RAPP Platform Wiki

v0.6.0

Generated by Doxygen 1.8.6

Tue Jul 19 2016 11:43:36

Contents

Chapter 1

_Footer

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

Chapter 2

_Sidebar

Home

Documentation:

Theory

- RAPP Architecture & component diagram
- RAPP Multithreading issues

RAPP Web services

- General Information
- RAPP HOP Web services description

RAPP Nodes

- RAPP Application Authentication
- RAPP Audio Processing
- RAPP Cognitive Exercise
- RAPP Caffe Wrapper
- RAPP Email
- RAPP Face Detection
- RAPP Geolocator
- RAPP Hazard Detection
- RAPP Knowrob wrapper
- RAPP MySQL wrapper
- RAPP News Explorer
- RAPP Object Recognition
- RAPP Path planner
- RAPP Platform (metapackage))

- RAPP Platform Launchers
- RAPP Platform ROS Communications
- RAPP QR Detection
- RAPP Speech Detection (Google API)
- RAPP Speech Detection (Sphinx4)
- RAPP Testing Tools
- RAPP Text-to-speech using Espeak & Mbrola
- RAPP Weather Reporter

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
- A full RAPP Platform service creation example

Full tutorials

- A full RAPP Platform service creation example: Simple addition (easy)
- A full RAPP Platform service creation example: Simple addition & RAPP Platform service invocation & testing (advanced)
- Remote application for NAO in Python: Move by speech (easy))
- In-robot application for NAO in Python: Move by speech (easy))
- Remote application for NAO in Python: Detect and track QR tags (normal))
- Remote application for NAO in Python: Use ROS & TLD tracker to approach arbitrary objects (hard))

Troubleshooting

- RAPP Improvement Center (RIC)
- RAPP Platform Web Server

Chapter 3

A-full-RAPP-Platform-service-creation-example-advanced

In this example, a basic guide will be presented that will cover the basics of creating an example RAPP Platform service. We assume that you have successfully installed the RAPP Platform and run the tests, if not please see our installation guide located at

https://github.com/rapp-project/rapp-platform/wiki/How-can-I-set-up-the-RAPP-Platform-in-my-PC%3F

It is addressed to the novice user and as such, experienced users may find it too detailed or need to skip over some sections. This tutorial assumes some basic ROS knowledge. Users not familiar with ROS are advised to first complete relevant ROS tutorials that can be found at

http://wiki.ros.org/ROS/Tutorials

By the end of the tutorial you should be able to call the service itself and obtain meaningful results. What you will learn:

- How to create a ROS service in the RAPP Platform and how to incorporate its launcher in the RAPP Platform launchers in order for the service to start along with the other services of the RAPP Platform
- How to call another ROS service within your service
- How to create a ROS test for your service
- How to create a HOP service for the ROS service that you created. The HOP service will be exposed on the network and will allow you to call the service over an http web request.

Section 1: Creating a ROS RAPP Platform example service

In this section we will cover how to create a basic ROS RAPP Platform service. Let us assume that the service we should to create takes as input two integer number, adds them together and returns the result. It also takes as input a string containing the username of a user and it returns a string containing the ontology alias of the user, by making a call to the appropriate ROS RAPP Platform service.

Create a ROS Node

The first thing to do is to create a new ROS Node which will be responsible for the service we want to implement. In order to create a new ROS node, navigate within the rapp-platform directory and create a new package:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/ $ catkin_create_pkg rapp_example_service std_msgs rospy rapp_platform_ros_communications “‘

This command creates a new package named rapp_example_service and adds the its dependencies upon the std-_msgs, rospy and rapp_platform_ros_communications packages. You can now navigate within the newly created package, and create a directory named cfg:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_example_service/ $ mkdir cfg $ cd cfg $ touch rapp_example_service_params.yaml $ echo ’rapp_example_service_topic: /rapp/rapp_example_-service/add_two_integers’ >> rapp_example_service_params.yaml “‘

We declared was the name of the service within the .yaml file. Now we will navigate back into the rapp_example_-service package and create the launch dir.

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_example_service $ mkdir launch $ cd launch $ touch rapp_example_service.launch “‘

The content of `rapp_example_service.launch` follows:

“‘xml <launch> <node name="rapp_example_service_node" pkg="rapp_example_service" type="rapp_example-_service_main.py" output="screen"> <rosparam file="$(find rapp_example_service)/cfg/rapp_example_service_-params.yaml" command="load"> </launch> “‘

This file declares the launcher of the service.

