Cloud Robotics: SLAM and Autonomous Exploration on PaaS

Giovanni Toffetti, Tobias Lötscher, Saken Kenzhegulov, Josef Spillner, Thomas Michael Bohnert
InIT, Zurich University of Applied Sciences (ZHAW)
[toff|loeh|kenz|spio|bohe]@zhaw.ch

ABSTRACT

Robots are moving out of factories, service robotics is bringing them to our homes, work environments, cities, and outdoors. While the Robot Operating System (ROS) is promising to open the world of robotics to developers, a proper platform and ecosystem supporting *robotic applications development* is still missing. This work presents an example of *cloud robotics* application in which cloud computing is not just complementing limited robot capabilities, but is leveraged to provide a development and operations environment supporting the complete life-cycle of a robotics-enabled application. We relate on our experience building cloud robotics applications spanning heterogeneous hardware (i.e., robots and cloud servers) through a use case scenario.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; *Robotics*;

KEYWORDS

Cloud robotics, PaaS, Kubernetes, micro-services, cloud native applications, cloud computing

ACM Reference format:
Giovanni Toffetti, Tobias Lötscher, Saken Kenzhegulov, Josef Spillner, Thomas Michael Bohnert. 2017. Cloud Robotics: SLAM and Autonomous Exploration on PaaS. In *Proceedings of UCC '17: 10th International Conference on Utility and Cloud Computing Companion, Austin, TX, USA, December 5–8, 2017 (UCC'17 Companion),* 7 pages.
https://doi.org/10.1145/3147234.3148100

1 INTRODUCTION

The connection between the physical world and the virtual world has never been as exciting, accessible, and economically viable as today. Sensors, actuators, and robots are able to deliver many physical services in several scenarios, including industrial production and home automation, elderly care, assisted living, logistics and cooperative maintenance.

In isolation, *computing capabilities of robots are however limited* by embedded CPUs and small on-board storage units. By connecting robots among each other and to cloud computing, cloud storage, and other Internet technologies centered around the benefits of converged infrastructure and shared services, three main advantages can be exploited. First, *computation can be outsourced to cloud services leveraging an on-demand pay-per-use elastic model.* Second, *robots can access a plethora of services complementing their capabilities* (e.g., speech analysis, object recognition and manipulation, knowledge sharing), enabling new complex functionalities and supporting incremental learning. Third and foremost concerning this work, *cloud development models and best practices* (e.g., PaaS, CI/CD, CNA, micro-services, containerization) *can be leveraged to simplify and support the entire application life-cycle*, from design and development all the way to deployment, operation, and update.

Cloud robotics is a natural extension to the Internet of Things (IoT). Where IoT devices gather information about an environment to help make smarter decisions, cloud robotics is able to use this information and act on it. Although there is clear recognition that cloud access is required to complement robotics computation and enable functionalities needed for some robotic tasks, it is still unclear how to best support these scenarios.

Together with our partners at Rapyuta Robotics (RR)[1], ZHAW is working within the framework of the Enterprise Cloud Robotics Platform[2] (ECRP) project to build the first PaaS (Platform as a Service) explicitly designed to target cloud robotics development. The project's goal is to enable robotic applications to take full advantage of cloud computing services, resources, best practices, and automation by integrating ROS nodes and robots just as other composable services in the cloud. Just as cloud computing is based on abstracting functionality from its implementing resources, we strive to build a platform that will allow developers to build robot-enabled applications without necessarily being robotic experts.

In this paper, we relate on our experience developing a simple proof-of-concept cloud robotics application for autonomous exploration, SLAM (simultaneous localization and

[1]Rapyuta Robotics is an ETH spinoff whose founding members were core researchers and developers of the EU-founded RoboEarth project, one of the first cloud Robotics projects in Europe.
[2]https://www.zhaw.ch/no_cache/de/forschung/personen-publikationen-projekte/detailansicht-projekt/projekt/3192/

UCC'17 Companion, December 5–8, 2017, Austin, TX, USA

ACM ISBN 978-1-4503-5195-9/17/12…$15.00
https://doi.org/10.1145/3147234.3148100

mapping), and navigation. Albeit we built some custom components for it, the application functionality is not novel per se. The contribution of this paper lays instead in the application architecture, its orchestration capabilities, its underlying development model deriving from the adoption of the ECRP, and our experience and plans in dealing with heterogeneous robots and servers in the cloud.

