# Rapyuta: A Cloud Robotics Platform

Gajamohan Mohanarajah, Dominique Hunziker, Raffaello D'Andrea, and Markus Waibel

***Abstract*—In this paper, we present the design and implementation of Rapyuta, an open-source cloud robotics platform. Rapyuta helps robots to offload heavy computation by providing secured customizable computing environments in the cloud. The computing environments also allow the robots to easily access the RoboEarth knowledge repository. Furthermore, these computing environments are tightly interconnected, paving the way for deployment of robotic teams. We also describe three typical use cases, some benchmarking and performance results, and two proof-of-concept demonstrations.**

***Note to Practitioners*—Rapyuta allows to outsource some or all of a robot's onboard computational processes to a commercial data center. Its main difference to other, similar frameworks like the Google App Engine is that it is specifically tailored towards multiprocess high-bandwidth robotics applications/middlewares and provides a well-documented open-source implementation that can be modified to cover a large variety of robotic scenarios. Rapyuta supports the outsourcing of almost all of the current 3000+ ROS packages out of the box and is easily extensible to other robotic middleware. A pre-installed Amazon Machine Image (AMI) is provided that allows to launch Rapyuta in any of Amazon's data center within minutes. Once launched, robots can authenticate themselves to Rapyuta, create one or more secured computational environments in the cloud and launch the desired nodes/processes. The computing environments can also be arbitrarily connected to build parallel computing architectures on the fly. The WebSocket-based communication protocol, which provides synchronous and asynchronous communication mechanisms, allows not only ROS based robots, but also browsers and mobiles phones to connect to the ecosystem. Rapyuta's computing environments are private, secure, and optimized for data throughput. However, its performance is in large part determined by the latency and quality of the network connection and the performance of the data center. Optimizing performance under these constraints is typically highly application-specific. The paper illustrates an example of performance optimization in a collaborative real-time 3-D mapping application. Other target applications include collaborative 3-D mapping, task/grasp planning, object recognition, localization, and teleoperation, among others.**

***Index Terms*—Cloud robotics, cloud-based mapping, networked robots, platform-as-a-service (PaaS).**

Manuscript received October 28, 2013; revised April 11, 2014; accepted May 16, 2014. Date of publication July 11, 2014; date of current version April 03, 2015. This paper was recommended for publication by Associate Editor D. Song and Editor A. Sarma upon evaluation of the reviewers' comments. This work was supported by the European Union Seventh Framework Programme FP7/2007–2013 under Grant 248942 RoboEarth and by AWS (Amazon Web Services) in Education Grant Award.

The authors are with the Institute of Dynamic Systems and Control (IDSC), ETH Zurich, Zurich, Switzerland (e-mail: gajamohan.m@gmail.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TASE.2014.2329556

## I. Introduction

THE past decade has seen the first successful, large-scale use of mobile robots. However, the vast majority of these robots either continue to use simple control strategies (e.g., robot vacuum cleaners) or are operated remotely by humans (e.g., drones, unmanned ground vehicles, and telepresence robots). One reason these mobile robots lack intelligence is because the costs of onboard computation and storage are high; this affects not only the robot's price point, but also results in the need for additional space and extra weight, which constrain the robot's mobility and operation time. Another reason is the absence of a common mechanism and medium to communicate and share knowledge between robots with potentially different hardware and software components.

The rapid progress of wireless technology and availability of data centers hold the potential for robots to tap into the cloud. Using the Web as a powerful computational resource, a communication medium, and a source of shared information could allow developers to overcome these current limitations by building powerful cloud robotics applications. Example applications include map building [1], task/grasp planning [2], object recognition, localization, and many others. Cloud robotics applications hold the potential for lighter, smarter, and more cost-effective robots.

Running robotics applications in the cloud falls into the platform-as-a-service (PaaS) model [4] of the cloud computing literature. In PaaS, the cloud computing platform typically includes an operating system, an execution environment, a database, and a communication server. Many existing cloud computing building blocks, including much of the existing hardware and software infrastructure for computation, storage, network access, and load balancing, can be directly leveraged for robotics. However, specific requirements (such as the need for multiprocess applications, asynchronous communication, and compatibility with existing robotics application frameworks) limit the applicability of existing cloud computing platforms to robot application scenarios.

For example, a general PaaS platform such as the popular Google App Engine [5] is not well suited for robotics applications since it exposes only a limited subset of program APIs tailored specifically for web applications, allows only a single process, and does not expose sockets, which are indispensable for robotic middlewares such as ROS [6]. The popular PaaS framework Heroku [7] overcomes some of these limitations, but lacks features required for many robotics applications, such as bidirectional data flow between robots and their computing environments. Other, more recent PaaS frameworks such as

Cloud Foundry [8] and OpenShift [9] are relatively liberal in terms of available runtimes and languages, but typically expect applications to be single processes or preconfigured set of parent and child processes[1] running inside a computing environment that has only an HTTP connection to the outside.[2] However, when it comes to robotics applications, such as the ones based on ROS, processes (ROS nodes) typically run as a computation graph and are dynamically configured to provide services for robots.

In summary, the inadequacy of the existing general PaaS platforms for robotics scenarios are mainly due to the differences in web applications and robotics applications. Web applications are typically stateless, single processes that use a request-response model to talk to the client. Meanwhile, robotic applications are stateful, multiprocessed, and require a bidirectional communication with the client. These fundamental differences may lead to different tradeoffs and design choices and may ultimately result in different software solutions for web and robotics applications.

Now, focusing on cloud robotics, the idea of having a remote brain for the robots can be traced back to the 1990s [10], [11]. During the past few years, this idea has gained traction (mainly due the availability of computational/cloud infrastructures), and several efforts to build a cloud computing framework for robotics have emerged. The DAvinCi Project [1] used ROS as the messaging framework to get data into a Hadoop cluster, and showed the advantages of cloud computing by parallelizing the FastSLAM algorithm [12]. It used a single computing environment without process separation or security; all inter-process communication were managed by a single ROS master. Unfortunately, the DAvinCi Project is not publicly available. While the main focus of DAvinCi was computation, the ubiquitous network robot platform (UNR-PF) [13], [14] focused on using the cloud as a medium for establishing a network between robots, sensors, and mobile devices. The project also made a significant contribution to the standardization of data-structures and interfaces. Finally, rosbridge [15], an open-source project, focused on the external communication between a robot and a single ROS environment in the cloud.

With the open-source project Rapyuta[3] we attempt to solve some of the remaining challenges of building a complete cloud robotics platform. Rapyuta is based on an elastic computing model that dynamically allocates secure computing environments (or clones [16]) for robots. These computing environments are tightly interconnected, allowing robots to share all or a subset of their services and information with other robots, through their corresponding clones. This interconnection makes Rapyuta a useful platform for multi-robot deployments such as those described in [17].

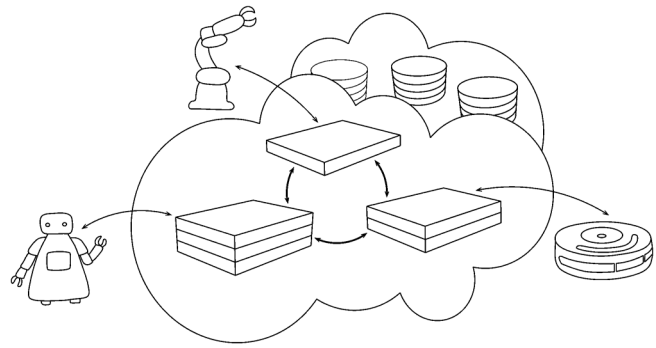

Fig. 1. Simplified overview of the Rapyuta framework. Each robot connected to Rapyuta has one or more secured computing environments (rectangular boxes), giving them the ability to move their heavy computation into the cloud. In addition, the computing environments are tightly interconnected with each other and have a high bandwidth connection to the RoboEarth [3] knowledge repository (stacked circular disks).

