# An Adaptive Robotics Middleware for a cloud-based bridgeOS

**Rafael Afonso Rodrigues**
Instituto Superior Tecnico
Lisbon, Portugal
rafael.afonso.rodrigues@tecnico.ulisboa.pt

## 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.

### Author Keywords

robotics middleware; cloud robotics; dynamic offloading; bridgeOS; docker containers; ROS; real-time monitoring;

## INTRODUCTION

Cloud Robotics is a fairly recent concept and field in robotics. It was first coined in November 2010 by James Kuffner [10], who presented through it a novel approach, inspired by the DAvinCI framework [1], 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 ([16] and [7]). The idea of creating separate databases, is not new and originated from the Networked Robotics field in the 90s [5], derived from advances made in telerobotics and telepresence systems. Back then, researchers 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 [19].

Since then, technology has progressed significantly, and is bound to continue at an exponential rate [11], 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 [12], 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 ranging from character and speech recognition, to language translation or object classification [17], and in some case even surpass human-level performance ([13] and [3]). Meanwhile, new frameworks, predominantly the Robot Operating System, 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,

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].

### 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. As addressed 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.

### Shortcomings of Current Solutions

The shortcomings of current solutions are addressed by many surveys [14] [4] [2]. Although not uniformly lacking from each existing solution, it includes:

- Non-generic or narrow scope of services;
- Use of proprietary or non web-oriented communication protocols, and custom data formats or drivers, instead of supporting ROS;
- Static or limited techniques for offloading computation, and no guarantees of service quality nor monitoring capabilities;
- Lack of data resynchronization mechanisms for handling network failures;
- Lack of security, privacy and anti-tampering mechanisms for network connections.

### Proposed Solution and Contributions

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 sets up procedures to mediate disruptions and failures. And includes mechanisms to data resynchronization and Firewall-friendly communication protocols to ease its deployment within any network and avoid possible traffic restrictions. It can also provide adaptive computation offloading based on available resources, general or service-dependent performance metrics, and according to the quality of service or optimization required.

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

## RELATED WORK

In this section, we address the research work and relevant systems related to our work, its domain and goals. Presenting successively, the core technologies integrated by our middleware.

### Architectures for Networked Robotics

There are many frameworks for interconnecting robots, be it for creating small peer-to-peer networks or establishing links over distributed infrastructures. We start by covering the renowned architecture for networking robots, ROS, and proceed to the cloud robotics framework extended by this work, bridgeOS.

#### *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, permitting rapid creation and deployment of modular applications [8].

The ROS network is composed of 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.

#### *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 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.

### Containers

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. This paradigm presents multiple advantages when compared to direct virtualization and virtual machines (VMs) [9]. 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 [18].

#### *Docker*

Docker is a recent container framework with high portability, supporting many platforms, including UNIX and Windows. The idea is to promote a "single service/application per container" model, synergistic with the microservices paradigm [15]. 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 [6]. 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.

#### *Docker Swarm*

Container Managers facilitate the orchestration and scheduling of highly scalable environments with large clusters of containers, through sets of management and supervision tools. Docker Swarm is the official manager and is nowadays included inside the Docker Engine. Each Swarm consists of a cluster of nodes, running the *Engine* in *swarm mode*, that can operate in either predefined roles, *manager* and *worker*. 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 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.

## ARCHITECTURE OF THE PROPOSED SOLUTION

The system-wide architecture for the cloud-based bridgeOS implementation, augmented by our proposed middleware, is depicted in Figure 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.

To deal with monitoring and middleware management, we developed a tailored event-based communications protocol to be used over WebSockets. As for other design choices, each middleware module, contains a web interface implementing said protocol, to permit bi-directional interactions.

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 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