Due to the strategic nature of the project results for our partners, we will not disclose the full ECRP architecture in detail here. However, our experience and results are general enough to be relevant for any similar endeavor building robot-enabled cloud-native applications.

2 STATE OF THE ART AND PRACTICE

Cloud Robotics

The term "cloud robotics" is credited to James J. Kuffner in 2010. A current definition of "cloud robot and automation systems" refers to them as any "robot or automation system that relies on either data or code from a network to support its operation, i.e., where not all sensing, computation, and memory is integrated into a single standalone system"[2].

Several research works fall into the broad definition above, including for instance the recent work on deep learning for grasping [3], or the RoboEarth[3] project results [5, 6]. However, from our cloud researchers' viewpoint, the above definition has more to do with distributed systems than cloud computing per se. As a matter of fact, it does not refer to any of the essential characteristics, nor service or deployment models commonly associated with cloud computing [4]. For the sake of this paper, we will simply define "cloud robotics" as *cloud computing research and practice applied to support the full life-cycle of robot-enabled applications*. We defer a more thorough definition to future work.

Robotic application development

The commercial domain of robotics is dominated by large enterprises, many of them multi-national conglomerates with origins in industry automation. Respectively, hardware platforms and software frameworks are closed and purpose-built for very specific applications. Despite the long existence of robotics and an active market of start-ups in recent time, there is no open and established eco-system anywhere near to the likes of iOS/Android, the entire Linux domain, Apache Hadoop ecosystem, OpenStack and other OSS Infrastructure as a Service stacks.

Recent developments, however, provide evidence for a disruption in the making. Comparable to the introduction of Linux or Android, the availability of and steadily maturing Robot Operating System (ROS)[4] is set to open-up and liberate the robotics market towards a truly open and participatory ecosystem.

ROS is a flexible framework for writing robot software. It is a "collection of tools, libraries, and conventions" with one top-most aim: to simplify the task of "creating complex and robust robot behaviors across the widest possible variety of robotic platforms". Why this? Because "creating truly robust, general-purpose robot software is hard". Dealing with a nearly unlimited heterogeneous and global environment is extremely challenging, so much that "no single individual, laboratory, or institution can hope to do it on their own"[5]. As a result, ROS was built from the ground up to encourage collaborative robotics software development.

Yet, ROS was designed with the focal point of an individual robot device in a local deployment. To truly unfold the potential of robotics, one has to take up a much wider vision, with large deployments of very heterogeneous robots, with very different abilities, in completely different environments. This adds different challenges for robot system development than targeting individual robots separately or as a small set. Furthermore, robots are meant to assist in application domains and those applications are equally diverse as the robots, ranging from personal all the way to enterprise and industrial application domains.

Cloud development

When it comes to the engagement of software developers, the adoption of Cloud Computing, Infrastructure as a Service (IaaS) but even more Platform as a Service (PaaS), has proven to significantly facilitate software-based innovations in almost any cloud-enabled application domain. Through cloud computing services, nearly unlimited resources (IaaS) combined with a nearly unlimited set of functionality (PaaS) have become available in a native, well-defined and unified, programmatic way to developers of software applications simply by software (code) itself, the lingua franca of any software engineer.

The potential, inherent to IaaS and even more to PaaS, to accelerate software-based innovations makes no exception of robotics. Rather, the emergence of a cloud-enabled software development environment appears overdue. Irrespectively, no public Platform as a Service (PaaS) for Robot-based Applications is on the market, and only one proposal has been made by academia, that is RoboEarth[10]. RoboEarth, however, is focused on providing compute, storage, and networking resources in an IaaS-like approach. To this date, all benefits of the modern software engineering with PaaS are unavailable to one of the most important and fastest growing markets, and this despite unquestioned innovation and commercial potential of PaaS for "connected things"[6] (e.g., robots).

