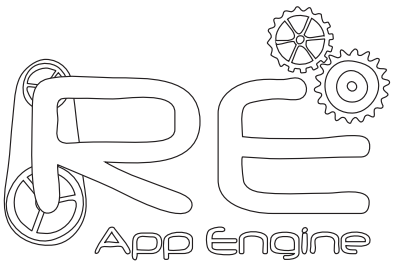

RoboEarth App Engine

Dominique Hunziker, Mohanarajah Gajamohan, Markus Waibel

Zurich, February 29, 2012

Introduction

Computational power is a key enabler for intelligent and efficient robot task performance. However, on-board computation entails additional power requirements, which may constrains robot mobility and operating duration and increases costs.

The RoboEarth App Engine makes powerful computation available to robots. It allows users to run their ROS software in the Cloud in a Software as a Service framework with minimal configuration. In addition, keeping the open source and RoboEarth spirit, robots can share compatible 'cloud-apps'.

The RoboEarth App Engine takes advantage of the rapid increase in mobile data transfer rates provided by the upcoming LTS standard (down-link peak rates of 300 Mbit/s, up-link peak rates of 75 Mbit/s) and the growing number of repositories and packages (3350 pkgs at the time of count) available under the ROS framework.

In addition, it sidesteps severe drawbacks of client-side robot apps, including high computational costs, configuration/setup overheads, dependence on custom middleware, as well as maintenance and update overheads.

RoboEarth App Engine Framework

The RoboEarth App Engine Framework allows users to create, select, and use apps. The apps interface of the framework acts as an interpreter between user requests and their apps. It allows communication with processes using ROS, process management using *roslaunch* and has a standardized language to communicate between processes. In this first implementation we focused on the usage of *nodes* which provide *services* as apps.

The client interface of the framework communicates with app users. To include a broader audience the interaction uses the standard HTTP protocol. This allows users who do not have ROS and opens the door to using apps from a web browser by using the framework as a web server. For this first implementation the focus is on the robots as users. For the API the implementation *django/piston* was used.

Requirements

Apps have to be ROS *nodes* providing a *service* which can be used by the user. No further restrictions are necessary.

For the communication between the user and the framework the HTTP protocol (request types GET, POST and DELETE) is used. To further distinguish and control the interaction dynamic URL are used. This leads basically to three levels, i.e. *Service*, *Environment* and *Task*, which results in the following URL pattern

```
BASE/api/reappengine/[envID/[taskID/[fileRef/]]]
```

The level *Service* represents the entry point for a new user. The first step is to create a new environment which essentially creates a new *namespace* in ROS. Also on this level the user can add or remove *nodes* from his environment and access general information.

On the next level the user can create a task which is essentially the input message for the selected ROS *service*. Again, he can get some information specific to his current environment.

And on the last level the user can retrieve the status or, if the task has been completed successfully, the result of a task. The forth level *fileRef* is only used to provide the possibility to download referenced files for the tasks result, e.g. an image file. The result is basically the response message of the selected service.

To exchange the data needed for all these requests the JSON format is used. Since the requests which create a new environment/node or task are implemented as a POST, it is also possible to send files. To simplify the construction of requests and the interpretation of the responses a small Python-based ServiceAPI is available. For a graphical representation of these interactions refer to figure 1.

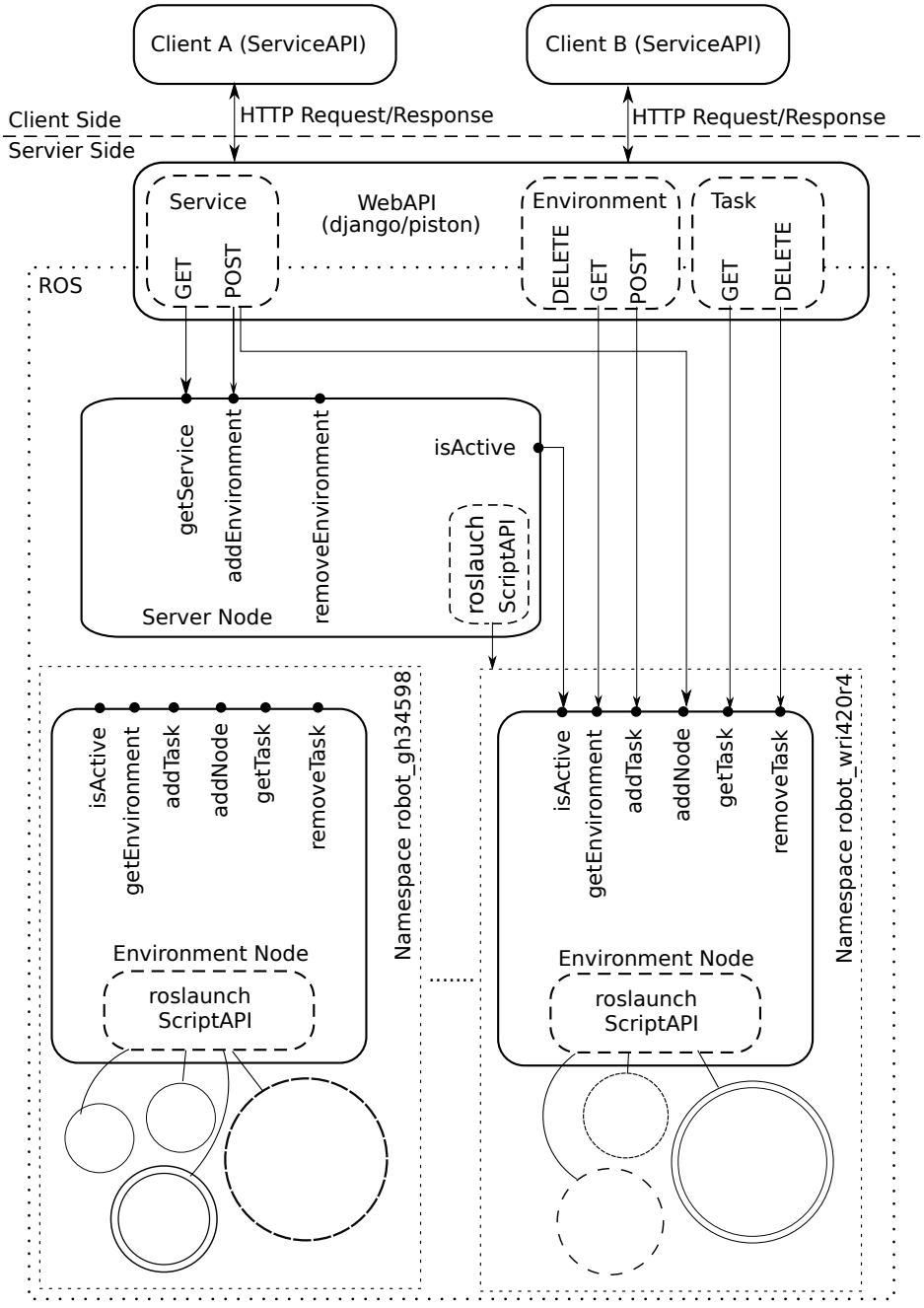

Figure 1: Organization of the framework (loosely from top to bottom): The client, which can use the provided *ServiceAPI*, communicates with the *DjangoInterface* over HTTP. The *DjangoInterface* then uses the necessary *services* of the *ServerNode* or *EnvironmentNode* (solid arrows). The *EnvironmentNode* is created and managed by the *ServerNode* (dashed arrow). The user selected app-*nodes* are then launched and managed by the *EnvironmentNode* (dashed arrow) which also provides the necessary *services* to create tasks and retrieve their results (solid arrow).