Now, we will create the source code files according to the coding standards of the RAPP project.

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_example_service/src $ touch rapp_-example_service_main.py “‘

The content of `rapp_example_service_main.py` follows

“‘python #!/usr/bin/env python

import rospy from rapp_example_service import ExampleService

if **name** == "__main__": rospy.init_node(’ExampleService’) ExampleServicesNode = ExampleService() rospy.spin() “‘

We have declared the main function of the ROS node and launched the ROS node with the rospy.spin() command. Now we will create the `rapp_example_service.py` below:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_example_service/src $ touch rapp_-example_service.py “‘

The content of `rapp_example_service.py` follows:

“‘python #!/usr/bin/env python

import rospy

from add_two_integers import AddTwoIntegers

from rapp_platform_ros_communications.srv import (tutorialExampleServiceSrv, tutorialExampleServiceSrv-Request, tutorialExampleServiceSrvResponse)

class ExampleService:

```
def __init__(self):
    self.serv_topic = rospy.get_param('rapp_knowrob_wrapper_create_ontology_alias')
    if(not self.serv_topic):
        rospy.logerror("rapp_knowrob_wrapper_create_ontology_alias param not found")
    rospy.wait_for_service(self.serv_topic)

    self.serv_topic = rospy.get_param("rapp_example_service_topic")
    if(not self.serv_topic):
        rospy.logerror("rapp_example_service_topic")
    self.serv=rospy.Service(self.serv_topic, tutorialExampleServiceSrv, self.tutorialExampleServiceDataHandler

def tutorialExampleServiceDataHandler(self,req):
    res = tutorialExampleServiceSrvResponse()
    it = AddTwoIntegers()
    res=it.addTwoIntegersFunction(req)
    return res
```

“‘

In this file, we initially declare python dependencies, one of them is the AddTwoIntegers class which we will define in a new file next. As you can see, we have imported an srv message files that needs to be created and declared in the rapp_platform_ros_communications package. We will cover this shortly. We also imported the name of the service from the .yaml file located in the cfg folder by the rospy.get_param() command. As our service depends on the `rapp_knowrob_wrapper_create_ontology_alias` we have it wait until it is available. The return message of the service is handled by the DataHandler function we created, named tutorialExampleServiceDataHandler.

The last file we need to create is the `add_two_integers.py`.

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_example_service/src $ touch add_two_integers.py “‘

The content of `add_two_integers.py` follows:

“‘python #!/usr/bin/env python

import rospy from rapp_platform_ros_communications.srv import (tutorialExampleServiceSrv, tutorialExampleServiceSrvRequest, tutorialExampleServiceSrvResponse, createOntologyAliasSrv, createOntologyAliasSrvRequest, createOntologyAliasSrvResponse)

class AddTwoIntegers:

```
def addTwoIntegersFunction(self,req):
    res = tutorialExampleServiceSrvResponse()
    res.additionResult=req.a+req.b
    res.userOntologyAlias=self.getUserOntologyAlias(req.username)
    return res

def getUserOntologyAlias(self,username):
    serv_topic = rospy.get_param('rapp_knowrob_wrapper_create_ontology_alias')
    knowrob_service = rospy.ServiceProxy(serv_topic, createOntologyAliasSrv)
    createOntologyAliasReq = createOntologyAliasSrvRequest()
    createOntologyAliasReq.username=username
    createOntologyAliasResponse = knowrob_service(createOntologyAliasReq)
    return createOntologyAliasResponse.ontology_alias
```

“‘

In this file, initially we define the imports again and following we define the AddTwoIntegers class and within it a simple function named addTwoIntegersFunction that accepts the service requirements as input and returns the service response. We initially construct the response named res, and we assign to the addition value the addition of the parameters a and b. The function getUserOntologyAlias defined, is tasked with calling an existing RAPP Platform service, the `rapp_knowrob_wrapper_create_ontology_alias`. This service will create an ontology alias for a user in case it does not already exist, or simply return the existing one if it already exists. The argument to call the service is the user’s username. In this way, by calling this existing RAPP Platform service, we obtain the userOntologyAlias of a user given his username, and we assign it to the appropriate service response variable.