3 MOTIVATION

Robotics and its applications domain have been rapidly changing in the recent years for a number of factors, including advancements of hardware and software capabilities as well as decreasing cost of sensors and actuators. The big shift is the

[3] http://roboearth.ethz.ch/
[4] http://ros.org
[5] http://www.ros.org/about-ros/
[6] http://www.gartner.com/newsroom/id/3241817

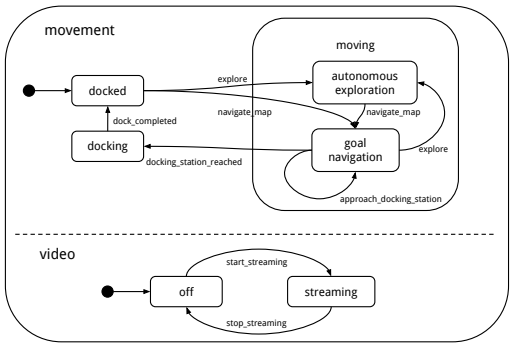

Figure 1: Application behavior statechart

change of focus from industrial robots working in constrained predictable environments (e.g., factories and robot cages) to small cheaper robots immersed in unknown or changing environments requiring context identification and adaptation. These situations require more complex processing for recognition, coordination, and planning (e.g., deep learning, fleet coordination) while at the same time robot manufacturers strive to produce devices which are accessible to the masses. One big obstacle to this is of course the cost of devices. While decreasing, the costs of sensors, actuators, and powerful processors are still too high to make most robots a commodity (notable exceptions are iRobot's Roombas). Still, combining cloud computing with cheaper robot processors is a viable solution towards building robot-enabled applications that can have mass market penetration yet have access to powerful computation when needed (i.e., on demand when a robot is in operation). Moreover, services deployed on the cloud are designed with reliability and availability targets that consumer robots cannot guarantee.

What is needed is an ecosystem that embraces the heterogeneity of robotic devices, applications, and related commercial imperatives to support robot-enabled applications in their complete life cycle, from design and implementation, all the way to operations. Such an ecosystem will enable all stakeholders to participate in the vision of global availability of a wide array of robotics services. The Cloud Computing "Platform as a Service" (PaaS) paradigm is the natural incarnation of such an ecosystem from a development perspective. PaaS provides a development and execution environment: an execution "platform" that, abstracting from the underlying system infrastructure (e.g., bare-metal servers, virtual machines, robots, generic devices), allows developers to focus on application functionality, building, deploying, and managing at runtime their applications as compositions of high level platform components and services. This is the goal of the ECRP project and of the work in this paper.

4 USE CASE

For the purpose of this paper, we concentrate on a simple use case of a robot patrolling application. We call the use case "RoboPatrol". It also covers the common scenario in domestic service robotics of setting up the robot in a new environment.

RoboPatrol is a simple patrolling application with a Turtlebot[7] 2. The user is given a Web UI through which she can drive the robot around on a map and see a live stream from its camera. One or more maps of the environment can be constructed by the robot through manual or autonomous exploration.

From a functional perspective, the application behavior can be represented with the statechart diagram in Figure 1. The system is composed of two orthogonal (sub) systems concerning the movement and video functionalities. The latter only has two states (streaming or not), while the former has different sub-states that can be triggered either by user or robot behavior (e.g., "dock_completed" event).

RoboPatrol embodies a "proper" cloud robotics application in the sense that it follows CNA (cloud-native application) design principles [9] such as, for instance, elasticity, including provisioning and disposing of its own required components on demand. This is an important feature in cloud robotics applications, enabling components to be shut down (hence reducing operational costs) while robots are not performing tasks that require them.

5 ARCHITECTURE

The high-level logical architecture of RoboPatrol is depicted in Figure 2. The application operates on a distributed heterogeneous system composed of one robot, containers running in a cloud infrastructure, and the web browser(s) of the application user(s).

The main application components running on the robot are: the "minimal" Turtlebot ROS script (enabling the ROS master and the moving base of the robot), the MS Kinect video streaming component, the "move_base" ROS node used for navigation, the "rplidar" node receiving data from the laser scanner mounted on top of the robot, "amcl" which is used for positioning of the robot on the map, and the self-docking functionality. These components will be started and stopped in the robot depending on the application state from Figure 1. Apart from application-specific components, two cloud-robotics platform components on the robot allow it to communicate with the cloud through a control and a data plane (respectively called device manager and cloud bridge).

