Rapyuta: The RoboEarth Cloud Engine

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

***Abstract*— In this paper we present the design and implementation of Rapyuta[1], the RoboEarth Cloud Engine. Rapyuta is an open source Platform-as-a-Service (PaaS) framework designed specifically for robotics applications. Rapyuta helps robots to offload heavy computation by providing secured customizable computing environments in the cloud. The computing environments also allow 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 specific use case configurations and present some performance results.**

I. INTRODUCTION

The past decade has seen robots begin to move from factories into household-like environments, folding towels [1] and serving drinks [2]. The demand for these service robots is predicted to grow steadily in the coming decades [3]. But moving from half a century of comfort in structured factory environments to highly unstructured, non-deterministic household environments presents the robotics community with several challenges.

Computing power is a key enabler for solving some of these challenges. However, on-board computation entails additional power requirements, which may constrain robot mobility, reduce operating duration, and increase costs. The computational burden of a service robot can be reduced by offloading those tasks that do not have hard real time requirements to a cloud computing infrastructure [4], [5]. These tasks include grasp planning [6], [7], mapping [8], and navigation. The rapid increase in mobile data transfer rates [9] makes more and more robotics tasks feasible for execution in the cloud.

Running robotics applications in the cloud falls into the Platform-as-a-Service (PaaS) model [10] 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. Unfortunately differences between web and robotics applications make it hard to use existing web solutions for robotics. Such differences include programming languages, the number of processes (robotics applications contain multiple processes while web applications are typically single processes), and the communication protocols (a request/response based stateless model is sufficient for most web applications, while most robotics applications require servers with stateful protocols to push information asynchronously to the robot). For example, a general PaaS platform such as the popular Google App Engine [11] is not well suited for robotics applications since it exposes only a limited subset of program APIs required for a web application, allows only a single process, and does not expose sockets, which are indispensable for robotic middlewares such as ROS [12]. Although Heroku [13] provides some freedom in the programming APIs and sockets, it does not allow the server to push data to the robot, and has no networking between the computing environments to allow robots to share information.

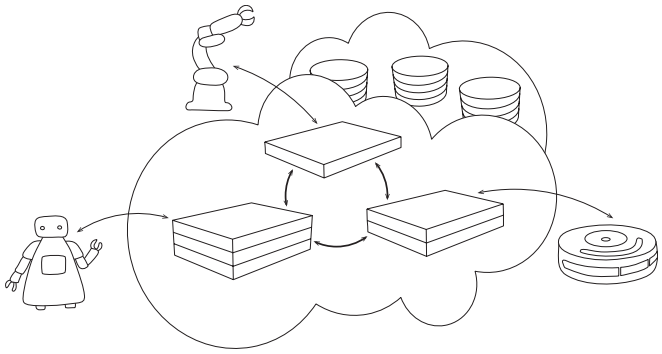

Fig. 1. Simplified overview of the Rapyuta framework: Each robot connected to Rapyuta has a secured computing environment (rectangular boxes) giving them the ability to move their heavy computation into the cloud. The computing environments are tightly interconnected with each other and have a high bandwidth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

During the past few years, first efforts to build a cloud computing framework for robotics have emerged. The DAvinCi Project [8] 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 [14]. It offered a single computing environment without process separation or security, while the interprocess communication was 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) [15], [16], an on-going project, focuses on using the cloud as a medium for establishing a network between robots, sensors, and mobile devices. The project also makes a significant contribution to the standardization of data-structures and interfaces. Finally, rosbridge [17], an open source project, focuses on the external communication between a robot and a single ROS environment in the cloud.

The authors are with the Institute for Dynamic Systems and Control, Swiss Federal Institute of Technology Zürich, Switzerland. The contact author is M. Gajamohan, e-mail: gajan@ethz.ch

[1]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 the open source[2] 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 [4] that dynamically allocates secure computing environments (or clones) for robots. These computing environments are tightly interconnected, allowing robots to share some of their services and information with the other robots. This interconnection makes Rapyuta a useful platform for multi-robot deployments [18].