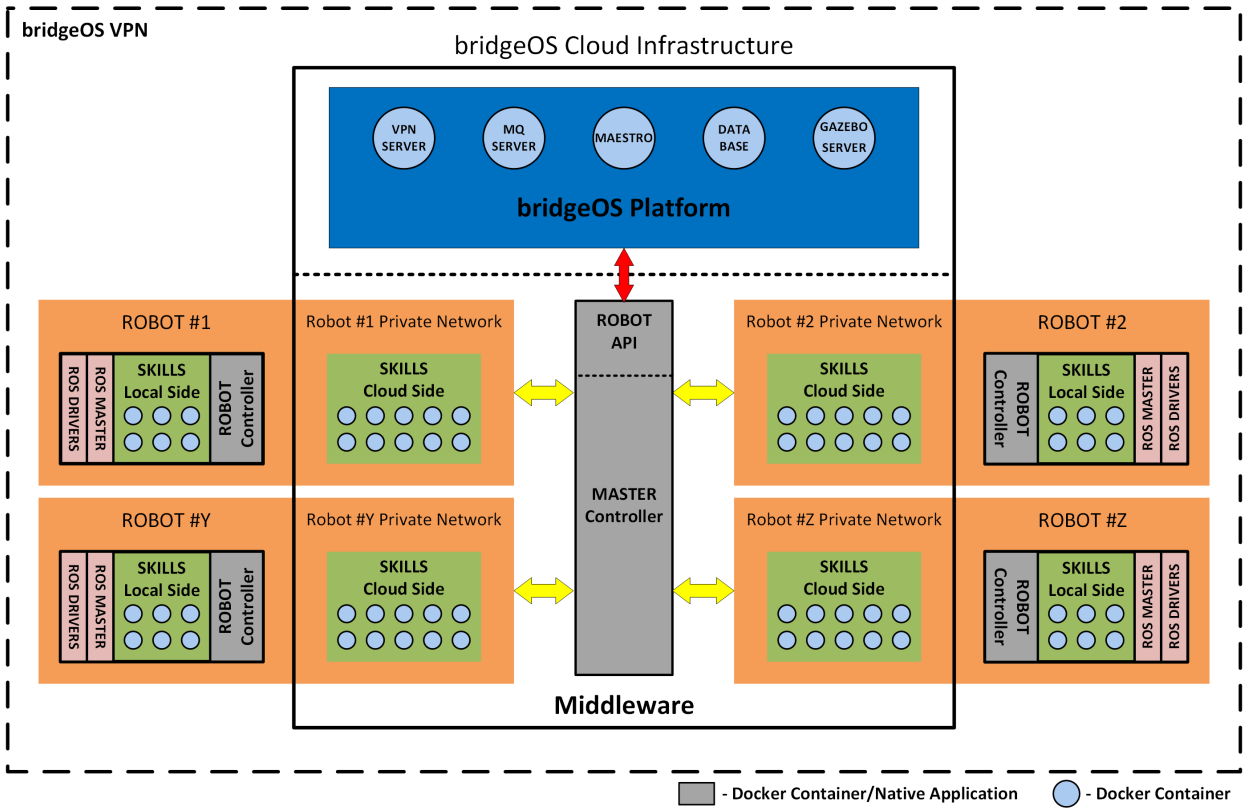


**Figure 1. Overview of the extended bridgeOS architecture**

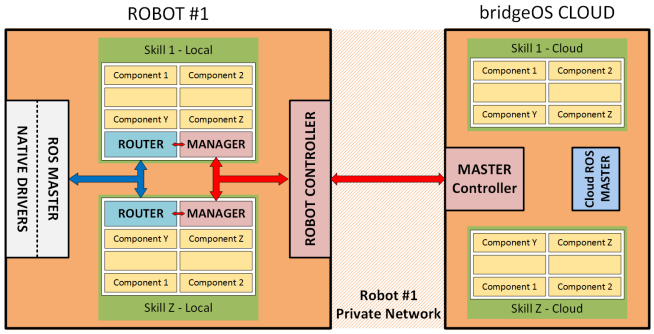


**Figure 2. Detailed architecture from a robot point-of-view. Red arrows represent communication channels using WebSockets, dedicated to monitoring and management purposes, while blue arrows represent communication channels for ROS. Not depicted, are all connections involving components, which can be over ROS or use any other type of protocol.**

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.

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. 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, avoiding lengthy pre-configurations.

### 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 created specifically for this middleware, although the database can be accessed by the bridgeOS platform through the API.

#### 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 establish 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 our common API, 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.

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 we developed a Resource Allocation algorithm for generating decision based on current resource availability and usage, heuristic functions and cost thresholds.

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, 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.

#### 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.

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.

#### Network 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 publishing a status update) and request-response (e.g. requesting the current location of a component) exchanges logic.

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.

## EVALUATION

In this section 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 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.

#### Simulated Setup

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 3. In which the Laptop acted as the bridgeOS Platform, providing the VPN server and our added PostgreSQL database, in addition to the regular bridgeOS services. And, 1 server provided the cloud part of our middleware. However, for testing the implemented use cases we used 1 T2.Micro cloud instance for simulating the robot, while for benchmarking purposes we employed the second server. We selected this setup as to better reflect the different aspects being tested in each suite.