On the cloud we have several application components that can be enabled or disabled depending on application requests and behavior. Only two components are always enabled as they provide application control, these are: the "front-end" component that provides the endpoint for user access and authentication through a browser and the "robopatrol-logic" component which implements the application behavior and orchestrates the provisioning and disposal of all other application components. The idea is always to minimize the number of active components while the robot is not doing anything (that is minimize operational costs for the application by taking advantage of cloud pay-per-use model). Application

[7] http://wiki.ros.org/Robots/TurtleBot

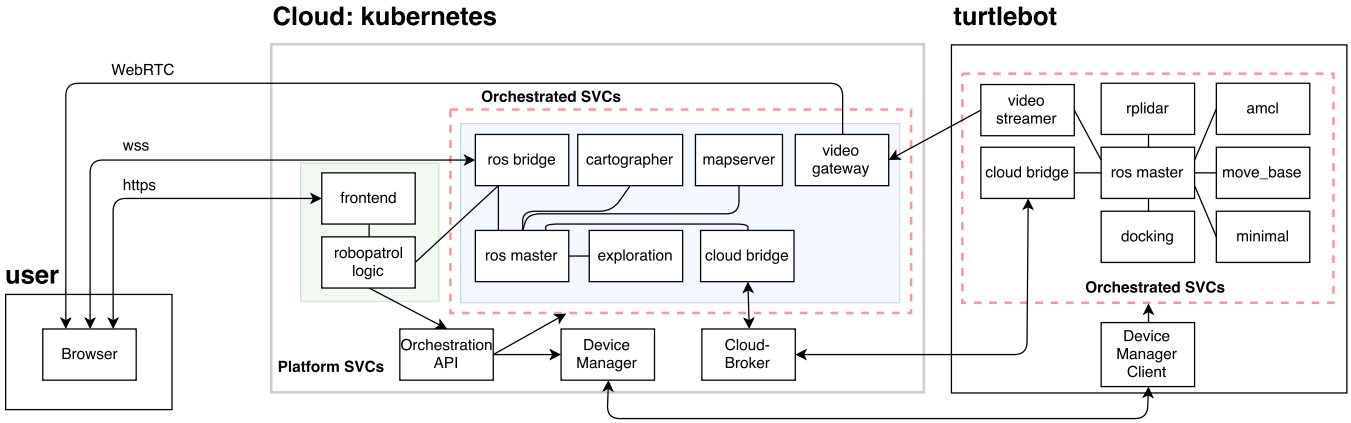

Figure 2: Logical architecture of the application

State	Robot	Cloud
docked	-	-
autonomous exploration	minimal, move_base, rplidar	cartographer, exploration, ROS bridge
navigation	minimal, move_base, amcl, rplidar	map_publisher, ROS bridge
docking	minimal, self-docking	-

Table 1: Active components in each application state

components that are instantiated on demand are instead: Google Cartographer (used for SLAM functionality while building maps), a static map publisher used for point to point navigation, the autonomous exploration logic, and the video streaming gateway.

Table 1 lists which components are active in each movement state of the application as depicted in Figure 1. Components required for video streaming are spawned both on the cloud and the robot when requested by the user.

Apart from hardware constraints, that is dependency of a component on a physical device (e.g., rplidar, minimal, move_base, a GPU in some cases), the placement on the cloud or the robot of the components listed above is arbitrary. Theoretically, any configuration on a spectrum ranging from running everything on the cloud or everything on the robot would be possible. In practice, typical constraints from placement and resource allocation research apply, namely in terms of capacity (i.e., CPU, memory, storage, bandwidth) as well as performance (e.g., latencies, response times). We relate on our experience concerning these aspects in the following section.

6 CHALLENGES AND EXPERIENCE

6.1 Hardware Heterogeneity

While the application was originally implemented to run on a Turtlebot 2, whose computing unit is typically based on a PC architecture (ASUS F200M Celeron 4GB RAM by default), the ECRP platform is meant to allow *composing applications leveraging heterogeneous robots and machines in the cloud*. To this end, the platform is intended to support the automated build and packaging in containers of developer-provided components for multiple architectures using the Docker manifest concept. Automated build is a common feature in PaaS cloud computing which we extended for heterogeneous platforms. When a component needs to be instantiated, the Docker daemon will pull the appropriate architecture-specific container to a specific robot. In the cloud, placement requirements can be expressed through annotations supported by common container management solutions (e.g., Kubernetes).

The ECRP platform makes one further assumption concerning robots that have been successfully on-boarded for development, and that is that the so-called "minimal" ROS node, which is providing access to the basic hardware functionality, is available and tested for a robot of a specific class (e.g., a Turtlebot, a DJI drone). The minimal node is expected to work properly and come with all required device drivers to correctly operate the robotic hardware.

In order to test our application on different platforms, we replaced the computing unit of our lab Turtlebot with a Raspberry Pi3 (the same unit used by default in Turtlebot 3 "Burger" models) which is based on a quad-core ARM Cortex-A53 processor equipped with 1 GB of RAM. We also experimented with a Raspberry Pi Zero (single core ARM11 with 512M RAM) and a laptop equipped with and Intel i5 processor and 4GB of RAM.

