An Adaptive Robotics Middleware for a Cloud-based bridgeOS

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

Luís Manuel Antunes Veiga
INESC-ID
Instituto Superior Tecnico
Lisbon, Portugal
luis.veiga@inesc-id.pt

***Abstract*—An innovative platform, bridgeOS, attempts to provide a new standard and generic solution for bridging robots and users with the cloud, by integrating recent Services paradigms, using a web-oriented approach and supporting ROS. To better take on such role, we propose a cloud-oriented extension for bridgeOS, 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 propose is robust, resilient, versatile, capable of scaling to hundreds of components. And, our experimental results show that it is a viable solution, whose benefits exceed the overhead it generates.**

***Index Terms*—Cloud Robotics, Dynamic Offloading, Robotics Middleware, bridgeOS, Docker, ROS, Real-Time Monitoring**

I. Introduction

Robotic applications and their capabilities, alongside their levels of mobility and autonomy, 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 expensive computation and data to the cloud, effectively creating the field of 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. Surveys [1]–[3], have examined current solutions and identified numerous shortcomings:

- Non-generic or narrow scope of services;
- 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;
- Use of proprietary (closed-source) or non web-oriented communication protocols;
- Lack of security, privacy and anti-tampering mechanisms for network connections.

While those features are not universally lacking from each one, it is our sentiment that a complete framework must be capable of implementing them all. Designed by Bridge Robotics, bridgeOS, attempts to take on the role of the "de facto" standard by providing a new solution, integrating recent Services paradigms, using a web-oriented approach and supporting existing technology such as ROS.

To attain this objective, we propose 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.

The work presented in this paper was developed, as a Master's Thesis, by request of Bridge Robotics for their bridgeOS solution.

II. Related Work

A. Robot Operating System

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.

A 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 central node, the ROS master, while the second layer is simply for direct communications between them. This ROS master 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. In turn, it will advertise each side, so they can open communication channels without intermediaries.

B. 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.

C. Docker

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. This paradigm presents multiple advantages when compared to direct virtualization or virtual machines (VMs) [4]. For instance, boot time is faster and containers can make direct calls to instructions of the host's CPU with performances near those of native applications [5].

Docker is currently the main platform providing containers. Its 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 [6]. This provides additional benefits, services can then be updated individually without disruption and scalability becomes more precise, only increasing the bottlenecked parts. Lastly, containers are created from layered templates, known as Docker images, that can be easily shared via public or private registries and reused, to extend or adapt to a particular need.

To facilitate the orchestration and scheduling of containers within highly scalable environments with large clusters of machines, additional sets of management and supervision tools are often require. Docker provide such tools through Docker Swarm, its official container manager. Each Swarm consists of a cluster of docker nodes, who 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. 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.

III. Architecture of the Proposed Middleware

The system-wide architecture for the cloud-based bridgeOS implementation, augmented by the proposed middleware, is depicted in Fig. 1. With it, the extensions undertaken by this work 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 the WebSocket protocol. 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. Fig. 2 offers a more in-depth look about how robots interact with and use our middleware, and subsequently, the cloud. Logically, some robots might already possess core functionalities (i.e. native ROS drivers controlling local actuators or sensors) and will turn to bridgeOS as an option to enhance them. It is then plausible, that such robots will have an on-board ROS master. Therefore, we have to anticipate the occurrence of a local ROS master, while also planning for the opposite possibility, by offering a dedicated ROS master instantiated in the cloud. The middleware adapts to both scenarios by managing network routes and enable packet forwarding.

Additionally, as a means to provide a portable middleware, both the Master and Robot Controller modules can be launched as native applications or as docker containers, similarly to bridgeOS Skills. For Robot controllers, the benefits of executing them as natively, are to allow greater control of the host and its resources, and enabled increased monitoring capabilities. On the other hand, using them as Docker containers helps achieve the interoperability characteristic desired for the middleware, considering that, any system supporting Docker

