## Motivation

Although sufficient computational power enables robots to perform tasks in an intelligent and efficient manner, high on-board computation demands extra power and constrains mobility of the robot.

Motivated by 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 we propose [re app engine], a Software as a Service framework, where the user, the robot, can upload and run its ROS software in the Cloud with almost zero changes. In addition, keeping the open source spirit robots, can also share compatible 'cloud-apps' between them.

An app store concept, where a robot downloads an app and runs locally, has several disadvantages: 1. Computational Cost 2. Configuration/setup overhead 3. Customization of the app to the specific middleware 4. Maintenance (update) overhead. If the data transfer rate is not a constraint, all of this can be avoided by putting the app in the cloud.

In the following sections we explain our first prototype and all the implemented functionalities.

## Approach

The task of the framework is to offer an interface for arbitrary apps to other users. Therefore, the framework has to provide a possibility to interact, i.e. create, select and use, the apps. This means that the framework acts as a manager of the apps and interpreter between the user and apps. As the apps are typically a process the framework essentially has to manage and communicate with them. As the basic interface between the apps and the framework we propose the usage of ROS as motivated above. It allows simple management of processes using *roslaunch* and has a standardized language to communicate between processes. As a first implementation we focused on the usage of *nodes* which provide *services* as apps.

On the other end of the framework it has to communicate with the users who want to use the apps. To include a broader audience the interaction uses the standard HTTP protocol. This allows users who do not have ROS and it would be possible to use the apps from a web browser. This can be achieved by using the framework as a web server. For the first implementation the focus is on the robots as users and, therefore, an API is necessary and the implementation *django*/*piston* is used.

## Specification

The specification for the development of apps (in the current stage of the framework) is, as mentioned above, that it has to be a ROS *node* which provides a *service* which can be used by the user. Besides that no further restrictions are necessary.

For the communication between the user and the framework the HTTP protocol is used. Here the request types GET, POST and DELETE are 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. Furthermore, some general information is available.

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.

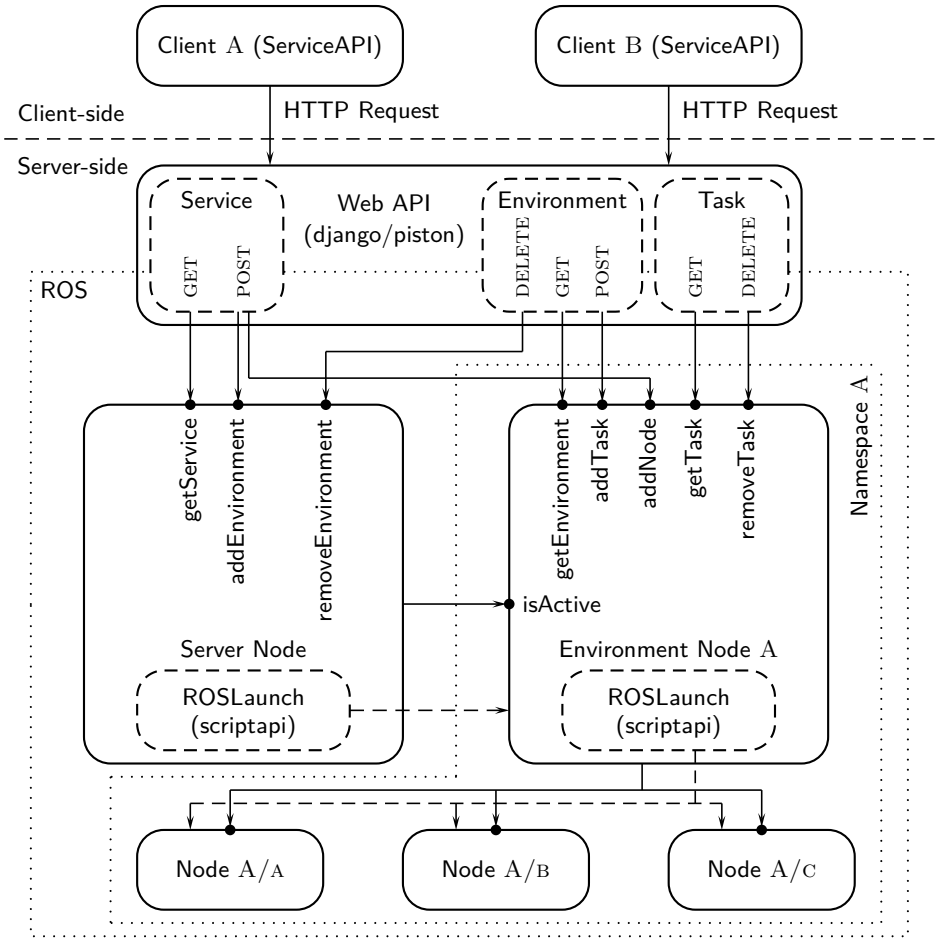


Figure 1: Organization of the framework (loosely from top to bottom): The client, which can use the provided *ServiceAPI*, communicates with the *DjangoInterface* over HTTP. The *DjangoInterface* then uses the necessary *services* of the *ServerNode* or *EnvironmentNode* (solid arrows). The *EnvironmentNode* 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).