In both Pi cases, we had to install the Turtlebot and rplidar ROS packages on the boards and make sure that the rplidar would get a consistent device identifier when plugged in. Moreover, we needed to install Docker for the two different ARM architectures. For the Raspberry Pi3 the official guide from Docker worked out without any issues. It is the same procedure as installing it on any other machine, such as the laptop we used first. On the Raspberry Pi Zero we had issues

Processing unit	CPU(%)	RAM(%)
ASUS F200M Celeron 4GB RAM	50	17
Raspberry Pi 3 1GB RAM	30	54
Pentium i5 4GB RAM	30	17

Table 2: Average CPU and memory usage running minimal, move_base, rplidar, cloud_bridge, and cartographer on robot

regarding the cgroup kernel features, which were not working when we installed the official Docker release. To fix those issues we found an adapted version created by hypriot[8]. This version of Docker is especially adapted for the Raspberry Pi platform. We followed the first step of the installation guide [9]. Using this version we could overcome the problems we had with the official release. Building architecture-specific components (and containers) for some of the application elements (e.g., turtlebot minimal, Google Cartographer) also required some effort as many of the required libraries had to be compiled from source on both the Pi3 and Pi Zero.

Table 2 lists the average CPU and RAM usage across the different processing units we used while running a manual exploration scenario. The cloud_bridge is sending few ROS topics[10] for visualization of the robot navigation in the user browser. Camera streaming was turned off.

The immediate take away of our simple experiment is that the Raspberry Pi Zero cannot be used to run Google Cartographer on board of the device, as it alone requires on average 270MB of RAM and combined with the other components it exceeds the RAM capacity of the Pi Zero. Hence, placing cartographer on the cloud enables using cheaper computational units on the robots, but this is not the only advantage: by running mapping components as cloud services, replicated storage solutions can be used to securely persist maps and provide them to user clients on request.

Albeit the use case we chose requires limited computational power, lower-end systems already show considerable resource utilization in both CPU and RAM compartments. This leaves little room for other simpler (e.g., video and audio streaming) or more complex (e.g., object and speech recognition, 3D mapping, pick and place planning) computational tasks an application might require justifying the invocation of external services or the delegation of computation to the cloud. Clearly, considerations with respect to code, data, resources mobility [1] and their different overheads apply.

6.2 Cloud service and billing model

We have previously discussed how the RoboPatrol application is self-managing in terms of controlling the life-cycle of a number of its own components in the cloud and on robots. While turning off components on robots might arguably only reduce electrical consumption for unneeded computation (i.e., at robots charging locations), shutting down components in the cloud can have a much more relevant cost reduction impact. In order to quantify it, one has to take into account the cloud service models and associated billing involved.

Containerization, cloud-native design principles, larger development teams and continuous delivery practices have had a serious impact on current cloud development practices. Most new cloud-based services are implemented using micro-services architectures [8], Robopatrol is no exception in this trend as it consists of micro-services built on a container management solution (Kubernetes) and a service catalog.

Current large IaaS providers still do not support per-container billing (they charge per container management cluster virtual machines (VM) usage), while smaller PaaS providers do offer container-based billing at a price that is currently much higher with respect to running your own container management cluster on IaaS.

Our experimental clusters run on AWS and GKE services both based in Frankfurt (Germany), and billing per VM. Hence, while shutting down unused containers would reflect in immediate cost saving in a proper container-based billing scenario, our cost reduction strategy would not work without being coupled with a cluster auto-scaling policy that reduces the cluster size in terms of VMs when "load is low". The drawbacks of such a strategy are common IaaS pitfalls: from defining correct triggers values for threshold based auto-scaling rules, to managing stateful components on scale-ins, to suffering VM startup times (i.e., in the order of minutes) when requiring scale outs.

While shutting down VMs is notoriously fast, and proper management of stateful application components becomes simpler with a container management solution and replication, VM startup times become a real hindrance when your robotic application behavior depends on new resources being timely added to a cluster. Apart from trivial VM over-provisioning policies, or applying sophisticated proactive solutions from the IaaS auto-scaling research literature which rely on accurate prediction expectations, there are no silver bullets to VM provisioning times problems. However, our prediction is that VM clusters for container management solutions will be used for running multiple concurrent applications, hence their management will have to rely on spare capacity and VM starting delay effects would be mitigated by application component churn.