Furthermore, Rapyuta's computing environments provide high bandwidth access to the RoboEarth [19] knowledge repository enabling robots to benefit from the experience of other robots. Note that until now robots only submitted and queried data in the RoboEarth repository, and all the processing, planning, and reasoning on this data happened locally on the robot. With Rapyuta, robots can perform these tasks in the cloud. Thus, Rapyuta is also called the RoboEarth Cloud Engine.

Rapyuta's ROS compatible computing environments allow it to run all, which is more than 3000, open source ROS packages without any modifications while sidestepping severe drawbacks of client-side robotics applications, including configuration/setup overheads, dependence on custom middleware, as well as maintenance and update overheads. Finally, Rapyuta's WebSocket-based communication server provides bidirectional, full duplex communications with the physical robot.

The remainder of this paper is structured as follows: In Section II we present the architecture with design choices and in Section III we give a detailed explanation of Rapyuta's communication protocols. This is followed by details on some standard use case configurations and performance measurements in Section IV. Finally, we conclude in Section V with a brief outlook on future developments.

II. ARCHITECTURE AND DESIGN CHOICES

Rapyuta consists mainly of the computing environments for robots to offload their tasks, a set of communication protocols, a set of core tasks 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 [20], which provide a lightweight and customizable solution. In principle, Linux Containers can be thought of as an extended version of `chroot` [21] which provides isolation of processes and system resources within a single host. 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.

[2]A public alpha release of Rapyuta is available from http://github.com/IDSCETHZurich/rce under Apache License, Version 2.0.

[3]Rapyuta is part of the RoboEarth initiative aimed at building a world wide web for robots. Visit http://www.roboearth.org/ for details.

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 [10] provider, or scaled down to just relay data to the Hadoop backend similar to the DAvinCI [8] framework.

The 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 interprocess communication. Having the well-established ROS protocol inside the environments allows them to run all existing ROS packages without any modification, and lowers the hurdle for application developers.

B. Communication Protocols

Rapyuta's communication protocols are split into three parts. The first part is the *internal communication protocol* that covers all communication between the Rapytua processes. The next part is the *external communication protocol* which defines the data transfer between the physical robot and the cloud infrastructure running Rapyuta. The last part is the communication between Rapyuta and the applications running inside the containers, which uses ROS as outlined above.

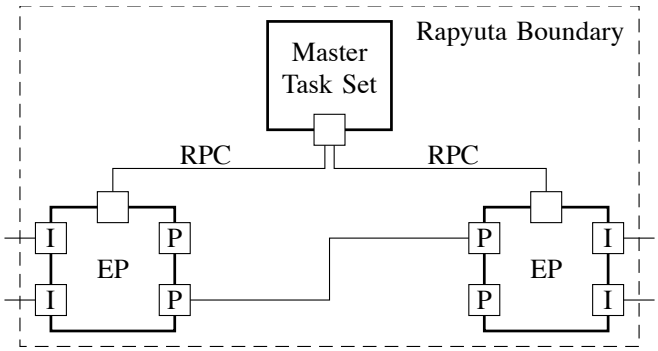

Fig. 2. The basic communication channels of Rapyuta: The *Endpoints* (EP) are connected to the master using a Remote Procedure Call (RPC) protocol. Additionally, the *Endpoints* have *Interfaces* (I) for connections to robots or computing environments as well as *Ports* (P) for communication between *Endpoints*.

Figure 2 shows the basic communication channels of Rapyuta. One of the basic building blocks of Rapyuta's communication protocol is the *Endpoint* which represents a process that consists of *Ports* and *Interfaces*.

The *Interfaces* are used for external communication with a non-Rapyuta process that can either be running on the robot or in the computing environment, and they implement an abstract class that can represent a service-provider, service-client, topic-publisher or a topic-subscriber. *Interfaces* of container *Endpoints* are standard ROS interfaces. Meanwhile, *Interfaces* of robot *Endpoints* provide converters, which convert a data message from the external communication format (JSON[4] object) to the internal communication format (serialized ROS message) and vice versa.

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

Meanwhile, the *Ports* are used for internal communication between *Endpoint* processes. Another form of internal communication, based on RPC[5], occurs between the *Endpoints* and the Master, the main controller process. Section III presents Rapyuta's communication protocols in detail.