Furthermore, Rapyuta's computing environments provide high bandwidth access to the RoboEarth [3] knowledge repository, enabling robots to benefit from the experience of other robots. Note that until now robots directly submitted and queried data in the RoboEarth repository, and all of the processing, planning, and reasoning on this data happened locally on the robot. With Rapyuta, robots can perform these tasks in the cloud by having a corresponding software agent/clone. Thus, Rapyuta is also called the RoboEarth Cloud Engine. See Fig. 1 for a simplified overview of the architecture.

Rapyuta's ROS-compatible computing environments allow it to run almost all open-source ROS packages without any modifications while sidestepping the severe drawbacks of client-side robotics applications, including requirements for expensive and/or power-hungry hardware, configuration/setup overheads, dependence on custom middleware, as well as often failure-prone maintenance and updates. In addition to its out-of-the-box ROS compatibility, Rapyuta can also be customized for other robotics middlewares.

Finally, Rapyuta's WebSocket-based communication server provides bidirectional, full duplex communication with the physical robot. Note that this design choice also allows the server to initiate the communication and send data or commands to the robot.

The remainder of this paper is structured as follows. Taking a bottom-up approach we present each of the main components of the architecture individually along with our design choices in Section II and Rapyuta's communication protocols in Section III. Section IV returns to a general picture and presents several use cases that combine the previous components and the communication protocols in different ways to fit a variety of deployment scenarios. Then, performance and benchmarking results are presented in Section V. This is followed by two robotics demonstrations that highlight various aspects of Rapyuta in Section VI. We conclude in Section VII with a

[1]Note that this requires some workarounds to implement and the developer has to take care of the interprocess communication.

[2]Some allow developers to use general messaging frameworks, such as RabbitMQ, for communication between computing environments, but this a significant undertaking, if someone wants to build a messaging framework similar to ROS/Rapyuta.

[3]Rapyuta is part of the RoboEarth initiative aimed at building a world wide web for robots. Visit http://www.roboearth.org/ for details. The source code of Rapyuta is available at http://github.com/rapyuta/rce under Apache License, Version 2.0.

The name is inspired from the movie ***Tenku no Shiro Rapyuta*** (English title: Castle in the Sky) by Hayao Miyazaki, where Rapyuta is the castle in the sky inhabited by robots.

with a brief outlook on Rapyuta's future developments and the potential future of cloud robotics in general.

## II. Main Components

Rapyuta's four main components are: the computing environments onto which robots offload their tasks, a set of communication protocols, four core task sets to administer the system, and a command data structure to organize the system administration.

### A. Computing Environments

Rapyuta's computing environments are implemented using Linux Containers [18], which provide a lightweight and customizable solution for process separation, security, and scaling. In principle, Linux Containers can be thought of as an extended version of `chroot` [19], which isolates processes and system resources within a single host machine. Since Linux Containers do not emulate hardware (similar to platform virtualization technologies), and since all processes share the same kernel provided by the host, applications run at native speed.

Furthermore, Linux Containers also allow easy configuration of disk quotas, memory limits, I/O rate limits, and CPU quotas, which enables a single environment to be scaled up to fit the biggest machine instance of the IaaS [4] provider, or scaled down to simply relay data to the Hadoop [20] backend, similar to the DAvinCI [1] framework.

Each computing environment is set up to run any process that is a ROS node, and all processes within a single environment communicate with each other using the ROS inter-process communication. Having the well-established ROS protocol inside the environments allows them to run all existing ROS packages without any modifications and lowers the hurdle for application developers.

### B. Communication and Protocols

For the communication, three main solutions were considered and evaluated. The first solution, a bus-based communication approach, used a *bus* process in each host machine to which all of the other Rapyuta processes running in the host machine were connected. By connecting all of the *bus* processes, the bus was providing the functionality to route the internal messages to the corresponding destination. Although, this solution allowed a relatively easy control of the message flow, the resulting latencies were high. The second solution distinguished itself by directly connecting ROS nodes without any intermediary processes. While this variant resulted in the lowest latencies, it was relatively complex to maintain during runtime and challenging in terms of securing the communication. The third solution, which is currently implemented in Rapyuta, strikes a balance between the complexity of the system and the resulting latencies. This solution used *Endpoint* processes to define a clear *Interface* for communicating with external processes. The *Endpoints* are directly connected with each other using *Ports*. Fig. 2 shows these building blocks and the basic communication channels of Rapyuta.

*Interfaces* are used for communicating between a Rapyuta process and a non-Rapyuta process running either on the robot or in the computing environment. They provide a synchronous (service-based) or an asynchronous (topic-based) transport mechanism. *Interfaces* used for communicating with robots provide converters, which convert a data message from the internal communication format to a desired external communication format and vice versa. *Ports* are used for communicating between Rapyuta processes.

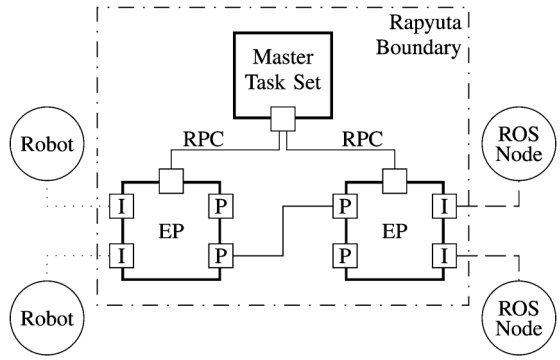


Fig. 2. Basic communication channels of Rapyuta. The *Endpoints* (EP) are connected to the *Master* task set using a two-way remote procedure call (RPC) protocol. Additionally, the *Endpoints* have *Interfaces* (I) for connections to robots or (ROS) nodes, as well as *Ports* (P) for communicating between *Endpoints*. The dotted lines represent the external communication, dashed lines represent the ROS-based communication between ROS nodes and Rapyuta, and, finally, all solid lines represent the internal communication between Rapyuta's processes.

The *Endpoints* allow to split the communication protocols into three parts. The first part is the internal communication protocol, which covers all communication between Rapytua's processes. The next part is the external communication protocol, which covers the data transfer between the physical robot and the cloud infrastructure running Rapyuta. The last part consists of the communication between Rapyuta and the applications running inside the containers. Each of these protocols are presented in more detail in Section III.

### C. Core Task Sets

Here, we present the four Rapyuta task sets that administer the system. A task set is a set of functionalities and one or more of these sets can be put together to run as a process depending on the use case (see Section IV).

*1) Master Task Set:* The *Master* task set is the main controller that monitors and maintains the *command* data structure, which includes:

- organization of connections between robots and Rapyuta;
- processing of all configuration requests from robots;
- monitoring the network of other task sets.

As opposed to the other task sets, only a single copy of the *Master* task set runs inside Rapyuta. Although running the *Master* task inside a single process creates a single point of failure, the *Master* task set is not duplicated or distributed in order keep the complexity of the system down. This drawback will be addressed in the upcoming design iterations of Rapyuta.

*2) Robot Task Set:* The robot task set is defined by the capabilities necessary to communicate with a robot. It includes:

- forwarding of configuration requests to the *Master*;
- conversion of data messages;
- communication with robots and other *Endpoints*.

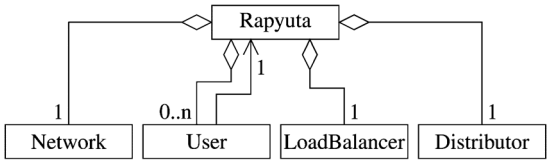


Fig. 3. Simplified UML diagram of Rapyuta's top level command data structure.

