RAPP Platform Wiki

v0.5.5

Generated by Doxygen 1.8.6

Tue Dec 8 2015 16:41:18

Contents

Chapter 1

_Footer

RAPP Project, http://rapp-project.eu/

Chapter 2

_Sidebar

Home

Documentation:

Theory

- RAPP Architecture & component diagram
- RAPP Multithreading issues

HOP services

- General Information
- RAPP HOP services

RAPP Nodes

- RAPP Audio Processing
- RAPP Cognitive Exercise
- RAPP Face Detection
- RAPP Knowrob wrapper
- RAPP MySQL wrapper
- RAPP Platform (metapackage))
- RAPP Platform Launchers
- RAPP Platform ROS Communications
- RAPP QR Detection
- RAPP Scripts
- RAPP Speech Detection (Google API)
- RAPP Speech Detection (Sphinx4)
- RAPP Testing Tools
- RAPP Text-to-speech using Espeak & Mbrola

Tutorials / Q&A:

General - Before the installation

- What is the RAPP Platform? Why is it useful?

After the installation

- How can I set-up the RAPP Platform in my PC?
- How do I launch the RAPP Platform?
- How can I see that everything is working properly?
- Everything is set-up and working! What now?
- I do not want to install RAPP Platform. Is there an easier way to use it?
- I do not even want to try the easier way. Do you have something up and running to test?

Ways to improve rapp-platform

- How can I contribute?
- How to create a new RAPP Platform ROS node?
- How to create a HOP service for a ROS service?
- How to write the API for a HOP service?
- How to call the HOP service I created?
- How to implement an integration test?
- Create and authenticate a new RAPP User

Chapter 3

Create-and-authenticate-a-new-RAPP-user

Use the `create_rapp_user.sh` to create and authenticate a new RAPP User.

The script is located under the `devel` directory of the `rapp_scripts` package.

“‘shell $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_scripts/devel $ ./create_rapp_user.sh “‘

The script will prompt to input required info

“‘shell $ ./create_rapp_user.sh

Minimal required fields for mysql user creation:

- username :
- firstname : User’s firstname
- lastname : User’s lastname/surname
- language : User’s first language

Username: rapp Firstname: rapp Lastname: rapp Language: el Password: “‘

Chapter 4

!-What-now? Everything-is-set-up-and-working!-What-now?

Well, now you have to use it! You can deploy the RAPP Platform using the instructions here and you can start invoking the HOP services from your PC or or your robot following this tutorial.

Chapter 5

RAPP Platform

Welcome to the rapp-platform wiki! Here you can find the documentation of every package existent in RAPP Platform, as well as tutorials for utilizing the RAPP Platform.

Documentation:

Theory

- RAPP Architecture & component diagram
- RAPP Multithreading issues

HOP services

- General Information
- RAPP HOP services

RAPP Nodes

- RAPP Audio Processing
- RAPP Cognitive Exercise
- RAPP Face Detection
- RAPP Knowrob wrapper
- RAPP MySQL wrapper
- RAPP Platform (metapackage))
- RAPP Platform Launchers
- RAPP Platform ROS Communications
- RAPP QR Detection
- RAPP Scripts
- RAPP Speech Detection (Google API)
- RAPP Speech Detection (Sphinx4)
- RAPP Testing Tools
- RAPP Text-to-speech using Espeak & Mbrola

Tutorials / Q&A:

General - Before the installation

- What is the RAPP Platform? Why is it useful?

After the installation

- How can I set-up the RAPP Platform in my PC?
- How do I launch the RAPP Platform?
- How can I see that everything is working properly?
- Everything is set-up and working! What now?
- I do not want to install RAPP Platform. Is there an easier way to use it?
- I do not even want to try the easier way. Do you have something up and running to test?

Ways to improve rapp-platform

- How can I contribute?
- How to create a new RAPP Platform ROS node?
- How to create a HOP service for a ROS service?
- How to write the API for a HOP service?
- How to call the HOP service I created?
- How to implement an integration test?
- Create and authenticate a new RAPP User

Chapter 6

HOP framework and HOP Web Services

Developed algorithms located under RAPP Improvement Center (RIC), can be utilized by robots via the RAPP Platform Web Services, exposed by RIC and furthermore the RAPP Platform. We consider RAPP Platform Web Services an imperative component of the RAPP Improvement Center (RIC) as they define the client-server interface layer.

HOP Web Services run under HOP Web Broker, as a part of RIC. Communication between remote clients and the HOP Server, and furthermore HOP Web Services, is achieved via **HTTP Post requests**. HOP Web Server is exposed to the web via a public IPv4 Address. Furthermore the HOP Web Services run on a specific system port and so those are accessible from remote clients via the HOP Web Server running IPv4 Address followed by a port (**xxx.xxx.xxx.xxx:8080**).

As HOP Web Services act as the interface layer between the Robot Platform and the Cloud Platform (along with the RAPP API), the development of those depends on both Client (User/Developer) and RIC requirements.

The communication workflow, from the client-side platform to RAPP Services can be severed into the following independent interface layers:

- Client-side Platform -> HOP Server.
- HOP Server -> HOP Web Services.
- HOP Web Services -> Rosbridge WebSocket Server.
- Rosbridge WebSocket Server -> RAPP Services (ROS Services).

[[images/platform-robot.png]]

The RAPP API, used from the client-side platform, integrates calls to the HOP Server and furthermore to the H-OP Web Services. A HOP Web Service delegates service calls to the RAPP Services through the `Rosbridge` interface layer.

Rosbridge Server provides a WebSocket transport layer to ROS and so it allows web clients to talk ROS using the rosbridge protocol. We use the Rosbridge Websocket Server in order to interfere with AI modules developed under ROS. That WebSocket server is not exposed to the public network and it is only accessible locally.

[[images/srv-rosnodes.png]]

Furthermore, user authentication on Hop Web Services requests, is handled by Hop Web Server. Currently we use **HTTP-Basic-Authentication** technique for enforcing access controls to the RAPP Platform web resources.

Initiate HOP Web Services

For initiating HOP Web Server (and HOP Web Services), execute, under the ***rapp-platform/rapp_web_-services*** directory.

“‘shell $ npm start “‘

Every js file, stored under the services/ directory with a .service.js extension, is considered to be a HOP Web Service js executable file. HOP Server is responsible for registering those services, on run-time, as workers.

Chapter 7

? How-can-I-contribute?

There are various ways to contribute to RAPP Platform.

- First of all, you can try our scripts and our services and if you identify any errors, please let us know by submitting an issue. We will be happy to help you out!
- If you want to engage more actively, you can fork our repository into your GitHub account and fix the issue yourself. Then you can create a pull request and we will add your contribution in our code.
- Finally, if you want to engage further, you can enrich the RAPP Platform with new algorithms, by creating ROS packages, as well as the corresponding HOP services and the API. Then with a pull request, we will check the code and we will merge your contributions.

Chapter 8

?
How-can-I-see-that-everything-is-working-properly?

In order to check that everything works properly, you can do the following:

Run the unit / functional tests

The RAPP Platform must not be running for these tests to run as they will attempt to launch it prior to running themselves, so please make sure you have not launched the platform, or terminate/kill the processes if you already have. Then use the following script:

“‘ cd ∼/rapp_platform/rapp-platform-catkin-ws/ catkin_make run_tests -j1 “‘

All the tests should be passing.

Run the integration tests

First you have to launch the RAPP Platform, as described here

Then you must execute:

“‘ rosrun rapp_testing_tools rapp_run_test.py “‘

Again all tests should be passing.

Check the RAPP Platform health webpage.

Invoke this service from your favourite web browser:

“‘javascript localhost:9001/hop/rapp_platform_status “‘

Chapter 9

? How-can-I-set-up-the-RAPP-Platform-in-my-PC?

The setup scripts are located in `rapp_scripts/setup`.

These scripts can be executed after a clean Ubuntu 14.04 installation, in order to install the appropriate packages and setup the environment.

Step 0 - Github installation

Install git in order to be able to clone the repositories:

```
sudo apt-get install git mercurial
```

WARNING: At least 10 GB's of free space are recommended.

Install RAPP Platform

It is advised to execute the clean_install.sh script in a clean VM or clean system.

Performs:

- initial system updates
- installs ROS Indigo
- downloads all Github repositories needed
- builds and install all repos (rapp_platform, knowrob, rosjava)
- downloads builds and installs depending libraries for Sphinx4
- installs MySQL
- installs HOP

What you must do manually

A new MySQL user was created with username = `dummyUser` and password = `changeMe` and was granted all on RappStore DB. It is highly recommended that you change the password and the username of the user. The username and password are stored in the file located at /etc/db_credentials. The file db_credentials is used by the RAPP platform services, be sure to update it with the correct username and password. It's first line is the username and it's second line the password.

Setup in an existing system

If you want to add rapp-platform to an already existent system (Ubuntu 14.04) you should choose which setup scripts you need to execute. For example if you have MySQL install you do not need to execute `8_mysql_install.sh`.

Scripts

- `1_system_updates.sh`: Fetches the Ubuntu 14.04 updates and installs them
- `2_ros_setup.sh`: Installs ROS Indigo
- `3_auxiliary_packages_setup.sh`: Installs software from apt-get, necessary for the correct RAPP Platform deployment
- `4_rosjava_setup.sh`: Fetches a number of GitHub repositories and compiles rosjava. This is a dependency of Knowrob.
- `5_knowrob_setup.sh`: Fetches the Knowrob repository and builds it. Knowrob is the tool that deploys the RAPP ontology.
- `6_rapp_platform_setup.sh`: Fetches the rapp-platform and rapp-api repositories and builds them
- `7_sphinx_libraries.sh`: Fetches the Sphinx4 necessary libraries, compiles them and installs them, Sphinx4 is used for ASR (Automatic Speech Recognition).
- `8_mysql_install.sh`: Installs MySQL
- `9_create_rapp_mysql_db.sh`: Adds the RAPP MySQL empty database in mysql
- `10_create_rapp_mysql_user.sh`: Creates a user to enable access the to database from the RAPP Platform code
- `11_hop_setup.sh`: Installs HOP and Bigloo, the tools providing the RAPP Platform generic services.

NOTES:

The following notes concern the manual setup of rapp-platform (not the clean setup form our scripts):

- To compile `rapp_qr_detection` you must install the `libzbar` library
- To compile `rapp_knowrob_wrapper` you must execute the following scripts:
  - https://github.com/rapp-project/rapp-platform/blob/master/rapp_-scripts/setup/4_rosjava_setup.sh
  - https://github.com/rapp-project/rapp-platform/blob/master/rapp_-scripts/setup/5_knowrob_setup.sh
  - **If you don't want interaction with the ontology, add an empty `CATKIN_IGNORE` file in the `rapp-platform/rapp_knowrob_wrapper/` folder**

Chapter 10

? How-do-I-launch-the-RAPP-Platform?

This is quite easy! Just follow the instructions below which describe the `deploy` folder in `rapp_scripts`:

There are two files aimed for deployment:

- `deploy_rapp_ros.sh`: Deploys the RAPP Platform back-end, i.e. all the ROS nodes
- `deploy_hop_services.sh`: Deploys the corresponding HOP services

If you want to deploy the RAPP Platform in the background you can use `screen`. Just follow the next steps:

- `screen`
- `./deploy_rapp_ros.sh`
- Press Ctrl + a + d to detach
- `screen`
- `./deploy_hop_services`
- Press Ctrl + a + d to detach
- `screen -ls` to check that 2 screen sessions exist

To reattach to screen session: “‘ screen -r [pid.]tty.host “‘ The screen step is for running rapp_ros and hop_services on detached terminals which is useful, for example in the case where you want to connect via ssh to a remote computer, launch the processes and keep them running even after closing the connection. Alternatively, you can open two terminals and run one script on each, without including the screen commands. It is imperative for the terminals to remain open for the processes to remain active.

Screen how-to: http://www.rackaid.com/blog/linux-screen-tutorial-and-how-to/

Chapter 11

? How-to-call-the-HOP-service-I-created?

As previously stated here we support API(s) for:

- Python
- JavaScript
- C++

For simplicity, we will describe the procedure of calling a HOP Web Service using Python API calls.

We assume that you have previously read on [How-to-write-the-API-for-a-HOP-Service] and implemented the relevant API call for the HOP Web Service.

Calling a HOP Web Service is pretry much simple.

Import the Python RAPP-API module into your code and invoke the relevant API function.

“‘python from RappCloud import RappCloud

rappCloud = RappCloud()

rappCloud.ontology_subclasses_of(’Oven’) “‘

The above sample presents a call to the ontology_subclasses_of HOP Web Service.

Chapter 12

? How-to-create-a-HOP-service-for-a-ROS-service?