C. Core Tasks

This sub-section presents Rapyuta's four task sets.

1) Master: The master is the main controller task that monitors and maintains the command data structure which includes

- organization of connection between robots and Rapyuta
- processing of all configuration requests from robots
- monitoring network of other task sets

As opposed to the other task sets, only a single copy of the Master task set runs inside a Rapyuta platform.

2) Robot: 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*

3) Environment: 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: 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 with four components (see Figure 3).

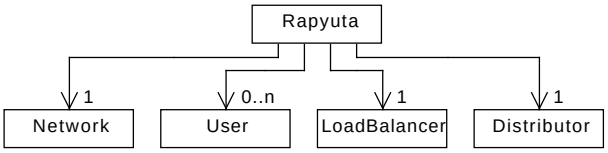

Fig. 3. Simplified UML diagram of Rapyuta's top level command data structure.

The *Network* (see Figure 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 other three components. It 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* for 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.

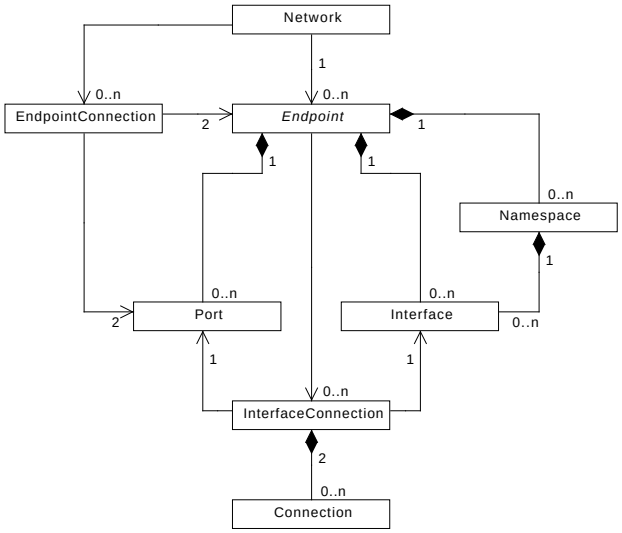

Fig. 4. Simplified UML diagram of Rapyuta's top level component *Network*.

The *User* (see Figure 5) represents generally a human who has one or more robots that need to be connected to the cloud. Each *User* has a unique API key, which is used by the robots for authentication. Additionally, all requests are initially processed by the *User* and handed off to another component for further processing if necessary. The *User* can have multiple *Namespaces* which, in turn, can have several *Interfaces*.

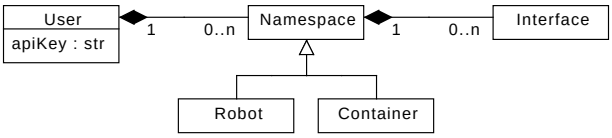

Fig. 5. Simplified UML diagram of Rapyuta's top level component *User*.

The *LoadBalancer* (see Figure 6) is used to manage the *Machines* which are intended to run the computing environments discussed in Section II-A. Therefore the *Machines* have a representation of each *Container* they are running. Additionally, the load balancer is used to assign new containers to the appropriate machine.

Finally, the *Distributor* is used to distribute the incoming connections from the robots over the available robot *Endpoints* which are discussed in detail in Section III.

III. COMMUNICATION PROTOCOLS

This section presents Rapyuta's internal and external communication protocols in more detail and shows a concrete

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

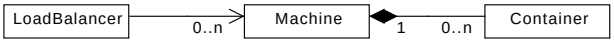

Fig. 6. Simplified UML diagram of Rapyuta's top level component *Load Balancer*.

example of the external communication procedure.

A. Internal Communication Protocol

All Rapyuta processes communicate with each other over UNIX sockets and the protocol is built using the Twisted framework [22], 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 the administrative messages used to configure Rapyuta. All these messages either originate or end in the process containing the command data structure, which is typically the Master task set. The *Perspective Broker*, an RPC implementation for the Twisted framework, is used as the protocol for the administrative messages. The second and the most frequent type is the data message. For this type of communication a length prefixed protocol is used. The data message itself is internally transported as 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 whose length has an upper limit of 255 bytes.

