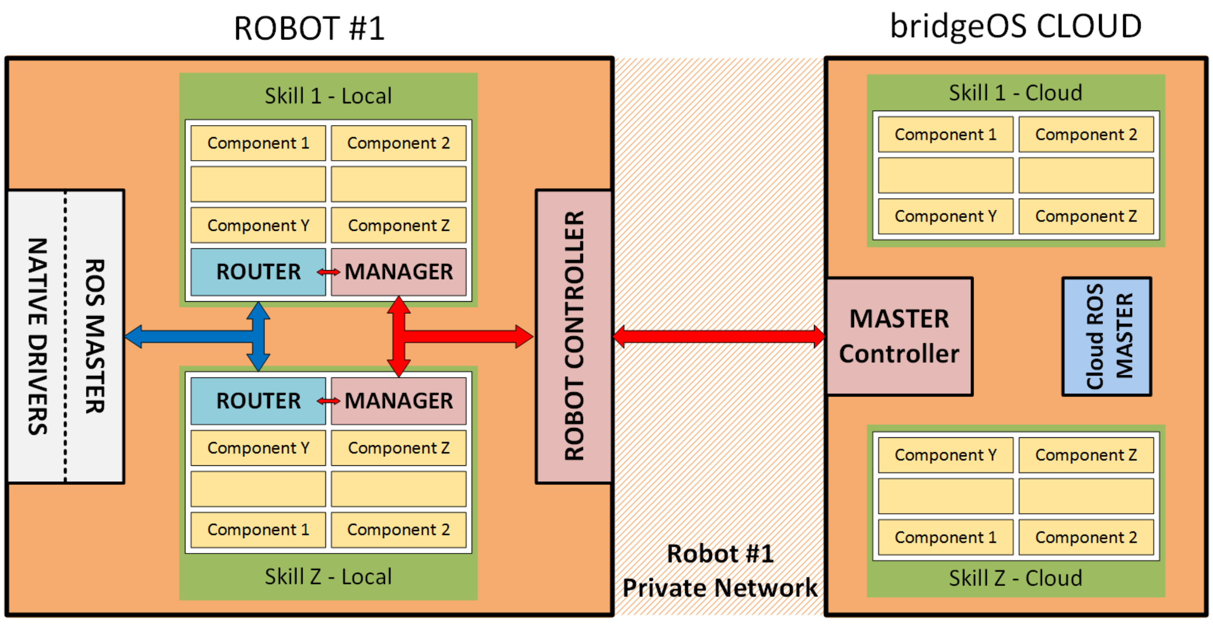


Figure 3.2: Detailed architecture from a robot point-of-view. Red arrows represent communication channels using WebSockets, dedicated to monitoring and management purposes, while blue arrows represent communication channels for ROS. Not depicted, are all connections involving components, which can be over ROS or use any other type of protocol.

Additionally, as a means to provide a portable middleware, and in substitution for being executed directly in the host like native applications, both the Master and Robot Controller modules can also be launched containerized through Docker, similarly to bridgeOS Skills. For Robot controllers, the benefits of executing them as native applications, are to allow greater control of the host and its resources, and enabled increased monitoring capabilities. For instance, even with its privileged mode, with Docker we are unable to obtain complete access to the host file system, a necessity if the user desires specific disk monitoring metrics or selects cost heuristics and functions that use them. With the Master Controller however, there is no real interest in a native deployment. Since it is located in a cloud environment, the oversight and resource management is most likely not handled by the same virtual machine, and rather performed through an API or some other tool supplied by the Cloud Provider.

On the other hand, using the controller modules as Docker containers helps achieve the interoperability characteristic desired for the middleware. Considering that, any system supporting Docker will be able to launch them without hassle. Containers also eliminate the disadvantage of lengthy pre-configurations, after all, dependencies are reduced solely to Docker, as any other dependency is dealt with during the Docker Image's creation; and of platform-dependent programs. Although, JavaScript and NodeJS are cross-platform, some packages used are themselves tied to certain operating systems. Lastly, initiating the middleware in a robot is greatly simplified, only a *docker run* command is necessary (or an initial *docker create* command and henceforth *docker start*, if we wish persist the containers).

We would also like to mention, that the choices regarding whether or not to containerize controllers, are user-related considerations to be taken when configuring their implementations and not an oversight of the solution proposed. Our position was to design the middleware in a way that sets up a highly customizable environment, malleable to each user's (i.e. the robots owners) specific requirements.

3.2 Master Controller

The Master Controller is the principal management component of our middleware, and has the purpose of serving multiple crucial roles. For starters, it acts as a gateway, providing robots with an entry point to the bridgeOS platform by exposing a common WebSockets interface for Robot Controllers. And is responsible for persisting and sharing their startup configuration. Inversely, it also enables bridgeOS services, users and applications to reach robots, by means of a HTTP web interface.

Located inside the cloud infrastructure, one of its core functions is to orchestrate cloud containers, using Docker Swarm, for Skill components, dedicated ROS masters and other containerized bridgeOS services. It this sense, it can manage and access all isolated subnetworks composing the robots virtual private networks.

Furthermore, another core function is to create a centralized information hub, logging data regarding the current states and offloading changes, and monitoring everything related to robots, skills, components and modules. In practice, the Master Controller is only charged with monitoring cloud containers and connections with robots, broadcasting any relevant data back to them. Meanwhile, Robot Controllers deal with the local side of their private network, forwarding to the Master all metrics and events generated by them, their Skill Managers and Skill Routers, such as ROS statistics and offloading decisions. This circumstance emerges from the design choice of implementing fault-tolerance and high-availability mechanisms locally (i.e. to provide cloud redundancy), essentially, each robot has its own Docker daemon, not connected to the cloud Docker Swarm. All this data horded by the Master Controller is persisted to a cloud database, described afterwards in 3.2.2. Although created specifically for this middleware, the database can be accessed by the bridgeOS platform through the API, or obviously directly within the cloud infrastructure (granted the credentials are known). Its schema can of course be fused into the existing back-end database of bridgeOS.

3.2.1 Robot API

The Robot API exposed by the Master Controller joins the existing bridgeOS solution with the thesis's middleware. It is a HTTP REST web interface, that implements management and monitoring services of the middleware and robots, porting part of the internal WebSockets interface to the remainder cloud infrastructure and, indirectly, users. Effectively maintaining the Master as an intermediary between all parties, for such matters.

This API provides therefore external access and control. And can be used for transmitting (de)activation commands for robots, skills or components, and letting users, provided they are properly authenticated, access their current state or even monitor metrics in real-time. The complete specification of the Robot API is disclosed in Table 3.1.

3.2.2 Database

For monitoring, administrative and modules integration purposes, the middleware uses a central database governed by the Master Controller. Figure 3.3 illustrates the database schema designed for our proposed middleware. Its structure was kept to the essential data needed by the different modules to operate autonomously, regrouping information about robots, skills and components, their latest statuses and all monitoring metrics. A separate table for logs was also included to facilitate reviewal and analysis due to the containerized environment, which can sometimes be burdensome to access. This structure also mixes relational data with fixed structure and other flexible and dynamic data, essentially, the JSON configurations of Skills and of their components. Which, if expressed traditionally as relational data, would consequently, vastly and unnecessarily, increase data redundancy and the complexity of the database schema.

Since relational data is suitable for SQL frameworks, and data with dynamic properties is suitable for NoSQL frameworks. To meet our hybrid requirement, we had to look for a compatible database management system (DBMS), able to offer the best of both environments.

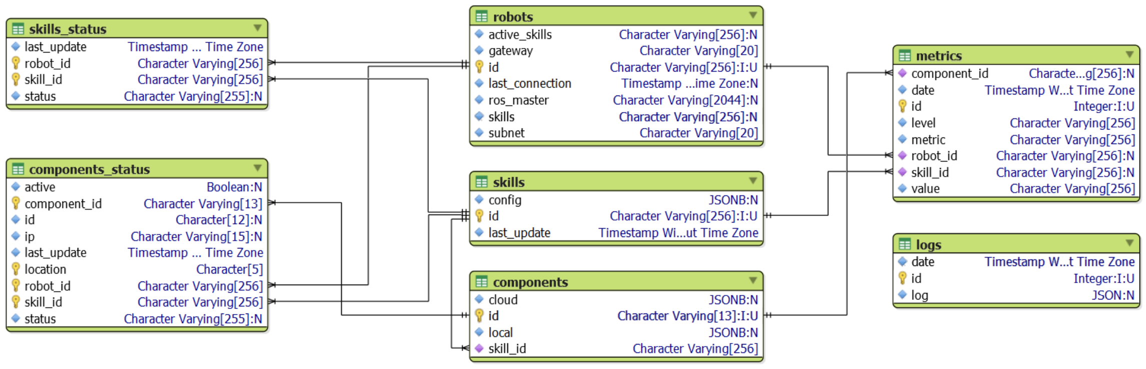

skills_status
last_update Timestamp ... Time Zone
robot_id Character Varying[256]
skill_id Character Varying[256]
status Character Varying[255]:N
components_status
active Boolean:N
component_id Character Varying[13]
id Character[12]:N
ip Character Varying[15]:N
last_update Timestamp ... Time Zone
location Character[5]
robot_id Character Varying[256]
skill_id Character Varying[256]
status Character Varying[255]:N
robots
active_skills Character Varying[256]:N
gateway Character Varying[20]
id Character Varying[256]:I:U
last_connection Timestamp ...ime Zone:N
ros_master Character Varying[2044]:N
skills Character Varying[256]:N
subnet Character Varying[20]
skills
config JSONB:N
id Character Varying[256]:I:U
last_update Timestamp Wi...ut Time Zone
components
cloud JSONB:N
id Character Varying[13]:I:U
local JSONB:N
skill_id Character Varying[256]
metrics
component_id Characte...g[256]:N
date Timestamp W...t Time Zone
id Integer:I:U
level Character Varying[256]
metric Character Varying[256]
robot_id Character Varying[256]:N
skill_id Character Varying[256]:N
value Character Varying[256]
logs
date Timestamp W...t Time Zone
id Integer:I:U
log JSON:N

Figure 3.3: Middleware Database Diagram

After pondering all factors and comparing different solutions, in terms of capabilities, performance and scalability, we selected PostgreSQL. PostgreSQL is a mature and powerful DBMS, that offers extensive SQL tools and features, some of which commonly provided by relational databases, including replication, key relationships and ACID [1] properties, a characteristic often overlooked by popular NoSQL databases due to their internal architecture. Moreover, it natively supports JSON, provides advanced functions and operators for handling JSON data, and is scalable to data-center wide applications.

Obviously, the database schema developed for the proposed solution is not dependent on PostgreSQL and can be ported to both SQL- or NoSQL- only databases, since the expected structure of the JSON data used remains known and, inversely, the relational data can be reorganized to suit the NoSQL document-based constraints. And of course, it can and most likely should be integrated with the existing database schema currently employed by a bridgeOS platform. To be noted, however, that user and authentication data pertinent to the current (non-extended) bridgeOS solution is excluded as it goes beyond the scope of this thesis.

3.3 Robot Controller

This module represents the local administrative part of the middleware, and is present within each robot, with the purpose of attaching them to the bridgeOS cloud. Locally, they take on a small network-related role, as they are responsible for bounding to the bridgeOS VPN and configuring its on-board firewall, to secure and conceal exchanges with the cloud. They also have to setup their local subnetwork, to be used by the robot's docker containers, and estab-

[1] Atomicity, Consistency, Isolation, Durability

lish the needed network routes, to ensure that all containers, modules, ROS nodes and other processes can communicate with each other, regardless of their location. When launched as a container, the Robot Controller appears to function as a network bridge, given that all inbound and outbound traffic is redirected through it, since the VPN tunnel is established inside its container.

When launched, Robot Controllers authenticate themselves with the Master Controller and retrieve their bridgeOS startup configuration, containing information regarding their local subnetworks and Skills. Earlier, during the overview section of this chapter, we acknowledged that a robot could already have a running ROS master. The Robot Controller is tasked with verifying this scenario and act accordingly when connected to the cloud. Therefore, when a ROS master is found, its ip address and port is disclosed, otherwise a request for launching a cloud ROS master is transmitted to the Master.

However, the core role of a Robot Controller is to manage the deployment of skills and orchestrate components, based on user configurations and requests from Skill Managers, to meet the desired performance, Quality of Service or any other quantifiable criterion. Although, for cloud containers, it merely forwards commands to the Master Controller. Additionally, it has to continuously monitor robot resources, local containers and the cloud availability, and share those metrics, both with its Skills and the cloud, to provide real-time information about its robot. The generated information is also exploited locally for allocating the available resources efficiently, during container deployments.