Create the service srv file

The srv file defines what the service will accept as input and what it will return as output. These must be declared in a special file, called the service’s srv file. RAPP Platform conveniently places all ROS service srv files, as well as ROS service message files in the

“‘ /rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_platform_ros_communications/ “‘

package. Let us assume that the service we should to create takes as input two integer number, adds them together and returns the result. First we need to create the .srv file declaring the inputs and outputs of the service as shown:

“‘bash $ cd /rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_platform_ros_communications/srv $ mkdir ExampleServices $ cd ExampleServices $ touch tutorialExampleServiceSrv.srv “‘

The content of `tutorialExampleServiceSrv.srv` follows:

“‘yaml Header header int32 a int32 b

string username

string userOntologyAlias int32 additionResult string error “‘

Toe declare the .srv file in the package, open the `/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform_platform_ros_communications/rapp_platform_ros_communications/CMakeLists.txt` file and add the following line within the `add_service_files()` block,

“‘ ExampleServices/tutorialExampleServiceSrv.srv “‘

The whole file now should look like this:

“‘cmake cmake_minimum_required(VERSION 2.8.3) project(rapp_platform_ros_communications) set(ROS_BUILD_TYPE Release)

find_package(catkin REQUIRED COMPONENTS message_generation message_runtime std_msgs geometry_msgs nav_msgs)

Declare ROS messages, services and actions

Generate messages in the ’msg’ folder

add_message_files(FILES StringArrayMsg.msg CognitiveExercisePerformanceRecordsMsg.msg MailMsg.msg NewsStoryMsg.msg WeatherForecastMsg.msg ArrayCognitiveExercisePerformanceRecordsMsg.msg CognitiveExercisesMsg.msg)

Generate services in the ’srv’ folder

add_service_files(FILES

/ExampleServices/tutorialExampleServiceSrv.srv

/CognitiveExercise/testSelectorSrv.srv /CognitiveExercise/recordUserCognitiveTestPerformanceSrv.srv /CognitiveExercise/cognitiveTestCreatorSrv.srv /CognitiveExercise/userScoresForAllCategoriesSrv.srv /CognitiveExercise/userScoreHistoryForAllCategoriesSrv.srv /CognitiveExercise/returnTestsOfTypeSubtypeDifficultySrv.srv

/HumanDetection/HumanDetectionRosSrv.srv

/CaffeWrapper/imageClassificationSrv.srv /CaffeWrapper/ontologyClassBridgeSrv.srv /CaffeWrapper/registerImageToOntologySrv.srv

/DbWrapper/checkIfUserExistsSrv.srv /DbWrapper/getUserOntologyAliasSrv.srv /DbWrapper/getUserLanguageSrv.srv /DbWrapper/registerUserOntologyAliasSrv.srv /DbWrapper/getUserPasswordSrv.srv /DbWrapper/getUsernameAssociatedWithApplicationTokenSrv.srv /DbWrapper/createNewPlatformUserSrv.srv /DbWrapper/createNewApplicationTokenSrv.srv /DbWrapper/checkActiveApplicationTokenSrv.srv /DbWrapper/checkActiveRobotSessionSrv.srv /DbWrapper/addStoreTokenToDeviceSrv.srv /DbWrapper/validateUserRoleSrv.srv /DbWrapper/validateExistingPlatformDeviceTokenSrv.srv /DbWrapper/removePlatformUserSrv.srv /DbWrapper/createNewCloudAgentServiceSrv.srv /DbWrapper/createNewCloudAgentSrv.srv /DbWrapper/getCloudAgentServiceTypeAndHostPortSrv.srv

/OntologyWrapper/createInstanceSrv.srv /OntologyWrapper/ontologySubSuperClassesOfSrv.srv /OntologyWrapper/ontologyIsSubSuperClassOfSrv.srv /OntologyWrapper/ontologyLoadDumpSrv.srv /OntologyWrapper/ontologyInstancesOfSrv.srv /OntologyWrapper/assertRetractAttributeSrv.srv /OntologyWrapper/returnUserInstancesOfClassSrv.srv /OntologyWrapper/createOntologyAliasSrv.srv /OntologyWrapper/userPerformanceCognitveTestsSrv.srv /OntologyWrapper/createCognitiveExerciseTestSrv.srv /OntologyWrapper/cognitiveTestsOfTypeSrv.srv /OntologyWrapper/recordUserPerformanceCognitiveTestsSrv.srv /OntologyWrapper/clearUserPerformanceCognitveTestsSrv.srv /OntologyWrapper/registerImageObjectToOntologySrv.srv /OntologyWrapper/retractUserOntologyAliasSrv.srv