Currently, there does not exist a tool to automate generation of web_services. Developers must be familiar with developing server-side applications using Nodejs and have a minor knowledge on developing Web-Services using the HOP Framework.

While developing RAPP Platform Web Services, you might want to read on `HOP/hopjs`.

If you are not familiar with server-side applications, you might also have to read on `Nodejs`.

Hop.js made simple - Overview

Hop.js is a multitier extension of JavaScript. It allows a single JavaScript program to describe the client-side and the server-side components of a Web application. Its runtime environment ensures a consistent execution of the application on the server and on the client.

HopScript is based on JavaScript and aims at ensuring compatibility with this language

- HopScript fully supports ECMAScript 5
- HopScript also support some ECMAScript 6 features:
  - arrow functions.
  - default function parameters.
  - rest arguments.
  - let and const bindings.
  - symbols.
  - template strings.
  - promises.
  - typed Arrays.
- HopScript supports most Nodejs APIs (see Section Nodejs)

Hop programs execute in the context of a builtin web server. They define services, which are super JavaScript functions that get automatically invoked when HTTP requests are received. Functions and services are almost syntactically similar but the latter are defined using the service keyword:

“‘javascript service hello() { return "hello world"; } “‘

To run this program put this code in the file hello.js and execute:

“‘shell $ hop -p 8080 hello.js “‘

Hop extends JavaScript with the geniune HTML. if we want to modify our service to make it return an HTML document, we can use:

“‘javascript service hello() { return <html>hello world</html>; } “‘

Hop is multitier. That is client-side codes are also implemented in Hop. The ∼{ mark switches from server-side context to client-side context:

“‘javascript service hello() { return <html><div onclick=∼{ alert("world") }>hello</html>; } “‘

Hop client-side code and server-side can also be mixed using the ${ mark:

“‘javascript service hello({ who: "foo" }) { return <html><div onclick=∼{ alert("Hi " + ${who} + "!") }>hello</html>; } “‘

Integration with the ROS Framework

In most cases a HOP Web Service must be able to "∗∗Talk ROS∗∗", meaning that a stable interface between the two, HOP Web Service and ROS, must exist.

For the RAPP Platform requirements, we use the lightweight, `RosBridgeJS` module. This module has been developed under the RAPP Project to provide to HOP Services developers, an **easy-to-use** intermediate transport interface layer for calling ROS-Services through their hopjs applications. It integrates a websocket client that allows connections to `rosbridge-websocket-server` and a service controller to call ROS-Service(s).

Initiate rosbridgejs:

“‘javascript /∗∗∗

- Import the RosBridgeJS module.

var Rosbridge = require(’<path_to_RosBridgeJs_sources>/RosBridgeJS.js’);

/∗∗∗

- Initiate connection to rosbridge_websocket_server.
-
- By default, it tries to connect to ws://localhost:9090.
- This is the default URL the rosbridge-websocket-server listens to.

var ros = new ROS({hostname: ”, port: ”, reconnect: true, onconnection: function(){ // . } });

“‘

Fill request parameters for the ROS-Service:

“‘javascript /∗∗∗

- Fill Ros Service request msg parameters here.
- var args = {param1: ”, param2: ”};
- var args = { imageFilename: ’face_image_frame.png’ }; “‘

Declare/Implement the callback functions:

“‘javascript

```
/***
 * Declare the ROS-Service response callback here!!
 * This callback function will be passed into the rosbridge service
 * controller and will be called when a response from rosbridge
 * websocket server arrives.

function callback(data){
   console.log(data);
}
```

```
  /***
   * Declare the onerror callback.
   * The onerror callack function will be called by the service
   * controller as soon as an error occures, on service request.
   *
  function onerror(e){
    console.log(e);
  }
```

‘‘‘

Call the ROS-Service (/rapp/rapp_face_detection/detect_faces):

‘‘‘javascript

```
 var rosSrvName = '/rapp/rapp_face_detection/detect_faces';

  /***
   * Call ROS-Service.
   * Input arguments:
   *   - rosSrvName: The name of the ROS-Service to call</li>
   *   - args: ROS-Service request message arguments</li>
   *   - objLiteral: An object literal to declare the onsuccess and
   *       onerror callbacks.
   *     { success: <onsuccess_callback>,fail: <onerror_callback> }
   *

  ros.callService(rosSrvName, args,
    {success: callback, fail: onerror});
```

‘‘‘

Use the provided by us, HOP Service Creation Templates:

To make development of HOP Web Services more attractive and understandable, we provide two template source files:

- web_service.template.js
- web_services_post_file.template.js

Using these templates as a reference while developing HOP Web Services for a ROS-Service, might save you alot of hours studying! In most cases, you might have to change just a few lines of code!

For more information read here.

Where to store HOP Service source files and how to register the service under the HOP Server.

HOP Service source files are stored under the **services**/ directory, of the rapp_web_services package.

Any ∗∗.js∗∗ file, stored under this directory with a ∗∗.service.js∗∗ extension, is considered to be a HOP Web Service source file.

HOP Server is responsible for registering those services, on run-time, as workers, so you will never have to write code to register your HOP Web Service. Just store the source file under the aforementioned directory with a ∗∗.service.js∗∗ extension.

Chapter 13

? How-to-create-a-new-RAPP-Platform-ROS-node?

In order to create a new functionality in the RAPP Platform, usually a new ROS package needs to be created. Some general guidelines follow:

- Rename the package to a specific name that shows what it does (beginning with `rapp_`). Split in multiple packages if necessary.
- Move the srv files in `rapp_platform_ros_communications` (in a new subfolder) and declare the correct dependencies in the package's `CMakeLists.txt` and `package.xml`. Do not forget to give an intuitive name at your service
- The ROS package should have the following folders and files:
  - `cfg`: Contains configuration files
  - `launch`: Contains the ROS package's launchers
  - `src`: Contains the source files (either C++ or Python)
  - `tests`: Contains the unit and functional tests (the folder's name **must** be `tests`!)
  - `README.md`: Contains the same information to the RAPP Platform wiki corresponding page
  - `setup.py`: If you have a python node this file should exist. Also a `__init__.py` file should exist in the `src` folder in order to create functional tests.
- Rename the srv file and the service topic to comply with the rest of the platform
- Rename the launch file to comply with the rest of the platform
- Call the launch file from `rapp_platform_launch.launch` in the `rapp_platform_launchers` package, in order to be spawned with everything else
- Add unit / functional tests
- Add Doxygen documentation in your code
- Add a description of your package here: `https://github.com/rapp-project/rapp-platform/wiki`
- Create a README.md in the ROS package with the same information as in the wiki registration

Chapter 14

? How-to-implement-an-integration-test?

It is recommended to study on `Rapp-Testing-Tools` first.

Basic documentation on "∗∗Developing Integration Tests∗∗" can be found `here`.

A more-in-depth information on how to write your first integration test is presented here.

The `template.py` will be used as a reference.

Each integration test must have the following characteristics:

- Each test is written as a seperate python source file (.py).
- Each test is implemented as a RappInterfaceTest class.
- The RappInterfaceTest class must define two member methods:
  - execute()
  - validate()
- The Python RAPP-API is imported into our code in order to invoke Platform. Web Service requests.

So lets first create our RappInterfaceTest class which includes the aforementioned member methods.

“‘python class RappInterfaceTest: """ Integration test class """ def **init**(self): pass

def execute(self): pass

def validate(self): pass “‘

Next we need to include the Python RAPP-API module and create an in instance of the RappCloud class which will allow us to invoke Platform Web Service calls.

“‘python from RappCloud import RappCloud

class RappInterfaceTest: """ Integration test class """ def **init**(self): self.rappCloud = RappCloud() pass

“‘

Load any files required for integration test on initialization of the RappInterfaceTest instance and declare the validation values.

“‘python def **init**(self): self.rappCloud = RappCloud()

Set the file path

self.file_uri = ’PATH_TO_LOADED_FILE’

Set the valid results.

self.valid_results = {} “‘

Next we need to implement the validation function, **validate(self)**.

“‘python def validate(self, response): error = response[’error’] if error != "": return [error, self.elapsed_time]

return_data = response[’qr_centers’]

Check if the returned data are equal to the expected

if self.valid_results == return_data: return [True, self.elapsed_time] else: return ["Unexpected result : " + str(return-_data), self.elapsed_time] “‘

Notice that the validate() function has a parameter named response. This is the response object returned from the Platform Service call. It is critical to return validation results as presented above!!

“‘python return [True, self.elapsed_time] “‘

If the response was succesfully validated,

“‘python return ["Unexpected result : " + str(return_data), self.elapsed_time] “‘

if the validation failed.

The `timeit` python module is used to time the execution of the test. **This will be raplaced in v0.6.0 release and execution time of each test will be calculated by the rapp-testing-core-engine.**

Implement the execution function. The execution function is called by the test-core-engine to execute implemented test. The execution function must:

- Call the, **to-test**, Platform Web Service through the Python RAPP-API call.
- Invoke the validation function.

“‘python def execute(self): start_time = timeit.default_timer()

Call the Python RappCloud service

response = self.rappCloud.qr_detection(self.file_uri) end_time = timeit.default_timer() self.elapsed_time = end_time - start_time return self.validate(response) “‘

You will notice that the above implementation of the execute() method calls the qr_detection Service!!

“‘python response = self.rappCloud.qr_detection(self.file_uri) “‘

Chapter 15

? How-to-write-the-API-for-a-HOP-service?

Currently, we support and maintain RAPP-API(s) for the following programming languages:

- Python
- JavaScript
- C++

Under the `rapp-api repository` you can find more information regarding implemented and tested Rapp--Platform Service calls.

As the integration tests are written in Python and uses the `Python-Rapp-APi` it is a good practice to start on the python-rapp-api.

Python Rapp-API Overview

The Python Rapp-API package can be found under the rapp-api/python/ directory, or `here`.

The package includes the following components:

- RappCloud.py: For each developed HOP Web Service this class includes a method that is used to call the relevant HOP Web Service.
- CloudInterface: Service controller that handles HOP Web Service calls. It is actually an implementation of an HTTP.Post service controller.
- Configuration files: Configuration files which includes information on:
  - Platform authentication credentials, `auth.cfg`. The default user for authentication is the **rapp** user. Platform-side authentication is handled by the HOP Server!!
  - Platform address information, `platform.cfg`. Here both the Platform ipv4-address and HOP Server listening port are declared.
  - Platform HOP Web Services information, `services.yaml`. Each developed HOP Web Service has its own entry into this file.

To extend the Python-Rapp-API with a new HOP Web Service call entry you have to:

- Implement the specific RappCloud class member function.
- Append the relevant HOP Web Service name into the **services.yaml** file.

Implement a RappCloud class member function as a HOP Web Service API call.

Lets say we want to implement the face_detection API call. The face_detection HOP Web Service has one input parameter that is a file (image frame).

We implement a member function, named face_detection with one input parameter, named **file_uri**. This parameter is the system path to an image file we want to perform face_detection on.

“‘python class RappCloud: ... ...

def face_detection(self, fileUri): pass “‘

As you can see the above code just declares the API method and does nothing.

HTTP .post requests can have the following fields:

- payload: The payload declares simple {parameter, value} pairs (param1: value).
- files: Used to transfer files over the Web, using multipart/form-data post requests.

So lets declare two variables into our code to describe the aforementioned fields.

“‘python def face_detection(self, fileUri): payload = {} files = {}

“‘

The face_detection HOP Web Service has one input parameter, named file_uri. We have to declare the **file_uri** field and append data into that field. Though keep in mind that the **file_uri** field holds the name of the trasfered, to the Platform HOP Web Service, file. To ensure that HOP Service incoming requests will never have conflicts on file_names because if this happens files on Platform will overwrite each other, we append a random identity as the file’s name postfix (**image1.png –> image1-akji12.png**). The RappCloud class holds an instance of the `Random String Generator` class to perform these kind of operations. To append a unique postfix to the file’s name call the ∗∗__appendRandStr∗∗ member:

“‘python def face_detection(self, fileUri): fileName = self.__appendRandStr(fileUri) payload = {} files = { ’file_uri’: (fileName, open(fileUri, ’rb’)) }

“‘

Notice that we leave the payload to be an empty field!!!

Finally we need call the CloudInterface service controller. RappCloud class holds an instance of the CloudInterface class.

“‘python def face_detection(self, fileUri): fileName = self.__appendRandStr(fileUri) payload = {} files = { ’file_uri’: (fileName, open(fileUri, ’rb’)) }

url = self.serviceUrl_[’face_detection’]

returnData = CloudInterface.callService(url, payload, files, self.auth_) return returnData

“‘

Though one last step is required in order to make the above code execute correctly We need to add the Face--Detection HOP Web Service name into the `services.yaml` file.