To permit cooperation with Skills, a Robot Controller operates a WebSocket server, implementing the common API (Table 3.3), that listens for incoming connections from Skill Managers. This enables administrative exchanges and constant feedback, shared up to and from the Master Controller. This continuous monitoring network, created between all modules, is then exploited by the *Master Controller*, for supervising current states, of the network and its components, and permitting logging any relevant information. And, by Skill Managers, to perform real-time performance checks, enabling adaptive offloading decisions. While the Robot Controller poses as an intermediary, on top of the monitoring information and management services it offers.

3.3.1 Robot Resources Allocation

A premise and driving factor of this thesis, is that robots have limited on-board resources, thus, Robot Controllers cannot blindly deploy containers locally simply based on Skill Managers requests, and have to analyze whether it is beneficial to do so. Consequently, developing an algorithm for this purpose became a necessity. The outcome of this development process is disclosed as Algorithm 1.

The Resource Allocation algorithm generates a decision based on current resource availability and usage, an heuristic function selected by the user, and a cost threshold. The computed decision can take on four different outcomes. The first two, are the expected binary boolean, True or False, stating clearly whether or not to deploy the requested Skill component. The third option, is also a positive outcome, albeit with a precondition, it specifies a component that has first to be migrated to the cloud, in order to free enough resources locally and confirm the request. The result is somewhat uncertain, because there are no guarantees that the migration will succeed, in which case the decision is overturned. Lastly, the fourth option is a negative outcome, but this time with a partial solution, launch the requested component in the cloud. Obviously, this response is only somewhat helpful when initializing a new Skill, since for requests emerging from offloading decisions, the desired and quantifiably best location, is the robot.

For each component we wish to launch locally, the algorithm procedure is as follows: first, we compute the entire resources it needs, including any links it might have. Linked components are essentially dependencies between components from the same Skill, stating they all must operate within the same location. Having computed the factual requirements, we examine if the currently available resources are enough to satisfy them. If so, the algorithm stops and announces its approval. Otherwise, the next step becomes finding a replacement. In this sense, we retrieve from the components instantiated locally, a list with all those movable, meaning those that can operate in either location. The goal now, is to find another component using enough resources to fill the gap between the current availability and the requirements. Of course, we must also take into account any eventual linked components of the replacement, since its dependencies remain in effect and are automatically propagated during a migration. This is the reason why the algorithm only needs to output the id of the base replacement. If no suitable replacement is found, the algorithm can merely make a final verification based on the required location, checking whether a cloud deployment could be attempted, before returning

Algorithm 1 Resource allocation algorithm for robot component deployment

```
Require: $skillId, componentId, heuristicFunction, threshold$
Ensure: $getRequiredLocation(skillId, componentId) \in \{local, either\}$
 1: $freeResources \leftarrow getAvailableResources()$
 2: $resources \leftarrow getRequiredResources(skillId, componentId)$
 3: $linkedComp \leftarrow linkedComponents(skillId, componentId)$
 4: for all $c \in linkedComp$ do
 5:     $resources \leftarrow resources + getRequiredResources(skillId, c)$
 6: end for
 7: if $resources \leq robotResources$ then
 8:     return true
 9: else
10:     $totalResources \leftarrow getTotalResources()$
11:     $movableComponents \leftarrow \varnothing$
12:     $replacement \leftarrow \varnothing$
13:     $minCost \leftarrow Infinity$
14:     for all $s \in getActiveSkills()$ do
15:         for all $c \in getComponents(s)$ do
16:             if $c \notin linkedComp$ and $getRequiredLocation(c) = “either”$ then
17:                 $movableComponents[s] \leftarrow movableComponents[s] \cup c$
18:             end if
19:         end for
20:     end for
21:     for all $s \in movableComponents$ do
22:         for all $c \in movableComponents[s]$ do
23:             $availableResources \leftarrow freeableResources + getResourceUsage(s, c)$
24:             for all $link \in linkedComponents(s, c)$ do
25:                 $availableResources \leftarrow availableResources + getResourceUsage(s, link)$
26:             end for
27:             if $resources > availableResources$ then
28:                 continue
29:             end if
30:             $heuristicCost \leftarrow heuristicFunction(s, c, requiredResources, availableResources,$
    $totalResources, constants, weights)$
31:             if $heuristicCost < minCost$ then
32:                 $replacement \leftarrow c$
33:                 $minCost \leftarrow heuristicCost$
34:             end if
35:         end for
36:     end for
37:     if $replacement \neq \varnothing \ \& \ minCost \leq threshold$ then
38:         return $replacement$
39:     else if $getRequiredLocation(skillId, componentId) = “either”$ then
40:         return $“cloud”$
41:     else
42:         return false
43:     end if
44: end if
```

a negative response.

Only a question now remains, when and why should a new component replace another? This is a valid question, even more so, when the latter is already operating. This issue however, is resolved with the use of a heuristic function to compute a commensurable deployment cost, that is leveled off by a threshold, dictating when such scenario is beneficial. Since not all robots serve the same purpose, a single common heuristic can hardly be an efficient response for the different necessities. That is why five confluent heuristics have been developed. Each offering a alternate approach and addressing a different objective. Even though the last three build on the previous heuristics. These heuristics and enumerate and described here after in this subsection. Both threshold and heuristic, can be picked by a user through its robot's configuration. A consideration to have relates to the trade-off between the added computational cost of using particular cost function and its potential efficiency in reaching the objective sought for the Resource Allocation algorithm. For example, maximize resource usage or minimize overall downtime.

To be noted, deployments are examined individually due to the asynchronous nature of the requests sent by the Skills Managers, either for Skill initialization or offloading purposes. Furthermore, there is few benefit to risk having groups of components not launched, solely because there are only enough resources, or valid replacement for a subset of them.

- **Resource Usage Heuristic (Algorithm 2):** The Resource Usage heuristic function aims to maximize resource allocation on the robot. To this end, it computes a ratio, for each type of hardware resource the robot has, of the expected waste of resources this replacement will generate. Afterwards, the ratios are returned, accompanied by their geometric mean, as a way to provide a single comparable cost.

  Clearly, the optimal case is the one where those ratios are all zero, representing a scenario without resource wastage. Anything less, the replacement does not satisfy the requirements, and anything more, resource allocation is not maximal. There is however an assumption that we make, which is, available resources are never null due to the dynamic environment of a computer.

- **Migration Cost Heuristic (Algorithm 3):** The Migration Cost heuristic goes on a different approach, it takes into account the expected cost of migrating the replacement, in terms of network bandwidth, both in absolute data (i.e. in bytes) and monetary cost, with the objective of minimizing the latter. The argument behind the utility of this heuristic function is that, with mobile devices such as robots, the second most important resource, after energy, is arguably network bandwidth. As it can be limited, possibly making large exchanges slow or borrowing network share from other processes, and might also have a monetary cost associated (e.g. mobile carrier rate for exchanging 1 gigabyte of data).

The bandwidth calculated, represents the amount of data needed to synchronize the replacement and its dependencies with their cloud counterparts. This includes stored ROS messages (e.g. replicating a map generated locally) and shared Docker volumes, which enable components to persist data they require or construct during their execution. Because the requested component is common for all replacements, its eventual cost is discarded from this measure.

- **Resource Expenditure Heuristic (Algorithm 4):** This heuristic combines three perspectives in order to produce a single measure of the overall resource expenditure caused by the substitution. It collects the average waste provided by the Resource Usage heuristic and the total bandwidth from the Migration Cost heuristic, weighting them together with a third measure computed locally.

  Fundamentally, the Resource Expenditure heuristic retrieves the network share currently used by the replacement and any linked components, and weighs it based on the proportion of bandwidth consumed locally. This establishes an assessment of the expected network usage variation resulting from the migrations of such replacement, since any local exchange will then have to involve the cloud. Of course, the measure is not perfect, because some of those exchanges might happen with linked components, which also migrate. However, given the monitoring capabilities at our disposal, performance considerations and the black-box approach of Skills, we cannot have access to such detailed metrics.

  Each factor is weighted, in function of what the user deems more important, based on his objectives, leading to an aggregated expenditure measure. Representing a trade-off between the gains of reduced resource wastage, the migration cost, and possible loss due to increased network usage.

- **Temporal Cost Heuristic (Algorithm 5):** The Temporal Cost heuristic function builds on both previous heuristics and introduces the temporal aspect. Basically, it linearizes over time the expected bandwidth cost produced by the migration, in function of the currently available network capacity, discounted by its expected variation. Its interest resides in

Algorithm 2 Resource Usage Heuristic

```
Require: $skill, component, requiredResources, freeResources, totalResources$
 1: $resources \leftarrow cpu, gpu, ram, bandwidth, IO$
 2: $ratios \leftarrow$ Infinity
 3: for all $res \in resources$ do
 4:     if $totalResources(res) = 0$ then
 5:         $ratios[res] \leftarrow 0$
 6:     else
 7:         $ratio \leftarrow \frac{freeResources[res] - requiredResources[res]}{totalResources[res]}$
 8:         if $ratio < 0$ then
 9:             exit for
10:         else
11:             $ratios[res] \leftarrow ratio$
12:         end if
13:     end if
14: end for
15: $average \leftarrow \left( \prod_{res \in resources} ratios[res] \right)^{\frac{1}{size(resources)}}$
16: return $ratios, average$
```

computing the amount of time necessary to complete the substitution, and consequently, the upper bound duration of the component's downtime.

- **Weighted Cost Heuristic (Algorithm 6):** This last heuristic is an accumulation of the work produced by the other heuristics. It aims to consolidate an overall measure, weighting both cost factors, time and resources. It is both more complete and computationally expensive, considering it is comparable to executing all other heuristics at once, on top of its final weighing. However, it gives the opportunity to not discard either cost aspects, and provides users with the option of determining a general best candidate for replacement.

3.3.2 Components Synchronization

Another important function of the Robot Controller, is to handle state synchronization of components during their migrations. Performing this, is not as straightforward as one would think. Normally, if we were using virtual machines, we could easily capture a snapshot of their current state and use it to replicate a given virtual machine somewhere else. However, this is not achievable with Docker containers, since Docker's layered storage architecture is much less transparent and makes it impossible to simply replicate specific layers in another host. Therefore, we have to rely on other mechanisms such as synchronizing shared Docker volumes,

Algorithm 3 Migration Cost Heuristic

```
Require: $skill, component, constants$
 1: $bw \leftarrow$ [Infinity]
 2: $bw[stateTransfer] \leftarrow getSyncSize(\text{“}DockerVolume\text{”}, skill, component)$
 3: $bw[rosSynchronization] \leftarrow getSyncSize(\text{“}ROS\text{”}, skill, component)$
 4: for all $c \in linkedComponents(skill, component)$ do
 5:     $bw[stateTransfer] \leftarrow bw[stateTransfer] + getSyncSize(\text{“}DockerVolume\text{”}, skill, c)$
 6:     $bw[rosSynchronization] \leftarrow bw[rosSynchronization] + getSyncSize(\text{“}ROS\text{”}, skill, c)$
 7: end for
 8: $bw[total] \leftarrow bw[stateTransfer] + bw[rosSynchronization]$
 9: $bw[cost] \leftarrow \frac{bw[total]}{1024^3} * constants[CostPerGB]$
10: return $bw$
```

which can be specific to each Skill or component, and storing ROS messages, based on user configurations, to be shared with the help of the Skill Router when a migration happens.

Components synchronization is not constrained solely to migrations. Some of the objectives tackled by this thesis regard questions of redundancy, robustness and fault-tolerance. For those reasons, we implemented mechanisms to enable the Robot Controller to adapt to unreliable circumstances and act accordingly. Hence, the added responsibility for resynchronizing data whenever confronted with disconnections and other network failures. In such events, it keeps any local messages, retransmitting them when possible, and informs the local managers, triggering any eventual data synchronization behavior specific to each Skill.

3.3.3 Cloud Role Takeover

Expanding on the topics addressed in the previous point, the mechanisms discussed expand beyond components synchronization purposes, to include cloud status monitoring and temporary role appropriation. To increase reliability on the middleware and enable robots to gain some independence vis-à-vis the cloud during disruptions, the Robot Controller should be capable of replacing the cloud functions, at least partially, whenever disconnected. This scenario is referred to as Cloud Role Takeover and its process is defined by the Algorithm 7.