#### 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. For both navigation and mapping Skills, we used a virtual instantiation of a robot called Husky, a 4x4 all-terrain mobile base, whose simulation is provided by ROS official website.

For each skill, every feasible and relevant combination of its components, location-wise, was be tested. This allows for a

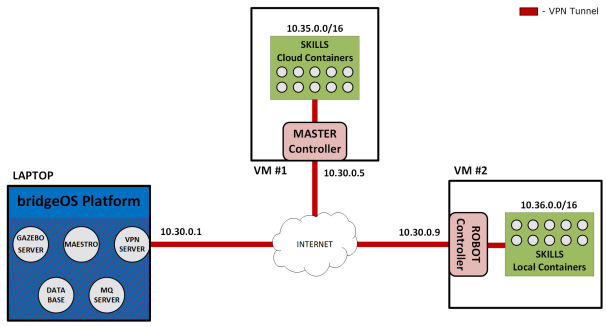


**Figure 3. Middleware Evaluation Setup**

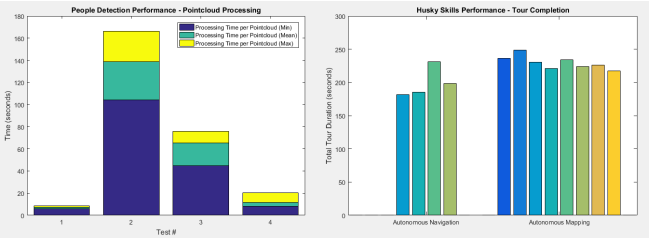


**Figure 4. Use Cases Results - Skills Resource Usage**

performance comparison of the middleware with relation to settings ranging from native to pure-cloud (every possible task offloaded into the cloud) execution. 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.

*Results*

We present in Figure 5, the performance measured for each Skill during the tests performed, and in Figure 4, their resource usage in the robot. To be noted, while navigation and mapping were simulated in the same environment, their trajectory were different, since the first used a path planner with scarce points and the latter a predefined route, therefore their durations albeit similar, are unrelated.

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 3, 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 persisted **under 5 seconds** during all tests, and surprisingly, the success rate of ROS messages transmission measured by the Skill Router remaining stable at **98.42±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.

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 was required. And, since the pointclouds were published periodically, the Skill rapidly generated a network bottleneck. Which is why, an improvement was observed in its last test.

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 components executed locally.). Based on Table 1, for navigation and mapping Skills, the conclusion is succinct, offloading components to the cloud is very advantageous. 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.

### 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,

**Table 1. Skill Offloading Comparison**

| Skill | Performance | CPU | RAM | Network IO | Power | Cloud Instances |
|---|---|---|---|---|---|---|
| **People Detection** | +64,76% | -88,32% | -45,85% | +137,21% | -63,10% | 2 containers |
| **Navigation** | +2,11% | -83,99% | -5,30% | +2,98% | -72,33% | 2 containers |
| **Mapping** | -8,18% | -92,78% | -67,87% | -80,85% | -87,93% | 3 containers |