Novel service models such as "Function as a Service" (FaaS) [7] are emerging especially for dealing with IoT devices. While many cloud applications have a number of components which are amenable of being re-implemented as pure "Function as a Service" elements, we feel this is not the case for the moment for RoboPatrol. The reason for this lays in the fact that the ROS components involved in the application rely on publish/subscribe messages being exchanged in order and with bounded latency, something that current FaaS implementation, with their on-request provisioning of functionality, cannot at the moment achieve with minimal (i.e., in the order of few milliseconds) latency.

[8] http://blog.hypriot.com
[9] https://github.com/alexellis/docker-arm/blob/master/ZERO.md
[10] i.e., map, laserscan, tf, odom

6.3 Placement constraints for service components

Before discussing placement, a short disclaimer concerning safety-critical / real-time components is due. Components that require real-time (or near real time) behavior (e.g., for safety, or stability control purposes on a drone) are typically run on real-time operating systems and on dedicated boards which are separated from the main computational unit running ROS nodes. Their placement is fixed on the robots. In this section we are instead discussing of the placement of all components that *can be* run both on robots and the cloud with an acceptable latency bound.

As mentioned on the previous section, the ECRP platform encourages the use of micro-services and the composition and reuse of components for the development of robot-enabled applications. The platform provides a set of basic building blocks that can be instantiated and combined to build basic functionality (e.g., device management, component execution, bridging cloud and robot messages), however the long term goal is to provide a complete ecosystem for developers to publish their components and monetize on their adoption in the same way as in cloud service marketplaces (e.g., in AWS or Openshift).

The additional challenge with respect to "pure cloud computing" services and components is that ECRP services are generally compositions of interconnected components running across cloud servers and (possibly heterogeneous) robots. Two kind of considerations have to be made: 1) notwithstanding ROS abstractions, functionality requiring specific sensors or actuators cannot be deployed on robots lacking them, and 2) due to latencies and resource constraints on robots, placement of components have a significant effect on overall application performance.

In order to tackle the first issue, upon definition of a new robotic service (i.e., a *service class* in the service catalog), the ECRP platform allows developers to attach hardware requirements to any service component. These requirements act as filters that match robotic features defined through a RDL (robot definition language). Hence, when a developer wants to provision an instance of a service from the catalog, the service orchestration logic checks whether all hardware requirements needed for deploying a service using one or more specific robots are met. If they are not, the provisioning request is not allowed.

Concerning placement of components across robots and the cloud, most ROS nodes require timely updates of the relative positions of the multiple coordinate frames of the robot sensors and components (i.e., its position with respect to a map, the position of its sensors with respect to the robot itself: think of a camera mounted on an arm) through the ROS */tf* topic to make correct use of sensor data and plan control actions. For instance, the cartographer node receives readings from the laser scanner at a rate of roughly 5 hertz on the */scan* topic, and the relative positions of the laser scanner with respect to the other coordinate frames on the separate */tf* topic. When these topics are not properly synchronized or delivered, warnings are thrown as the SLAM algorithm cannot make sense of the sensor data.

The latency we measured using public cloud infrastructure based in Frankfurt (average ping time from Zurich is around 8ms) was low enough to allow us to place latency-sensible components such as cartographer off the robot and in the cloud without experiencing errors nor warnings. However, proper placement feasibility for a component would in general depend on its tolerance to delays as well as input data size, frequency of updates, available bandwidth and latency of the connection. We are currently investigating the possibility of using an automated mechanism for adaptive dynamic placement of ROS components across cloud and robots that is able to identify the optimal placement of a component based on several contextual conditions (e.g., robot position and availability of different communication technologies / QoS while moving around).

6.4 Configuration dependencies

While ROS is very effective in abstracting some functionality from low-level robotic details through higher level interfaces (e.g., navigation with move_base), robotic algorithms typically have to be tuned through a plethora of configuration parameters that stem from physical details of a robot and its sensors. Some configuration dependencies are static and obvious: for instance, move_base planning algorithms clearly need to know the size of a robot to properly plan navigation through a narrow passage. Also, navigation planning is currently executed in our application for a single robot.