The algorithm outlines which monitoring events to look for and when to trigger or revert the takeover. It is a continuous task performed throughout a robot's active state. The first step of its cycle, is to monitor cloud connections, particularly with the Master Controller. If they become unavailable after a certain delay, the takeover initiates.

This immediately triggers a Skill recovery mechanism. From one side, Skill Managers are warned, driving them to halt their dynamic offloading decisions, shift the active locations of

Algorithm 4 Resource Expenditure Heuristic

```
Require: $skill, component, requiredResources, freeResources, totalResources, constants, weights$
 1: $variation \leftarrow 0$
 2: $expenditure \leftarrow 0$
 3: $localBandwidth \leftarrow getLocalBandwidth(skill, component)$
 4: $cloudBandwidth \leftarrow getCloudBandwidth(skill, component)$
 5: if $localBandwidth \neq 0$ & $cloudBandwidth \neq 0$ then
 6:     $localWeight \leftarrow \frac{localBandwidth}{localBandwidth+cloudBandwidth}$
 7:     $variation \leftarrow getResourceUsage(skill, component, \text{“}bandwidth\text{”}) * localWeight$
 8: end if
 9: for all $c \in getLinks(skill, component)$ do
10:     $localBandwidth \leftarrow getLocalBandwidth(skill, c)$
11:     $cloudBandwidth \leftarrow getCloudBandwidth(skill, c)$
12:     $localWeight \leftarrow \frac{localBandwidth}{localBandwidth+cloudBandwidth}$
13:     $variation \leftarrow variation + getResourceUsage(skill, c, \text{“}bandwidth\text{”}) * localWeight$
14: end for
15: $RUH \leftarrow ResourceUsageHeuristic(skill, component, requiredResources, freeResources,$
    $totalResources)$
16: $MCH \leftarrow MigrationCostHeuristic(skill, component, constants)$
17: $expenditure \leftarrow RUH[average] * weights[gain] + MCH[total] * weights[cost] + variation *$
    $weights[loss]$
18: return $expenditure, variation$
```

Algorithm 5 Temporal Cost Heuristic

```
Require: $skill, component, requiredResources, freeResources, totalResources, constants, weights$
 1: $MCH \leftarrow MigrationCostHeuristic(skill, component, constants)$
 2: $REH \leftarrow ResourceExpenditureHeuristic(skill, component, requiredResources, freeResources,$
    $totalResources, constants, weights)$
 3: return $\frac{MCH[total]}{getAvailableResources(bandwidth)-REH[variation]}$
```

components to local whenever possible, and expect error statuses about the disruption. While at the same time, the Robot Controller analyses active Skills to try and deploy their cloud segment locally. Unlike with the Resource Allocation algorithm, the goal is not a quantitative one (i.e. fitting the most components locally), but rather a qualitative one, where we attempt to restore the proper operation of Skills. For this reason, we take each Skill individually and examine whether it is possible to deploy all of its cloud components at once in the robot. Skills that cannot satisfy this requirement, or have components that must exclusively remain in the cloud, are ignored since their functionality remains hindered.

Upon cloud connectivity reestablishment, the takeover ceases and its stoppage is propagated to the Skill Managers, letting them revert their actions. And, since the migrations that occurred during the role appropriation are known, the duplicated components are stopped.

Algorithm 6 Weighted Cost Heuristic

```
Require: $skill, component, requiredResources, freeResources, totalResources, constants, weights$
1: $REH \leftarrow ResourceExpenditureHeuristic(skill, component, requiredResources, freeResources, totalResources, constants, weights)$
2: $TCH \leftarrow TemporalCostHeuristic(skill, component, requiredResources, freeResources, totalResources, constants, weights)$
3: return $REH[expenditure] * weights[cost] + TCH * weights[time]$
```

An apparent giveaway of this mechanism, is that robots can operate without the cloud. This is true, but up to a certain degree. For starters, their services remain limited to those provided by the Skills restored. And another implication is that, any remote control and access, happening through bridgeOS, is lost. However, an incorrect interpretation would be to assume robots can be launched and initialize Skills on their own, without an initial cloud connectivity[2]. While this might seem like an oversight, this characteristic stems in fact from a security precaution: ensure offline configuration changes are imperatively applied by robots. The argument is to avoid unforeseen risks arising between environment updates (e.g. robot was moved or its location might include new elements such as people) and previously used Skills.

3.4 bridgeOS Skills

BridgeOS Skills are assemblies of components, working together to provide particular functionalities such as navigation, speech, grasping and so on. They already implement an architecture based-on microservices, where each component executes limited processing tasks or expose services (e.g. a ROS node), and adopt the containerized approach of Docker.

With our middleware, bridgeOS Skills are embodied slightly differently. Their components can be launched both locally or in the cloud, thus ceasing to be represented by single containers. However, only one container of each is kept active, while the other is either stopped or made incapable of interacting. And, they are bundled together with two dedicated middleware modules, Skill Manager and Skill Router, operating with the purpose of organizing them. Hence, each skill instantiation can now become transparently distributed between the robot and the bridgeOS cloud. Furthermore, to enhance cooperation and data sharing between components, a common Docker volume, kept synchronized and replicated across locations, is provided and accessible by all within the same Skill, including the respective middleware modules.

[2]Robots can, logically, launch the middleware without any network connectivity. However, they will simply not instantiate Skills before being provided by the Master Controller of their latest configuration.

Algorithm 7 Cloud monitoring algorithm for role takeover

```
Require: $disconnectionDelay$
1: $takeoverActive \leftarrow$ false
2: $migrations \leftarrow \varnothing$
3: repeat
4:     $status \leftarrow monitorCloudConnection()$
5:     if $status \in [ConnectionClosed, ConnectionLost, NoResponse]$ then
6:         if $now() - getLastestPing() \geq disconnectionDelay$ then
7:             $takeoverActive \leftarrow$ true
8:             for all $skill \in getActiveSkills()$ do
9:                 $sendSkillManager(skill, “ConnectionLost”)$
10:                $components \leftarrow \varnothing$
11:                $resources \leftarrow \varnothing$
12:                for all $c \in getComponents(skill)$ do
13:                    if $getActiveLocation(skill, c) = “local”$ then
14:                        continue
15:                    else if $getRequiredLocation(skill, c) = “cloud”$ then
16:                        exit for
17:                    else
18:                        $components \leftarrow components \cup c$
19:                        $resources \leftarrow resources + getRequiredResources(skill, c)$
20:                    end if
21:                end for
22:                if $components \neq \varnothing$ & $resources \leq getAvailableResources()$ then
23:                    for all $c \in components$ do
24:                        $launchComponent(skill, c)$
25:                    end for
26:                    $migrations[skill] \leftarrow components$
27:                end if
28:            end for
29:        else
30:            continue
31:        end if
32:    else if $takeoverActive$ then
33:        $takeoverActive \leftarrow$ false
34:        for all $skill \in migrations$ do
35:            $sendSkillManager(skill, “ConnectionEstablished”)$
36:            for all $c \in migrations[skill]$ do
37:                $stopComponent(skill, c)$
38:            end for
39:        end for
40:        $migrations \leftarrow \varnothing$
41:    end if
42: until $robot\ shutdown$
```

The Skill Manager supervises Skill performance and the state of all components, along some other metrics related to the robot itself, such as network bandwidth or battery available, to generate offloading decisions in real-time using user policies. While the Skill Router essentially abstracts the location of all components, rerouting communications accordingly, based on the dynamic offloading decisions received. An intended benefit of this design, is that it allows specific components to concurrently operate during migrations, switching the active location only when their counterparts are available, thus reducing any possible downtime and avoiding conflicts. Additionally, for the purpose of increasing middleware resilience and robustness, they both reside exclusively in the robot, so that, if the Master Controller fails or the cloud infrastructure becomes unavailable, they can continue operating and decide the appropriate course of action, as addressed before with the fault-tolerance mechanisms implemented by the Robot Controller and the interactions depicted between all three modules.

3.4.1 Skill Manager

As stated before, Skill Managers have the critical role of providing the adaptive offloading capabilities sought by this middleware. Specific to a single Skill, each is a local information hub that regroups data it deems pertinent from all available sources, constantly sharing it both upwards, to the Robot Controller and through it the bridgeOS Cloud, and downwards to the components via its API. However, the responsibilities of a Skill Manager are even broader, since it has to initialize skills and manage components as well, whereas the Robot Controller simply orchestrates their deployment.

To better operate, it needs to persist some data across Skill launches, including variable configuration or metrics, to avoid redundant retrievals when queried by components. Consequently, it uses an embedded and lightweight file-based NoSQL datastore for NodeJS, NeDB, stored in its common Docker volume. By persisting such data, previous Skill launches can serve as a backbone for selecting the deployment locations of components during initialization.

Generating Dynamic Offloading Decisions

Using all collected data, it generates offloading decisions in real-time for components, that take into account the offloading Policies emboldening user preferences and goals. Policies can be adhered to, combined or discarded, and are intended to be either maximized or minimized. All of them are implemented from the perspective of the robot, by means of two cost functions, one per component location. Currently, 4 Policies targeting different perspectives are supported:

- **Energy Usage Policy**: measures the energy consumption of a component based on the resources it uses. However, for cloud components, we only consider the usage of the robot's network interfaces resulting from their exchanges.
- **Network Usage Policy**: computes a network cost based on the usage made by a component, with regards to the opposite location, meaning exchanges between the cloud and robots.
- **Cost Policy**: quantifies a cost derived from the energy consumption, network usage and other resources utilized by a component. Unlike other policies, with cloud components, it also provides the possibility of computing a cost based on cloud resources, whenever appropriate (e.g. Cloud instance renting).
- **Performance Policy**: measures the performance of a component in terms of some activity metric and resource availability. Such policy is heavily influenced by the purpose each component is serving and its type. For instance, with simpler components, measuring their network latency might suffice, while with ROS nodes, their frame rate or packet loss rate might be more adequate. Which is why it was implemented with customization and adaptability in mind, enabling more intricate and specific measures to be integrated. Further details on this topic are provided in Chapter 4.

Of course, simply computing the policy costs are not enough for deciding whether a component should migrate. First, we have to estimate if it would be beneficial to do so, and by how much. Estimations, by definition, always have some degree of inaccuracy attached to them, therefore using a model that can learn and be optimized, all of which in real-time to support dynamic policy updates is a must. Neural networks satisfy those requirements, hence our choice for selecting them as models for computing efficient decisions. Nevertheless, in order to become useful, they still have to be trained, at least partially. For this reason, a set of functions were also developed to manually estimate the expected migration reward, temporarily serving as a basis for computing offloading decisions while the neural networks are being initialized.

The materialization of these procedures is represented by Algorithm 8, which is implemented by Skill Managers and ran periodically. First, this Dynamic Offloading algorithm utilizes numerous preconditions to filter out components that cannot migrate. Either by default,

Algorithm 8 Dynamic offloading algorithm for skills components

```
Require: $componentId, policy, threshold$
Ensure: $getRequiredLocation(componentId) = \text{“}either\text{”}$
 1: $network \leftarrow getNeuralNetwork(componentId)$
 2: repeat
 3:     $migrate \leftarrow$ false
 4:     $location \leftarrow getActiveLocation(componentId)$
 5:     $resources \leftarrow resourceUsage(componentId)$
 6:     if $location = \text{“}local\text{”}$ or $resources < getAvailableResources(\text{“}robot\text{”})$ then
 7:         if $hasRecommendation(componentId)$ then
 8:             $migrate \leftarrow getRecommendation(componentId)$
 9:         else if $offloadingStrategy = \text{“}dynamic\text{”}$ then
10:             $metrics \leftarrow getStatistics(componentId)$
11:             $costs \leftarrow computeCosts(policy, metrics)$
12:             $networkDecision \leftarrow computeNetworkDecision(network, policy, threshold,$
    $metrics, costs)$
13:             if $networkDecision \neq \varnothing$ then
14:                 $migrate \leftarrow networkDecision$
15:             else
16:                 $reward \leftarrow estimateMigrationReward(policy, costs, metrics)$
17:                 if $reward \geq threshold$ then
18:                     $migrate \leftarrow$ true
19:                 end if
20:                 $trainNetwork(network, migrate, reward)$
21:                 if $trainingCount(network) \geq 1000$ then
22:                     $setTrained(network)$
23:                 end if
24:             end if
25:         end if
26:     end if
27:     return $migrate$
28: until $skill\ shutdown$
```

for instance if the component has a fixed location or dynamic offloading is disabled, or due to expected runtime circumstances, such as lack of free resources. And afterwards, it fetches the latest statistics of a component to compute the Policy costs, and forwards the results to the respective neural network so a decision can be generated.