“‘yaml

services: list:

- face_detection

... “‘

Make sure to check the Platform Authentication credentials (`auth.cfg`) and Platform Parameters (`platform.-cfg`).

By default, Platform Parameters point to the locally installed RAPP-Platform. Read the **platfrom.cfg** file on how to change these parameters, if for example you installed the RAPP-Platform on a remote machine.

Chapter 16

? NOTE: Under development!

If you do not want to setup / install the RAPP Platform, or even use the ready-to-deploy ova file we provide, but you want to use some of its functionalities, you can do so by invoking the RAPP Platform services in our already deployed instance.

The IP address of this instance is `0.0.0.0` and you can invoke any HOP service using the following url:

0.0.0.0:9001/hop/HOP_SRV_NAME

You can find the HOP service names here.

Chapter 17

? NOTE: Under development

If you do not want to "pollute" your PC with the RAPP Platform installation, but you still want to use it, you can utilize our `ready-to-deploy ova VM`. There the entire RAPP Platform is setup in a clean Ubuntu 14.04 OS and everything is ready for you to use.

Chapter 18

RAPP-Architecture

The overall design of the RAPP Architecture is depicted in the following figure.

[[images/rapp_platform_architecture.png]]

As evident, the RAPP ecosystem consists of two major components: the RAPP Platform (upper half) and the respective Robot Platform (bottom half). These systems are not only functionally, but also physically separated, as the Platform resides in the cloud and the Robot Platform obviously represents each robot supported by the RAPP system, constituting our architecture two layered.

The upper layer is the RAPP ecosystem's cloud part, consisting of three main parts: The RAPP Store, the Platform Agent and a RApp's Cloud agent. The RAPP Store provides the front-end of the application store, allowing the creation of accounts either from the robotic application developers or from the end users. Once a developer creates his account he/she is able to submit RApps via an interface, which are cross-compiled for the supported robots, packaged and indexed. Then they can be distributed at the corresponding robots. If any errors occur during this procedure, the developer is informed in order to correct them and resubmit the application. On the other hand, the end users are able to log in and select the RApps they desire for their robot to execute.

The main part of the RAPP Platform is the RAPP Platform Agent, where the core of the developed artificial intelligence resides. This includes a MySQL database, where all the eponymous information is safely being stored, offline learning procedures and the RAPP Improvement Centre (RIC). This constitutes of the ontology knowledge base, machine learning algorithms and various robotic-oriented algorithms. These are either developed or wrapped by ROS nodes. Finally, these algorithms can be utilized by robots via the HOP services exposed by RIC. The HOP services can be invoked by the RAPP API with specific arguments, providing access uniformity and authentication security. The final RAPP Platform part is the Cloud Agent, which will be described later for better comprehension reasons.

The lower layer of the RAPP architecture is the Robot Platforms. There three specific components exist: The Core agent, the Dynamic agent and the Communication layer. The Core agent is robot specific and is provided by the RAPP platform. It uptakes the tasks of downloading / updating the RApps and providing robot-specific services to the applications. These usually are high level interfaces to the robot's hardware or to locally embedded algorithms, such as mapping, navigation or path planning. A robot can utilize either low level drivers for interfacing with its hardware or higher level tools, such as ROS2. When a RApp is downloaded and executed by the Core agent, a Dynamic agent is created, essentially being another naming for the RApp's robot part. The Dynamic agent can be a ROS package, a JavaScript file, or a combination of them. This uses the RAPP API to invoke services either on the Platform or in the robot. One important aspect of the overall architecture is that the developer may choose for his application to be executed in a decentralized way, meaning that according to his submission, one part of the code may execute in the robot and another part of the cloud. The second part is the Cloud agent described before, which is initiated and killed synchronously to the Dynamic agent.

Considering the RAPP database, it resides on the RAPP Platform and can be accessed a) by the RAPP Store, in order to keep track of the user accounts and the submitted / downloaded applications and b) by the RIC for machine learning / statistical purposes and data acquisition. The latter is performed via a ROS MySQL wrapper, providing interfaces to all the nodes in need for stored data.

Regarding the semantic / knowledge part of RAPP, we decided to employ already used and tested ontologies, aiming at not reinventing the wheel. The only limitations towards this selection were that the ontologies a) should

have ROS support, since the RAPP Platform will be ROS-based, b) should have a common file format to easily merge the information and c) to be state-of-the-art in their respective fields. Since RAPP is multidisciplinary, the selected ontologies contain concepts from heterogeneous scientific fields. After researching in the ontology domain the KnowRob and OpenAAL ontologies were selected for deployment.

KnowRob was part of the RoboEarth FP7 project and was specially designed to be used by robots, extending classical ontologies and concepts in a machine-usable way. KnowRob is a state-of-the-art robotic ontology, used by many scientific teams in cloud robotics projects for concepts storage, inference and distribution of knowledge. One great advantage to our cause is that bindings to ROS exist, as it would be challenging to utilize it otherwise, being developed in SWI-Prolog. Finally, The ROS bindings are created using the JPL interface (Java / Prolog bidirectional Interface) and the rosjava package. Finally, KnowRob provides an abundance of ontology packages, all encoded in OWL files that can be dynamically loaded in the KnowRob software tool and queried. On the other hand, regarding the ambient assisted living field, the OpenAAL ontology was selected, created for the SOPRANO integrated project, providing a middleware for AAL scenarios. It should be stated that similarly to KnowRob, OpenAAL is not just an ontology but a software tool providing methods for context management, multi-paradigm context augmentation, context aware behaviours and others. In our case we will not utilize the overall software package but only the ontology semantic information, encoded in OWL files.

As obvious, KnowRob and OpenAAL are two different ontologies, intended for heterogeneous applications, thus means to utilize them efficiently and meaningfully must be implemented. An initial thought is to load both OWL files in the KnowRob software tool, in order to be able to access the information via ROS interfaces. Even though this would be possible, no higher level semantic processes can be deployed, since the two ontology upper level taxonomies would be semantically separated. Thus we decided to semantically merge the two taxonomies and to enrich them with classes relevant to the RAPP's scope. Regarding the RAPP ontology's access a KnowRob ROS wrapper was created that will provide querying capabilities to external services.

The RAPP Platform component diagram is evident in the image below.

[[images/rapp_platform_class_diagram.png]]

Chapter 19

RAPP-Audio-Processing

The audio processing node was created in order to perform necessary operations for the speech recognition modules to operate for all audio cases. Even though both the Google and Sphinx4 speech recognition modules were functional when the input was captured from a headset, the same did not happen with the audio captured from the NAO robot.

NAO is able to record a single audio file at a time (wav or ogg), either from all microphones (4 channels at 48kHz) or from any single microphone (1 channel, 16kHz). The RAPP Speech detection modules can operate either with ogg or with wav (1 and 4 channels) by employing the Audio processing node. Nevertheless, the one-channel audio is the most appropriate selection, since Sphinx-4 requires single channel wav files, with a 16kHz sample rate and 16 bit little-endian format. The NAO captured audio contains considerable background static noise, being probably the result of a cooling fan that also exists in the NAO head. The problem raised is that the high noise levels cause Sphinx-4, as well as Google API to fail by producing no output.

It is obvious that in order for the speech recognition modules to operate successfully, denoising operations must take place. Additionally, since each NAO robot creates its own noise with different spectral characteristics, a personalization effort must be performed, storing silence samples from each robot and extracting the NAO noise's DFT coefficients. These denoising operations are offered as ROS services by the Audio Processing package. This node utilizes the SoX Unix audio library in order to perform spectral denoising, along with other custom made techniques.

ROS Services

Set noise profile service

This service was created in order to store each robot's noise profile. The service expects three inputs: a string containing the audio file, the audio type and the user that owns the robot. The supported audio types are nao_ogg, nao_wav_1_ch and nao_wav_4_ch.

Initially, the user is authenticated against the MySQL database, in order to check if they have the necessary access rights to execute speech recognition and audio processing functionalities. Then, the audio file (ogg, wav 1 or 4 channels) is converted to wav, single channel with a sampling rate of 16kHz, employing the SoX library. Finally, the noise profile is acquired using the SoX noiseprof tool and the respective file is stored in the RAPP Platform under the user's folder.

Service URL: `/rapp/rapp_audio_processing/set_noise_profile`

Service type:

"'bash #The stored audio file containing silence string noise_audio_file #The audio type [nao_ogg, nao_wav_1_ch, nao_wav_4_ch] string audio_file_type #The user

string user

#Possible error string error "'

Denoise service

This ROS service utilizes the user's stored noise profile in order to perform spectral subtraction against the input audio signal. For this reason the SoX library is used, and specifically the noisered plugin.

Service URL: `/rapp/rapp_audio_processing/denoise`

Service type: "'bash #The stored audio file containing the user's input string audio_file #The audio type [nao_ogg, nao_wav_1_ch, nao_wav_4_ch] string audio_type #The denoised audio file string denoised_audio_file #The user string user #The denoising scale

float32 scale

Possible error

string error "'

Energy denoise service

The energy denoise ROS service performs a signal hard gating in the time domain, based on the RMS metric (Root Mean Squared). The hard signal gating is applied in the individual sample's power when compared with the RMS value.

Service URL: `/rapp/rapp_audio_processing/energy_denoise`

Service type: "'bash #The stored audio file containing the user's input string audio_file #The audio type [nao_ogg, nao_wav_1_ch, nao_wav_4_ch] string audio_type #The denoised audio file string denoised_audio_file #The user string user #The denoising scale

float32 scale

#Possible error string error "'

Detect silence service

There are cases where the captured audio files from NAO do not contain any speech. Since the recording length is limited (e.g. 3 seconds) it is possible for some cases for the actual speech to miss this critical time slot. If this happens, the detect silence service is capable of indicating this issue in order for the robot to ask again the question it was not answered.

In order to detect if the signal contains silence, we follow a statistical approach. We suppose that if the file does not contain a voice, the samples' power levels will be homogeneous to a certain extend. Thus, we calculate the RSD (Relative Standard Deviation) of the signal's power and compare each sample with it. If one sample has a higher value, the signal is considered to contain voice.

Service URL: `/rapp/rapp_audio_processing/detect_silence`

Service type: "'bash #The stored audio file containing the user's input string audio_file #The silence threshold

float32 threshold

#The result bool silence #Possible error string error "'

Launchers

Standard launcher

Launches the **rapp_audio_processing** node and can be launched using “‘ roslaunch rapp_audio_processing audio_processing.launch “‘

HOP services

Set denoise profile RPS

The only RPS Audio Processing is the set_denoise_profile. The set_denoise_profile RPS is of type 3 since it contains a HOP service frontend, contacting a RAPP ROS service, which utilizes the SoX audio library.

Service URL: `localhost:9001/hop/set_noise_profile`

Input/Output

The set_noise_profile RPS has three input arguments, which are the input file, the audio file type and the user. These are encoded in JSON format in an ASCII string representation.

The set_noise_profile RPS returns the success status. The encoding is in JSON format.

“‘ Input = { “noise_audio_file”: “THE_AUDIO_FILE” “audio_file_type”: “nao_ogg, nao_wav_1_ch, nao_wav_4_ch” “user”: “THE_USERNAME” } Output = { “error”: “Possible error” } “‘

Chapter 20

RAPP-Cognitive-Exercise

The RAPP Cognitive exercise system aims to provide the Robot users a means of performing basic cognitive exercises. The cognitive tests supported belong to three distinct categories, a) Arithmetic, b) Reasoning/Recall, c) Awareness. A number of subcategories exist within each category. Tests have been implemented for all subcategories in different variations and difficulty settings. The NAO robot is used in order to dictate the test questions to the user and record the answers. A user performance history is being kept in the ontology that aids in keeping track of the user's cognitive test performance and in adjusting the difficulty of the tests he is presented with. Based on past performance the test difficulty adapts to the user's specific needs in each test category separately in order to accurately reflect the user's individual cognitive strengths and weaknesses. To preserve the user's interest different tests are selected for each category using a least recently used model. To further enhance variation, even tests of the same subcategory exist in different variations. The RAPP Cognitive exercise system component diagram is depicted below.

[[images/cognitive_exercise_system_component_diagram.png]]

ROS Services

Test Selector

This service was created in order to select the appropriate test for a user given its type. The service will load the user's past performance from the ontology, determine the appropriate difficulty setting for the specific user and return the least recently used test of its type. In case no test type is provided then the least recently used test type category will be selected.