B. External Communication Protocol

The robots connect to Rapyuta using the WebSockets protocol [23], similar to rosbridge [17]. The protocol was implemented using the Autobahn tools [24], which, in turn, are again based on the Twisted framework [22]. Unlike a common web server, which uses pull technology, the use of WebSockets allows Rapyuta to push the results. Note that this protocol is very general compared to the ROS protocol used in the DAvinCI [8] 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 messages that have the following top level structure

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

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

- CC - Create Container message that creates a secure computing environment in the cloud
- DC - Destroy Container message destroys an existing computing environment
- CN - Configure Components message allows to launch/stop ROS nodes, set/remove parameters in the ROS parameter server, and add/remove *Interfaces*
- CX - Configure Connections message allows to connect/disconnect *Interfaces*
- DM - Data messages are used to send/receive sensor/-command messages to/from application nodes (for more examples see Section III-D)
- ST - Status message pushed from Rapyuta to the robot
- ER - Error message also pushed from Rapyuta to the robot

C. Handling Big 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, 842x595), 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 corresponding ROS message must be provided on the Rapyuta 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. Basic Communication Example

In order to illustrate the usage and communication protocols, in this subsection we provide a simple example where 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: Using the user ID *roombaOwner*, the robot ID *roomba*, and the API key *secret*, the Roomba performs the initialization by sending the following HTTP request to the master containing the command data structure:

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

A robot *Endpoint* is assigned on an available machine and the machine's URL together with an authentication key is returned to the Roomba as a JSON encoded response.

```
{
  "url":"ws://[domain]:[port]/",
  "key":"8f42eeedefb0463a834c582782a9e2bc"
}
```

As the final step of the initialization process, Roomba makes a connection using the received URL and registers with the assigned robot *Endpoint* using the following URL.

```
ws://[domain]:[port]/?userID=roombaOwner&robotID=
    roomba&key=8f42eeedefb0463a834c582782a9e2bc
```

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 and container creation also includes starting the necessary 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 Roomba can use the added node the two *Interfaces* have to 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 2D-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 type at *roomba/pos* and is sent to *roombaClone/-pos*. Finally, *roombaClone/pos* transfers the message to the *posRecorder.py* node via the ROS environment.

E. Communication with RoboEarth

After creating a container, Rapyuta launches the `re_comm_core`[6] node inside the container by default. This 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.

IV. USE CASES AND PERFORMANCE

A. Use cases

Figure 7 shows the four tasks sets (see Section II-C) split up into the four processes, Master process, Robot EP (Endpoint) process, Environment EP process, and Container process, and how they are combined to build a PaaS framework with interconnected computing units. The Master process runs on a single dedicated server. In every other machine a Robot EP and a Container process are running. The two task sets are run separately, because 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 Robot EP process. The fourth process, the Environment EP process, is running in each computing environment. Note that this configuration allows all three elastic computing models to be deployed for cloud robotics, as proposed in [4], the peer-based, proxy-based, and the clone-based model.

An extreme case of the above use case is where everything runs in a single server with one container. This mimics a rosbridge [17] system and can be used as a sandbox to develop cloud robotics applications and investigate latencies.

Finally, in Figure 8 we show how to set up a network of robots using the Robot EP and Master processes. Although Figure 8 shows a single server, multiple machines with interconnected Robot EP processes are also feasible.

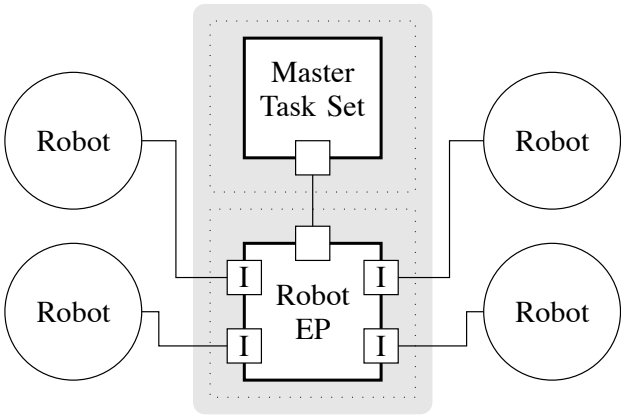

Fig. 8. Use case 2: Process configuration for setting up a network of robots running a Robot EP and Master process in a single server (light-gray block).