Algorithm 9 formulates with greater detail how neural networks are used and handled, including the scenarios in which they are unable to return a decision. In such eventualities, it resorts to manually quantifying the expected migration reward. In either case, the computed prediction is compared with a user-defined threshold before reaching a decision. Thresholding is used since it is logical to assume migrations generate some cost, as we inferred when present-

Algorithm 9 Compute Network Decision algorithm

```
Require: $network, policy, threshold, metrics, costs$
 1: $migrate \leftarrow$ false
 2: if $network\ is\ not\ Initialized$ then
 3:     $network \leftarrow initializeNeuralNetwork(network)$
 4:     return $\varnothing$
 5: end if
 6: if $network\ is\ Trained$ then
 7:     $prediction \leftarrow network(costs)$
 8:     if $prediction \geq threshold$ then
 9:         $migrate \leftarrow$ true
10:     end if
11:     if $training\ interval\ Reached$ then
12:         $reward \leftarrow estimateMigrationReward(policy, costs, metrics)$
13:         $trainNeuralNetwork(network, migrate, reward)$
14:     end if
15: else
16:     return $\varnothing$
17: end if
18: return $migrate$
```

ing the Resource Allocation mechanisms and algorithms implemented by the Robot Controller (Subsection 3.3.1), thus they should only happen when it remains beneficial from an overall standpoint and not solely component-wise.

It is apparent in both algorithms, how neural networks evolve, from their initialization to their continued live learning. To train them, we use each prediction and policy costs to determine in retrospect, if the decision generated was indeed beneficial. Of course, even this conclusion is imperfect since there are external factors in play (e.g. sudden spike in resource usage, connectivity disruptions, new Skills launched), however our premise is that neural networks will be able to form an internal representation that accounts for them, as a result of periodic training.

Components API

Skill Managers provide components with a small REST API, presented in Table 3.2, letting them access and publish metrics or monitoring events relating to their Skill, themselves or other components. Another functionality of this API, is to enable components to manually push offloading recommendations, which are then processed as depicted in Algorithm 8. Besides custom metrics that can shared by components, Skill Managers also offer by default numerous monitoring metrics, that can be useful, such as resource usage, ROS statistics and other statistics (i.e. network latency).

3.4.2 Skill Router

The core function of a Skill Router, as its name suggests, is to route messages between components and operate as a proxy for those collaborating through each other's services, essentially abstracting the location of all components. It communicates with the Skill Manager to remain up to date regarding the offloading decisions, and keeps track of the components ip addresses. And given its convenient position within a bridgeOS network, helps complement our monitoring capabilities of components, by tracking all forwarded exchanges and collecting statistics about those communications, including those happening over ROS. The generated data is then shared with the Skill Manager and offered via ROS under the topic **/statistics**, which is commonly used for the same monitoring purposes.

The allow this transparent distribution to happen, where components can be deployed independently anytime and in any location, we have to implement a mechanism capable of satisfying our preconditions: container images cannot be modified to respect the interoperability characteristic sought by bridgeOS, components must continue interacting without (or with minimal) interruptions during offloading, and, we have to ensure idle components do not engage with others. Therefore, options consisting of injecting hosts directly onto containers or using private DNS servers were not viable.

Our solution consists in having Skill Routers map services we know components offer or will use from one another. For instance, with components operating within ROS, it can map topics and services, they published or subscribed, to specific namespaces that are different for cloud and robot components. Similarly, the Skill Router maps known URIs and HTTP methods, redirecting afterwards requests and responses to both ends, and achieving the same transparency provided to ROS nodes, for web services components offer or consume. By operating in such manner, the Skill Router vastly facilitates switching the active location of components, since applying the correct route can be virtually reduced to a boolean flag. Ensuring, consequently, that idle components stop operating, while remaining able to coexist without disrupting their counterparts. This was a desired feature for our middleware and a requirement of our strategy for minimizing downtime, which consists in waiting for a component to be ready in the new location before committing to its migration.

3.5 Private Networks and Communications Protocols

As addressed during the overview of the middleware architecture, all entities interacting within a bridgeOS ecosystem are protected by a global VPN securing all communications. While conjointly, further isolated robots by bounding them to dedicated internal private networks. Fundamentally, robots are allocated subnets for local and cloud use, with their access overseen by the VPN server. To provide such network capabilities, we selected a robust open-source solution, OpenVPN.

With regards to our middleware needs, communications between the developed modules occur using the web-oriented WebSocket protocol and exposing web services access points. Of course, the benefit of this, is to take advantage of its inherent capabilities, namely bidirectional exchanges, interoperability, performance, Firewall-friendly approach and high-throughput, well suited for real-time web exchanges. For it, we designed an event-based protocol dealing with management and monitoring exchanges using JSON messages, that defines fault-tolerance and retransmission mechanisms, and sets up push (e.g to publish a status update) and request-response (e.g. request the location of a component) exchanges logic. Table 3.3, provides an overview of the events we deemed essential for our communications protocol.

Finally, we assess a possible risk, robots can sometimes be unable to join the bridgeOS VPN, either because the server failed (highly unlikely) or their local network restricts VPN traffic (increasingly possible nowadays). In such scenarios, we must to still be capable of providing basic security guarantees. Therefore, whenever VPNs are unavailable, our modules will switch to the secure version of WebSocket, WSS, which is protected by TLS, akin to HTTPS for HTTP. And all docker subnetworks will be hid, protected by firewall rules, setup in both sites, to restrict their access. On top of that, if the Robot Controller is containerized, all external traffic is still redirected through it, easing network monitoring.

Summary

This Chapter described the core aspects of our middleware proposal, addressing its overall architecture, network arrangement and software modules. We also described the relevant aspects that allowed us to implement and achieve some of the objectives defined for the middleware. This is the main outline of the architecture of the system. Next chapter delves into the technical intricacies of the middleware and its modules, and how we integrated them with bridgeOS.

Table 3.1: Master Controller REST API

Method	**URI**	**Description**
GET	/auth	Authenticates a user and opens a new session
GET	/auth/renew	Extends the validity an open session
GET	/robot	Retrieves exhaustive information about all connected robots, their skills and components
POST	/robot/start	Starts all idle Robot controllers
POST	/robot/stop	Executes a soft stop of all connected robots
POST	/robot/shutdown	Shuts down all connected robots permanently
GET	/robot/*/info	Retrieves exhaustive information about the current state of a robot, his skills and components
POST	/robot/*/start	Starts, a previously stopped, Robot controller
POST	/robot/*/stop	Executes a soft stop of the robot, its Robot Controller will shutdown everything but become idle, awaiting a start command
POST	/robot/*/shutdown	Shuts down the middleware at the robot side
POST	/robot/*/skill/add	Adds a new Skill to the robot
POST	/robot/*/skill/remove	Removes a particular Skill from the robot's currently available Skills
POST	/robot/*/skill/*/start	Instantiates a Skill
POST	/robot/*/skill/*/stop	Shuts down an active Skill.
GET	/robot/*/skill/*/info	Retrieves extensive information regarding the current state of a Skill
POST	/robot/*/skill/*/component/*/start	Starts a Skill component at the requested location
POST	/robot/*/skill/*/component/*/stop	Stops a Skill component operating at the specified location
POST	/robot/*/skill/*/component/*/migrate	Migrates the active location of a component to the other side
GET	/robot/*/skill/*/component/*/info	Retrieves information about a component and its containers
POST	/database/query	Queries the database for the latest metrics and activities about robots.

(Note: The asterisk symbol present in the URIs, stands as a placeholder for the resources, namely Robots, Skills and components IDs.)

Table 3.2: Skill Manager REST API

Method	**URI**	**Description**
GET	/components	Retrieve information about all components of its Skill.
GET	/components/location	Retrieve current active locations of all components.
GET	/components/{id}	Request current status of a component.
GET	/components/{id}/location	Request current active location of a component.
POST	/components/{id}/location	Publish a manual offloading decision for a component.
GET	/metrics	Query the latest values of multiple metrics.
POST	/metrics	Update the values of multiple metrics.
GET	/metrics/{metric}	Retrieve the latest value of a single metric.
POST	/metrics/{metric}	Update the value of a single metric.

Table 3.3: WebSocket Communications Protocol

Event	Description
authentication_client	Client authentication request for accessing the Master Controller through its WebSocket API.
authentication_robot	Authentication request for connection robots to the cloud platform.
authentication_success	Explicit response indicating a successful authentication.
authentication_failure	Authentication failure response.
connection_established	Signals that the connection between the cloud platform and a robot has been (re)established.
connection_lost	Signals that the connection between the cloud platform and a robot has been severed.
error_message	Transmission of errors resulting from processing failures or format inconsistencies.
shutdown	Requests or signals a robot permanent shutdown.
robot_start	Orders a robot to resume operations.
robot_stop	Orders a robot to stop operating and remain idle.
robot_config	Transmission of the robot configuration for initializing the middleware locally.
robot_config_get	Requests the robot configuration from the Master Controller.
robot_config_update	Forwards minor configuration updates during runtime.
offloading	Sets or reports the current offloading mode used by a robot.
offloading_get	Requests the offloading mode currently used by a robot.
skill_start	Orders the launch of a Skill.
skill_stop	Orders the shutdown of a Skill.
skill_status	Forwards status updates of a Skill.
skill_status_get	Requests the current status of a Skill.
skill_connnect	Connects a Skill Manager with the Robot Controller.
skill_config	Transmits the local configuration of a Skill.
skill_config_get	Requests the local configuration file of a Skill.
component_start	Requests the launch of a Skill component.
component_stop	Requests the shutdown of a Skill component.
component_information	Forwards the latest information about a Skill component container, including its ID and IP address.
component_status	Forwards status updates of a Skill component.
component_status_get	Requests the current status of a Skill component.
component_location	Reports the current active location of a Skill component.
component_migrate	Orders the deployment of a skill component to the requested location.
metrics	Forwards one or more metrics.
metrics_get	Requests the latests measures of one or more metrics.

4 Implementation

This chapter covers more exhaustively the intricacies and inner-workings resulting from the implementation of the solution that was proposed in Chapter 3.

4.1 Managing Private Robot Networks

We proclaimed previously that all communications made within the bridgeOS ecosystem are protected by a common VPN, and went further stating robots are isolated from each other and have their own private subnetworks. We achieved those claims using OpenVPN coupled with tailored firewall and routing rules.

Robots are associated with specific certificates, allowing them to connect to our VPN server and consequently open a secure network tunnel. Using the client management functionalities of OpenVPN, we define specific configurations for each one, to express their subnetworks and push network robots enabling them to reach their cloud components. In our cloud back-end configuration, we dynamically define which subnetworks are available, and through cooperation between Master and Robot Controllers, have them create the local docker networks that will contain the Skill components.

While the local network routes enable subnetworks to interact, whenever required, firewall rules protect them from unauthorized access. A question might arise from this architecture, since robots are isolated, how can they interact and cooperate? With bridgeOS, robot cooperation happens at a higher-level, through the bridgeOS applications. This middleware only plays an indirect role regarding this topic, connecting robots and Skills with the bridgeOS Cloud Platform. The interactions that occur afterwards are beyond the scope of this middleware.

4.2 Performing Dynamic Offloading

As covered before, Skill Managers use neural networks to generate offloading decisions based on the policies used. We created a JavaScript class, ComponentDecisionModule, that encloses the decision making processes together with the neural networks. It implements the algorithms we presented in the previous chapter and provides an interface for handling neural networks, including their persistence across Skill launches. Each component is attributed a single object of this class throughout its life-cycle, independently of its location.

The cost functions of policies were however implemented in a separate class, PolicyEstimator. The main motivation behind this choice, was to provide developers with an easy solution, in case they ever need to extend the policies. To be capable of computing costs, we inject into it user-defined constants (e.g. costs, weights) and types of resources, specific to each robot and Skill. Then, provide the latest component metrics, whenever their cost functions are called. The equations below (4.1 through 4.6) detail the current cost functions we selected for our PolicyEstimator. Regarding the Performance Policy, we decided the use a common cost function, used for both locations, that computes costs based the component type. For ROS nodes, it outputs their message loss rate within the ROS network, while for the remaining it returns their network latency. This simplicity was intended to encourage customization, since Skills are too diverse to use the rely on the same function.