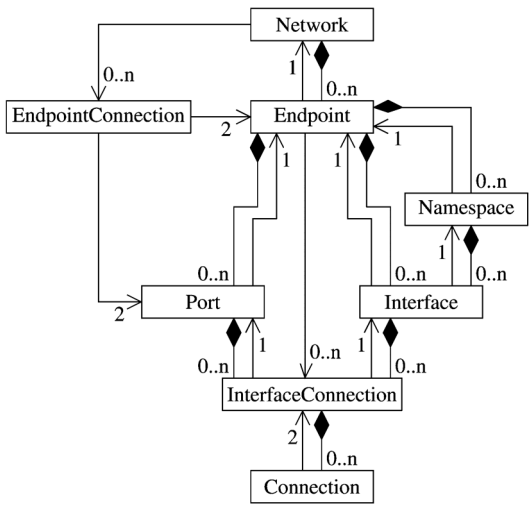


Fig. 4. Simplified UML diagram of Rapyuta's top level component *Network*.

*3) Environment Task Set:* The environment task set is defined by the capabilities necessary to communicate with a computing environment. It includes:

- communication with ROS nodes and other *Endpoints*;
- launching/stopping ROS nodes;
- adding/removing parameters.

A process containing the environment task set runs inside every computing environment.

*4) Container Task Set:* The container task set is defined by the capabilities necessary to start/stop computing environments. A process containing the container task set runs inside every machine.

### D. Command Data Structure

Rapyuta is organized in a centralized command data structure. This data structure is managed by the *Master* task set and it consists of the four components shown in Fig. 3.

The *Network* (see Fig. 4) is the most complex part of the data structure. Its elements are used to provide the basic abstraction of the whole platform and are referenced by the *User*, *LoadBalancer*, and *Distributor* components. The *Network* is also used to organize the internal and external communication, which will be discussed in detail in Section III. The addition of *Namespaces* in the command data structure enables an *Endpoint* to group *Interfaces* of a single robot or a computing environment and the addition of the connection classes (*EndpointConnection*, *InterfaceConnection*, and *Connection*) simplifies the reference counting for the connections.

The *User* (see Fig. 5) generally represents a human who has one or more robots that need to be connected to the cloud. Each

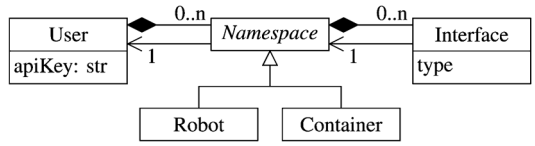


Fig. 5. Simplified UML diagram of Rapyuta's top-level component *User*.

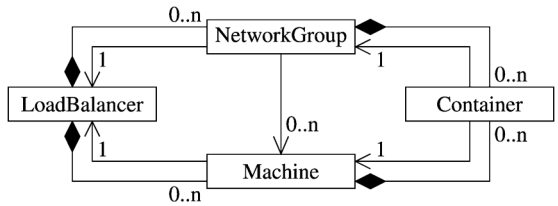


Fig. 6. Simplified UML diagram of Rapyuta's top-level component *LoadBalancer*.

*User* has a unique API key, which is used by the robots for authentication. The *User* can have multiple *Namespaces* which is an abstracted representation of a container or robot. A *Namespace*, in turn, can have several *Interfaces*.

The *LoadBalancer* (see Fig. 6) is used to manage the *Machines* which are intended to run the computing environments. To allow these computing environments to communicate directly with each other without Rapyuta (see Section III-E), the computing environments can be added to a *NetworkGroup*. Therefore, the *NetworkGroups* have a representation of each *Container* included in the group and references to the participating *Machines*. Similarly, the *Machines* have a reference of each *Container* they are running. Additionally, the *LoadBalancer* is used to assign new containers to the appropriate machine.

Finally, the *Distributor* is used to distribute incoming connections from robots between available robot *Endpoints*.

## III. Communication Protocols

Here, we present Rapyuta's internal and external communication protocols in more detail. The third protocol, as mentioned in Section II-A, is based on ROS [6] and is not covered in this paper and the reader is referred to the ROS documentation.

### A. Internal Communication Protocol

All Rapyuta processes communicate with each other over UNIX sockets and the protocol is built using the Twisted framework [21], an event-driven networking engine that uses asynchronous messaging. The type of messages used for the internal communication can be split into two categories. The first type consists of all administrative messages used to configure Rapyuta. All of these messages either originate or end in the *Master* process (runs the *Master* task set) containing the command data structure. The Perspective Broker, a two-way RPC[4] implementation for the Twisted framework, is used as the protocol for administrative messages. The second and the most frequent type is the data protocol. For this type of communication, a length prefixed protocol is used. The content of a data message

[4]RPC (Remote Procedure Call) is a communication protocol that allows a process to execute a procedure in another process.

is a serialized ROS message. For Rapyuta, an additional header containing the ID of the sending *Interface*, an optional destination ID (necessary for service type interfaces), and the message ID (which is used also for the external communication) is added. This results in a header length of 22 or 38 bytes plus the message ID, which has a length upper bounded by 255 bytes.

### B. External Communication Protocol

The robots connect to Rapyuta using the WebSockets protocol [22], similar to rosbridge [15]. The protocol was implemented using the Autobahn tools [23], which also runs on top of the Twisted framework [21]. Unlike a common web server, which uses pull technology, the use of WebSockets allows Rapyuta to push results. Note that this protocol is very general compared to the ROS protocol used in the DAvinCI [1] framework, allowing easy integration of non-ROS robots, mobile devices and even web browsers into the system.

The messages between the robot and Rapyuta are pure ASCII JSON[5] messages that have the following top level structure:

```
{"type":"...", "data": ...},
```

which is an unordered collection of key/value pairs. Note that a value can, in turn, be a collection of key/values pairs. The value of the `type` key is a string and denotes the type of message found in `data`:

| | |
|---|---|
| CC | Create container message creates a secure computing environment in the cloud. |
| DC | Destroy container message destroys an existing computing environment. |
| CN | Configure components message enables the launching/stopping of ROS nodes, the setting/removal of parameters in the ROS parameter server, and the adding/removal of *Interfaces*. |
| CX | Configure connections message enables the connection/disconnection of *Interfaces*. |
| DM | Data messages are used to send/receive any kind of messages to/from application nodes (for more examples see Section IV-B). |
| ST | Status messages are pushed from Rapyuta to the robot. |
| ER | Error messages are also pushed from Rapyuta to the robot. |

### C. Handling Large Binary Messages

The WebSocket interface supports transportation of binary blobs and, for some types of data, it is better to transport them as a binary blob instead of using their corresponding ROS message type encoded as a JSON string. For example, the RoboEarth logo (RGBA, 842 $\times$ 595), if transported as PNG (lossless data compression), takes 18 kB in bandwidth but uses approximately 2.0 MB when transported as a serialized ROS message. Converting the ROS message into a JSON string would result in an even larger message size.

To exploit this method of transportation, special converters between the binary format and the corresponding ROS message must be provided on the Rapyuta's *interface* side. Rapyuta provides a default PNG-to-sensor_msgs/Image converter as an example of how to build new converters.

When sending a binary message, first a standard data message is sent as a JSON string with a reference to the binary blob that will follow. The message is a DM type message having a `data` key with value:

```
"iTag": "converter_modifyImage",
"type": "sensor_msgs/Image",
"msgID": "msgID_0",
"msg*": "f9612e9b3c7945ef8643f9f590f0033a"
```

The '$*$' in the last line indicates that the value/resource will follow as a binary blob with the given ID as header. Note that the ID must be unique only within the current connection.