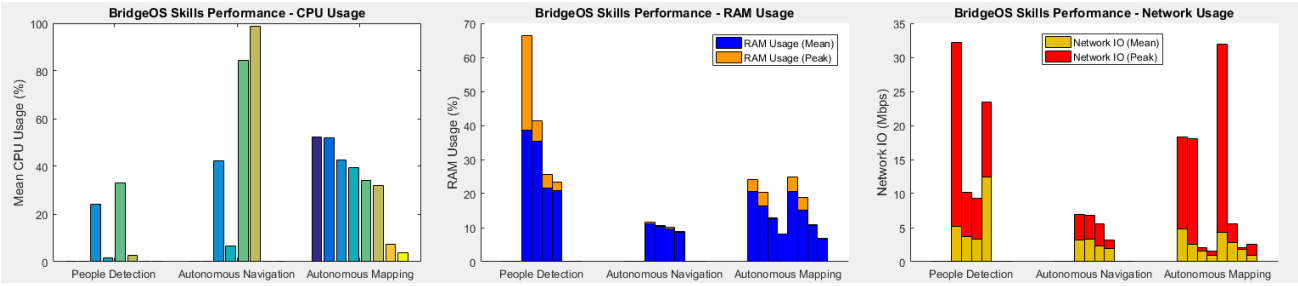


**Figure 5. Use Cases Results - Skills Performance**

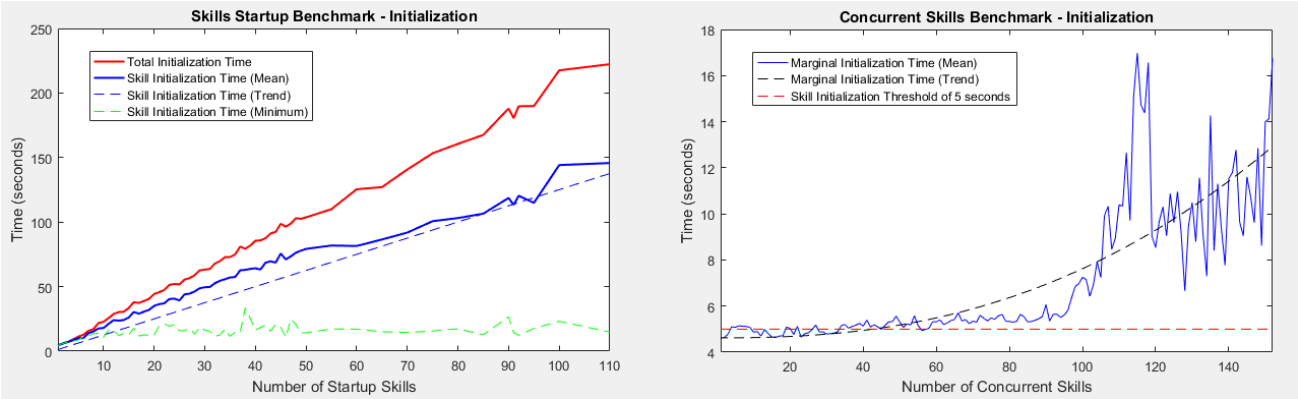


**Figure 6. Skills Benchmarking Results - Initialization**

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 bridgOS, we focused on the Skill Router to test its ROS routing capabilities. Assessing first its capacity 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 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.

### *Results*

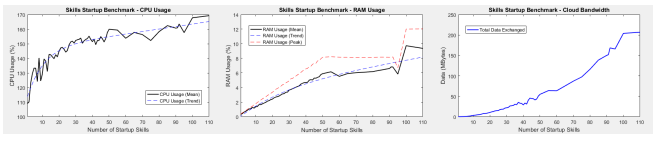

**Figure 7. Skills Benchmarking Results - Resource Usage**

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 usage made by all Skills and the Robot Controller, we realized the results were actually limited by the resources of the robot. 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 exponential increase in data exchanged with the cloud, as bandwidth grows proportionally to the number of containers monitored.

Analyzing the temporal costs of Figure 6, 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 one measured previously about the implemented use cases. Meanwhile, parallel launches of Skills can slash such average to merely **2.19±0.33 seconds**.

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. Since, upon reaching such volumes, Skill Managers began experiencing some failures, network-wise and internally. Some experiments managed to exceed 300 components, though they

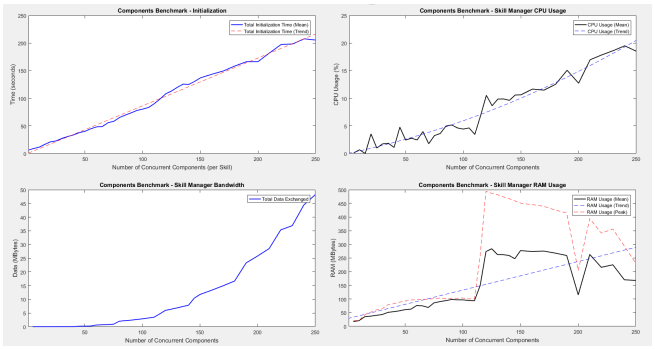


**Figure 8. Components Benchmarking Results**

were unreliable and not always reproducible. We also want to state that, these limits 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. 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.

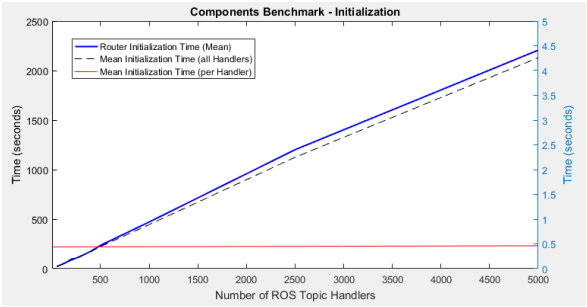


**Figure 9. Skill Router Benchmarking Results - Initialization**

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 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