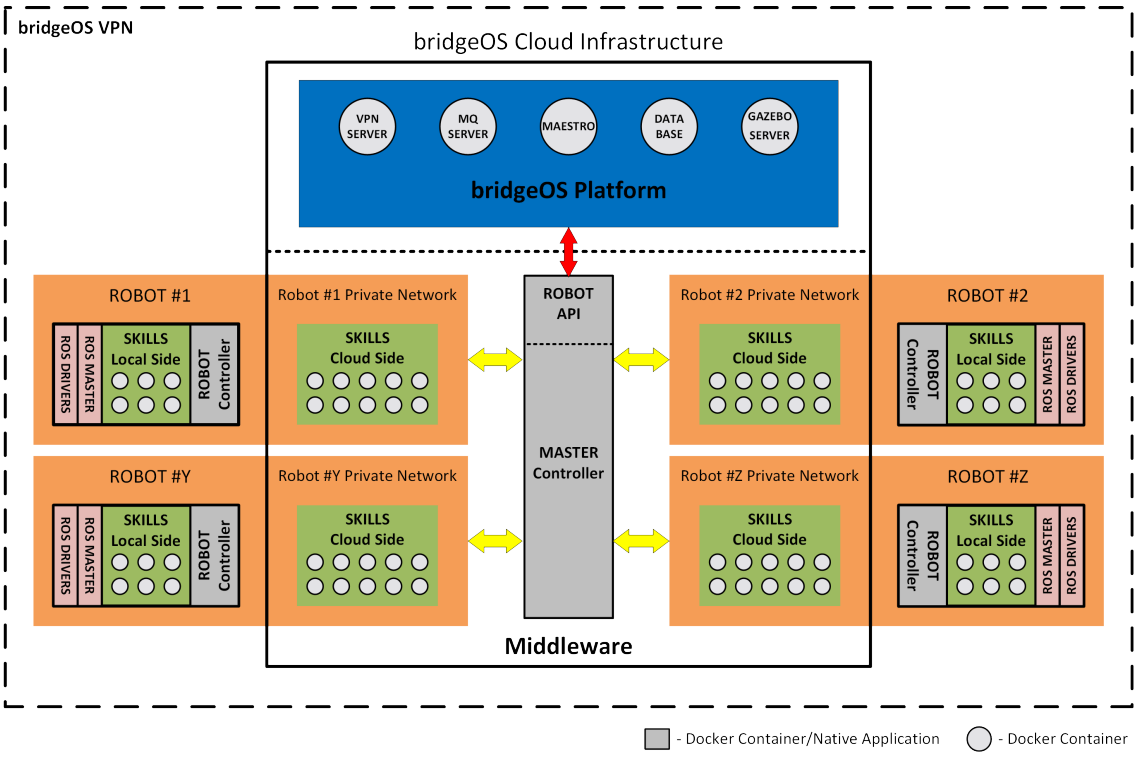

Fig. 1. Overview of the extended bridgeOS architecture

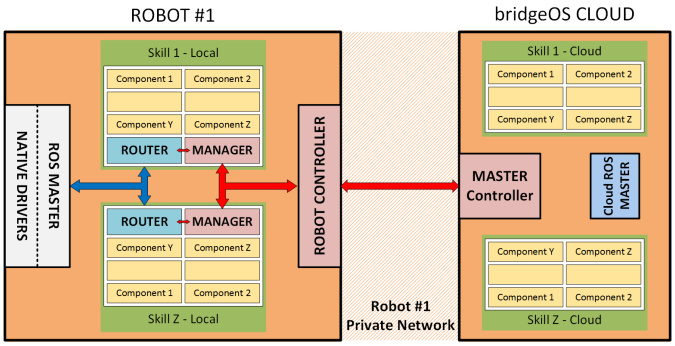

Fig. 2. Detailed architecture from a robot point-of-view. Red arrows represent communication channels using WebSocket, dedicated to monitoring and management purposes, while blue arrows represent communication channels for ROS. Connections involving components are not depicted.

is able to launch them without hassle, avoiding lengthy pre-configurations.

A. Master Controller

The Master Controller is the principal management component of our middleware, and serves multiple crucial roles. For starters, it acts as a gateway, providing robots with an entry point to the bridgeOS platform by exposing a common WebSocket 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 REST interface.

Located inside the cloud infrastructure, one of its core functions is to orchestrate cloud containers, 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.

Another core function is to create a centralized information hub, logging data regarding the current states, and monitoring everything related to robots, skills, components and modules. While Robot Controllers monitor the local side of their private network and forward 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.