### D. Communication With RoboEarth

By default, every container has a `py_re_comm`[6] node running inside it. This ROS node exposes the RoboEarth repository by providing services to download, upload, update, delete, and query action recipes, object models, and environments stored in the RoboEarth repository. Since the RoboEarth repository is also typically hosted in the same data center, all applications running on Rapyuta have high bandwidth access to the data using the wired networks, in contrast to applications running on board the robot with wireless connectivity over the Internet.

### E. Virtual Networks

As described in Section III-A, processes running in different computing environments (containers) communicate through Rapyuta's internal communication protocol built on top of ROS. However, some applications that are distributed over multiple containers, may require a less abstracted version of the network to use different protocols such as Open MPI [24]. Containers within a common host could communicate using the LXC bridge, which is the default network interface/bridge (layer 2 or data link layer of the OSI model[25]) for containers. However, the LXC bridges of different host machines cannot be connected directly. Therefore, the current version of Rapyuta includes the functionality to create a virtual network with an arbitrary topology between containers that belong to a specific user. The virtual network is realized using Open vSwitch [26], which is connected to an additional network interface of the container. Open vSwitch was selected mainly due to its high (production) quality, openness (licensed under Apache 2.0 similar to Rapyuta), and scalability with almost no compromise in usability (compared to the standard Linux bridge). See benchmarking results in Section V-C for comparisons between the virtual networks and Rapyuta's internal communication protocol.

[5]JSON (JavaScript Object Notation) is a lightweight data-interchange format with a focus on human readability.

[6]See http://github.com/rapyuta/re_comm_core for more details.

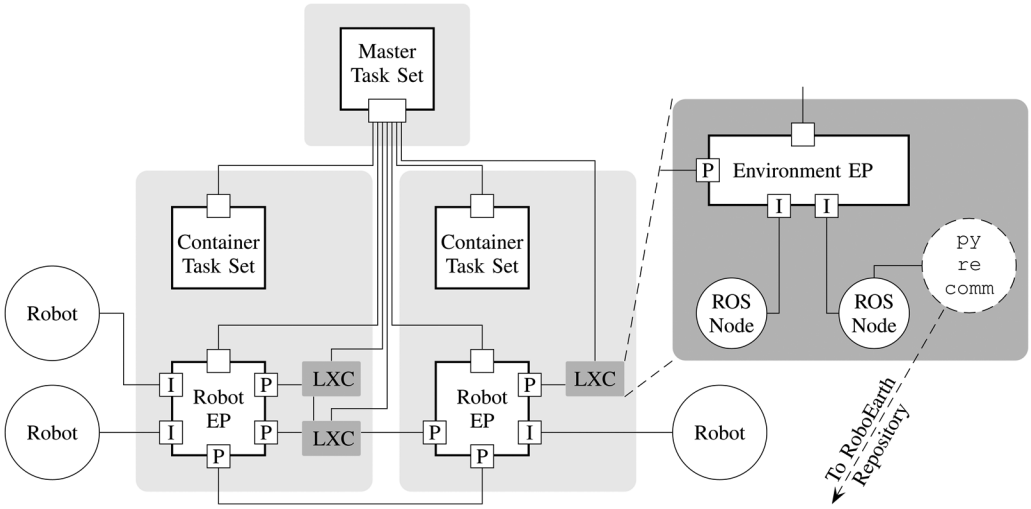


Fig. 7. Use Case 1. The typical use case of Rapyuta processes deployed on three machines (light-gray blocks) to build a PaaS framework with interconnected computing environments (LXC, dark-gray blocks). Here the *Master* task set runs as a single process on one of the machines and the other two machines are used to deploy containers. Inside each machine that hosts containers, the robot task set runs as a single process, and inside each container the environment task set runs as a single process. The computing environment denoted by LXC (Linux Containers) is enlarged in the right side of the figure. Note that the dashed arrow from the `py_re_comm` node denotes the connection to the RoboEarth knowledge repository within the same cluster/data center, thus providing a high-bandwidth access.

## IV. Deployment

The core components and communication protocols described in the previous sections can be combined in different ways to meet the specifications of a robotic scenario. Here, we present three typical use cases, a basic example of the communication process, and some useful tools.

### A. Use Cases

Fig. 7 shows the standard use case[7] where the four task sets are split up into the four processes (the *Master* process, *RobotEndpoint* process, *EnvironmentEndpoint* process, and the *Container* process), and combined with interconnected computing environments to build a PaaS framework. The *Master* process runs on a single dedicated machine. Other machines each run both a *RobotEndpoint* and a *Container* process. The two task sets are run separately, since the *Container* process requires super user privileges to start and stop containers which could pose a severe security risk when combined with the open accessible *RobotEndpoint* process. The fourth process, the *EnvironmentEndpoint* process, is running inside every computing environment. Note that this configuration allows all three elastic computing models to be deployed for cloud robotics, as proposed in [16]; the peer-based, proxy-based, and the clone-based model. From an administrative point of view, the standard use case can be deployed in the following ways:

- Private cloud: Rapyuta, the applications running on it, and the robots belong to a single entity. This is better suited for some commercial entities where trust and security is the highest concern.
- Software-as-a-service: Rapyuta and the applications running on it belong to a single entity, and several users connect and use the applications. This allows the single entity to better protect its intellectual property, keep the software up to date, and provide better support.

[7]See http://rapyuta.org/install and http://rapyuta.org/usage for more details on the setup and usage of the standard use case.

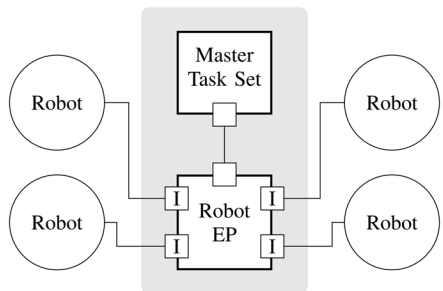


Fig. 8. Use Case 3. Process configuration for setting up a network of robots running a *RobotEndpoint* and *Master* process in a single machine (light-gray block).

- Platform-as-a-service: Here, only the Rapyuta platform is managed by a single entity, while a community of developers develop and share/host the applications. In addition to the advantages stated above, this allows for easy benchmarking and ranking of various solutions to robotics.

The second use case is an extreme case of the standard use case where everything runs on a single machine with one container. This mimics a rosbridge [15] system and can be used as a sandbox to develop cloud robotics applications and investigate latencies.

Finally, the third use case presented in Fig. 8 shows how to set up a network of robots using the *RobotEndpoint* and *Master* processes. Although Fig. 8 shows a single machine, multiple machines with interconnected *RobotEndpoint* processes are also feasible. Although this use case is similar to the ROS multimaster implementations [27] in terms of functionality, Rapyuta has several advantages such as connections over nonlocal networks and fine-grained control over which topics and services of a ROS system are available for another ROS system. However, if all robots are connected to a local network and if no malicious activity is expected from any of the robots, a ROS multimaster implementation would be easier to setup and debug.

Note that the machines mentioned in all three use cases (light-gray blocks in Figs. 7 and 8) can also be instances of an IaaS [4] provider such as Amazon EC2 [28] or Rackspace [29].