Service URL: `/rapp/rapp_cognitive_exercise/cognitive_exercise_chooser`

Service type: "'bash #Contains info about time and reference Header header #The username of the user string username #The test type requested

String testType

#The test's name string test #The test's type string testType #The test's sub type string testSubType #The test's language string language #The test's questions string[] questions #The list of answers for each test string[][] answers #The correct answers for each test string[] correct_answers #Possible error string error "'

Record User Performance

This service was created in order to record the performance, meaning the score, of a user after completing a cognitive exercise test. The name of the test, its type and the user's score are provided as input arguments. Within the service, a Unix timestamp is automatically generated. The timestamp reflects the time at which the test was performed. The service returns the name of the test performance entry that was created in the ontology.

Service URL: `/rapp/rapp_cognitive_exercise/record_user_cognitive_test_performance`

Service type: “‘bash #Contains info about time and reference Header header #The username of the user string username #The type of the test string testType #The test which was performed string test #The score the user achieved on the test

String score

#Container for the subclasses String user_cognitive_test_performance_entry #Possible error string error “‘

Cognitive Test creator

This service was created in order to create a cognitive test in the special xml format required and register it to the ontology. It accepts as an input a specially formatted text file containing all the information required for the test including its type, subtype, difficulty, questions, answers etc. The service formats the above information in an xml file and registers the test along with some vital information like it’s type and difficulty setting to the ontology. The service returns a bool variable that is true if xml creation and ontology registration were successful. Any possible error is contained in the error string variable also returned.

Service URL: `/rapp/rapp_cognitive_exercise/cognitive_test_creator`

Service type: “‘bash #Contains info about time and reference Header header #The text file containing the test information

string inputFile

#True if test creation and registration to the ontology was successfull bool success #Possible error string error “‘

Launchers

Standard launcher

Launches the **rapp_cognitive_exercise** node and can be launched using “‘ roslaunch rapp_cognitive_exercise cognitive_exercise.launch “‘

HOP services

Test Selector RPS

The test_selector RPS is of type 3 since it contains a HOP service front-end, contacting a RAPP ROS ontology wrapper, which performs queries to the KnowRob ontology repository. The get subclass of RPS can be invoked using the following URL.

Service URL: `localhost:9001/hop/test_selector`

Input/Output

The test_selector RPS has two arguments, which are the username of the user and the requested test type. This is encoded in JSON format in an ASCII string representation.

The test_selector RPS the questions, the possible answers and the correct answers of the selected test. The encoding is in JSON format.

“‘ Input = { “username”: “THE_USERNAME” “testType”: “THE_TEST_TYPE” } Output = { “test”: “The test’s name” “testType”: “The test’s type” “testSubType”: “The test’s sub type” “language”: “The test’s language” “questions”: “The questions of the test” “answers”: “The possible answers for each question of the test” “correct_answers”: “The correct answer for each question of the test” “error”: “Possible error” }

“‘

Example

An example input for the test_selector RPS is “‘ Input = { “instance_name”: “Person_1” “attribute_names”: “-ArithmeticCts” } “‘

For this specific input, the result obtained was

“‘ Output = { “test”: “ArithmeticCts_XXXXXXX” “testType”: “ArithmetiCts” “testSubType”: “BasicArithmetic” “language”: “en” “questions”: “[How much is 2 plus 2?, How much is 5 plus 5?]” “answers”: “[2,3,4,5],[7,8,9,10]” “correct_answers”: “[4,10]" } “‘

Record user cognitive test performance RPS

The record_user_cognitive_test_performance RPS is of type 3 since it contains a HOP service frontend, contacting a RAPP ROS ontology wrapper, which performs queries to the KnowRob ontology repository. The record user cognitive test performance of RPS can be invoked using the following URL.

Service URL: `localhost:9001/hop/record_user_cognitive_test_performance`

Input/Output

The record_user_cognitive_test_performance RPS has four arguments, which are the username of the user the test which was taken, the test’s type and the score achieved. This is encoded in JSON format in an ASCII string representation.

The record_user_cognitive_test_performance RPS outputs only a possible error message which is empty when the query was successful. The encoding is in JSON format.

“‘ Input = { “username”: “THE_USERNAME” “test”: “THE_TEST” “testType”: “THE_TEST_TYPE” } Output = { “error”: “Possible error” } “‘

Example

An example input for the record_user_cognitive_test_performance RPS is “‘ Input = { “instance_name”: “Person_1” “test”: “Test1” “testType”: “ArithmeticCts” “score”: “90” } “‘ For this specific input, the result obtained was

“‘ Output = { “error”: “‘

Chapter 21

RAPP-Face-Detection

In the RAPP case, the face detection functionality is implemented in the form of a C++ developed ROS node, interfaced by a HOP service. The HOP service is invoked using the RAPP API and gets an RGB image as input, in which faces must be detected. The second step is for the HOP service to locally save the input image. At the same time, the Face Detection ROS node is executed in the background, waiting to server requests. The HOP service calls the ROS service via the ROS Bridge, the ROS node make the necessary computations and a response is delivered.

ROS Services

Face detection

Service URL: `/rapp/rapp_face_detection/detect_faces`

Service type: “‘bash #Contains info about time and reference Header header #The image’s filename to perform face detection

string imageFilename

#Container for detected face positions geometry_msgs/PointStamped[] faces_up_left geometry_msgs/PointStamped[] faces_down_right string error “‘

Launchers

Standard launcher

Launches the **face detection** node and can be launched using “‘ roslaunch rapp_face_detection face_detection.launch “‘

HOP services

URL

`localhost:9001/hop/face_detection`

Input / Output

“‘ Input = { “image”: “THE_ACTUAL_IMAGE_DATA” } Output = { “faces”: [{ “top_left_x” : “T_P_X”, “top_left_y” : “T_P_Y”, “bottom_right_x” : “B_R_X”, “bottom_right_y” : “B_R_Y” }, { . . . }] } “‘

Chapter 22

RAPP-HOP-services

RAPP Platform Web Services are developed under the `HOP web broker`. These services are used in order to communicate with the RAPP Platform ecosystem and access RIC(RAPP Improvement Center) AI modules.

Platform Services	Description
face_detection	Performs face detection on given input image frame
qr_detection	Performs face detection on given input image frame
text_to_speech	Performs text-to-speech on given input plain text
denoise_profile	Used in order to perform given user's audio profile
speech_detection_sphix4	Performs speech-detection using the Platform integrated Sphinx4 engine
speech_detection_google	Performs speech-detection using the Platform integrated Google engine
available_services	Returns a list of the Platform available services (up-to-date)
ontology_subclasses_of	Perform Ontology, subclasses-of, query
ontology_superclasses_of	Perform Ontology, superclasses-of, query
ontology_is_supsuperclass_of	Perform Ontology, is-subsuperclass-of, query
cognitive_test_chooser	Returns a Cognitive Exercise literal that describes the test
record_cognitive_test_performance	Record user's performance on given Cognitive Exercise

Services

QR-Detection

“‘javascript qr_detection ({file_uri: ”}) “‘

Input parameters

- file_uri: Destination where the posted file data (**image file**) are saved by hop-server. Data are posted using a multipart/form-data post request using this field. e.g.

“‘python file = {’file_uri’: open(<to-send-file-path>, ’rb’)} “‘

Response/Return-Data

The returned data are in *JSON* representation. A JSON.load() from client side must follow in order to decode the received data.

“‘javascript { qr_centers: [], qr_messages: [], error: ” } “‘

- qr_centers: Vector that containes points (x,y) of found QR in an image frame.

- qr_messages: Vector that containes message descriptions of found QR in an image frame.
- error: If error was encountered, an error message is pushed in this field and returned to the client.

Response Sample:

“‘javascript { qr_centers: [{ y: 165, x: 165 }], qr_messages: [’rapp project qr sample’], error: ” } “‘

Face-Detection

“‘javascript face_detection ({file_uri: ”}) “‘

Input parameters

- file_uri: Destination where the posted file data (**image file**) are saved by hop-server. Data are posted using a multipart/form-data post request using this field. e.g.

“‘python file = {’file_uri’: open(<to-send-file-path>, ’rb’)} “‘

Response/Return-Data

The returned data are in *JSON* representation. A JSON.load() from client side must follow in order to decode the received data.

“‘javascript { faces: [{ up_left_point: {}, down_right_point: {} }], error: ” } “‘

Point coordinates are presented in Cartesian Coordinate System as:

“‘javascript {x: <value_int>, y: <value_int>} “‘

- faces: Dynamic vector that contains recognized faces in an image frame.
- up_left_point: This Object literal contains the up-left point coordinates of detected face.
- up_left_point: This Object literal contains the down-right point coordinates of detected face.
- error: If error was encountered, an error message is pushed in this field and returned to the client.

Response Sample:

“‘javascript { faces: [{ up_left_point: { y: 200, x: 212 }, down_right_point: { y: 379, x: 391 } }], error: ” } “‘

Speech Detection related services.

Denoise Audio Profile

“‘javascript set_denoise_profile ({file_uri: ”, audio_source: ”, user: ”}) “‘

Input parameters

- ∗∗’file_uri’∗∗: Destination where the posted file data (**audio data file**) are saved by hop-server. Data are posted using a multipart/form-data post request using this field. e.g.

“‘python file = {’file_uri’: open(<to-send-file-path>, ’rb’)} “‘

- ∗∗’audio_source’∗∗: A value that presents the <robot>_<encode>_<channels> information for the audio source data. e.g "nao_wav_1_ch".
- ∗∗’user’∗∗: User’s name. Used for per-user profile denoise configurations.

Response/Return-Data

The returned data are in *JSON* representation. A JSON.load() from client side must follow in order to decode the received data.

“‘javascript { error: ’<error_message>’ } “‘

- error: If error was encountered, an error message is pushed in this field and returned to the client.

“‘javascript { error:"RAPP Platform Failure!" } “‘

Speech-Detection-Sphinx4

“‘javascript speech_detection_sphinx4 ({ file_uri: ”, language: ”, audio_source: ”, words: [], sentences: [], grammar: [], user: ”}) “‘

Input parameters

- **’file_uri’**: Destination where the posted file data (**audio data file**) are saved by hop-server. Data are posted using a multipart/form-data post request using this field. e.g.

“‘python file = {’file_uri’: open(<to-send-file-path>, ’rb’)} “‘

- **’language’**: Language to be used by the speech_detection_sphinx4 module. Currently valid language values are ‘gr’ for Greek and ‘en’ for English.
- **’audio_source’**: A value that presents the <robot>_<encode>_<channels> information for the audio source data. e.g "nao_wav_1_ch".
- **’words[]’**: A vector that carries the words to search for into the voice-audio-source.
- **’sentences[]’**: The under consideration sentences.
- **’grammar[]’**: Grammars to use in speech recognition.
- **’user’**: User’s name. Used for per-user profile denoise configurations.

Response/Return-Data

The returned data are in *JSON* representation. A JSON.load() from client side must follow in order to decode the received data.

“‘javascript { words: [], error: ’<error_message>’ } “‘

- **’error’**: If error was encountered, an error message is pushed in this field and returned to the client.
- **’words[]’**: A vector that contains the "words-found"

Speech-Detection-Google

“‘javascript speech_detection_sphinx4 ({ file_uri: ”, audio_source: ”, user: ”, language: ”}) “‘

Input parameters

- **’file_uri’**: Destination where the posted file data (**audio data file**) are saved by hop-server. Data are posted using a multipart/form-data post request using this field. e.g.

“‘python file = {’file_uri’: open(<to-send-file-path>, ’rb’)} “‘

- **’language’**: Language to be used by the speech_detection_sphinx4 module. Currently valid language values are ‘gr’ for Greek and ‘en’ for English.
- **’audio_source’**: A value that presents the {robot}_{encode}_{channels} information for the audio source data. e.g "nao_wav_1_ch".
- **’user’**: User’s name. Used for per-user profile denoise configurations.

Response/Return-Data

The returned data are in *JSON* representation. A JSON.load() from client side must follow in order to decode the received data.

“‘javascript { words: [], alternatives: [] error: ’<error_message>’ } “‘

- **’error’**: If error was encountered, an error message is pushed in this field and returned to the client.
- **’words[]’**: A vector that contains the "words-found" with highest confidence.
- **’alternatives[[]]’**: Alternative sentences. e.g. [[’send’, ’mail’], [’send’, ’email’], [’set’, ’mail’]...]

Ontology related services.

The following Platform services give access to the Platform integrated Ontology system.

Ontology-SubClasses-Of

“‘javascript ontology_subclasses_of ({ query: ”}) “‘

Input parameters

- **’query’**: The query to the ontology database.

Response/Return-Data

The returned data are in *JSON* representation. A JSON.load() from client side must follow in order to decode the received data.

“‘javascript { results: [], error: ’<error_message>’ } “‘

- **’results’**: Query results returned from ontology database.
- **’error’**: If error was encountered, an error message is pushed in this field and returned to the client.