Note that the servers mentioned in both use cases (light-gray blocks in Figures 7 and 8) can also be instances

[6] `re_comm_core` is a stripped down version of the `re_comm` ROS package containing only the communications aspects. `re_comm_core` was provided to us by our RoboEarth colleague Alexander Perzylo of Technical University Munich.

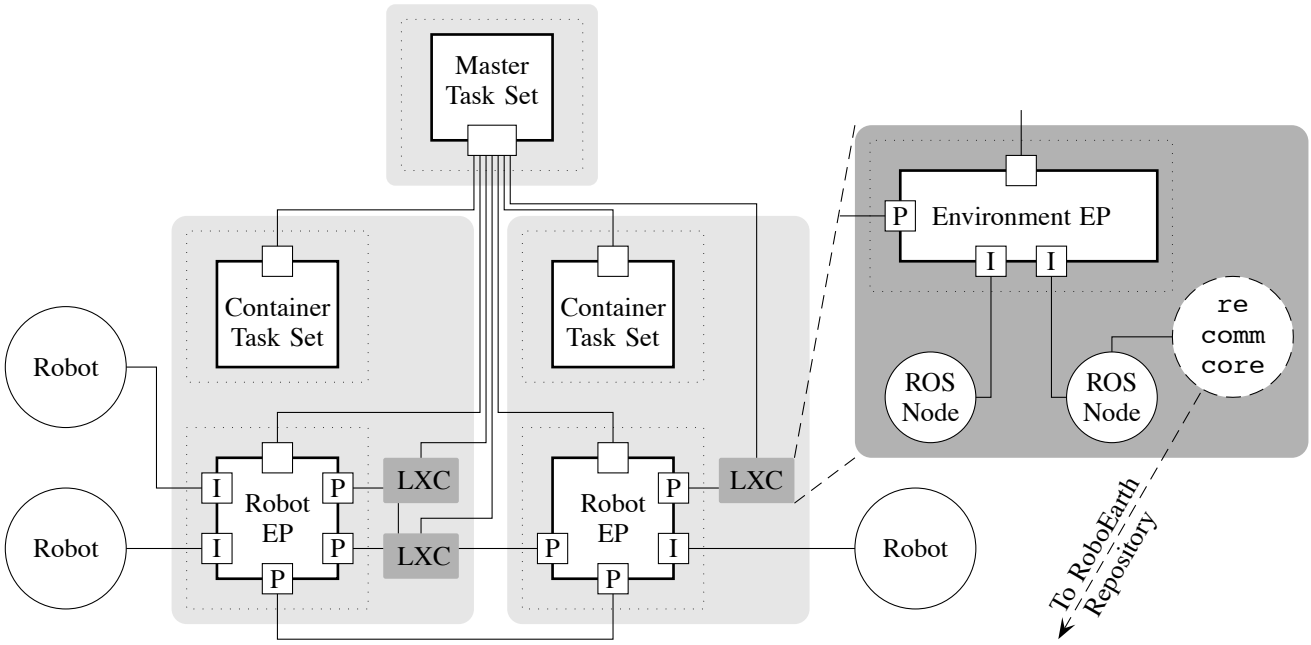

Fig. 7. Use case 1: The typical use case of Rapyuta processes (see Section IV-A) deployed on three servers (light-gray blocks) to build a PaaS framework with interconnected computing environments (LXC, dark-gray blocks). Here the master task set runs as a single process on one of the servers and the other two servers are used to deploy containers. Inside each server that hosts containers, robot task set run as a single process and inside each container the environment task set run 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 `re_comm_core` node denotes the connection to the RoboEarth knowledge repository within the same cluster/data center, thus providing a high bandwidth access.

of an IaaS [10] provider such as Amazon EC2 [25] or Rackspace [26].