### *B. Basic Communication Example*

In order to illustrate the usage and communication protocols, this subsection provides a simple example of a communication process with Rapyuta's standard use case setup (see Fig. 7). Here, a Roomba vacuum cleaning robot with a wireless connection uses Rapyuta to record/log its 2D pose. The communication takes place in the following order:

*1) Initialization:* The first step for the Roomba is to contact the process running the *Master* task set using the user ID `roombaOwner` to get the address of a *RobotEndpoint*. This is done with the following HTTP request:

```
http://[domain]:[port]?userID=roombaOwner
&version=[version]
```

A *RobotEndpoint* is selected of the available *Endpoints* and the *Endpoint*'s URL is returned to the Roomba as a JSON encoded response.

```
{
    "url":"ws://[domain]:[port]/"
}
```

In the second step of initialization, Roomba makes a connection using the received URL of the assigned *RobotEndpoint*, registers using the robot ID `roomba`, and logs in using the API key `secret`. This is done with the following HTTP request:

```
ws://[domain]:[port]/?userID=roombaOwner
&robotID=roomba&key=secret
```

*2) Container Creation:* The Roomba creates a computing environment and tags it with a CC-type message having a `data` key with value:

```
"containerTag": "roombaClone"
```

Note that the tag must be unique within the robots that use the same user ID. Container creation also automatically starts the necessary Rapyuta processes inside the container.

*3) Configure Nodes:* The Roomba launches the logging node (posRecorder.py) and starts two *Interfaces* with tags using a CN-type message having a `data` key with value:

```
"addNodes": [{
        "containerTag": "roombaClone",
        "nodeTag": "positionRecorder",
        "pkg": "testPkg",
        "exe": "posRecorder.py"
    }],
"addInterfaces": [{
        "endpointTag": "roomba",
        "interfaceTag": "pos",
        "interfaceType":
        "SubscriberConverter",
        "className": "geometry_msgs/Pose2D"
    }, {
        "endpointTag": "roombaClone",
        "interfaceTag": "pos",
        "interfaceType":
        "PublisherInterface",
        "className": "geometry_msgs/Pose2D",
        "addr": "/posPub"
    }]
```

Note that the above complex message can be split into multiple messages that launches the node and start *Interfaces* separately.

*4) Binding Interfaces:* Before the Roomba can use the added node the two *Interfaces* must be connected. This is achieved with a CX-type message having a `data` key with value:

```
"connect": [{
        "tagA": "roomba/pos",
        "tagB": "roombaClone/pos"
    }]
```

*5) Data:* Finally, the Roomba starts sending the data message that contains the 2-D pose information, i.e., a DM-type message having a `data` key with value:

```
"iTag": "pos",
"type": "geometry_msgs/Pose2D",
"msgID": "id",
"msg": {
        "x": 3.57,
        "y": -44.5,
        "theta": 0.581
    }
```

This data message (ASCII JSON) is converted to a ROS message at the *Interface* `roomba/pos` and is sent to the *Interface* `roombaClone/pos`. The *Interface* `roombaClone/pos` then transfers the message to the `posRecorder.py` node via the ROS environment. Now, for example, by adding *Interfaces* of type `PublisherConverter` and `SubscriberInterface`, a topic can also be transferred from the nodes running in `RoombaClone` to the robot `roomba`.

### *C. Tools*

Managing a cloud-based environment is a complex and cumbersome task. To simplify the management of Rapyuta, a console client for administrators and users is provided. This tool allows users to monitor Rapyuta's components based on their privileges and to interact with Rapyuta similar to the external protocol described in Section III-B.[8]

[8]For more details on the console client, see http://rapyuta.org/Console

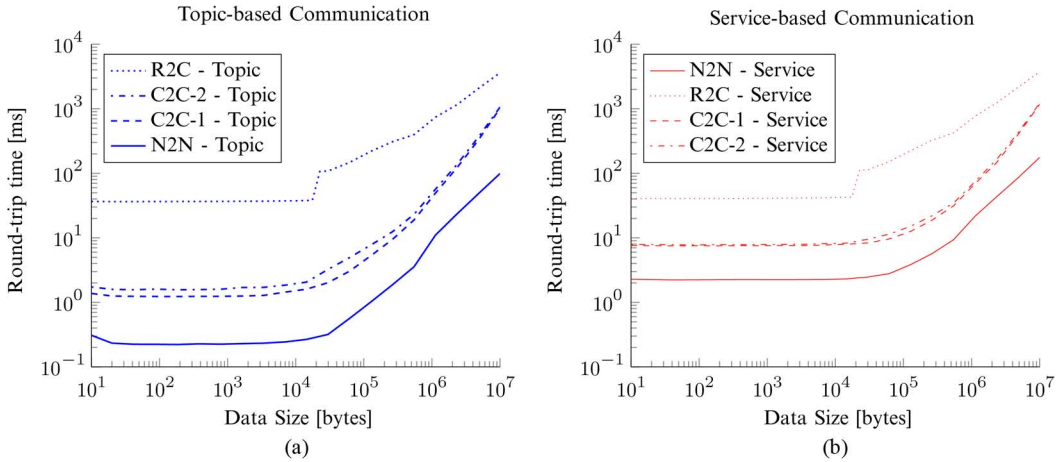


Fig. 9. Round-trip times (RTTs) of different transport mechanisms and data routes in the standard use case (see Fig. 7). N2N denotes the communication of two processes (nodes) within the same ROS environment inside a container; C2C denotes two processes in two different containers, where for C2C-1 the containers are running on the same machine and for C2C-2 on different host machines. Finally R2C denotes the communication between a remote process and a process running inside Rapyuta's containers. (a) RTTs for different data routes in the standard use case (see Fig. 7) under the topic transport mechanism. (b) RTTs for different data routes in a standard use case (see Fig. 7) under the service transport mechanism.

Setting up Rapyuta can be the first and the biggest hurdle for a beginner. To address this issue, Rapyuta provides a provisioning script for users who want to setup and use Rapyuta in their own hardware. The script sets up the full system, including the networking, containers, and their file system. This provisioning script is compatible with almost all of the recent Ubuntu (12.04,12.10,13.04) and ROS (Fuerte, Groovy) variants at the time of writing. For Amazon EC2 users, Rapyuta provides an Amazon Machine Image (AMI) with the latest stable version, which they can copy and start using within minutes.[9]

## V. Performance and Benchmarking

Here, we provide various performance measures of Rapyuta under different configurations and provide benchmarking results with rosbridge. All experiments were conducted by measuring the round-trip times (RTTs) of different sized messages between two processes. Note that all experiments are a variation of: where the two processes were running, their communication route, and the transport mechanism. A topic-based and a service-based transport mechanism were used. For each experiment, 25 message sizes (log-uniformly distributed between 10 B and 10 MB) were selected and, for each size, 20 samples were measured. All experiments, except for the remote process/robot case, were performed on a machine instance in Amazon's Ireland data center. A machine located at ETH Zurich, Switzerland was used to replicate the remote process/robot.

In addition to the transmission delays (message size/bandwidth), round-trip times also contain other factors such as queuing time, processing time and propagation delay (distance/propagation speed). For smaller messages, the influence of these other factors diminish the effects of transmission delays, giving a flat RTT for message of sizes up to 10 kB. After 10 kB, the transmission delays start to dominate resulting in an almost linear increase in RTTs with respect to the message size (after a brief transient phase).

[9]For more details on the setup tools for installation, see http://rapyuta.org/Install

For interpretation and comparison, note that a tf-typed message that contains the relationship between multiple coordinate frames of the robot discussed in Section VI is around 100 B; whereas an RGB-D frame, which contains a PNG compressed RGB and depth image in VGA resolution, is around 500 kB.

### A. Rapyuta Core

In this part, we compare RTTs for all of the core data routes of the standard use case as shown in Fig. 7. The results of the experiments are shown in Fig. 9(a) for the topic-based transport mechanism and Fig. 9(b) for the service-based transport mechanism. Since a new connection must be established for every service call that is made, services show a higher latency than topics. However, the trends with respect to RTTs of different routes remain the same under both transport mechanisms.

The results in Fig. 9(a) and (b) show the following.