“‘javascript { results: [’http://knowrob.org/kb/knowrob.owl#Oven’, ’http://knowrob.org/kb/knowrob.owl#MicrowaveOven’,’http://knowrob.org/kb/knowrob.owl#RegularOven’, ’http://knowrob.org/kb/knowrob.owl#ToasterOven’], error: ” } “‘

Ontology-SuperClasses-Of

“‘javascript ontology_superclasses_of ({ query: ”}) “‘

Input parameters

- **’query’**: The query to the ontology database.

Response/Return-Data

The returned data are in *JSON* representation. A JSON.load() from client side must follow in order to decode the received data.

“‘javascript { results: [], error: ’<error_message>’ } “‘

- **’results’**: Query results returned from ontology database.
- **’error’**: If error was encountered, an error message is pushed in this field and returned to the client.

Ontology-Is-SubSuperClass-Of

“‘javascript ontology_is_subsuperclass_of ({ parent_class: ”, child_class: ”, recursive: false }) “‘

Input parameters

- **'parent_class'**: The parent class name.
- **'child_class'**: The child class name.
- **'recursive'**: Defines if a recursive procedure will be used (true/false).

Response/Return-Data

The returned data are in *JSON* representation. A JSON.load() from client side must follow in order to decode the received data.

"'javascript { results: Bool, error: '<error_message>' } "'

- **'result'**: Success index on ontology-is-subsupeclass-of query
- **'error'**: If error was encountered, an error message is pushed in this field and returned to the client.

Text-To-Speech (tts) related services

Text-To-Speech

"'javascript text_to_speech({ text: '', language: ''}) "'

Input parameters

- **'text'**: Input text to translate to audio data.
- **'language'**: Language to be used for the TTS module. Valid values are currently **el** and **en**

Response/Return-Data

The returned data are in *JSON* representation. A JSON.load() from client side must follow in order to decode the received data.

"'javascript { payload: <audio_data>, basename: <audio_file_basename>, encoding: <payload_encoding>, error: <error_message> } "'

- **'payload'**: The audio data payload. Payload encoding is defined by the 'encoding' json field. Decode the payload audio data (client-side) using the codec value from the 'encoding' field.
- **'encoding**: Codec used to encode the audio data payload. Currently encoding of binary data is done using base64 codec. Ignore this field. May be used in future implementations.
- **'basename'**: A static basename for the audio data file, returned by the platform service. Ignore this field. May be usefull in future implementations.
- **'error'**: If error was encountered, an error message is pushed in this field.

Cognitive Exercises related services

Cognitive-Test-Selector

"'javascript cognitive_test_chooser({ user: '', test_type: '' }) "'

Input parameters

- **'user'**: Username of client used to retrieve information from database. e.g "klpanagi"
- **'test_type'**: Cognitive Exercise test type. Can be one of ['ArithmeticCts', 'AwarenessCts', 'Reasoning-Cts']

Response/Return-Data

The returned data are in *JSON* representation. A JSON.load() from client side must follow in order to decode the received data.

“‘javascript { questions: [], possib_ans: [], correct_ans: [], test_instance: ”, test_type: ”, test_subtype: ”, error: ” } “‘

- **’questions’**: The exercise set of questions.
- **’possib_ans’**: The set of answers for each question. vector<vector<string>>
- **’correct_ans’**: The set of correct answers for each question. vector<string>
- **’test_instance’**: Returned test name. For example, ’ArithmeticCts_askw0Snwk’
- **’test_type’**: Cognitive exercise class/type. Documentation on Cognitive Exercise classes can be found here
- **’test_subtype’**: Cognitive exercise sub-type. Documentation on Subtypes can be found here
- **’error’**: If error was encountered, an error message is pushed in this field.

Cognitive-Test-Selector

“‘javascript record_cognitive_test_performance({ user: ”, test_instance: ”, score: 0 }) “‘

Input parameters

- **’user’**: Username of client used to retrieve information from database. e.g "klpanagi"
- **’test_instance’**: Cognitive Exercise test instance. The full cognitive test entry name as reported by the **cognitive_test_chooser()**.
- **’score’**: User’s performance score on given test entry.

Response/Return-Data

The returned data are in *JSON* representation. A JSON.load() from client side must follow in order to decode the received data.

“‘javascript { performance_entry: ”, error: ” } “‘

- **’performace_entry’**: User’s cognitive test performance entry in ontology.
- **’error’**: If error was encountered, an error message is pushed in this field.

Health - RAPP Platform Status

For RAPP developers.

Rapp-Platform-Status

“‘javascript rapp_platform_status() “‘

Invoke this service from your favourite web browser:

“‘javascript <rapp_platform_pub_ipaddr>/9001/hop/rapp_platform_status “‘

Chapter 23

RAPP-Knowrob-wrapper

The RAPP Knowrob wrapper ROS node is needed to expose specific services to the other ROS nodes and RPSs (similarly to the MySQL wrapper concept). In our implementation, the pure KnowRob ontology was enhanced after the insertion of extra classes derived from the OpenAAL ontology, as well as others needed to implement the desired RApps. The RAPP Knowrob wrapper component diagram is depicted in the figure.

[[images/knowrob_wrapper_component_diagram.png]]

The services of the RAPP Knowrob wrapper are detailed below.

ROS Services

Subclasses of

This service was created in order to return the subclasses of a specific ontology class. Apart from the basic functionality, one can perform recursive search in the ontology and not just in the classes' immediate lower level connections.

Service URL: `/rapp/rapp_knowrob_wrapper/subclasses_of`

Service type: "'bash #Contains info about time and reference Header header #The class whose subclasses we want string ontology_class #True if the query is recursive

bool recursive

#Container for the subclasses string[] results #Possible error string error #true if successful bool success "'

Superclasses of

This service was created in order to return the superclasses of a specific ontology class. Apart from the basic functionality, one can perform recursive search in the ontology and not just in the classes' immediate higher level connections.

Service URL: `/rapp/rapp_knowrob_wrapper/superclasses_of`

Service type: "'bash #Contains info about time and reference Header header #The class whose superclasses we want string ontology_class #True if the query is recursive

bool recursive

#Container for the superclasses string[] results #Possible error string error #true if successful bool success "'

Is sub-super-class of

This service was created in order to investigate two classes' semantic relations. Apart from the basic functionality, one can perform recursive search in the ontology and not just in the classes' immediate higher or lower level connections.

Service URL: `/rapp/rapp_knowrob_wrapper/is_subsuperclass_of`

Service type: "'bash #Contains info about time and reference Header header #The parent class string parent_class #The child class string child_class #True if the query is recursive

bool recursive

#True if the semantic condition applies bool result #Possible error string error #true if successful bool success "'

Create instance

This service was created in order to create an instance related to a specific ontology class. One can also store a file and comments. This service is not public but it is employed from the RIC nodes. Ownership of the new instance is assigned to the provided user which is an instance of the class Person within the ontology. The assignment involves setting an ontology attribute. The username provided in the input is the username of the user in the MySQL database and not the name of the user's instance in the ontology. The second is also stored within the MySQL database the service acquires it by querying the MySQL database (with the username as input) through the MySQL wrapper. In case no ontology instance name (alias) is defined a new one is created utilizing the Create ontology alias service described below and the Ontology and MySQL database are updated accordingly.

Service URL: `/rapp/rapp_knowrob_wrapper/create_instance`

Service type: "'bash #Contains info about time and reference Header header #The user to whom the instance belongs string username #The instance's ontology class string ontology_class #A file url (if needed) string file_url #Comments (if needed)

string comments

#A unique id string instance_name #Possible error string error #true if successful bool success "'

Create ontology alias

This service was created in order to create an alias for a user within the ontology. A user ontology alias is basically an instance of the class Person that exists within the ontology. The user's ontology alias is stored within the MySQL database in the respective column of the table User. This service accepts the MySQL username of the user and performs a check by querying the MySQL database in order to identify if an ontology alias has already been defined. If that is the case it simply returns that ontology alias. If not, it creates the ontology alias instance within the ontology, stores this information in the MySQL database and finally it returns the newly created user ontology alias. In the case that a new ontology alias is created both the ontology and the MySQL need to be updated. The service ensures that either both are updated in tandem or in case one fails no modifications take place in the other. This is critical to preserving proper synchronization between the MySQL database and the ontology.

Service URL: `/rapp/rapp_knowrob_wrapper/create_ontology_alias`

Service type: "'bash #Contains info about time and reference Header header #The user to whom the instance belongs

string username

#A unique id string ontology_alias #Possible error string error #true if successful bool success "'

User instances of class

This service allows an ML algorithm to retrieve user-specific information in order to compute and extract personalized data. This service is not exposed via a HOP service, but is employed internally by other RIC nodes. It returns all the ontology instances that are assigned to the ontology alias of the provided user. The ontology alias is acquired again by querying the MySQL database.

Service URL: `/rapp/rapp_knowrob_wrapper/user_instances_of_class`

Service type: "'bash #Contains info about time and reference Header header #The user whose instances must be returned string username #The ontology class whose instances we desire

string ontology_class

#The instances String[] results #Possible error string error #true if successful bool success "'

Load / Dump ontology

Since the KnowRob ontology framework does not provide online storage functionality, the ontology (along with the new information) must be stored in predefined time slots. This way, if a system crash occurs, the stored data won't be lost but can be retrieved using the Load ontology ROS service. Both of these services have a common representation.

Service URL: `/rapp/rapp_knowrob_wrapper/load_ontology` Service URL: `/rapp/rapp_knowrob_wrapper/dump_ontology`

Service type: "'bash #Contains info about time and reference Header header #The file intended for loading or dumping the ontology

string file_url

#Possible error string error #true if successful bool success "'

Create cognitive test

This service creates a new cognitive exercise test in the ontology as an instance. This service is not exposed via a HOP service, but is employed internally by the RAPP Cognitive exercise system node. It accepts as input that parameters of the test which include its type, subtype, difficulty, variation and file path and returns the name of the ontology instance created.

Service URL: `/rapp/rapp_knowrob_wrapper/create_cognitve_tests`

Service type: "'bash #Contains info about time and reference Header header #The type of the test string test_type #The sub type of the test string test_subtype #The difficulty of the test int32 test_difficulty #The variation id of the test int32 test_variation #The file path where the test is located

string test_path

#The created test name string test_name #Possible error string error #true if successful bool success "'

Return cognitive tests of type

This service returns all cognitive tests of the given type that exist within the ontology. This service is not exposed via a HOP service, but is employed internally by the RAPP Cognitive exercise system node. It accepts as input the test type and returns the names of the tests and their parameters which include their subtypes, file paths, difficulties and variation ids.

Service URL: `/rapp/rapp_knowrob_wrapper/cognitive_tests_of_type`

Service type: “‘bash #Contains info about time and reference Header header #The type of the tests to be returned

string test_type

#The names of the tests string[] tests #The subtype of the tests string[] subtype #The file paths of the tests string[] file_paths #The difficulty of the tests string[] difficulty #The variation of the tests string[] variation #Possible error string error #true if successful bool success “‘

Record user cognitive test performance

This service records the user’s (patient’s) performance on a given test at a given time. Performance is measured as integer value in the 0-100 range. This service is not exposed via a HOP service, but is employed internally by the RAPP Cognitive exercise system node.

Service URL: `/rapp/rapp_knowrob_wrapper/record_user_cognitive_tests_performance`

Service type: “‘bash #Contains info about time and reference Header header #The name of the test string test #The type of the test string test_type #The ontology alias of the patient who is taking the test string patient_ontology_alias #The score the patient achieved in the test int32 score #The timestamp at which the test was performed

int32 timestamp

#The name of the cognitive test performance entry string cognitive_test_performance_entry #Possible error string error #true if successful bool success “‘

Return user cognitive test performance

This service returns all the tests of the requested type that a specific user (patient) has undertaken along with the scores achieved, the time at which they were performed and the difficulty and variation ids of the tests. This service is not exposed via a HOP service, but is employed internally by the RAPP Cognitive exercise system node.

Service URL: `/rapp/rapp_knowrob_wrapper/user_performance_cognitve_tests`

Service type: “‘bash #Contains info about time and reference Header header #The ontology alias of the patient string ontology_alias #The type of the tests of interest

string test_type

#The names of the tests string[] tests #The scores of the tests string[] scores #The difficulty of the tests string[] difficulty #The variation ids of the tests string[] variation #The timestamps at which the tests were performed string[] timestamps #Possible error string error #true if successful bool success “‘

Launchers

Standard launcher

Launches the **rapp_knowrob_wrapper** node and can be launched using “‘ roslaunch rapp_knowrob_wrapper knowrob_wrapper.launch “‘

HOP services

Subclasses-of RPS

The subclasses_of RPS is of type 3 since it contains a HOP service frontend, contacting a RAPP ROS ontology wrapper, which performs queries to the KnowRob ontology repository. The get subclass of RPS can be invoked using the following URL.