B. Performance Measurement

In this subsection we present experimental results for round-trip times of a Rapyuta instance running in two machines. The following four communications paths were analyzed for 10 data sizes.

- R2C: Robot to Container
- C2C-1: Container to Container where both containers are hosted by the same machine
- C2C-2: Container to Container where the containers are hosted by different machines
- N2N: Communication path between two nodes running in the same container (baseline)

To evaluate the round-trip times, a string message with different string sizes was passed between two processes. Note that here the robot was a Python client with a wireless connection in ETH Zurich, and Rapyuta was deployed in RackSpace [26] servers located in Dallas, Texas.

The results in Figure 9 show:

1) External communication (R2C) is the biggest constraint of Rapyuta's throughput.
2) The difference between containers running in the same machine and different machines, due to the iptables and port forwarding overheads, is relatively small (< 1 [ms]).
3) Rapyuta introduces an overhead of 1 [ms] for data sizes up to 1 MB, which can be seen from the differences between C2C-1 and N2N.

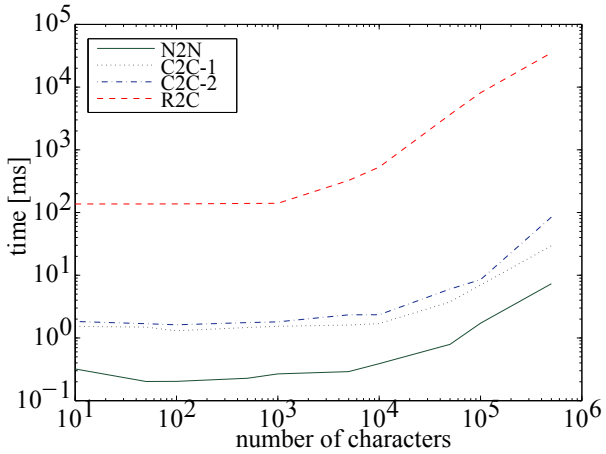

Fig. 9. Round-trip times (RTT) along four different type of paths in the Rapyuta framework. The External communication has the largest RTT. For container to container communications, the RTT is slightly higher when containers are hosted in different machines. Finally, the RTT within a container under the ROS protocol is given as a baseline.

V. CONCLUSION AND OUTLOOK

In this paper we described the design and implementation of Rapyuta, a PaaS framework for robots. Rapyuta, based on an elastic computing model [4], dynamically allocates secured computing environments for robots.

We showed how the computing environments and the communications protocols allow the robots to 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. Furthermore, we explained how the computing environment exposes the RoboEarth database to the applications running inside it.

Communication protocols were presented with comments on the design choices and an example was provided to clarify different types of messages and to show how they work together. With respect to communications we also provided some performance results of the communication protocols.

Finally, we showed the flexibility of Rapyuta's modular design by giving two specific cloud robotic configurations.

A public alpha release is available from http://github.com/IDSCETHZurich/rce under Apache License, Version 2.0. Additionally, clients in Python and C++ are provided.

For the planned beta release several enhancements are being considered. To increase the stability of Rapyuta the command data structure (see Section II-D) will be stored in a redundant manner such that a crash of the Master process does not result in a complete failure of Rapyuta. As a second enhancement besides the JSON encoded data structure used in the external communication protocol we plan to support Google's protocol buffers for flexibility and efficiency [27]. Furthermore, the possibility of using protocol buffers for internal communications will be investigated. Finally, we plan to look into the integration of software frameworks other than ROS into Rapyuta.

ACKNOWLEDGMENTS

This research was funded by the European Union Seventh Framework Programme FP7/2007-2013 under grant agreement no. 248942 RoboEarth. The authors would like to express their gratitude towards Carolina Flores and Christine Waibel for helping with the promotional video, Alexander Perzylo of Technical University Munich for providing the `re_comm_core` package, and all RoboEarth colleagues for their valuable feedback and motivation.

REFERENCES

[1] J. Maitin-Shepard, M. Cusumano-Towner, J. Lei, and P. Abbeel, "Cloth grasp point detection based on multiple-view geometric cues with application to robotic towel folding," in *Proc. IEEE Int. Conf. Robotics and Automation*, May 2010, pp. 2308 –2315.