Customization is a principle we wanted to encourage. For this reason, we allow users, through the configuration files of both robots and Skills, to vastly customize the parameters used by Skill Managers for dynamic offloading, while retaining their black box aspect. This includes not only the architecture of the neural networks (e.g. number of layers and nodes, activation functions), but also policy cost functions. Thanks to the characteristics of JavaScript, we can load strings as functions during runtime. Therefore users can design cost functions more appropriate to their needs, knowing in advance the variables they have access (constants, resources and metrics).

$$EnergyCost_{local} = \sum_{h \in RHR} S_h * Ec_h \tag{4.1}$$

$$EnergyCost_{cloud} = \sum_{i \in RWI} S_i * Ec_i \tag{4.2}$$

$$NetworkCost_{local} = Network_{IO} * \frac{BW_{cloud}}{BW_{total}} \tag{4.3}$$

$$NetworkCost_{cloud} = Network_{IO} * \frac{BW_{local}}{BW_{total}} \tag{4.4}$$

$$MonetaryCost_{local} = E_{total} * C_{KW} + BW_{cloud} * C_{GB} \tag{4.5}$$

$$MonetaryCost_{cloud} = E_{wireless} * C_{KW} + BW_{local} * C_{GB} + \sum_{r \in CRR} S_r * C_r \tag{4.6}$$

Where :

BW	= Bandwidth expended, in GigaBytes
C	= Monetary cost for a given resource.
E	= Energy expended, in kilowatts.
Ec	= Hourly energy consumption.
S	= Usage share of a resource, in percentage.
CRR	= Cloud Hardware Resources
RHR	= Robot Hardware Resources
RWI	= Robot Wireless Interfaces

4.3 Developing Skills

With bridgeOS, Skills are collections of reusable components, that can be launched together as a single service. For our middleware, however, components contain two sets of configurations detailing how their containers should be run in depending on their deployment location. To develop our augmented bridgeOS Skills, we bundled their components with our Skill modules, and encapsulate them through a Skill configuration file, whose format is presented in Table 4.1. Additionally, we provide Skills with access to dedicated Docker volumes, that can be used to persist and share data over multiple instantiations.

Since the Skill modules remain in the robot, we chose to launch them together as a Docker service. To that end we define their Docker configuration in a format readable by their official tool for services deployment, Docker Compose. This allows their containers to communicate with runtime injections. The configuration files of Skill modules contain basic information regarding middleware communications, including which ports to use and connect to. For the Skill Manager, it also contains the parameters used for decision making, as detailed previously. While for the Skill Router, it includes the mapping and routing configuration required by its components. Currently, the routing functionalities available for components include support for publishing and subscribing to ROS topics, ROS services and HTTP web services. We believe, those 3 aspects, coupled with the shared docker volumes and the Skill Manager API, which provides ip addresses of containers, are enough to accommodate the vast majority of components.

Regarding the routing of ROS related communications, since ROS nodes can use tailored packages and messages, we to keep scouring components for new formats and giving them to the Skill Router. Our approach is to maintain a single Docker Image for the Skill Router, to which new formats are added. The alternative was to maintain one image version per user or Skill, however this lead to increased complexity, and given the fact the size of formats is quite small, unnecessary. This question remains nonetheless open, since it will depend on the scale bridgeOS is employed, and in the long term, a compromised between both approaches seems like the best choice.

Finally, to integrate, through our middleware, Skills with the bridgeOS platform, during the deployment of components, the Master and Robot Controllers respectively inject into containers, hosts and other environment variables mapping the bridgeOS services and providing access to the their ROS master.

Table 4.1: Attributes of a Skill configuration file

Property	Data Type	Description
skill	String	Skill name
skill_id	String	Skill unique identifier
volumes	Array	List of Docker volumes required by the Skill
compose	Object	Docker Compose configuration for deploying the Manager and Router components of a Skill
manager	Object	Skill Manager configuration
router	Object	Skill Router configuration
components	Object	List with all configurations of the components composing this Skill

Summary

We unveiled during this chapter the relevant implementation options followed and depicted the internal mechanics of our middleware, by subsequently disclosing how the modules developed interact and fuse with the current bridgeOS framework.

5 Evaluation

5.1 Overview

In this Chapter, we introduce the results obtained through two series of experiments designed to assess and evaluate the validity of the middleware presented in this thesis. First, to primarily test the potential of our offloading capabilities, we implemented 3 use cases commonly used by robots. Second, to benchmark the different modules developed and characterize the soundness of such middleware as an extension to bridgeOS and more generally for integrating cloud computing with robotics.

Testing was performed in a simulated yet representative test environment, enclosed by a VPN, consisting of a bridgeOS Cloud Infrastructure, divided into a bridgeOS Platform and an instantiation of the Master Controller, with resources to launch cloud components. And a robot, hosting our robot-side middleware modules, and an instantiation of the Robot Controller.

5.2 Experimental Testbed

To assess the developed middleware, we had at our disposal 1 laptop with an Intel Core i7-3610QM CPU at 2.30GHz, 8151MB of available RAM memory, and HDD 7200 RPM SATA 3Gb/s 16 MB Cache, connected by a 220 Mb LAN. Two servers, provided by INESC-ID and IST in Lisbon, with an Intel Core i7-2600K CPU at 3.40GHz, 11926MB of available RAM memory, and HDD 7200RPM SATA 6Gb/s 32MB cache, connected by a 1 Gb LAN. And, from Amazon Web Services, 1 T2.Micro cloud instance, located in Ohio (USA), using 1 Virtual Core of an Intel Xeon E5-2676 v3 @ 2.40GHz, 990MB of RAM and 16GB SSD limited to 160Mb/s.

For both evaluation suites, we implement the network environment depicted in Figure 5.1. The Laptop acted as the bridgeOS Platform, providing the VPN server and our added PostgreSQL database, in addition to the regular bridgeOS services. And, one server provided the cloud part of our middleware. However, for testing the implemented use cases we used one T2.Micro cloud instance for simulating the robot, while for benchmarking purposes we employed the second server. We selected this setup as to better capture the different aspects being tested in each suite.

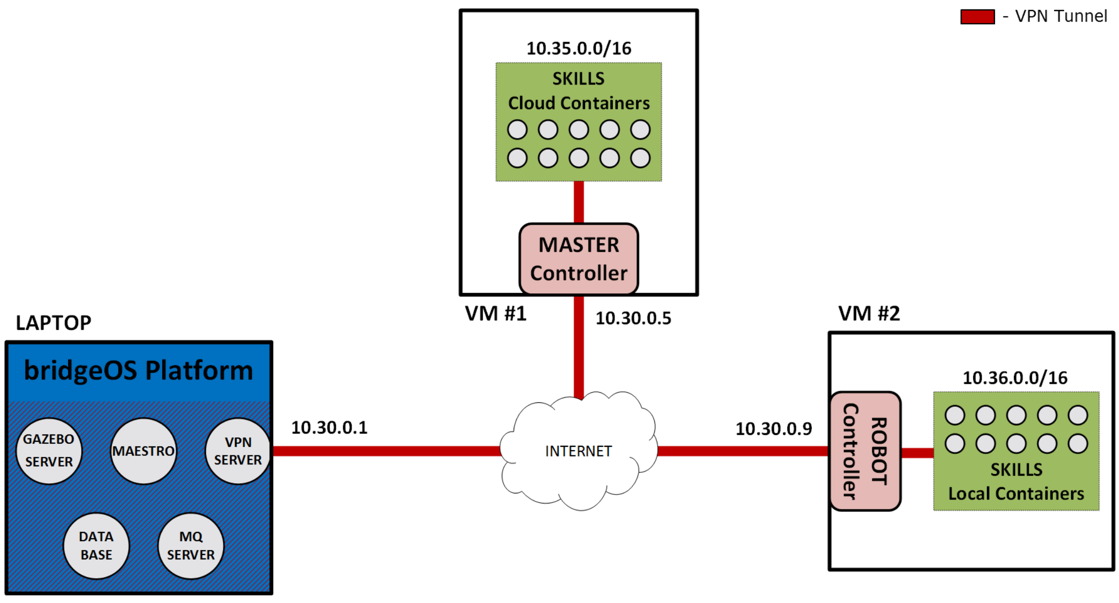


Figure 5.1: Middleware Evaluation Setup

5.3 Offloading Performance

The goal of this series of tests was to observe the potential of our middleware, in terms of its offloading capabilities, and measure the overhead it may cause. To that end, we selected 3 existing bridgeOS Skills relevant for robotics, and integrated them with our architecture as to create 3 standalone use cases that use ROS, presented in Figure 5.2. For both navigation and mapping Skill, we used an instantiation of a robot called Husky, a 4x4 all-terrain mobile base, whose simulation is provided by ROS official website.

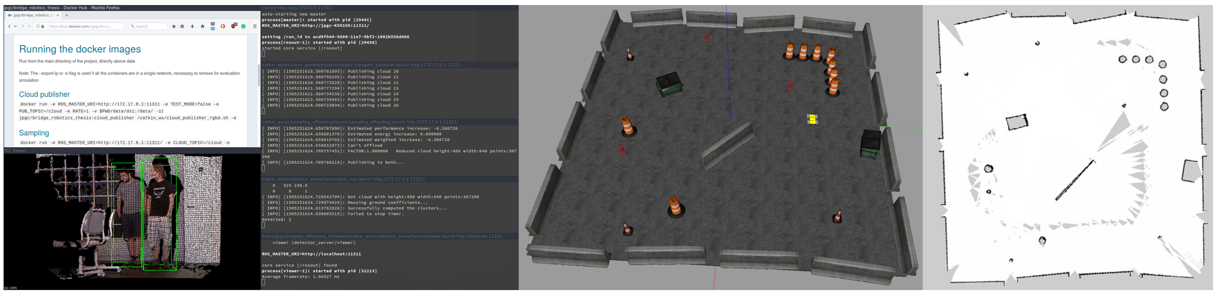
- **People detection:** detects humans using pointclouds from RGB and Depth images;
- **Autonomous Navigation:** navigates robots autonomously using known maps;
- **Autonomous Mapping:** maps the surroundings of a robot moving using odometry.

For each skill, every feasible and relevant combination of its components, location-wise, was tested. This allows for a performance comparison of the middleware with relation to settings ranging from fully-local to pure-cloud (every possible task offloaded into the cloud) execution. Table 5.1, presents the different components composing the Skills, while Figure

5.3, provides the order in which each combination was tested. During testing, we monitored the resource usage of Skills and the middleware from the robot's point-of-view, and measured some performance metrics pertinent to each use case, namely, loss rate of ROS messages, time required to process a single pointcloud, time needed for completing a map tour.

Table 5.1: Skills decomposition

Skill	Component	Description	Location
People Detection	Publisher	Publishes pointclouds generated from RGB and Depth image data.	robot
People Detection	Sampling	Samples points from pointclouds, reducing its number if necessary.	either
People Detection	People Detector	Detects and locates people present within pointclouds.	either
Autonomous Navigation	Path Planner	Generates paths for robots based on maps and sensor data.	cloud
Autonomous Navigation	Map Server	Manages maps and provides path planners and known data.	either
Autonomous Navigation	Robot Mover	Issues desired location goals for robots.	either
Autonomous Mapping	Robot Controller	Publishes commands to control robot movements.	either
Autonomous Mapping	Mapping	Generates maps based on sensor data.	either
Autonomous Mapping	Odometry	Processes and publishes odometry data received from robots.	either



Running the docker images

Run from the main directory of the project, directly above data

Note: The --export-ip or -e flag is used if all the containers are in a single network, necessary to remove for evaluation simulation

Cloud publisher

```
docker run -e ROS_MASTER_URI=http://172.17.0.1:11311 -e TEST_MODE=false -e
PUB_TOPIC=/cloud -e RATE=1 -v $PWD/data/ds1:/data/ -it
jpgc/bridge_robotics_thesis:cloud_publisher /catkin_ws/cloud_publisher_rgbd.sh -e
```