/FaceDetection/FaceDetectionRosSrv.srv

/NewsExplorer/NewsExplorerSrv.srv /Geolocator/GeolocatorSrv.srv

/WeatherReporter/WeatherReporterCurrentSrv.srv /WeatherReporter/WeatherReporterForecastSrv.srv

/QrDetection/QrDetectionRosSrv.srv

/Email/SendEmailSrv.srv /Email/ReceiveEmailSrv.srv

/SpeechDetectionGoogleWrapper/SpeechToTextSrv.srv

/SpeechDetectionSphinx4Wrapper/SpeechRecognitionSphinx4Srv.srv /SpeechDetectionSphinx4Wrapper/SpeechRecognitionSphinx4ConfigureSrv.srv /SpeechDetectionSphinx4Wrapper/SpeechRecognitionSphinx4TotalSrv.srv

/AudioProcessing/AudioProcessingDenoiseSrv.srv /AudioProcessing/AudioProcessingSetNoiseProfileSrv.srv /AudioProcessing/AudioProcessingDetectSilenceSrv.srv /AudioProcessing/AudioProcessingTransformAudioSrv.srv

/TextToSpeechEspeak/TextToSpeechSrv.srv

/HazardDetection/LightCheckRosSrv.srv /HazardDetection/DoorCheckRosSrv.srv

/PathPlanning/PathPlanningRosSrv.srv /Costmap2d/Costmap2dRosSrv.srv /PathPlanning/MapServer/MapServerGetMapRosSrv.srv /PathPlanning/MapServer/MapServerUploadMapRosSrv.srv

/ApplicationAuthentication/UserTokenAuthenticationSrv.srv /ApplicationAuthentication/UserLoginSrv.srv /ApplicationAuthentication/AddNewUserFromStoreSrv.srv /ApplicationAuthentication/AddNewUserFromPlatformSrv.srv)

Generate added messages and services with any dependencies listed here

generate_messages(DEPENDENCIES std_msgs # Or other packages containing msgs geometry_msgs nav_msgs)

catkin specific configuration

The catkin_package macro generates cmake config files for your package

Declare things to be passed to dependent projects

INCLUDE_DIRS: uncomment this if you package contains header files

LIBRARIES: libraries you create in this project that dependent projects also need

CATKIN_DEPENDS: catkin_packages dependent projects also need

DEPENDS: system dependencies of this project that dependent projects also need

catkin_package(

INCLUDE_DIRS include

LIBRARIES rapp_platform_ros_communications

CATKIN_DEPENDS other_catkin_pkg

CATKIN_DEPENDS message_generation message_runtime std_msgs geometry_msgs) “‘

This line will declare the srv file we created and stage it to be compiled.

Now we need to recompile the package. We will navigate to the root RAPP Platform catkin workspace directory and compile the code as shown below:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/ $ catkin_make –pkg rapp_platform_ros_communications “‘

this will recompile only the specific package. Sometimes it is preferable to recompile the whole project, in that case please delete the folders build and devel and compile the whole project, as shown below:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/ $ rm -rf ./build ./devel $ catkin_make “‘

Launch and manually test the service

In order to test that our ROS service works:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/ $ roslaunch rapp_example_service rapp_example_service.launch “‘

With our service launched, use a different terminal and type:

“‘bash $ rosservice call /rapp/rapp_example_service/add_two_integers “‘

and immediately press tab twice in order to auto complete the service requirements. Please assign values on the parameters a and b and in the username put ’rapp’ and hit enter. Voila! You can see the result in the additionResult parameter and you can see the username of the user rapp in the userOntologyAlias parameter. In case you use a different username which does not exist, the field will return blank, ”.