Service URL: `localhost:9001/hop/subclasses_of`

Input/Output

The sublasses_of RPS has two input arguments, which are the ontology class for which the subclasses must be found and the recursive flag. These are encoded in JSON format in an ASCII string representation.

The subclasses_of RPS returns the subclasses of the input class in a ontology URL form. The encoding is in JSON format.

“‘ Input = { “class”: “THE_ONTOLOGY_CLASS” “recursive”: “True of False” } Output = { “subclasses”: [“SUBCL_-URL_1”, “SUBCL_URL_2”,. . . , “SUBCL_URL_3”] “error”: “Possible error” } “‘

Example

An example input for the subclasses_of RPS is “‘ Input = { “class”: “Oven” “recursive”: “False” } “‘

For this specific input, the result obtained was

“‘ Output = { “subclasses”: [“http://knowrob.org/kb/knowrob.owl#Oven”, “http://knowrob.org/kb/knowrob.owl#-MicrowaveOven”, “http://knowrob.org/kb/knowrob.owl#RegularOven”, “http://knowrob.org/kb/knowrob.owl#Toaster-Oven”] “error”: “” } “‘

Superclasses-of RPS

The superclasses_of RPS is of type 3 since it contains a HOP service frontend, contacting a RAPP ROS ontology wrapper, which performs queries to the KnowRob ontology repository. The superclasses_of RPS can be invoked using the following URL.

Service URL: `localhost:9001/hop/superclasses_of`

Input/Output

The superclasses_of RPS has two input arguments, which are the ontology class for which the superclasses must be found and the recursive flag. These are encoded in JSON format in an ASCII string representation.

The superclasses_of RPS returns the superclasses of the input class in an ontology URL form. The encoding is in JSON format.

“‘ Input = { “class”: “THE_ONTOLOGY_CLASS” “recursive”: “True of False” } Output = { “superclasses”: [“SUPCL-_URL_1”, “SUPCL_URL_2”,. . . , “SUPCL_URL_3”] “error”: “Possible error” } “‘

Example

An example input for the superclasses_of RPS is “‘ Input = { “class”: “SpatialThing” “recursive”: “False” } “‘

For this specific input, the result obtained was

“‘ Output = { “superclasses”: [“http://www.w3.org/2002/07/owl#Thing”] “error”: “” } “‘

Is Sub-Superclass-of RPS

The is_subsuperclass_of RPS is of type 3 since it contains a HOP service frontend, contacting a RAPP ROS ontology wrapper, which performs queries to the KnowRob ontology repository.

Service URL: `localhost:9001/hop/is_subsuperclass_of`

Input/Output

The is_subsuperclass_of RPS has three arguments, which are the parent and child ontology class, as well as the recursive flag. This is encoded in JSON format in an ASCII string representation.

The is_subsuperclass_of RPS returns true if the semantic relation holds. The encoding is in JSON format.

“‘ Input = { “parent_class”: “THE_PARENT_ONTOLOGY_CLASS” “child_class”: “THE_CHILD_ONTOLOGY_CLASS” “recursive”: “True of False” } Output = { “result”: “True or False” “error”: “Possible error” } “‘

Example

An example input for the is_subsuperclass_of RPS is “‘ Input = { “parent_class”: “SpatialThing” “child_class”: “Oven” “recursive”: “True” } “‘

For this specific input, the result obtained was

“‘ Output = { “result”: “True” “error”: “” } “‘

Chapter 24

RAPP-Multithreading-issues

Since RIC (or in other words the RAPP Platform) is a cloud-based service provider, it makes absolute sense for its algorithms to be able to handle multiple and concurrent requests. This requirement translates into the requirement for HOP (which receives the RPSs) and the ROS nodes to be able to handle simultaneous calls.

HOP is currently in version 3.0 and is indeed capable of receiving simultaneous calls by assigning a different thread in each service call. Next, a rosbridge socket is used for the HOP service to invoke the ROS service.

ROS has a special way of treating the ROS services invocation. In C++ nodes, each ROS node is handled by a single thread, i.e. if two simultaneous calls arrive the one will be served and the other will wait in queue. If the ROS node is a C++ node, a specific configuration exists that allows the node to accept a predefined number of threads. It should be made clear that we can only allow a number of threads for the whole node and not for a specific service. The number of concurrent threads a C++ node can serve is defined as a parameter in each node's configuration file. For example, in the face detection case, the parameter `rapp_face_detection_threads` is equal to 10 (thus 10 concurrent threads can be served). This parameter can be found here:

rapp-platform/rapp_face_detection/cfg/face_detection_params.yaml

On the other hand, if we have a Python ROS node, this configuration is not available, but each ROS service call creates a new thread. This means that we do not have any problems regarding parallel calls but we cannot limit the number of calls as well.

Considering the ROS threads handling, it becomes apparent that ROS nodes that are purely functional (i.e. do not hold state or a back-end procedure) have no issues at all with simultaneous calls. On the other hand, nodes such as the Knowrob Wrapper or the Sphinx4 ASR system, which depend on the deployment of other packages, need special handling. In the RAPP Platform case, we decided that only the Sphinx4 ASR ROS node is in need of service handling, since this will be the most invoked one. Our approach involved the creation of N back-end Sphinx4 processes, along with a handler, which accepts the requests and decides which process should serve them. At any time, we keep track of which processes are occupied and what their configurations are. Thus, if for example a speech recognition invocation occurs with a specific configuration, the pool of unoccupied processes is researched in case one of them is already configured as requested. If true, the handler proceeds directly to the speech recognition or in the opposite case, selects a random unoccupied process and performs both configuration and recognition. This approach was selected, as the configuration process is quite expensive in time resources, thus we prefer to avoid it whenever possible.

Chapter 25

RAPP-MySQL-wrapper

The RAPP MySQL wrapper ROS node provides the means to interact with the developed MySQL database by providing 26 ROS services that aid in selecting, inserting, updating or deleting information from some of the tables of the RAPP MySQL Database.

The scheme of the MySQL Database is depicted below.

[[images/mysql_scheme.png]]

The component diagram of the MySQL wrapper is depicted below. [[images/mysql_wrapper_component_diagram.-png]]

ROS Services

A write, fetch, update and delete service exists for each of the tables of the MySQL database that are accessible through the wrapper. These four services follow a general approach, irrespective of the actual table they affect. The URLs of the available services are shown below.

“‘ /rapp/rapp_mysql_wrapper/tbl_apps_robots_delete_data /rapp/rapp_mysql_wrapper/tbl_apps_robots_fetch-_data /rapp/rapp_mysql_wrapper/tbl_apps_robots_update_data /rapp/rapp_mysql_wrapper/tbl_apps_robots-_write_data /rapp/rapp_mysql_wrapper/tbl_model_delete_data /rapp/rapp_mysql_wrapper/tbl_model_fetch-_data /rapp/rapp_mysql_wrapper/tbl_model_update_data /rapp/rapp_mysql_wrapper/tbl_model_write_data /rapp/rapp_mysql_wrapper/tbl_rapp_delete_data /rapp/rapp_mysql_wrapper/tbl_rapp_fetch_data /rapp/rapp-_mysql_wrapper/tbl_rapp_update_data /rapp/rapp_mysql_wrapper/tbl_rapp_write_data /rapp/rapp_mysql_-wrapper/tbl_robot_delete_data /rapp/rapp_mysql_wrapper/tbl_robot_fetch_data /rapp/rapp_mysql_wrapper/tbl-_robot_update_data /rapp/rapp_mysql_wrapper/tbl_robot_write_data /rapp/rapp_mysql_wrapper/tbl_user_-delete_data /rapp/rapp_mysql_wrapper/tbl_user_fetch_data /rapp/rapp_mysql_wrapper/tbl_user_update_data /rapp/rapp_mysql_wrapper/tbl_user_write_data /rapp/rapp_mysql_wrapper/tbl_users_ontology_instances_delete-_data /rapp/rapp_mysql_wrapper/tbl_users_ontology_instances_fetch_data /rapp/rapp_mysql_wrapper/tbl-_users_ontology_instances_update_data /rapp/rapp_mysql_wrapper/tbl_users_ontology_instances_write_-data /rapp/rapp_mysql_wrapper/view_users_robots_apps_fetch_data /rapp/rapp_mysql_wrapper/what_rapps-_can_run “‘ Below is a detailed explanation of how the fetch, write, update and delete services work in general. Examples of how these services are called can be found in: https://github.-com/rapp-project/rapp-platform/blob/master/rapp_mysql_wrapper/test/mysql_-wrapper/db_test.py#L217-L304

Fetch Data

The general format of the service responsible for fetching data from an array is presented below.

“‘bash #the requested column names in the MySQL select query string[] req_cols #a 2 dimensional array each line of which is a where relationship in the select query

rapp_platform_ros_communications/StringArrayMsg[] where_data

#the resulting column names string[] res_cols #the resulting column data (in 2 dimensional array) where each line is an entry rapp_platform_ros_communications/StringArrayMsg[] res_data #possible error information string[] trace #true if successful std_msgs/Bool success “‘

Write Data

The general format of the service responsible for writing data to an array is presented below.

“‘bash #the to be written column names string[] req_cols #the values to be written to the above columns

rapp_platform_ros_communications/StringArrayMsg[] req_data

#possible error information string[] trace #true if successful std_msgs/Bool success “‘

Update Data

The general format of the service responsible for updating data of an array is presented below.

“‘bash #the to be set columns along with their new values in ’columnName=newValue’ format string[] set_cols #a 2 dimensional array each line of which is a where relationship

rapp_platform_ros_communications/StringArrayMsg[] where_data

#possible error information string[] trace #true if successful std_msgs/Bool success “‘

Delete Data

The general format of the service responsible for deleting data of an array is presented below.

“‘bash #a 2 dimensional array each line of which is a where relationship

rapp_platform_ros_communications/StringArrayMsg[] where_data

#possible error information string[] trace #true if successful std_msgs/Bool success “‘

whatRappsCanRun

This service returns what RApps can run on a robot when given the robot’s model ID and it’s Core Agent version.

“‘bash #the robot’s model id string model_id #the robot’s core agent version

string core_agent_version

#the resulting column names string[] res_cols #the resulting column data (in 2 dimensional array) where each line is an entry rapp_platform_ros_communications/StringArrayMsg[] res_data #possible error information string[] trace #true if successful std_msgs/Bool success “‘

Chapter 26

(metapackage) RAPP-Platform-(metapackage)

The `rapp_platform` folder is essentially the RAPP Platform metapackage, where all the RAPP Platform dependencies are declared. As stated in the ROS webpage:

> Metapackages are specialized Packages in ROS (and catkin). They do not install files (other than their package.xml manifest) and they do not contain any tests, code, files, or other items usually found in packages.
>
> A metapackage is used in a similar fashion as virtual packages are used in the debian packaging world. A metapackage simply references one or more related packages which are loosely grouped together.

Chapter 27

RAPP-Platform-Launchers

The `rapp_platform_launchers` is a ROS package containing the necessary launchers to deploy RAPP Platform. For now, only the `rapp_platform_launch.launch` launcher exists, which deploys the entire RAPP Platform. This can be launched with the following command:

```
roslaunch rapp_platform_launchers rapp_platform_launch.launch
```

Chapter 28

RAPP-Platform-ROS-Communications

The `rapp_platform_ros_communications` ROS package contains all the necessary ROS message, service and action files the RAPP Platform nodes require to operate. These are placed in the respoctive `msg`, `srv` and `action` folders. In these folders, each file is being kept under a dedicated folder for each ROS package.

This way, each ROS node has to declare the `rapp_platform_ros_communications` as dependency, in order to use ROS messages, services or actions.

Chapter 29

RAPP-QR-Detection

A QR code (Quick Response code) is a visual two dimensional matrix, firstly introduced in Japan's automotive industry. A QR is a kind of barcode, with the exception that common barcodes are one dimensional, whereas QRs are two dimensional, thus able to contain much more information. As known, several types of data can be encoded in a barcode or QR, such as strings, numbers etc. The QR code has become a standard in the worldwide consumers' field, as they are widely used for product tracking, item identification, time tracking, document management and general marketing. One of the main reasons behind its widespread is that efficient algorithms are developed that provide real-time QR detection and identification even from mobile phones. It should be stated that QR tags can have different densities, therefore include different amount of information.

[[images/qr_sample.png]]

A QR consists of square black and white patterns, arranged in a grid in the plane, which can be detected by a camera in order to perform the decoding process. Regarding the RAPP implementation, a ROS node was developed that uses the well-known ZBar library, in conjunction to OpenCV for image manipulation.