Sampling

```
docker run -e ROS_MASTER_URI=http://172.17.0.1:11311/ -e CLOUD_TOPIC=/cloud -e
```

```
roscore http://jpgc-K501UX:11311/
auto-starting new master
process[master]: started with pid [29441]
ROS_MASTER_URI=http://jpgc-K501UX:11311/

setting /run_id to acd9f944-9800-11e7-9bf2-1002b556d466
process[rosout-1]: started with pid [29458]
started core service [/rosout]
```

```
/catkin_ws/src/cloud_generators/launch/rgbd_transport_publisher.launch http://172.17.0.1:11311
[ INFO] [1505251618.560781895]: Publishing cloud 20
[ INFO] [1505251619.560756295]: Publishing cloud 21
[ INFO] [1505251620.560772920]: Publishing cloud 22
[ INFO] [1505251621.560777294]: Publishing cloud 23
[ INFO] [1505251622.560734536]: Publishing cloud 24
[ INFO] [1505251623.560734563]: Publishing cloud 25
[ INFO] [1505251624.560723034]: Publishing cloud 26
```

```
/catkin_ws/src/sampling_offloading/launch/sampling_offloading.launch http://172.17.0.1:11311/
[ INFO] [1505251624.658787890]: Estimated performance increase: -6.380728
[ INFO] [1505251624.658801376]: Estimated energy increase: 0.000000
[ INFO] [1505251624.658819768]: Estimated weighted increase: -6.380728
[ INFO] [1505251624.658832873]: Can't offload
[ INFO] [1505251624.709757452]: FACTOR:1.000000   Reduced cloud height:480 width:640 points:307
200
[ INFO] [1505251624.709788218]: Publishing to both...
```

```
/catkin_ws/src/detector_server/launch/pub_sub.launch http://172.17.0.1:11311/
    0   525 239.5
    0     0     1
[ INFO] [1505251624.728543799]: Got cloud with height:480 width:640 points:307200
[ INFO] [1505251624.729974424]: Reusing ground coefficients...
[ INFO] [1505251624.813782026]: Successfully computed the clusters...
[ INFO] [1505251624.830003515]: Failed to stop timer.
detected: 2
```

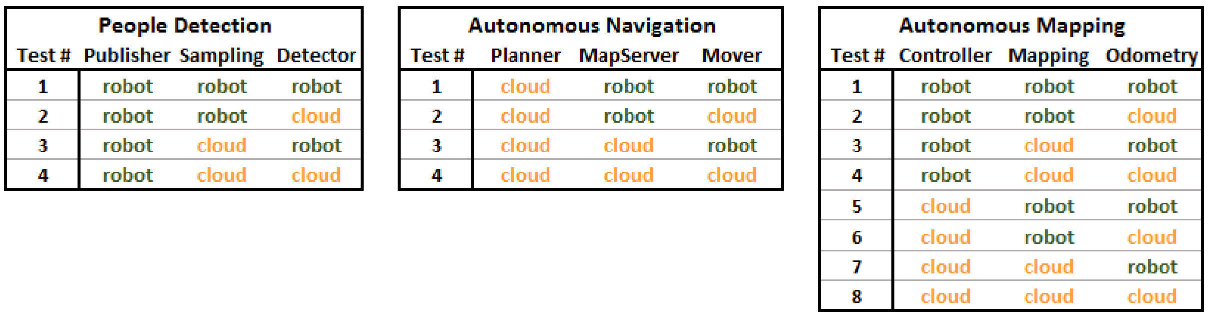
```
/home/jpgc/bridgeos_offloading_framework/catkin_ws/src/detector_server/launch/viewer.launch http://localhost:11311
    viewer (detector_server/viewer)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[viewer-1]: started with pid [31213]
Average framerate: 1.06927 Hz
```

a) People Detection **b) Autonomous Navigation** **c) Autonomous Mapping**

Figure 5.2: Middleware Evaluation Skills

5.3.1 Results

We present in Figure 5.5, the performance measured for each Skill during the tests performed, and in Figure 5.4, their resource usage in the robot. To be noted, while navigation and mapping were simulated in the same environment and have similar durations, their trajectories were different, since the first used a path planner and the latter a predefined route.



People Detection			
Test #	Publisher	Sampling	Detector
1	robot	robot	robot
2	robot	robot	cloud
3	robot	cloud	robot
4	robot	cloud	cloud

Autonomous Navigation			
Test #	Planner	MapServer	Mover
1	cloud	robot	robot
2	cloud	robot	cloud
3	cloud	cloud	robot
4	cloud	cloud	cloud

Autonomous Mapping			
Test #	Controller	Mapping	Odometry
1	robot	robot	robot
2	robot	robot	cloud
3	robot	cloud	robot
4	robot	cloud	cloud
5	cloud	robot	robot
6	cloud	robot	cloud
7	cloud	cloud	robot
8	cloud	cloud	cloud

Figure 5.3: Components location during the performed tests

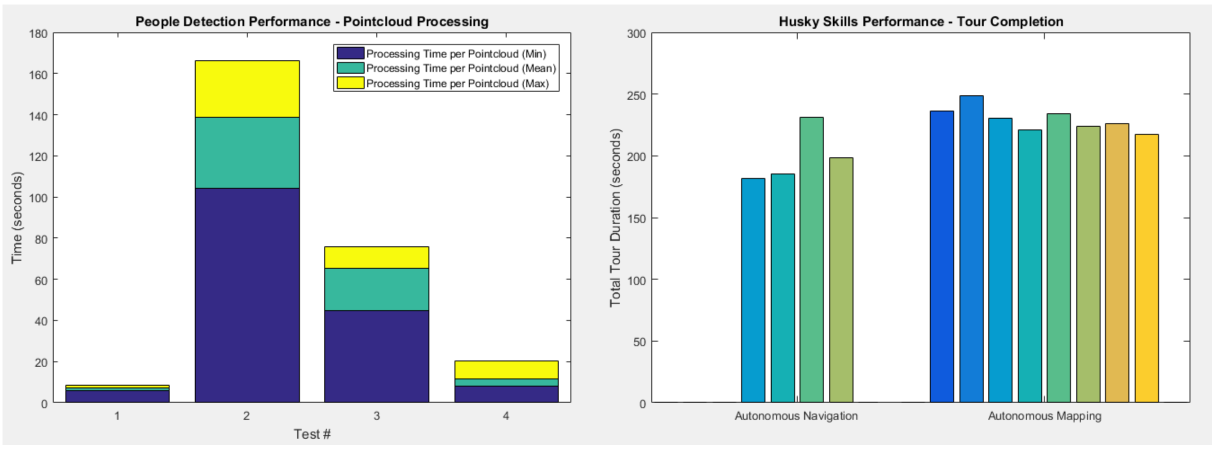


Figure 5.4: Use Cases Results - Skills Performance: each bar represents the combination tested and is displayed in the order defined by Figure 5.3.

Based on resource usage alone, an overall decreasing tendency is clearly apparent, although with some notable exceptions. Case in point being the spike in CPU usage for navigation Skill during Tests 3 and 4, correlating with a performance loss. Of course, a decrease is to be expected from the perspective of the robot, since we are offloading components to the cloud. However, if we examine tests with the most number of cloud components, their results also prove that the overhead caused by the middleware modules of a Skill, Router and Manager, can be insignificant.

With regards to performance, Skill initialization remained **under 5 seconds** during all tests, and surprisingly, the success rate of ROS messages transmission measured by the Skill Router was kept stable at **98.42$\pm$0.63%**. In terms of Skill-wise performance, we note a slight increase in tour duration for navigation, correlated as stated before with a resource spike, and a stable outlook with mapping. A stability in the duration is actually a very good result, since the conditions remain the same (i.e. a robot moving at the same speed over the same path, will take the same time), demonstrating that our routing mechanism does not hinder certain tasks.

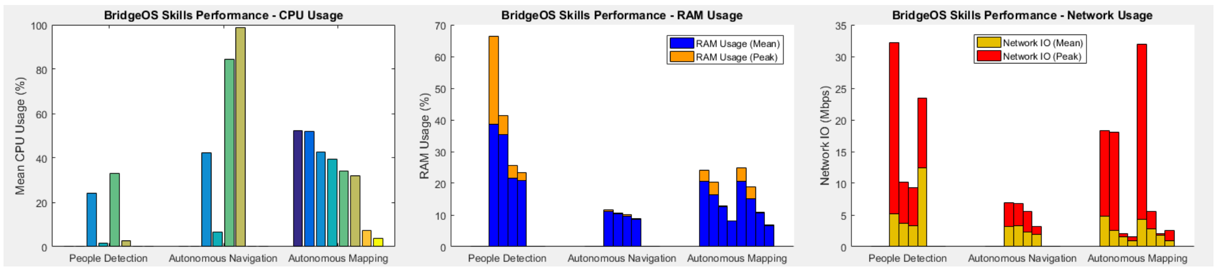


Figure 5.5: Use Cases Results - Skills Resource Usage: each bar represents the combination tested and is displayed in the order defined by Figure 5.3.

For people detection Skill the takeaway is different. With a dramatic increase in the mean processing time of pointclouds, we cannot at first assume the middleware performed well. However, it is easily explained when we consider how the Skill works and the locations of its components. The pointclouds generated by the Publisher had sizes of around 30-40 MBytes, and were forwarded successively to the Sampling and, from it, to the Detector. So, whenever they were in different locations, and additional round-trip with significant data transfer was required. And, since the pointclouds were published periodically, the Skill rapidly generated a network bottleneck. We subsequently observe an improvement in its last test, when both those components stayed in the same location.

Table 5.2: Skill Offloading Comparison

Skill	Best	Performance	CPU	RAM	Network IO	Power
People Detection	Test 4	+64,76%	-88,32%	-45,85%	+137,21%	-63,10%
Navigation	Test 2	+2,11%	-83,99%	-5,30%	+2,98%	-72,33%
Mapping	Test 8	-8,18%	-92,78%	-67,87%	-80,85%	-87,93%

Lastly, with the results obtained, we were able to determine which, if any, combination of components performed best with relation to the native execution (i.e. Test 1 of each Skill, the combination with most robot components.). Based on Table 5.2, for navigation and mapping Skills, the conclusion is succinct, offloading components to the cloud is very advantageous. This is a clear task that motivated our work. For people detection Skill, the verdict is more ambiguous, as we have a trade-off to consider between decreased on-board resource usage and decreased performance and network availability. With a live robot, it would depend on the priority given for such functionality and its degree of importance, for example if it was for live remote viewing or to be used with another local application for people recognition.

5.4 Middleware Benchmarking

The intended objective of this series of experiments was to test the overall resiliency, robustness, reliability and scalability of our middleware, and determine the overhead consumed by its modules. To achieve such analysis, the middleware modules were stress tested through a series of tests and benchmarks, where the parameters were amplified until their failure or impossibility to continue.

Since each module has a different role within our middleware, we defined numerous experiments designed to assess different aspects of their core functions. First, to benchmark our middleware, we gradually increased and measured the number of both, startup and concurrent, Skills the Robot Controller was capable of handling. Startup Skills are also concurrent, but are launched in parallel at the robot's startup, using the Robot Controller's internal pooling mechanisms, whereas for concurrent Skills, we launched them sequentially. Then, we concentrated on the Skill modules, repeating the same concurrency experiment, but this time with components. The purpose is also to analyze the overhead caused by the neural networks employed by the Skill Manager. Finally, given the prime importance of ROS for robotics and bridgeOS, we focused on the Skill Router to test its ROS routing capabilities. Assessing first its ability of handling topics, in absolute terms, and then of routing messages, by experimenting with both the number of publishers and subscribers, and their message publishing rate.

During those tests, we monitored the usage made by the modules of robot resources, and measured temporal statistics about their initializations. To obtain a more accurate and precise representation of their overhead, we also created a mockup Skill composed of a dummy component, which performs a repetitive and meaningless task. Except for concurrent components benchmarking, Skills were composed of a single dummy component launched in the cloud.

5.4.1 Results

The results for Skills benchmarking are provided in Figures 5.6 and 5.7, while Figure 5.8 portrays the results for components. Figures 5.9 and 5.10, concern the experiments made specifically for the Skill Router. Before analyzing results, we would like to precise that for benchmarking tests, each absolute amount of units plotted in the figures (e.g. Skills, components, topics), represent individual tests, each starting from 0, and not continued iterations of a unique test.