- Communication with an external process (R2C) is the biggest constraint of Rapyuta's throughput.[10]
- The difference between containers running in the same machine (C2C-1) and different machines (C2C-2), resulting from `iptables`' port forwarding overheads, can be neglected in comparison to the difference to the communication between two nodes in the same ROS environment (N2N).
- Rapyuta introduces an overhead of $<2$ ms for topics and 5 ms for services for data sizes up to 10 kB (see Fig. 9(a)), which can be seen from the differences between C2C-1 and N2N.

### B. Rosbridge

Here, we compare Rapyuta with rosbridge with respect to RTTs. Fig. 10(a) and (b) show RTTs for communication with an external non-Rapyuta process that runs on the same machine

[10]The abrupt increase of RTTs for data sizes between $10^4$ and $10^5$ bytes is due to an artifact of Amazon's networking infrastructure at the time of writing. This is reproducible even with a basic server using a normal TCP socket with a straightforward echo protocol.

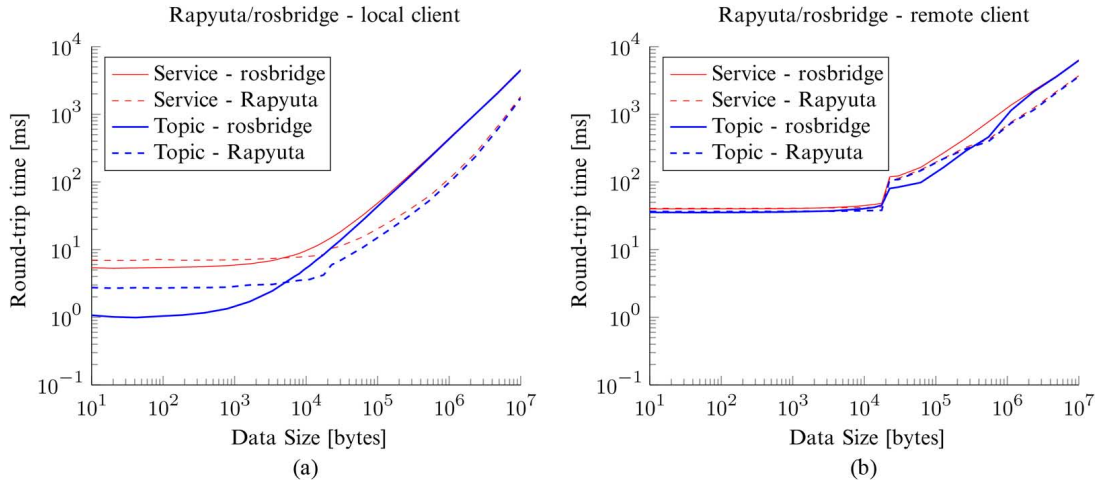


Fig. 10. RTTs of different transport mechanisms and data routes in Rapyuta and rosbridge [15]. (a) RTTs with the external process running on the same host machine that is running Rapyuta/rosbridge. (b) RTTs with a remote external process running at ETH Zurich, Switzerland while Rapyuta and rosbridge ran on Amazon's Ireland data center.

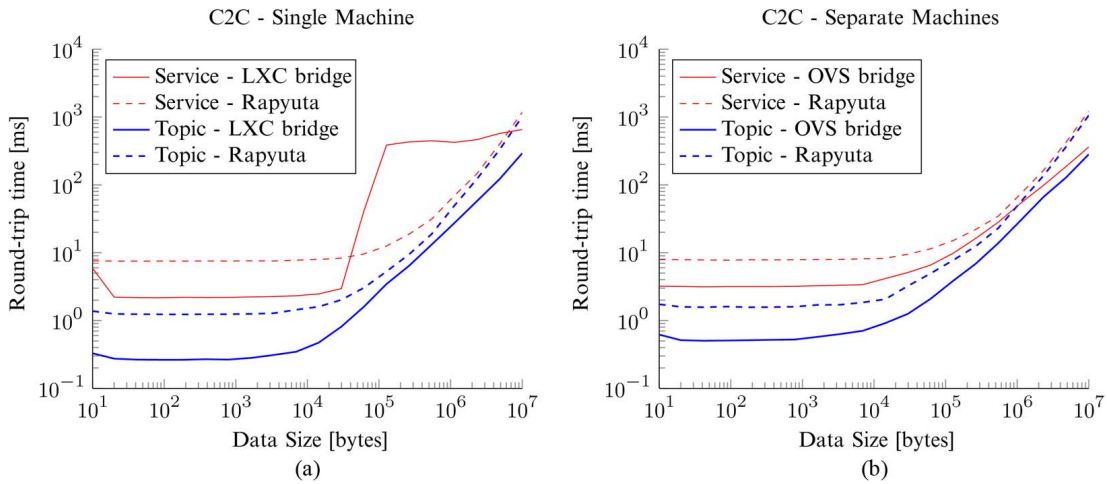


Fig. 11. RTTs for communication between containers under different transport mechanisms. (a) RTTs with LXC bridge-based virtual networks and Rapyuta. The two containers that were running the communication processes were hosted by a single host machine. (b) RTTs with Open vSwitch [26]-based virtual networks and Rapyuta. The two containers that were running the communication processes were hosted separately on two host machines.

where the framework (Rapyuta/rosbridge) is running and on a remote machine, respectively.

For external processes running on the same machine [Fig. 10(a)], RTTs are dominated by the queuing and processing times. For small message sizes, both services and topics show lower RTTs for rosbridge compared to Rapyuta. Conversely, Rapyuta shows lower RTTs than rosbridge for larger message sizes. The larger RTTs of Rapyuta for small message sizes is due to the fact that the message has to pass through two endpoints (processes) whereas in the case of rosbridge the message only has to pass through a single process. For larger messages the implementation of the transport becomes the dominating factor. Due to the fact that Rapyuta's WebSocket library (Autobahn [23] version 0.6.0) has a better performance compared to rosbridge's WebSocket library (tornado [30] version 2.3),[11] the RTTs for larger message sizes of Rapyuta are smaller.

For remote external processes [Fig. 10(b)], RTTs are dominated by the transmission delays of the Internet connection. These transmission delays diminish rosbridge's advantage over small message, and results in a similar performance between Rapyuta and rosbridge for smaller messages. For larger messages (10–300 kB), rosbridge RTTs are lower for topics compared to Rapyuta, but higher for services. For message sizes larger than 500 kB, Rapyuta offers lower RTTs. This is again due to the better performance of Rapyuta's WebSocket library.

### C. Virtual Network-Based Internal Communication

Here, we compare the virtual networks discussed in Section III-E to Rapyuta's internal communication protocol. Note that all virtual network-based experiments between two containers had one single ROS master managing the communication as opposed to one ROS master per container in a typical Rapyuta use case. Fig. 11(a) and (b) compare the communication between two containers under the virtual network using ROS and Rapyuta's internal communication protocol. Except for LXC bridge's artifact for service calls with larger messages, the virtual networks are superior to Rapyuta's internal communication protocol. Note that this comes at the cost of losing security and encapsulation, and is thus only allowed within a single user's computing environments.

[11]This fact was verified by using a straightforward echo protocol.

Fig. 12. Two low-cost (∼600$) robots used in our demonstrations: Each robot consists mainly of a differential drive base (iRobot Create), an RGB-D sensor (PrimeSense), an ARM-based single board computer (ODROID-U2), and a dual-band USB wireless device.

## VI. Demonstrations

In addition to the tutorial applications that come with the source code, we also developed two typical robotic applications to highlight various aspects of Rapyuta.

### A. Demonstration: Cloud-Based Mapping

As a first proof-of-concept demonstration, we implemented a cloud-based collaborative mapping service in Rapyuta. The robots shown in Fig. 12 consist mainly of off-the-shelf components. They use the iRobot Create as their base, and this differential drive base provides a serial interface to send control commands and receive sensor information. PrimeSense CARMIN 1.08 is used for the RGB-D sensing. A 48 × 52 mm embedded board with a smartphone-class multi-core ARM processor is used for onboard computation. The embedded board runs a standard Linux operating system and connects to the cloud through a dual-band USB wireless device. This hardware setup was used with different software configurations to highlight different aspects of the challenges and opportunities.