[2] J. Bohren, R. Rusu, E. Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mosenlechner, W. Meeussen, and S. Holzer, "Towards autonomous robotic butlers: Lessons learned with the pr2," in *Proc. IEEE Int. Conf. Robotics and Automation*, May 2011, pp. 5568 –5575.

[3] International Federation of Robotics, "World robotics - service robots 2011," 2011.

[4] G. Hu, W. P. Tay, and Y. Wen, "Cloud robotics: architecture, challenges and applications," *Network, IEEE*, vol. 26, no. 3, pp. 21–28, May-June 2012.

[5] K. Goldberg and B. Kehoe, "Cloud robotics and automation: A survey of related work," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2013-5, Jan 2013. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-5.html

[6] 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. Robotics and Automation*. IEEE, May 2012, pp. 576–583.

[7] B. Kehoe, A. Matsukawa, S. Candido, J. Kuffner, and K. Goldberg, "Cloud-Based Robot Grasping with the Google Object Recognition Engine," in *IEEE International Conference on Robotics and Automation*. IEEE, May 2013.

[8] R. Arumugam, V. R. Enti, K. Baskaran, and A. S. Kumar, "DAvinCi: A cloud computing framework for service robots," in *Proc. IEEE Int. Conf. Robotics and Automation*. IEEE, May 2010, pp. 3084–3089.

[9] L. Garber, "Wi-fi races into a faster future," *Computer*, vol. 45, no. 3, pp. 13 –16, march 2012.

[10] P. Mell and T. Grance, "The NIST definition of cloud computing," National Institute of Standards and Technology, Special Publication 800-145, 2011, available http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[11] Google, "Google App Engine," 2008. [Online]. Available: https://developers.google.com/appengine/

[12] 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 *ICRA Workshop on Open Source Software*, 2009.

[13] J. Lindenbaum, A. Wiggins, and O. Henry, "Heroku," 2007. [Online]. Available: http://www.heroku.com/

[14] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[15] K. Kamei, S. Nishio, N. Hagita, and M. Sato, "Cloud Networked Robotics," *Network, IEEE*, vol. 26, no. 3, pp. 28–34, May-June 2012.

[16] M. Sato, K. Kamei, S. Nishio, and N. Hagita, "The ubiquitous network robot platform: Common platform for continuous daily robotic services," in *System Integration (SII), 2011 IEEE/SICE Int. Symp.*, Dec 2011, pp. 318 –323.

[17] G. T. Jay, "brown_remotelab: rosbridge," 2012. [Online]. Available: http://www.rosbridge.org/

[18] R. D'Andrea and P. Wurman, "Future challenges of coordinating hundreds of autonomous vehicles in distribution facilities," in *Technologies for Practical Robot Applications, 2008, IEEE Int. Conf.*, nov. 2008, pp. 80 –83.

[19] 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," *Robotics Automation Mag., IEEE*, vol. 18, no. 2, pp. 69 –82, june 2011.

[20] "Linux Containers," 2012. [Online]. Available: http://lxc.sourceforge.net/

[21] "chroot, Linux programmer's manual," 2012. [Online]. Available: http://www.kernel.org/doc/man-pages/online/pages/man2/chroot.2.html

[22] G. Lefkowitz, "Twisted," 2012. [Online]. Available: http://twistedmatrix.com/

[23] I. Fette and A. Melnikov, "The WebSocket Protocol, RFC 6455," 2011. [Online]. Available: http://tools.ietf.org/html/rfc6455

[24] Tavendo GmbH, "Autobahn WebSockets," 2012. [Online]. Available: http://autobahn.ws/

[25] Amazon.com Inc., "Amazon Elastic Compute Cloud," 2012. [Online]. Available: http://aws.amazon.com/ec2/am

[26] Rackspace US, Inc., "The Rackspace Open Cloud," 2012. [Online]. Available: http://www.rackspace.com/

[27] Google Inc., "protocol buffers," 2012. [Online]. Available: http://developers.google.com/protocol-buffers