Skills Benchmarking

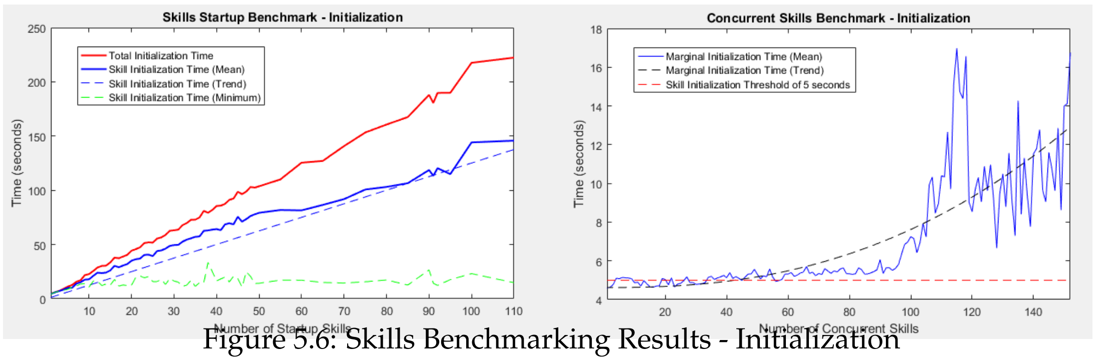


Figure 5.6: Skills Benchmarking Results - Initialization

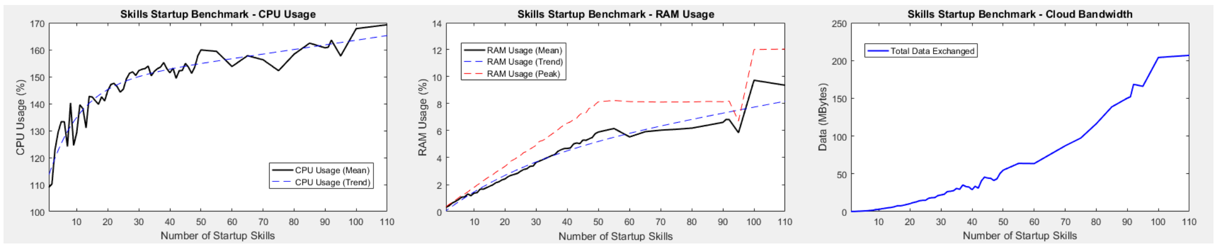


Figure 5.7: Skills Benchmarking Results - Resource Usage

Overall, our middleware was able to surpass the symbolic bar of 100 concurrent Skills. Specifically, it managed to complete **startups of 110 Skills** without failures, and even cross the line of **150 concurrent Skills** when launched sequentially. However, shortly above the 100 Skills threshold, we began noticing sporadic Skill failures, leading to their shutdown. In reality, those failures can be explained by two factors, on-board resources and monitoring. By aggregating the resource usage made by all Skills and the Robot Controller, we realized the results were actually limited by the resources of the robot. Further motivating the need to migrate to the cloud, even if partially. Secondly, in order to closely simulate a live setup, we left active their monitoring functionalities, which generated periodic increases in resource usage, explaining why failures began to happen sooner. This factor also accounts for the significant increase in data exchanged with the cloud, as bandwidth grows proportionally to the number of containers monitored.

Analyzing the temporal costs obtained during the benchmarking of Skills, we observed that up to 100 concurrent Skills, their initialization remained stable at **5.26±0.61 seconds** each, afterwards it deteriorates rapidly due to the scarcity of resources. An average similar to the time registered during use case testing. Meanwhile, parallel launches of Skills can slash such average to merely **2.19±0.33 seconds**.

Components Benchmarking

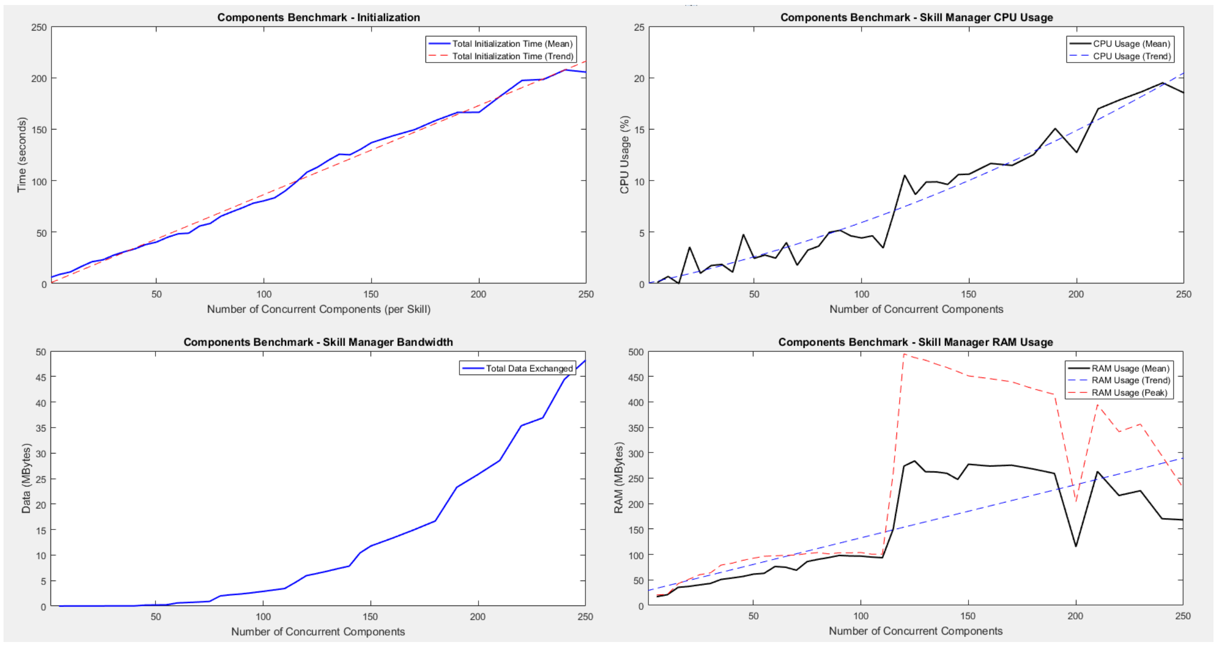


Figure 5.8: Components Benchmarking Results

Regarding components, our middleware was able to guarantee deployment of skills with up to **250 robot components**. More than that led to failures of our Skill Manager, and unlike with Skills benchmarking, the cause was not a lack of resources. Instead, upon reaching such volumes, Skill Managers began experiencing some failures, network-wise and internally. Some experiments managed to exceed 300 components, though they were unreliable and not always reproducible. We also want to state that these limits only regard components that can be offloaded. Components with fixed locations are expected to be a minority, and would portray less accurate results since they do not require neural networks.

Nonetheless, the results indicate that even when instantiating skills of 250 components, the mean duration required for initializing each component remains stable at **0.89±0.16 seconds**. This measure includes both the container launch and the neural network initialization, besides the cost resulting from network exchanges between modules, and is, in our opinion, quite positive and significant to the middleware adoption. In terms of resource usage, Skills Managers follow a linear trend, an expected exception being bandwidth due to component monitoring, we discovered that they have a memory baseline of **19.97±1.3 MBytes** of RAM. Lastly, the spikes in RAM usage correlate with the processing of monitoring metrics, whose periodicity can coincide with the initialization and skew results.

Routing Benchmarking

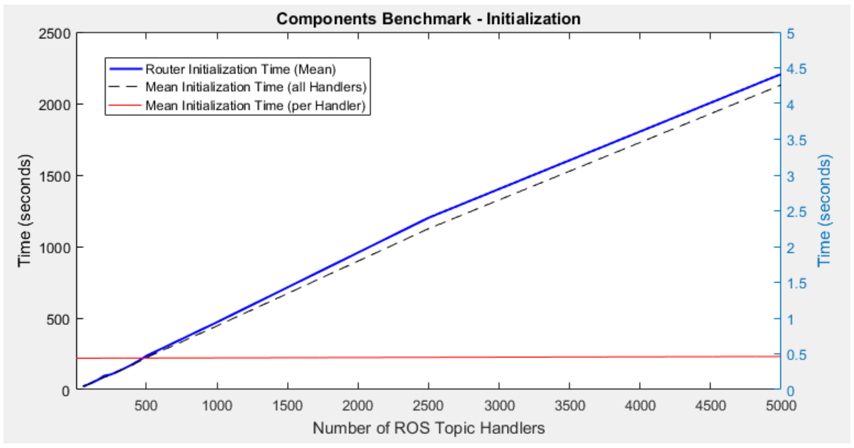


Figure 5.9: Skill Router Benchmarking Results - Initialization

In terms of the mapping capacity of ROS topics by Skill Routers, the results are also quite good. We were able to map up to **5000 topics** with a single Skill Router. An artificial limit, since we considered the total initialization time required and concluded that there was no interested in further testing such feature. The main takeaway of Figure 5.9 is that the mapping a ROS topic lasts in average **443±20 ms**, and remains stable under 0.5 seconds even with 5000 topics. A downside is that Skill Routers required an increasingly more time to initialize, although it only hinders their capacity to route all components, since they are still able to communicate with the remaining modules and thus, operate partially. We also discovered that each Skill Router has a memory baseline of **46.47±1.3 MBytes** of RAM, while each Topic Handler only consumed an additional **10 KBytes** of RAM. We pinpointed the cause for this relatively high RAM baseline to the rospy library, and found that a simple python library import originated in an excess of 35MBytes of RAM. Although indispensable for now, creating a lighter library is certainly a question to consider in the future.

Regarding our routing capabilities, a Skill Router was able to easily handle **100 concurrent pairs of ROS publishers and subscribers**, instantiated in the cloud. Once again, this boundary does not correspond to an actual limitation of our middleware, but rather of the available cloud resources. Given this upper bound of publishers, we decide to subsequently increase their message publishing rate, going from their baseline of 1Hz and up to 500Hz, reaching a total throughput of **50 000 messages each second**. Each iteration was sustained for a period of 10 seconds, counting from the moment when all pairs were instantiated. An attempt was made at 1kHz, and although the cloud server crashed midway, the Skill Router was able to keep up.

Those are surprisingly good results, considering ROS nodes with high framerates (i.e. cameras, transformations, statistics) almost never surpass 100Hz. Of course, in those cases the message payload is usually much larger. Resource-wise, the Skill Router also performs positively, as its usage trends remain linear with relation to publishing rates and number of publishers, although with slightly more pronounced slopes.

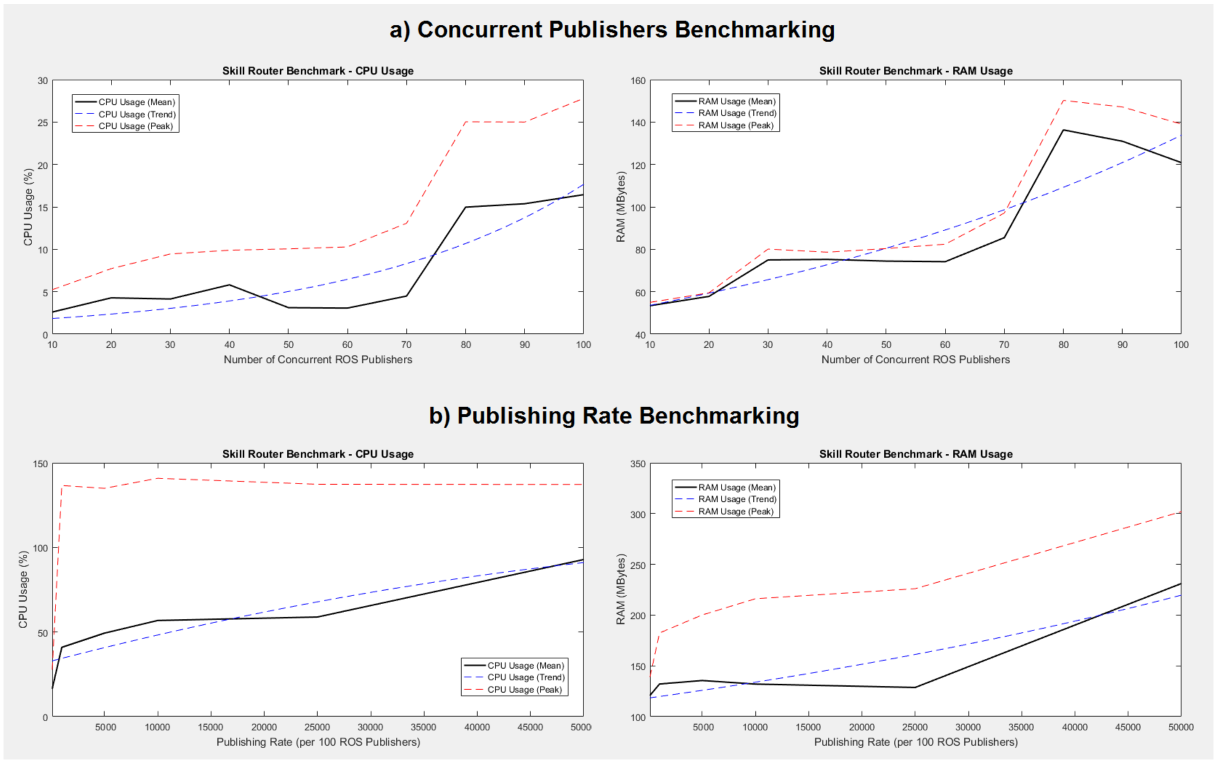


Figure 5.10: Skill Router Benchmarking Results - Resource Usage

Summary

In this chapter, we described the assessment of the developed middleware and its different modules, regarding its properties, performance, and resource overhead. This evaluation was performed from two different optics, one designed to quantify the potential this middleware has, an another to evaluate its overall capacity and reliability in guaranteeing such potential.