B. 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, used by the robot's containers, and establish the needed network routes, to ensure that all containers and Skill 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 network traffic is redirected through it, since the VPN tunnel is established inside its container.

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. 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 the 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 Skill Managers, to perform real-time performance checks, enabling adaptive offloading decisions.

A premise and driving factor of the work presented, 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 decisions 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, with virtual machines, we could easily capture a snapshot of their current state and replicate it 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, specific to Skills or components, and storing ROS messages, based on user configurations, to be shared with the help of the Skill Router during migrations.

Components synchronization is not constrained solely to migrations. Some of the objectives tackled by this work 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, for instance, by partially replicating the Master Controller functions whenever disconnected and recover disrupted Skills.

C. 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, components can be launched both locally and in the cloud, although, only one of its containers is kept active. Skills are embodied slightly differently, as they are bundled together with two dedicated middleware modules, **Skill Manager** and **Skill Router**, operating with the purpose of organizing the components. 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.

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.

IV. Evaluation Methodology

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.

To assess the developed middleware, we had at our disposal one 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 located in Lisbon (Portugal), 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, one 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 implemented the network environment depicted in Fig. 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.

A. Offloading Performance Evaluation

The goal of this series of tests was to observe the potential of our middleware, in terms of its offloading capabilities, and

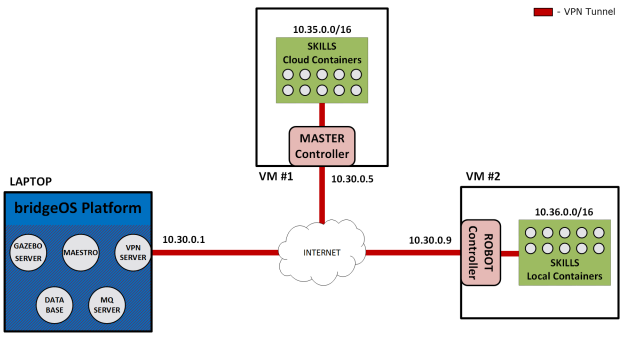

Fig. 3. Evaluation Setup

measure the overhead it may cause. To that end, we selected 3 existing bridgeOS Skills using ROS and integrated them with our architecture: people detection, autonomous navigation and autonomous mapping. For both navigation and mapping Skills, we used an 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 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. 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.

B. Middleware Benchmarking

The intent of this series of experiments was to test the overall 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 benchmarks, where the parameters were amplified until failure or impossibility to continue.

Since each module has a different role, we defined numerous experiments designed to assess different aspects of their core functions. First, to benchmark the middleware, we gradually increased and measured the number of both, startup (parallel launch using the Robot Controller's internal pooling mechanisms) and concurrent (sequential launch), Skills the Robot Controller was capable of handling. Then, we concentrated on the Skill modules, repeating the same concurrency experiment for components. 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 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 local resource usage made by the different modules, 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 performing a repetitive and computationally inexpensive task.

V. Results

A. Offloading Performance Results

TABLE I
USE CASES RESULTS: SKILL OFFLOADING COMPARISON BETWEEN THE BEST DECOMPOSITION TESTED W.R.T. FULLY-LOCAL EXECUTION

Skill	TPT	CPU	RAM	Network	Power*
People Det.	+64,76%	-88,32%	-45,85%	+137,21%	-63,10%
Navigation	+2,11%	-83,99%	-5,30%	+2,98%	-72,33%
Mapping	-8,18%	-92,78%	-67,87%	-80,85%	-87,93%

*Power consumption is merely an estimate based on the usage of hardware resources.

Based on resource usage shown in Fig. 4, we noted an overall decreasing tendency, albeit with some notable exceptions. Case in point being the spike in CPU usage for navigation Skill, which correlated 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, 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 explained by the massive increase of network bandwidth. The pointclouds generated had sizes of around 30-40 MBytes, and were forwarded successively between its components. So, whenever they were in different locations, additional round-trips were required. And, since the pointclouds were published periodically, the Skill rapidly generated a network bottleneck.

Lastly, with the results obtained, we were able to examine whether any combination of components performed better than native execution. Based on Table I, 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 the overall degree of importance it would have.