*1) Complete Offloading:* In this setup, almost all of the processing was moved to Rapyuta and the only computation done on board was the compression of the RGB-D data. At ROSCON 2013 (Stuttgart, Germany) this setup was demonstrated by doing a frame-by-frame compression with Rapyuta running at Amazon's data center in Ireland, which resulted in a 5-MB/s throughput at 30 Hz in QVGA resolution. This setup demonstrated the practical limits of off-the-shelf wireless devices and helped us stress test Rapyuta in terms of throughput. Under this frame-by-frame compression, any resolution higher than the QVGA resulted in dropped frames due to the bandwidth limitation. For more effective ways of sending visual and depth information, see Sections VI-A-2 and VI-B.

*2) Local Visual Odometry:* In this setup,[12] a dense visual odometry algorithm [31] was run on board the robot, and only the RGB-D key frames were sent to Rapyuta for the global optimization. This setup used a bandwidth of 300–500 kB/s with QVGA-resolution key frames and demonstrated a good trade off between the data rates and computation given the wireless speed, and the robots' speed and available computation. Furthermore, this setup also had a component running on Rapyuta that directed the robot to its next exploration point based on predefined set points. This component can be extended to automatically provide set points based on the map being built.

*3) Collaborative Mapping:* In this setup, multiple robots were used to collaboratively build a 3-D map of an environment. This setup demonstrated that the cloud can serve not only as a computational resource, but also as a common medium for collaborative tasks. A 3-D model of an environment created by this method is shown in Fig. 13.

For more details and quantitative evaluations on Sections VI-A-2 and VI-A-3, see [32].

[12]The source code of the dense visual odometry-based mapping demonstration is available at http://github.com/rapyuta/rapyuta-mapping under the Apache 2.0 license.

(a)

(b)

Fig. 13. Point cloud map of a room at ETH Zurich built in real-time by the two robots shown in Fig. 12. The individual maps generated by the two robots are merged and optimized in a process running on the cloud. The robots are re-localized and the robot model is over-layed in the merged map. (a) Top view. (b) Side view with a photograph taken in a similar perspective.

### B. Demonstration: Dense Mapping With Rapyuta

As a second demonstration, we implemented a dense mapping service, which compressed and streamed all of the RGB-D data at VGA resolution to the GPU processes running on Rapyuta. This demonstration used the point cloud library's (PCL) [33] implementation of the KinectFusion [34] algorithm,

Fig. 14. Frame of the KinectFusion output running on an Amazon GPU instance in Ireland.

which does not rely on keyframes as in the previous demonstration, but utilizes all images and all of their pixels to reconstruct the map. This prevents the mapping algorithm from being split up (as suggested previously) to reduce the necessary bandwidth, and is therefore used as an example for an application where a large amount of data must be exchanged. Libav [35] (an open-source video and audio processing library) was used to compress the RGB-D stream as two separate video streams. For this demonstration, we compressed the color images using the common compression scheme h264. For the depth images we used the FFV1 video codec, since FFV1 natively supports the lossless compression of a 16-bit monochrome image stream, whereas h264 does not support this type of pixel format.[13]

Fig. 14 shows a sample frame of the KinectFusion output. The bandwidth usage of the RGB images in this particular case was 1.1 MB/s for a dynamic scene and 800 kB/s for a static scene. Compared with 27.6 MB/s (which is the bandwidth requirement of raw RGB images), the compression results in a reduction of 96%. Similarly, the bandwidth required for the depth images was 1.6 MB/s for a dynamic scene and 1.4 MB/s for a static scene. This resulted in a reduction of 92%, compared with the 18.4 MB/s requirement of the raw depth images. Note that all values are for a frame rate of 30 frames per second. The second notable aspect of this demonstration is that it uses the GPU to process the incoming data in a timely manner. This allows us to demonstrate that although the algorithm runs isolated in a virtual machine, the container is flexible enough to allow direct access to the GPU, a hardware component of the host machine.

Mapping was specifically selected due its high data rate, which is an important constraint in cloud robotics. Other tasks that can benefit from the cloud and that are closer to manufacturing include grasp planning [2] and object recognition [36]. Both tasks are computationally expensive, highly parallelizable, and require a significant amount of storage.

To build a reasonable alternative to Rapyuta (to run the above cloud-based application) a developer would have to:

- provision servers from an IaaS provider;
- connect the servers together with some technology like Open vSwitch [26] (see Section III-E) and common ROS master/multimaster setup [27] based on the application requirements;
- connect the external robots to the servers using rosbridge [15].

However, compared to Rapyuta, the developer will lose the generality (architecture has to be manually changed based on the application), scalability, and security (multiple users can use this system only under the assumption of complete trust) with this alternative.[14]

## VII. Conclusion and Outlook

In this paper, we described the design, implementation, benchmarking results, and the first demonstrations of Rapyuta, a PaaS framework for robots. Rapyuta, based on an elastic computing model, dynamically allocates secured computing environments for robots.

We showed how the computing environments and the communication protocols allow robots to easily offload their computation to the cloud. We also described how Rapyuta's computing environments can be interconnected to share specific resources with other environments, making it a suitable framework for multi-robot control and coordination.

Our choice of communication protocols were explained, and an example was provided to clarify the different types of messages and to show how they work together. With respect to communication, we also provided some benchmarking results for different protocols.

Next, we showed the flexibility of Rapyuta's modular design by giving three specific use cases as a guide. Finally, two robotics demonstrations were presented as examples to highlight various aspects of Rapyuta and cloud robotics in general. These demonstrations also provided practical examples on how to handle application-specific tradeoffs between available onboard computation, the application's bandwidth and real-time requirements, the reliability of communication with the cloud, and the available cloud infrastructure.

Together with the RoboEarth Knowledge Repository, Rapyuta provides an appropriate cloud robotics platform that has the potential to improve robotics in the following ways.

- It provides massively parallel computation on demand for sample-based statistical modelling and motion planning [37].
- It leverages the cloud as a real-time communication medium for collaborative task performance and information sharing.
- It serves as a global repository to store and share object models, environment maps, and actions recipes between various robotic platforms; enabling life-long learning.
- The robotic application-as-a-service eliminates setup and update overhead for the end users, serves as a better model for protecting intellectual property for the developers, and functions as a common platform to benchmark different algorithms and solutions.

[13]The source code of the depth mapping demonstration can be found at http://github.com/rapyuta/rapyuta-kinfu. under the Apache 2.0 license.

[14]There is also the option of modifying an existing open source PaaS framework for web applications to run robotics application. This is a significant undertaking, and we do not recommend it.

- It allows humans to monitor or intervene and help robots when they are lost; this not only makes the robotic system more robust but also provides a lot of labelled data to learn from as an intermediary step before humans are taken out of the loop.

Many more applications can be found in the field of intelligent transportation, environmental monitoring, smart homes, and defence [16].

## ACKNOWLEDGMENT

The authors would like to express their gratitude towards V. Usenko, TUM, for help building the mapping demonstration, D. Sathe and M. Singh for their help with the software development, C. Flores and C. Waibel for helping with the promotional video, M. Ciocarlie, Willow Garage, for the excellent feedback on the draft, and all RoboEarth colleagues for their support and feedback.