Create a ROS test for the service

Now we will create a ROS test that will perform a test to ensure our service is working correctly. The first thing to do is edit the `CMakeLists.txt` of the rapp_example_service package, to incorporate the testing dependencies. The file located at

“‘ ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_example_service/CMakeLists.txt “‘

must now contain the following content:

“‘cmake cmake_minimum_required(VERSION 2.8.3) project(rapp_example_service)

Find catkin macros and libraries

if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)

is used, also find other catkin packages

find_package(catkin REQUIRED COMPONENTS rapp_platform_ros_communications rospy std_msgs rostest)

catkin specific configuration

catkin_package(CATKIN_DEPENDS rospy std_msgs rostest rapp_platform_ros_communications INCLUDE_DIRS)

Build

include_directories(${catkin_INCLUDE_DIRS})

Build

if (CATKIN_ENABLE_TESTING) #catkin_add_nosetests(tests/cognitive_exercise_system_services_functional_-tests.py) add_rostest(tests/example_service_tests.launch) endif() “‘

As you can see we have added rostest in the dependencies and enabled testing. Similarly, the file located at

“‘ ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_example_service/package.xml “‘

must be updated in order to contain the following:

“‘xml <?xml version="1.0"?> <package> <name>rapp_example_service</name> <version>0.0.0</version> The rapp_example_service package

<maintainer email="thanos@todo.todo">thanos</maintainer>

<license>TODO</license>

<buildtool_depend>catkin</buildtool_depend> <build_depend>rapp_platform_ros_communications</build-_depend> <build_depend>rospy</build_depend> <build_depend>std_msgs</build_depend> <run_-depend>rapp_platform_ros_communications</run_depend> <run_depend>rospy</run_depend> <run_-depend>std_msgs</run_depend> <run_depend>rostest</run_depend> <build_depend>rostest</build_-depend>

<export>

</export> </package> “‘

We have added the rostest run and build dependencies. Now we can proceed to creating the test source files. Navigate into the package folder and create the necessary files:

“‘bash cd /rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_example_service mkdir tests touch example_service_tests.launch touch example_service_tests.py “‘

The content of the `example_service_tests.launch` file is:

“‘xml <launch> <test time-limit="100" test-name="rapp_example_service_tests" pkg="rapp_example_service" type="example_service_tests.py"> </launch> “‘

The launch file declares the other ROS node dependencies and the source file of tests to run. The content of the `example_service_tests.py` file is:

“‘python #!/usr/bin/env python

#Copyright 2015 RAPP

PKG=’rapp_example_service’

import sys import unittest import rospy import roslib

from rapp_platform_ros_communications.srv import (tutorialExampleServiceSrv, tutorialExampleServiceSrv-Request, tutorialExampleServiceSrvResponse)

class ExampleServiceTests(unittest.TestCase):

```
def test_example_service_basic(self):
    ros_service = rospy.get_param("rapp_example_service_topic")
    rospy.wait_for_service(ros_service)

    test_service = rospy.ServiceProxy(ros_service, tutorialExampleServiceSrv)

    req = tutorialExampleServiceSrvRequest()
    req.a=10
```

```
req.b=25
req.username="rapp"
response = test_service(req)
self.assertEqual(response.userOntologyAlias, "Person_DpphmPqg")
self.assertEqual(response.additionResult, 35)
```

The main function. Initializes the Cognitive Exercise System functional tests

if **name** == '**main**': import rosunit rosunit.unitrun(PKG, 'ExampleServiceTests', ExampleServiceTests) "'

We have created a python unit test, where the input is checked against the expected output in order to validate the results and consequently that the service is working correctly. The last thing to do now is to perform the test. Please first recompile the project:

"'bash $ cd /rapp_platform/rapp-platform-catkin-ws/ $ rm -rf ./build ./devel $ catkin_make "'

And now, in order to launch the test run the command:

"'bash $ cd /rapp_platform/rapp-platform-catkin-ws/ $ catkin_make run_tests_rapp_example_service "'

The result should be that 1 test executed successfully.

Section 2: Create the HOP web service

Read on `How-to-create-a-HOP-service-for-a-ROS-service`, if you have not done so already.

Also, this web service implementation requires to have previously read on the `simple-example`. The steps are the same, with a difference in the Web Service onrequest callback implementation.

Follow the steps described `here` to create the respective directories and source files.

The Web-Service response message includes the `sum_result` and `error` properties. The ROS Service request message has two integer properties, `a` and `b` and the `username` value.