ROS Services

Face detection

Service URL: `/rapp/rapp_qr_detection/detect_qrs`

Service type: "'bash #Contains info about time and reference Header header #The image's filename to perform the detection

string imageFilename

#Container for detected qr positions geometry_msgs/PointStamped[] qr_centers string[] qr_messages string error "'

Launchers

Standard launcher

Launches the **qr detection** node and can be launched using "' roslaunch rapp_qr_detection qr_detection.launch "'

HOP services

URL

`localhost:9001/hop/qr_detection`

Input / Output

“‘ Input = { “image”: “THE_ACTUAL_IMAGE_DATA” } Output = { “qrs”: [{ “qr_x” : “QR_X”, “qr_y” : “QR_Y”, “message” : “MESSAGE” }, { . . . }] } “‘

Chapter 30

RAPP-Scripts

The `rapp_scripts` folder contains scripts necessary for operations related to the RAPP Platform. The scripts are divided into four folders:

backup

The following scripts are offered:

- `dumpRAPPMysqlDatabase.sh`: Dumps the RAPP MySQL database in a .sql file
- `importRAPPMysqlDatabase.sh`: Imports the RAPP MySQL database from a .sql file
- `dumpRAPPOntology.sh`: Dumps the RAPP ontology in an .owl file
- `importRAPPOntology.sh`: Imports the RAPP ontology from an .owl file

continuous_integration

This folder contains the Travis script, used to support continuous integration of the RAPP Platform. The RAPP Platform Travis page is located `here`. For every push performed in rapp-platform Travis:

- Checks the clean_install script
- Builds the rapp-platform repository
- Runs the unit, functional and integration tests
- Creates the code documentation and uploads it `online`

deploy

There are two files aimed for deployment:

- `deploy_rapp_ros.sh`: Deploys the RAPP Platform back-end, i.e. all the ROS nodes
- `deploy_hop_services.sh`: Deploys the corresponding HOP services

If you want to deploy the RAPP Platform in the background you can use `screen`. Just follow the next steps:

- `screen`
- `./deploy_rapp_ros.sh`

- Press Ctrl + a + d to detach
- `screen`
- `./deploy_hop_services`
- Press Ctrl + a + d to detach
- `screen -ls` to check that 2 screen sessions exist

To reattach to screen session: "' screen -r [pid.]tty.host "' The screen step is for running rapp_ros and hop_services on detached terminals which is useful, for example in the case where you want to connect via ssh to a remote computer, launch the processes and keep them running even after closing the connection. Alternatively, you can open two terminals and run one script on each, without including the screen commands. It is imperative for the terminals to remain open for the processes to remain active.

Screen how-to: http://www.rackaid.com/blog/linux-screen-tutorial-and-how-to/

setup

These scripts can be executed after a clean Ubuntu 14.04 installation, in order to install the appropriate packages and setup the environment.

Step 0 - Get the scripts

You can get the setup scripts either by downloading the rapp-platform repository in a `zip format`, or by cloning it in your PC using git:

`git clone https://github.com/rapp-project/rapp-platform.git`

WARNING: At least 10 GB's of free space are recommended.

Install RAPP Platform

It is advised to execute the clean_install.sh script in a clean VM or clean system.

Performs:

- initial system updates
- installs ROS Indigo
- downloads all Github repositories needed
- builds and install all repos (rapp_platform, knowrob, rosjava)
- downloads builds and installs depending libraries for Sphinx4
- installs MySQL
- installs HOP

What you must do manually

A new MySQL user was created with username = `dummyUser` and password = `changeMe` and was granted all on RappStore DB. It is highly recommended that you change the password and the username of the user. The username and password are stored in the file located at /etc/db_credentials. The file db_credentials is used by the RAPP platform services, be sure to update it with the correct username and password. It's first line is the username and it's second line the password.

Setup in an existing system

If you want to add rapp-platform to an already existent system (Ubuntu 14.04) you should choose which setup scripts you need to execute. For example if you have MySQL install you do not need to execute `8_mysql_install.sh`.

Scripts

- `1_system_updates.sh`: Fetches the Ubuntu 14.04 updates and installs them
- `2_ros_setup.sh`: Installs ROS Indigo
- `3_auxiliary_packages_setup.sh`: Installs software from apt-get, necessary for the correct RAPP Platform deployment
- `4_rosjava_setup.sh`: Fetches a number of GitHub repositories and compiles rosjava. This is a dependency of Knowrob.
- `5_knowrob_setup.sh`: Fetches the Knowrob repository and builds it. Knowrob is the tool that deploys the RAPP ontology.
- `6_rapp_platform_setup.sh`: Fetches the rapp-platform and rapp-api repositories and builds them. This script has an input argument which is the git branch the rapp-platform will be checked out. If you decide to execute this script manually it is recommended to give `master` as input.
- `7_sphinx_libraries.sh`: Fetches the Sphinx4 necessary libraries, compiles them and installs them, Sphinx4 is used for ASR (Automatic Speech Recognition).
- `8_mysql_install.sh`: Installs MySQL
- `9_create_rapp_mysql_db.sh`: Adds the RAPP MySQL empty database in mysql
- `10_create_rapp_mysql_user.sh`: Creates a user to enable access the to database from the RAPP Platform code
- `11_hop_setup.sh`: Installs HOP and Bigloo, the tools providing the RAPP Platform generic services.

NOTES:

The following notes concern the manual setup of rapp-platform (not the clean setup form our scripts):

- To compile `rapp_qr_detection` you must install the `libzbar` library
- To compile `rapp_knowrob_wrapper` you must execute the following scripts:
  - https://github.com/rapp-project/rapp-platform/blob/master/rapp_-scripts/setup/4_rosjava_setup.sh
  - https://github.com/rapp-project/rapp-platform/blob/master/rapp_-scripts/setup/5_knowrob_setup.sh
  - **If you don't want interaction with the ontology, add an empty `CATKIN_IGNORE` file in the `rapp-platform/rapp_knowrob_wrapper/` folder**

documentation

Scripts to automatically create documentation from the RAPP Platform code, the services and this wiki, using Doxygen. All documents can be bound in `${HOME}/rapp_platform_files/documentation`.

- `create_source_documentation.sh` : Creates the platform's source code documentation (cpp, h, py, java).
- `create_wiki_documentation.sh` : Creates the platform's GitHub Wiki documentation.
- `create_test_documentation.py` : Creates the platform's test source code documentation.
- `create_web_services_documentation.sh` : Creates the rapp_web_services documentation.
- `create_documentation.sh` : Creates all aforementioned documentations.
- `update_rapp-project.github.io.sh` : Creates the documentation and pushes it in the `gh-pages` branch of rapp-platform, in order to update the online pages. NOTE: This is functional only when executed in the Travis environment, so do not try to run it!

devel

Use the `create_rapp_user.sh` to create and authenticate a new RAPP User.

The script is located under the `devel` directory of the `rapp_scripts` package.

“‘shell $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_scripts/devel $ ./create_rapp_user.sh “‘

The script will prompt to input required info

“‘shell $ ./create_rapp_user.sh

Minimal required fields for mysql user creation:

- username :
- firstname : User’s firstname
- lastname : User’s lastname/surname
- language : User’s first language

Username: rapp Firstname: rapp Lastname: rapp Language: el Password: “‘

Chapter 31

RAPP-Speech-Detection-using-Google-API

For the initial Speech recognition implementation, a proof of concept approach was followed, employing the Google Speech API . This API was created to enable developers to provide web-based speech to text functionalities. There are two modes (one-shot speech and streaming) and the results are returned as a list of hypotheses, along with the most dominant one. Since the aforementioned API is for developing purposes some limitations exist, such as that the input stream cannot be longer than 10-15 seconds of audio and the requests per day cannot be more than 50 if a personal Speech API Key is used. In our implementation we use the one-shot speech functionality, meaning that an audio file must be locally stored.

Another limitation of the Google Speech API is that the input audio file must be of certain format. Specifically the file must be a flac file with one channel, having a sample rate of 16kHz. In order for this service to be able to be used with any audio file that arrives as input, in case where the input file has not the desired characteristics, a system call to the flac library is performed, converting the file to the correct format. Additionally, if the audio originates from the NAO robot, the file is denoised using the respective services of the Audio Processing node, according to its specific characteristics.

The component diagram of the RAPP Speech detection system using the Google API is depicted below.

[[images/google_speech_component_diagram.png]]

ROS Services

Speech to text service using Goolge API

The Google Speech Recognition node provides a single ROS service.

Service URL: `/rapp/rapp_speech_detection_google/speech_to_text`

Service type: “‘bash #The stored audio file string audio_file #The audio type [nao_ogg, nao_wav_1_ch, nao_wav_-4_ch] string audio_file_type #The user

string user

#The words string[] words #The confidence returned by Google float64 confidence #A list of alternative phrases returned from Google string[][] alternatives #Possible error string error “‘

Test Selector RPS

The QR detection RPS is of type 4 since it contains a HOP service frontend that contacts a ROS node wrapper, which in turn invokes an external service. The speech detection RPS can be invoked using the following URL.

Service URL: `localhost:9001/hop/speech_detection_google`

Input/Output

As described, the speech detection RPS takes as input the audio file in which we desire to detect the words. The file path is encoded in JSON format in a binary string representation.

The speech detection RPS returns as output an array of words determining the dominant guess, the confidence in a probability form and the suggested alternative sentences. The encoding is in JSON format.

“‘ Input = { “audio_file”: “THE_AUDIO_FILE” “audio_file_type”: “nao_ogg, nao_wav_1_ch, nao_wav_4_ch” “user”: “USERNAME” } Output = { “words”: [“WORD_1”, “WORD_2”, . . . , “WORD_N”], “confidence” : “PROBABILITY”, “alternatives”: [{ “words”: [“WORD_1”, “WORD_2”, . . . , “WORD_N”] }, { . . . }] } “‘

Example

An example input for the speech detection RPS was created. The actual input was a flac file, having a size of 242.6 KB, where the “I want to use the Skype” sentence was recorded. For this specific input, the result obtained was:

For this specific input, the result obtained was “‘ Output = { “words”: [“I”, “want”, “to” , “use” , “Skype”], “confidence” : “0.9400593”, “alternatives”: [{ “words”: [“I”, “want”, “to” , “use” , “the” , “Skype”]}, { “words”: [“I”, “want”, “to” , “use” , “Skype”]}, { “words”: [“I”, “want”, “to” , “use” , “the” , “Skype” , “app”]}] } “‘

Chapter 32

RAPP-Speech-Detection-using-Sphinx4

In our case we desire a flexible speech recognition module, capable of multi-language extension and good performance on limited or generalized language models. For this reason, Sphinx-4 was selected. Since we aim for speech detection services for RApps regardless of the actual robot at hand, we decided to install Sphinx-4 in the RAPP Cloud and provide services for uploading or streaming audio, as well as configuring the ASR towards language or vocabulary-specific decisions.

Before proceeding to the actual description, it should be said that the RAPP Sphinx4 detection was create in order to handle **limited vocabulary** speech recognition in various languages. This means that it is not suggested for detecting words in free speech or words from a vocabulary larger than 10-15 words.

Before describing the actual implementation, it is necessary to investigate the Sphinx-4 configuration capabilities. In order to perform speech detection, Sphinx-4 requires:

- An acoustic model containing information about senones. Senones are short sound detectors able to represent the specific language's sound elements (phones, diphones, triphones etc.). In order to train an acoustic model for an unsupported language an abundance of data must be available. CMU Sphinx supports a number of high-quality acoustic models, including US English, German, Russian, Spanish, French, Dutch, Mexican and Mandarin. In the RAPP case we desire multi speakers support, thus as Sphinx-4 suggests, 50 hours of dictation from 200 different speakers are necessary `ref`, including the knowledge of the language's phonetic structure, as well as enough time to train the model and optimize its parameters. If these resources are unavailable (which is the current case), a possibility is to utilize the US English acoustic model and perform grapheme to phoneme transformations.
- A language model, used to restrict word search. Usually n-gram (an n-character slice of a bigger string) language models are used in order to strip non probable words during the speech detection procedure, thus minimizing the search time and optimizing the overall procedure. Sphinx-4 supports two language models- : grammars and statistical language models. Statistical language models include a set of sentences from which probabilities of words and succession of words are extracted. A grammar is a more "strict" language model, since the ASR tries to match the input speech only to the provided sentences. In our case, we are initially interested in detecting single words from a limited vocabulary, thus no special attention was paid in the construction of a generalized Greek language model.
- A phonetic dictionary, which essentially is a mapping from words to phones. This procedure is usually performed using G2P converters (Grapheme-to-Phoneme), such as `Phonetisaurus` or `sequitur-g2p3`. G2P converters are used in almost all TTSs (Text-to-Speech converters), as they require pronunciation rules to work correctly. Here, we decided to not use a G2P tool, but investigate and document the overall G2P procedure of translating Greek graphemes directly into the CMU Arpabet format (which Sphinx supports).