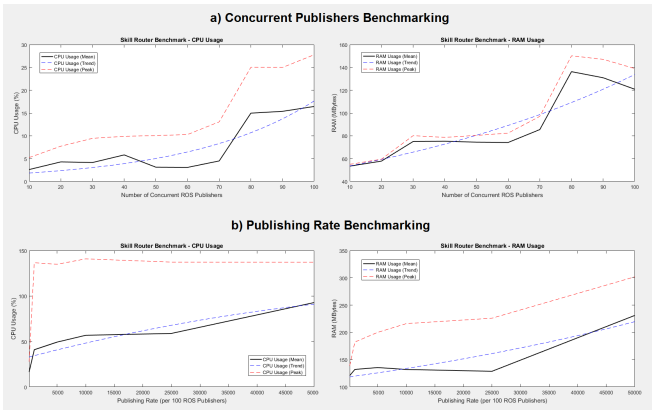


**Figure 10. Skill Router Benchmarking Results - Resource Usage**

also discovered that each Skill Router has a memory baseline of **46.47±1.3 MBytes** of RAM, and that Topic Handler consumes an additional **10 KBytes** of RAM.

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 instead increase their message publishing rate, successively increasing it, 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 publishers and subscribers were instantiated. An attempt was made at 1kHz, and although the cloud server crashed after a few seconds, the Skill Router was able to keep up during that time.

Those are surprisingly good results, considering ROS nodes with high framerates (i.e. cameras, transformations, statistics) almost never surpass 100Hz. Of course, usually in those cases the message payload is also 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.

## CONCLUSION

To conclude, 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.

### Future Work

During the completion of this work, 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.

Offloading mechanisms are currently 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.

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

### REFERENCES


1. Rajesh Arumugam and others. 2010. DAvinCi: A cloud computing framework for service robots. In *International Conference on Robotics and Automation*. IEEE, 3084–3089.
2. Abdelghani Chibani and others. 2013. Ubiquitous robotics: Recent challenges and future trends. *Robotics and Autonomous Systems* 61, 11 (2013), 1162–1172.
3. Kaiming He and others. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *CoRR* abs/1502.01852 (2015). `http://arxiv.org/abs/1502.01852`
4. Guoqiang Hu, Wee Peng Tay, and Yonggang Wen. 2012. Cloud robotics: architecture, challenges and applications. *IEEE Network* 26, 3 (2012), 21–28.
5. Masayuki Inaba. 1994. Remote-brained robotics: Interfacing ai with real world behaviors. Vol. 6. The International Foundation of Robotics Research, 335–344.
6. David Jaramillo, Duy V Nguyen, and Robert Smart. Leveraging microservices architecture by using Docker technology. In *SoutheastCon 2016*. IEEE, 1–5.
7. Ben Kehoe and others. 2015. A survey of research on cloud robotics and automation. *IEEE Transactions on Automation Science and Engineering* 12, 2 (2015), 398–409.
8. Anis Koubaa. 2016. *Robot Operating System (ROS): The Complete Reference*. Springer International Publishing.
9. Zhanibek Kozhirbayev and Richard O. Sinnott. 2016. A performance comparison of container-based technologies for the Cloud. *Future Generation Computer Systems* 68 (2016), 175–182. Elsevier.
10. James J Kuffner. 2010. Cloud-enabled Humanoid Robots. In *IEEE-RAS 10th international conference on humanoid robotics, Nashville, TN*.
11. R. Kurzweil. 2005. *The Singularity Is Near: When Humans Transcend Biology*. Penguin Publishing Group.
12. Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.
13. Chaochao Lu and Xiaoou Tang. 2014. Surpassing Human-Level Face Verification Performance on LFW with GaussianFace. *CoRR* abs/1404.3840 (2014). `http://arxiv.org/abs/1404.3840`
14. Bishwadeep Mainaly and Dhruba Ningombam. 2014. A Survey on Cloud Robotics. *Communication, Cloud and Big Data: Proceedings of CCB 2014* (2014).
15. Sam Newman. 2015. *Building Microservices*. " O'Reilly Media, Inc.".
16. J Quintas, P Menezes, and J Dias. 2011. Cloud robotics: Towards context aware robotic networks. In *International Conference on Robotics*. 420–427.
17. Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (2015), 85–117.
18. Kyoung-Taek Seo and others. 2014. Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters* 66 (2014), 105–111.
19. B. Siciliano and O. Khatib. 2016. *Springer Handbook of Robotics*. Springer International Publishing, Chapter Networked Robotics, 1109–1134.