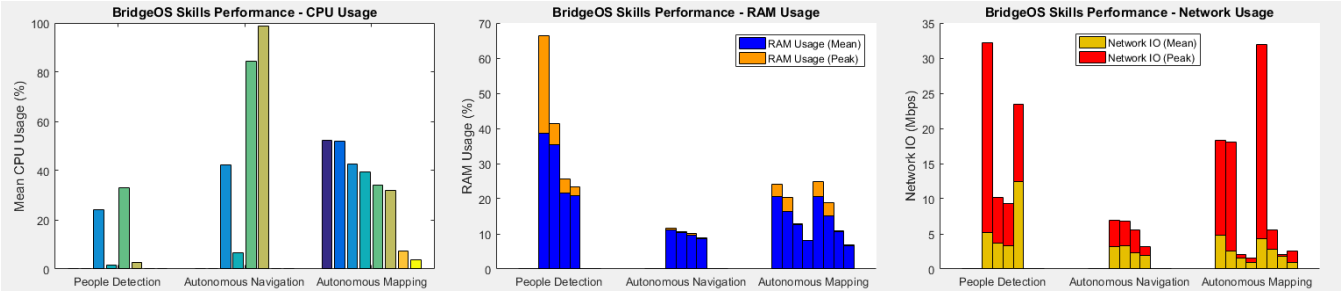

Fig. 4. Use Cases Results: each bar represents a different combination of component locations, going left-to-right, from fully-local to fully-cloud execution.

B. Middleware Benchmarking Results

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. In reality, those numbers were constrained by two factors, the limited on-board resources and the periodic usage surge pertaining to their monitoring functionalities.

Analyzing the temporal costs obtained, we observed that up to 100 concurrent Skills, their initialization remained stable, and similar to the previous test suite, at **5.26±0.61 seconds** each, afterwards it deteriorates rapidly due to the scarcity of resources. 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 occasional network-related or internal failures of our Skill Manager. To be noted, this boundary only regards components that can be offloaded, since those with fixed locations generate less overhead.

Nonetheless, the results indicate that even when instantiating skills of 250 components, the mean duration for initializing each one remains stable at **0.89±0.16 seconds**. This measure includes both the container launch and offloading model initialization, on top of 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 other than bandwidth, due to component monitoring, Skills Managers follow linear trends.

In terms of the mapping capacity, the results are also quite good. Mapping ROS topics lasts in average **443±20 ms** each, and a single Skill Router can map at least **5,000 topics**. An artificial limit, since we considered the total initialization time required and concluded there was no interested in further testing such feature. 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 corresponds to limitations in our cloud resources. Given this upper bound, we decide to successively increase their message publishing rate instead, sustaining each iteration for a period of 10 seconds. The Skill Router was capable of handling rates of up to 500Hz, reaching a total throughput of **50 000 messages each second**. 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 larger. Resource-wise, the Skill Router also performs positively, as its usage trends remain linear.

VI. Conclusion

To conclude, the middleware, represented by the different modules conceived, is able to achieve most requirements we sought. 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. However, we feel that further testing, especially in live scenarios, is still required.

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 with help of technological advances in cloud computing and telecommunications, newer applications and others to finally become viable.

Acknowledgment

The authors thank INESC-ID and Tiago Costa, from Bridge Robotics, for supplying the necessary hardware to perform the tests presented in this paper.

References

[1] B. Mainaly and N. Dhruba, “A Survey on Cloud Robotics” in Communication, Cloud and Big Data: Proceedings of CCB 2014 (2014).
[2] G. Hu, W. P. Tay, Y. Wen, “Cloud robotics: architecture, challenges and applications.” IEEE network, 26(3).
[3] A. Chibani, et al. “Ubiquitous robotics: Recent challenges and future trends.” Robotics and Autonomous Systems 61.11 (2013): 1162-1172.
[4] K. T. Seo, et al. “Performance comparison analysis of linux container and virtual machine for building cloud” in Advanced Science and Technology Letters 66 (2014): 105-111.
[5] Z. Kozhirbayev and R. O. Sinnott, “A performance comparison of container-based technologies for the Cloud” in Future Generation Computer Systems 68 (2017): 175-182.
[6] D. Jaramillo, D. V. Nguyen, and R. Smart. “Leveraging microservices architecture by using Docker technology.” SoutheastCon, 2016. IEEE, 2016.