The overall Speech Detection architecture utilizing the Sphinx4 library is presented in the figure below:

[[images/sphinx_diagram.png]]

As evident, two distinct parts exist: the NAO robot and RAPP Cloud part. Let's assume that a robotic application (RApp) is deployed in the robot, which needs to perform speech detection. The first step is to invoke the Capture Audio service the Core agent provides, which in turn captures an audio file via the NAO microphones. This audio

file is sent to the cloud RAPP ASR node in order to perform ASR. The most important module of the RAPP ASR is the Sphinx-4 wrapper. This node is responsible for receiving the service call data and configuring the Sphinx-4 software according to the request. The actual Sphinx-4 library is executed as a separate process and Sphinx-4 wrapper is communicating with it via sockets.

Between the RAPP ASR and the Sphinx-4 wrapper lies the Sphinx-4 Handler node which is responsible for handling the actual service request. It maintains a number of Sphinx wrappers in different threads, each of which is capable of handling a different request. The Sphinx-4 handler is responsible for scheduling the Sphinx-4 wrapper threads and for this purpose maintains information about the state of each thread (active/idle) and each thread's previous configuration parameters. Three possible situations exist:

1. If a thread is idle and its previous configuration matches the request's configuration, this thread is selected to handle the request as the time consuming configuration procedure can be skipped.
2. If no idle thread's configuration matches the request's configuration, an idle thread is chosen at random.
3. If all threads are active, the request is put on hold until a thread is available.

Regarding the Sphinx-4 configuration, the user is able to select the ASR language and if they desire ASR on a limited vocabulary or on a generalized one stored in the RAPP cloud. If a limited vocabulary is selected, the user can also define the language model (the sentences of the statistical language model or the grammar). The configuration task is performed by the Sphinx-4 Configuration module. There, the ASR language is retrieved and the corresponding language modules are employed (currently Greek, English and their combination). If the user has requested ASR on a limited vocabulary, the corresponding language module must feed the Limited vocabulary creator with the correct grapheme to phoneme transformations, in order to create the necessary configuration files. In the English case, this task is easy, since Sphinx-4 provides a generalized English vocabulary, which includes the words' G2P transformations. When Greek is requested, a simplified G2P method is implemented, which will be discussed next. In the case where the user requests a generalized ASR, the predefined generalized dictionaries are used (currently only English support exists).

The second major task that needs to be performed before the actual Sphinx-4 ASR is the audio preparation. This involves the employment of the **SoX** audio library utilizing the `Audio processing` node. Then the audio file is provided to the Sphinx4 Java library and the resulting words are extracted and transmitted back to the RApp, as a response to the HOP service call.

Regarding the Greek support, first a description of some basic rules of the Greek language will be presented. The Greek language is equipped with 24 letters and 25 phonemes. Phonemes are structural sound components defining a word's acoustic properties. Some pronunciation rules the Greek language has follow:

- Double letters are two letters that contain two phonemes.
- Two digit vowels are two-letter combinations that sound like a single vowel phoneme.
- There is a special "s" letter, which is placed at the word's end instead of the "normal" 's' letter.
- The common consonants are two common letter combinations that sound exactly like the single case letter.
- There are some special vowel combinations that are pronounced differently according to what letter is next.
- A grammatical symbol is the acute accent (Greek tonos), denoting where the word is accented.
- Another special symbol is diaeresis.
- There are some sigma rules, where if sigma is followed by specific letters it sounds like z.

Finally, there are several other trivial and rare rules that we did not take under consideration in our approach.

Let's assume that some Greek words are available and we must configure the Sphinx4 library in order to perform speech recognition. These words must be converted to the Sphinx4-supported Arpabet format which contains 39 phonemes. The individual steps followed are:

- Substitute upper case letters by the corresponding lower case

- Substitute two-letter phonemes and common consonants with CMU phonemes
- Substitute special vowel combinations
- Substitute two digit vowels by CMU phonemes
- Substitute special sigma rules
- Substitute all remaining letters with CMU phonemes

For more information you can read the full description `here - 2bchanged`

Then, the appropriate files are created (custom dictionary and language model) and the Sphinx4 library is configured. Then the audio pre-processing takes place, performing denoising similarly to the Google Speech Recognition module by deploying the ROS services of the Audio processing node.

The RAPP Speech Detection using Sphinx component diagram is depicted in the figure.

[[images/sphinx_speech_component_diagram.png]]

ROS Services

Speech Recognition using Sphinx service

The Sphinx4 ROS node provides a ROS service, dedicated to perform speech recognition.

Service URL: `/rapp/rapp_speech_detection_sphinx4/batch_speech_to_text`

Service type: "'bash #The language we want ASR for string language #The limited vocabulary. If this is empty a general vocabulary is supposed string[] words #The language model in the form of grammar string[] grammar #The language model in the form of sentences string[] sentences #The audio file path string path #The audio file type string audio_source #The user requesting the ASR

String user

#The words recognized string[] words #Possible error string error "'

HOP services

Speech recognition sphinx RPS

The speech_recognition_sphinx RPS is of type 3 since it contains a HOP service frontend, contacting a RAPP ROS node, which utilizes the Sphinx4 library. The speech_recognition_sphinx RPS can be invoked using the following URI:

Service URL: `localhost:9001/hop/speech_recognition_sphinx`

Input/Output

The speech_recognition_sphinx RPS has several input arguments, which are encoded in JSON format in an ASCII string representation.

The speech_detection_sphinx RPS returns the recognized words in JSON forma.

"' Input = { "language": "gr, en" "words": "[WORD_1, WORD_2 ...]" "grammar": "[WORD_1, WORD_2 ...]" "sentences": "[WORD_1, WORD_2 ...]" "path": "AUDIO_FILE_URI" "audio_source": "nao_ogg, nao_wav_1_ch, nao_wav_4_ch, headset" "user": "USERNAME" } Output = { "words": "[WORD_1, WORD_2 ...] "error": "Possible error" } "'

The request parameters are:

- `language`: The language to perform ASR (Automatic Speech Recognition). Supported values:
  - `en`: English
  - `el`: Greek (also supports English)
- `words[]`: The limited vocabulary from which Sphinx4 will do the matching. Must provide individual words in the language declared in the language parameter. If left empty a generalized vocabulary will be assumed. This will be valid for English but the results are not good.
- `grammar[]`: A form of language model. Contains either words or sentences that contain the words declared in the words parameter. If grammar is declared, Sphinx4 will either return results that exist as is in grammar or <nul> if no matching exists.
- `sentences[]`: The second form of language model. Same functionality as grammar but Sphinx can return individual words contained in the sentences provided. This is essentially used to extract probabilities regarding the phonemes succession.
- `path`: The audio file path.
- `audio_source`: Declares the source of the audio capture in order to perform correct denoising. The different types are:
  - headset: Clean sound, no denoising needed
  - nao_ogg: Captured ogg file from a single microphone from NAO. Supposed to have 1 channel.
  - nao_wav_1_ch: Captured wav file from one microphone of NAO. Supposed to have 1 channel, 16kHz.
  - nao_wav_4_ch: Captured wav file from all 4 NAO's microphones. Supposed to have 4 channels at 48kHz.
- `user`: The user invoking the service. Must exist as username in the database to work. Also a noise profile for the declared user must exist (check rapp_audio_processing node for set_noise_profile service)

The returned parameters are:

- `error`: Possible errors
- `words[]`: The recognized words

Chapter 33

RAPP-Testing-Tools

Testing tools used for RAPP Platform integration tests.

The RAPP Testing Tools have been developed in consideration of providing to **RAPP developers**, a system integration suite. While developing independent RIC modules, Web Services, Platform Agents, Robot Agents, the rapp_testing_tools package provides a way to test the functionality, system-wide.

All tests use the RappCloud python module,`python-rapp-api` to communicate with RAPP Platform AI modules.

A test, written under the RAPP testing tools framework, is composed of, at-least, the following ingredients:

- The source code that describes the test (Python).
- (**Optional**) Data files used as a part of the test, if any are used. Data files can be, for example, image data files that will be transmitted to the RAPP Platform in order to perform image processing algorithms on those.

The implementation of a test defines two stages;

- The test execution.
- The validation of the response.

    The developer, of the integration tests, is responsible to declare this set of valid results.

[[images/rapp-testing-tools-dia.png]]

Using the RAPP testing framework, several tests have been developed. At least one test, for each module located under the RAPP Improvement Center, has been developed. Integration test are used to check on the integration functionality, of the independent components which constitute the RAPP system. Those are developed in a way to also test the functionality of the interface layers between a client-side process and the RAPP Services.

Executing integration tests

The rapp_run_test.py script is used in order to execute developed tests.

Execution time arguments:

- [**-i**] : Give as input, the test name, to execute.

“‘shell $ ./rapp_run_test_.py -i face_detection_test_1.py “‘

- [**-n**] : Give number of execution of test(s).

“‘shell $ ./rapp_run_test.py -i face_detection_test_1.py -n 10 “‘

- [∗∗-t∗∗] : Run the test in threaded mode which means that multible invocations can be done, simultaneous. Each test execution is handled by a standalone thread.

“‘shell $ ./rapp_run_test.py -i face_detection_test_1.py -n 10 -t “‘

- [∗∗-c∗∗] : Test classes can be used to execute a spesific family of tests. Using test classes all tests, under this class, will be executed. In most cases a test class is named after the relevant RAPP Platform ROS-Package; [**face-detection, qr-detection, speech-detection, speech-detection-sphinx4, speech-detection-google, ontology, cognitive, tts**]

“‘shell $ ./rapp_run_test.py -i -c face-detection “‘

Note: If no test_name is provided as argument, all tests located under tests paths are executed!!

There are two ways of executing developed tests:

- Execute directly.
- Execute under the ROS framework.

Using the python executable, directly

The below example executes the qr_detection_test_1, five times in sequential mode.

“‘shell $ ./rapp_run_test.py -i qr_detection_test_1.py -n 5 “‘

Using ROS framework

The below example executes all tests, once.

“‘shell $ rosrun rapp_testing_tools rapp_run_test.py “‘

Developing RAPP integration tests using the RAPP-Testing-Tools

For **more-in-depth** information (how to write and execute your own integration tests, using the rapp_testin_tools framework), please have a look at the technical documentation of the **rapp_testing_tools** package, here.

Chapter 34

RAPP-Text-to-speech-using-Espeak-&-Mbrola

In order to provide speech capabilities to the robots that do not have a Text-To-Speech system installed, but are equipped with microphones, the `rapp_text_to_speech_espeak` ROS node. This module utilizes the `espeak` speech synthesis library, as well as the `mbrola` project for altering the speech synthesis voices.

ROS Services

Text to speech

Service URL: `/rapp/rapp_text_to_speech_espeak/text_to_speech_topic`

Service type: “‘bash #Contains info about time and reference Header header #The text to be spoken string text #The audio output file string audio_output #Language (en, gr)

string language

#Possible errors string error “‘

Launchers

Standard launcher

Launches the **rapp_text_to_speech_espeak** node and can be launched using “‘ roslaunch rapp_text_to_speech_espeak text_to_speech_espeak.launch “‘

HOP services

URL

`localhost:9001/hop/text_to_speech`

Input / Output

“‘ Input = { “text”: “THE_TEXT_TO_BECOME_SPEECH” “language”: “THE_TEXT_LANGUAGE” } Output = { THE_AUDIO_FILE } “‘

Chapter 35

?-Why-is-it-useful? What-is-the-RAPP-Platform?-Why-is-it-useful?

RAPP Platform is a collection of ROS nodes and back-end processes that aim to deliver ready-to-use generic services to robots. The main concept of RAPP Platform aligns with the cloud robotics approach.

As stated in Wikipedia:

Cloud robotics is a field of robotics that attempts to invoke cloud technologies such as cloud computing, cloud storage, and other Internet technologies centred on the benefits of converged infrastructure and shared services for robotics.

RAPP Platform is divided in two main parts: the ROS nodes functionalities and the HOP services. The ROS nodes are back-end processes the provide generic functionalities, such as Image processing, Audio processing, Speech-to-text and Text-to-speech, Ontology & Database operations, as well as ML procedures.

The second part consists of the various HOP services, which are the front-end of the RAPP Platform. These expose specific RAPP Platform functionalities to the world, thus any robot can call specific algorithms, making easy the work of developers.