## REFERENCES

[1] R. Arumugam, V. R. Enti, K. Baskaran, and A. S. Kumar, "DAvinCi: A cloud computing framework for service robots," in *Proc. IEEE Int. Conf. Robot. Autom.*, May 2010, pp. 3084–3089.
[2] B. Kehoe, D. Berenson, and K. Goldberg, "Toward cloud-based grasping with uncertainty in shape: Estimating lower bounds on achieving force closure with zero-slip push grasps," in *Proc. IEEE Int. Conf. Robot. Autom.*, May 2012, pp. 576–583.
[3] M. Waibel, M. Beetz, J. Civera, R. D'Andrea, J. Elfring, D. Galvez-Lopez, K. Haussermann, R. Janssen, J. Montiel, A. Perzylo, B. Schiessle, M. Tenorth, O. Zweigle, and R. van de Molengraft, "Roboearth," *IEEE Robot. Autom. Mag.*, vol. 18, no. 2, pp. 69–82, Jun. 2011.
[4] P. Mell and T. Grance, "The NIST Definition of cloud computing," Nat. Inst. of Standards and Technol., Special Publication 800-145, 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf
[5] Google, Inc., Google App Engine, 2014. [Online]. Available: https://developers.google.com/appengine/
[6] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: An open-source Robot Operating System," in *Proc. ICRA Workshop on Open Source Software*, 2009.
[7] J. Lindenbaum, A. Wiggins, and O. Henry, "Heroku," 2007. [Online]. Available: http://www.heroku.com/
[8] *"Cloud Foundry,"* VMware, Inc., 2013. [Online]. Available: http://www.cloudfoundry.com/
[9] *"OpenShift,"* Red Hat, Inc, 2013. [Online]. Available: https://openshift.com/
[10] , K. Goldberg and R. Siegwart, Eds., *Beyond webcams: An introduction to online robots*. Cambridge, MA, USA: MIT, 2002.
[11] M. Inaba, S. Kagami, F. Kanehiro, Y. Hoshino, and H. Inoue, "A platform for robotics research based on the remote-brained robot approach.," *Int. J. Robot. Res.*, vol. 19, no. 10, pp. 933–954, 2000.
[12] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. Cambridge, MA, USA: MIT, 2005.
[13] K. Kamei, S. Nishio, N. Hagita, and M. Sato, "Cloud networked robotics," *IEEE Network*, vol. 26, no. 3, pp. 28–34, May–Jun. 2012.
[14] M. Sato, K. Kamei, S. Nishio, and N. Hagita, "The ubiquitous network robot platform: Common platform for continuous daily robotic services," in *Proc. IEEE/SICE Int. Symp. Syst. Integr.*, Dec. 2011, pp. 318–323.
[15] C. Crick, G. Jay, S. Osentoski, and O. C. Jenkins, "ROS and rosbridge: Roboticists out of the loop," in *Proc. Annu. ACM/IEEE Int. Conf. Human–Robot Interaction*, 2012, pp. 493–494.
[16] G. Hu, W. P. Tay, and Y. Wen, "Cloud robotics: Architecture, challenges and applications," *IEEE Network*, vol. 26, no. 3, pp. 21–28, May–Jun. 2012.
[17] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," *AI Mag.*, vol. 29, no. 1, pp. 9–20, 2008.
[18] Linux Containers, 2012. [Online]. Available: http://lxc.sourceforge.net/
[19] *"Linux Programmer's Manual,"* Chroot, 2012. [Online]. Available: http://www.kernel.org/doc/man-pages/online/pages/man2/chroot.2.html
[20] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.
[21] G. Lefkowitz, "Twisted," 2012. [Online]. Available: http://twistedmatrix.com/
[22] I. Fette and A. Melnikov, "The WebSocket Protocol," RFC 6455, 2011. [Online]. Available: http://tools.ietf.org/html/rfc6455
[23] Tavendo GmbH, "Autobahn WebSockets," 2014. [Online]. Available: http://autobahn.ws/
[24] Community Project, "Open MPI: Open Source High Performance Computing," 2013. [Online]. Available: http://www.open-mpi.org/
[25] *Information Technology—Open Systems Interconnection—Basic Reference Model: The Basic Model*, ISO/IEC, Nov. 1994.
[26] Community Project, "Open vSwitch," 2013. [Online]. Available: http://openvswitch.org/
[27] ROS Community Project, "Multimaster Special Interest Group (SIG)," 2014. [Online]. Available: http://wiki.ros.org/sig/Multimaster/
[28] Amazon.com Inc., "Amazon Elastic Compute Cloud," 2014. [Online]. Available: http://aws.amazon.com/ec2
[29] Rackspace US, Inc., "The Rackspace Open Cloud," 2012. [Online]. Available: http://www.rackspace.com/
[30] Facebook, Inc., "Tornado Framework," 2014. [Online]. Available: https://github.com/facebook/tornado
[31] F. Steinbrucker, J. Sturm, and D. Cremers, "Real-time visual odometry from dense RGB-D images," in *Proc. ICCV Comput. Vision Workshop*, 2011, pp. 719–722.
[32] G. Mohanarajah, V. Usenko, M. Singh, M. Waibel, and R. D'Andrea, "Cloud-based collaborative 3D mapping in real-time with low-cost robots," *IEEE Trans. Autom. Sci. Eng.* vol. 12, no. 2, pp. 423–431, Apr. 2015.
[33] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, Shanghai, China, May 9–13, 2011, pp. 1–4.
[34] R. A. Newcombe, A. J. Davison, S. Izadi, P. Kohli, O. Hilliges, J. Shotton, D. Molyneaux, S. Hodges, D. Kim, and A. Fitzgibbon, "Kinectfusion: Real-time dense surface mapping and tracking," in *Proc. 10th IEEE Int. Symp. Mixed and Augmented Reality*, 2011, pp. 127–136.
[35] libav, 2013. [Online]. Available: http://libav.org/
[36] W. Zhu, C. Luo, J. Wang, and S. Li, "Multimedia cloud computing," *IEEE Signal Process. Mag.*, vol. 28, no. 3, pp. 59–69, May 2011.
[37] USA Robotics VO, "A Roadmap for U.S. Robotics, From Internet to Robotics, 2013 Edition," 2013.

**Gajamohan Mohanarajah** received the B.S. and M.S. degrees from Tokyo Institute of Technology, Tokyo, Japan. He is currently working toward the Ph.D. degree at ETH Zurich supervised by Prof. Raffaello D'Andrea and co-supervised by Prof. Andreas Krause.

His current interests include cloud robotics, controls, and large-scale machine learning.

**Dominique Hunziker** received the B.S. degree in mechanical engineering from ETH Zurich, Zurich, Switzerland, in 2011, where he is currently working toward the M.S. degree.

His current interests include cloud robotics, Python development, and piloting helicopters.

**Raffaello D'Andrea** received the B.Sc. degree in engineering science from the University of Toronto, Toronto, ON, Canada, in 1991, and the M.S. and Ph.D. degrees in electrical engineering from the California Institute of Technology, Pasadena, CA, USA, in 1992 and 1997, respectively.

He held positions as an Assistant Professor and later an Associate Professor with Cornell University, Ithaca, NY, USA, from 1997 to 2007. He is currently a Full Professor of dynamic systems and control with ETH Zurich, Zurich, Switzerland, and Technical Co-founder and Chief Technical Advisor with Kiva Systems.

**Markus Waibel** received the M.Sc. degree in physics from the Technical University of Vienna, Vienna, Austria, in 2003, and the Ph.D. degree in robotics from the EPF Lausanne, Lausanne, Switzerland, in 2007.

He is currently a Senior Researcher with ETH Zurich, Zurich, Switzerland, and Program Manager of the cloud robotics project RoboEarth. He is also the Cofounder of the ROBOTS Association and its flagship publications *Robohub* and the ROBOTS Podcast, and the founder of Robotics by Invitation, an online panel discussion of 30 high-profile roboticists.