Globally, the results obtained consolidated our position stating that cloud computing and dynamic computation offloading can be successfully adapted for robotics, and demonstrated that the proposed solution can be integrated flawlessly with existing technology while being able to respond beyond what can be expected from real life applications, especially in terms of scalability. As, very rarely would we find realistic scenarios where single robots would require hundreds of Skill or components.

6 Conclusion

6.1 Concluding Remarks

To summarize the thesis briefly, Chapter 1 introduced the research proposal and main ideas behind the work presented, with Chapter 2 delving into the core technologies and concepts that would help achieve its fruition. Later, Chapters 3 and 4 divulged the architecture of the proposed solution and thoroughly presented how its main components were designed and implemented. Then, Chapter 5 provided a comprehensive evaluation of the middleware, disclosing its performance results. Finally, this Chapter concludes the thesis with some succinct points regarding the work accomplished and the goals fulfilled, closing with general recommendations of possible future work.

We began with a portrayal of the current state of cloud robotics, presenting the opportunities and shortcomings it manifested, and how the proposed solution could contribute to it. Then, we examined the most prominent frameworks available for networked robotics and current solutions for integrating robotics with cloud computing, while outlining the benefits of each approach. We also identified state of the art technologies and paradigms that would assist us in developing more sophisticated mechanisms for incorporating transparent distribution and adaptive offloading with robotics.

Then, ensued with a characterization of the extensions we propose for bridgeOS, helping it fuse with the cloud and complement its offering, in order to take advantage of the opportunities currently available for cloud robotics. And, presented the algorithms and protocols we designed to implement adaptive offloading and enable robots to optimize on-board resources and extend their functionalities.

Overall, the middleware, represented by the different modules we conceived, is able to achieve most requirements we sought for this thesis. It consolidates mechanisms for fault-tolerance, data retransmission and resynchronization, disruption resiliency and cloud replication. The assessment provided afterwards, demonstrated its ability to scale and operate in geographically distributed environments facing real-time constraints, displaying benefits that outweigh its overhead.

Given the broad scope and diversity inherent to robots and their requirements, characteristics and capacities, we enforced a principle of customization into our modules, enabling versatile configurations that adapt to specific needs, without requiring technical modifications to their code.

To conclude, we believe that our bridgeOS middleware, will be beneficial for cloud robotics and useful for extending robot functionalities, permitting newer applications and others to finally become viable.

6.2 Future Work

During the completion of this thesis, we identified some next steps, worthy of study, that could be undertaken as a continuation of the work we initiated via this middleware.

Due to the architecture of ROS, a loss of connection with the ROS master, will most likely force improperly designed nodes to shutdown. So even though our takeover mechanism is able to restore Skills, some local components will still have to be restarted. A solution would be to replicate the ROS master across both sides and have the Robot Controller further cement its role as a network bridge between both robots and cloud networks, mirroring topics that are needed on both sides and effectively acting as a proxy. This brings an additional level of complexity and redundancy, with benefits and disadvantages, that certainly need to be analyzed. For instance, there would be no disruption of ROS-based functionalities during loss of connectivity, components interacting only within the cloud boundary would avoid going through the Skill Router first. Thus, increasing the black box aspect and blind integration of existing functionalities as Skills. On the other hand, it could degrade performance due to redundant connections and mapping.

Regarding the offloading mechanisms, currently they are are applied on a *per component* basis, however a Skill-level decision module could instead be implemented. It would require much larger quantities of real data, which is the argument behind our current selected. There are also some challenges in constructing a generic, scalable and sophisticated architecture, that remains versatile enough. In our opinion, a single decision module per robot would be to broad and lose the ability to adapt to specific needs of Skills. Therefore, the Robot Controller is better let off solely with the resource allocation aspect.

Skills with resource-intensive cloud components can also benefit from additional features such as container replication and load-balancing. They could be achieved by using a new cloud module, either centralized or per-Skill basis, for proxying requests. Currently, it is already possible to implement such features in a Skill, albeit in a arduous manner. Since the Skill developer would need to create a dedicated component for load-balancing and then manually duplicate the configuration of the cloud component requiring replication. Therefore, providing them internally would simplify and generalize this process.

A final welcoming idea, is the creation of Developer Tools to help generate bridgeOS Skills and Docker images for components. Though, components only using a specific technology, such as elementary ROS nodes, NodeJS or Python applications, and so on, can be fused directly into the official Docker Images provided for those technologies and programming languages, without the need for further customization. Back-end Tools for bridgeOS, enabling automatic generation of Skill configurations based on container analysis and ROS packages retrieval would also be interesting additions.

Bibliography

Arumugam, R. et al. (2010). Davinci: A cloud computing framework for service robots. In *International Conference on Robotics and Automation*, pp. 3084–3089. IEEE.

Chibani, A. et al. (2013). Ubiquitous robotics: Recent challenges and future trends. *Robotics and Autonomous Systems 61*(11), 1162–1172.

Estrada, R. & I. Ruiz (2016). The manager: Apache mesos. In *Big Data SMACK*, pp. 131–164. Springer.

Feilner, M. (2006). *OpenVPN: Building and integrating vpns*. Packt Publishing Ltd.

Fernando, N., S. W. Loke, & W. Rahayu (2013). Mobile cloud computing: A survey. *Future Generation Computer Systems 29*(1), 84–106.

Furrer, J. et al. (2012). Unr-pf: An open-source platform for cloud networked robotic services. In *SICE International Symposium on System Integration*, pp. 945–950. IEEE.

He, K. et al. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR abs/1502.01852*.

Hindman, B. et al. (2011). Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, Volume 11, pp. 22–22.

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

Hunt, P. et al. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, Volume 8, pp. 9.

Inaba, M. (1994). Remote-brained robotics: Interfacing ai with real world behaviors. Volume 6, pp. 335–344. The International Foundation of Robotics Research.

Jaramillo, D., D. V. Nguyen, & R. Smart. Leveraging microservices architecture by using docker technology. In *SoutheastCon 2016*, pp. 1–5. IEEE.

Ji, X. et al. (2016). Data transmission strategies for resource monitoring in cloud computing platforms. *Optik 127*(16), 6726–6734. Elsevier.

Kamei, K., S. Nishio, N. Hagita, & M. Sato (2012). Cloud networked robotics. *IEEE Network 26*(3), 28–34.

Kehoe, B. et al. (2015). A survey of research on cloud robotics and automation. *IEEE Transactions on Automation Science and Engineering 12*(2), 398–409.

Khan, M. A. (2015). A survey of computation offloading strategies for performance improvement of applications running on mobile devices. *JNCA 56*, 28–40. Elsevier.

Kintsakis, A. M. et al. (2015). Robot-assisted cognitive exercise in mild cognitive impairment patients. In *E-Health and Bioengineering Conference (EHB), 2015*, pp. 1–4. IEEE.

Koubaa, A. (2016). *Robot Operating System (ROS): The Complete Reference*. Springer International Publishing.

Kozhirbayev, Z. & R. O. Sinnott (2016). A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems 68*, 175–182. Elsevier.

Kratzke, N. (2014). A lightweight virtualization cluster reference architecture derived from open source paas platforms. *Open J. Mob. Comput. Cloud Comput.*

Kuffner, J. J. (2010). Cloud-enabled humanoid robots. In *IEEE-RAS 10th international conference on humanoid robotics, Nashville, TN.*

Kurzweil, R. (2005). *The Singularity Is Near: When Humans Transcend Biology*. PPG.

LeCun, Y., Y. Bengio, & G. Hinton (2015). Deep learning. *Nature 521*(7553), 436–444.

Lee, J. (2012). Web applications for robots using rosbridge. *Brown University*.

Lei, K. et al. (2014). Performance comparison and evaluation of web development technologies in php, python, and node. js. In *2014 17th CSE*, pp. 661–668. IEEE.

Liu, Q. & X. Sun (2012). Research of web real-time communication based on web socket. *International Journal of Communications, Network and System Sciences 5*(12). SRP.

Mainaly, B. & D. Ningombam (2014). A survey on cloud robotics. *Communication, Cloud and Big Data: Proceedings of CCB 2014.*

Mohanarajah, G. et al. (2015). Rapyuta: A cloud robotics platform. *IEEE Transactions on Automation Science and Engineering 12*(2), 481–493.

Namiot, D. & M. Sneps-Sneppe (2014). On micro-services architecture. *International Journal of Open Information Technologies 2*(9).

Park, S. et al. (2014). Design and evaluation of mobile offloading system for web-centric devices. *Journal of Network and Computer Applications 40*, 105–115.

Pimentel, V. & B. G. Nickerson (2012). Communicating and displaying real-time data with websocket. *IEEE Internet Computing 16*(4), 45–53.

Quintas, J., P. Menezes, & J. Dias (2011). Cloud robotics: Towards context aware robotic networks. In *International Conference on Robotics*, pp. 420–427.

Riazuelo, L. et al. (2015). Roboearth semantic mapping: A cloud enabled knowledge-based approach. *IEEE Transactions on Automation Science and Engineering 12*(2), 432–443.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks 61*, 85–117.

Seo, K.-T. et al. (2014). Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters 66*, 105–111.

Siciliano, B. & O. Khatib (2016). *Springer Handbook of Robotics*, Chapter Networked Robotics, pp. 1109–1134. Springer International Publishing.

Tauber, J. A. (1996). *Issues in building mobile-aware applications with the Rover toolkit*. Ph. D. thesis, Citeseer.

Tenorth, M. et al. (2012). The roboearth language: Representing and exchanging knowledge about actions, objects, and environments. In *ICRA 2012*, pp. 1284–1289. IEEE.

Tilkov, S. & S. Vinoski (2010). Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing 14*(6), 80.

Zhang, X. et al. (2010). Towards an elastic application model for augmenting computing capabilities of mobile platforms. In *MOBILWARE*, pp. 161–174. Springer.

A Comparison Properties

Table A.1: Properties of container managers

Property	Description
Scalability	Supported scalability and potential, in relative terms and in comparison with all other platforms compared.
Overhead	Resource overhead for managing containers, in relative terms and in comparison with all other platforms compared.
Monitoring	Monitors state of nodes, containers or tasks, and provides APIs for accessing that data.
Redundancy	Implements fault-tolerance and high-availability mechanisms for management nodes.
Service Discovery	Implements service discovery and mapping mechanisms.
Service Replication	Provides service replication mechanisms.
Image Sharing	Provides built-in tools for sharing containers images.
General Constraints	Enables restricting execution of containers in relation to higher-level concepts (e.g. datacenter, geographical location, etc. . .)
Node constraints	Supports constraints regarding the worker nodes' operational environments.
Non-standard restrictions	Supports custom restrictions, unrelated to CPU, RAM, storage and network.
Multi-level restrictions	Permits differentiation of restrictions for containers, nodes or other components.
Docker only	Supports Docker containers only.
Load-balancing	Internal load-balancing service provided.
Single system	If all machines and resources are aggregated as a single virtual system.
Multi-cluster	Supports multiple independent clusters.
Secure communications	Protects internal communications.

Table A.2: Properties of Cloud Robotics frameworks

Property	Description
Scope	Scope of applicability and services targeted.
Scalability	Supported scalability and potential, in relative terms and in comparison with all other frameworks compared.
Redundancy	Implements fault-tolerance and high-availability mechanisms locally for whenever the cloud becomes unavailable.
Offloading	Type of offloading techniques implemented, if any.
Modular	Follows a modular approach for developed and operating services and applications.
QoS monitoring	Monitors the Quality of Service of services and applications provided.
ROS compatible	Implements or supports ROS nodes and messages.
Ontology	Type of ontology integrated.
Shared skills	Allows sharing services and related knowledge between robots.
Skill Templates	Offers templates for generating Skills based on specific robot types, hardware characteristics and other capabilities requirements.
On-demand	Performs on-demand deployment of cloud services or resources.
Dynamic apps	Supports deployment of new functionalities into robots in real-time.
App store	Offers a marketplace for retrieving applications or services for robots.
User apps	Offers applications intended for end-users.
Uploadable apps	Lets users upload private applications through the framework.
User API	Provides APIs for user access and control.
Web dashboard	Provides web-based user interfaces to enable direct interactions.
Private data	Protects proprietary or personal information.
Built-in security	Implements security mechanisms for authenticating users or robots and securing communications.