However, our assumption of reusable services deployable on demand for common tasks partially clashes with the reality of these configuration parameter dependencies. To provide an example, we mentioned how we are able to run the cartographer component in the cloud; still, in order to properly function, it will need to be configured (at startup!) to know details about devices on the robot (e.g., the laser scanner in use with its range in centimeters). These static dependencies hinder the more dynamic scenarios we foresee the ECRP should support where robots can, while performing a task, replace one another at runtime depending on contextual conditions (e.g., low battery, malfunction, position).

7 FUTURE WORK

Through our work on a first reference application which combines recent cloud computing technologies with state-of-the-art open robotics frameworks, we have identified the need for a developer-centric cloud robotics platform. Our work has furthermore extracted necessary requirements for such a platform, one of which is the ability to quickly provision and dispose of containers and container clusters.

In the continuation of this work, we intend to work on applying additional principles from cloud-native application design to the reference application prototype in order to derive design guidelines for future cloud-robotics applications. As a part of this work, we will consider a coherent replication strategy for all involved components such as containers and

ROS nodes. Our preliminary experiments have shown issues with replicated ROS nodes which suggests that this will be a non-trivial problem to solve. Furthermore, as cloud-native applications differentiate strictly between stateless and stateful services, we will look into persistent storage options: on the device, in the cloud, or hybrid. Finally, our next steps include larger-scale field tests with fleets of multiple robots with heterogeneous hardware and capabilities.

8 CONCLUSION

This work has presented the motivations and main concepts of a PaaS for cloud robotics development through an example use case prototype. We reported our experience and learnings from implementing the application and running it on distributed heterogeneous computing resources leveraging cloud-native application principles and solutions.

ACKNOWLEDGEMENTS

This work has been partially funded by grant nr. 18235.2PFES-ES of the Swiss Commission for Technology and Innovation (KTI).

It has also been supported by an AWS Cloud Credits for Research grant, and a Google Cloud Platform Education grant which helped us run our experiments on public clouds.

We wish to thank the guys at Rapyuta Robotics for all the discussions and exchange of ideas, in particular: Gajamohan Mohanarajah, Dominique Hunziker, Dhananjay Sathe, Vivek Bagade, Bharat Khatri, and Alankrita Pathak.

REFERENCES

[1] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. 1997. Designing Distributed Applications with Mobile Code Paradigms. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*. ACM, New York, NY, USA, 22–32. https://doi.org/10.1145/253228.253236

[2] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg. 2015. A Survey of Research on Cloud Robotics and Automation. *IEEE Transactions on Automation Science and Engineering* 12, 2 (April 2015), 398–409. https://doi.org/10.1109/TASE.2014.2376492

[3] Jeffrey Mahler, Jacky Liang, Sherdil Niyaz, Michael Laskey, Richard Doan, Xinyu Liu, Juan Aparicio Ojea, and Ken Goldberg. 2017. Dex-Net 2.0: Deep Learning to Plan Robust Grasps with Synthetic Point Clouds and Analytic Grasp Metrics. (2017).

[4] Peter Mell, Tim Grance, et al. 2011. The NIST definition of cloud computing. (2011).

[5] Gajamohan Mohanarajah, Dominique Hunziker, Raffaello D'Andrea, and Markus Waibel. 2015. Rapyuta: A cloud robotics platform. *IEEE Transactions on Automation Science and Engineering* 12, 2 (2015), 481–493.

[6] Gajamohan Mohanarajah, Vladyslav Usenko, Mayank Singh, Raffaello D'Andrea, and Markus Waibel. 2015. Cloud-based collaborative 3D mapping in real-time with low-cost robots. *IEEE Transactions on Automation Science and Engineering* 12, 2 (2015), 423–431.

[7] Josef Spillner and Serhii Dorodko. 2017. Java Code Analysis and Transformation into AWS Lambda Functions. *arXiv preprint arXiv:1702.05510* (2017).

[8] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Florian Dudouet, and Andrew Edmonds. 2015. An architecture for self-managing microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*. ACM, 19–24.

[9] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Josef Spillner, and Thomas Michael Bohnert. 2017. Self-managing cloud-native applications: Design, implementation, and experience. *Future Generation Computer Systems* 72 (2017), 165–179.

[10] Markus Waibel, Michael Beetz, Javier Civera, Raffaello d'Andrea, Jos Elfring, Dorian Galvez-Lopez, Kai Häussermann, Rob Janssen, JMM Montiel, Alexander Perzylo, et al. 2011. Roboearth. *IEEE Robotics & Automation Magazine* 18, 2 (2011), 69–82.