"'js var clientRes = function(sum_result, user_ontology_alias, error) { sum_result = sum_result || 0; error = error || ''; user_ontology_alias = user_ontology_alias || '' return { sum_result: sum_result, user_ontology_alias: user_ontology_alias, error: error } }

var rosReqMsg = function(a, b, username) { a = a || 0; b = b || 0; username = username || ''; return { a: a, b: b, username: username } } "'

The rapp_example_service ROS Service url path is `/rapp/rapp_example_service/add_two_integers`

"'js var rosSrvUrlPath = "/rapp/rapp_example_service/add_two_integers" "'

Next you need to implement the WebService **onrequest** callback function. This is the function that will be called as long as a request for the rapp_example_web_service arrives.

A question comes. How will we obtain the `username` value to be passed to the ROS Service? The easy way is for the clients to pass the `username` value on the request body, but this means that the RAPP Application Authentication mechanisms break. As explained `here` and `here`, the **username** of the client can be found in `username` property of the `req` object, `req.username`, after authentication of the incoming client request.

Below is the implementation of the respective Web Service onrequest callback.

"'js function svcImpl(req, resp, ros) { // Get values of 'a' and 'b' from request body. var numA = req.body.a; var numB = req.body.b; var username = req.username;

// Create the rosMsg var rosMsg = new rosReqMsg(numA, numB, username);

/***

- ROS-Service response callback.

function callback(rosResponse){ // Get the sum result value from ROS Service response message. var response = clientRes(rosResponse.additionResult, rosResponse.userOntologyAlias); // Return the sum result, of numbers 'a'

and 'b', and the `user_ontology_alias` to the client. resp.sendJson(response); }

/***

- ROS-Service onerror callback.

function onerror(e){ // Respond a "Server Error". HTTP Error 501 - Internal Server Error resp.sendServerError(); }

/***

- Call ROS-Service.

ros.callService(rosSrvUrlPath, rosMsg, {success: callback, fail: onerror}); } ‘‘‘

Export the service onrequest callback function (svcImpl):

‘‘‘js module.exports = svcImpl; ‘‘‘

Add the `rapp_example_web_service` entry in `services.json` file:

‘‘‘json "rapp_example_web_service": { "launch": true, "anonymous": false, "name": "rapp_example_web_service", "url_name": "add_two_ints", "namespace": "", "ros_connection": true, "timeout": 45000 } ‘‘‘

The Web Service will listen at `http://localhost:9001/hop/add_two_ints` as defined by the `url_name` value.

You can set the url path for the Web Service to be `rapp_example_web_service/add_two_ints` by setting the url_name and namespace respectively:

‘‘‘json "rapp_example_web_service": { "launch": true, "anonymous": false, "name": "rapp_example_web_service", "url_name": "add_two_ints", "namespace": "rapp_example_web_service", "ros_connection": true, "timeout": 45000 } ‘‘‘

We have also set this Web Service to timeout after 45 seconds (`timeout`). This is critical when WebService-to--ROS communcation bridge breaks!

For simplicity, we will configure this WebService to be launched under an already existed Web Worker thread (**main-1**).

The `workers.json` file contains Web Workers entries. Add the `rapp_example_web_service` service under the `main-1` worker:

‘‘‘json "main-1": { "launch": true, "path": "workers/main1.js", "services": [... "rapp_example_web_service"] } ‘‘‘

The newly implemented `rapp_example_web_service` Web Service is ready to be launched. Launch the RAPP Platform Web Server:

‘‘‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_web_services $ pm2 start server.yaml ‘‘‘

If you dont want to launch the Web Server using pm2 process manager, just execute the `run.sh` script in the same directory:

‘‘‘bash $ ./run.sh ‘‘‘

You will notice the following output from the logs:

‘‘‘bash info: [] Registered worker service {`http://localhost:9001/hop/add_two_ints`} under worker thread {main-1} info: [] { worker: 'main-1', path: '/hop/add_two_ints', url: '`http://localhost:9001/hop/add_two_ints'`, frame: [Function] } ‘‘‘

All set! The RAPP Platform accepts requests for the `rapp_example_web_service` at `http://rapp-platform-local:9001/hop/add_two_ints`

You can test it using `curl` from commandline:

‘‘‘bash $ curl –data "a=100&b=20" `http://localhost:9001/hop/add_two_ints` ‘‘‘

Notice that the RAPP Platform will return `Authentication Failure` (HTTP 401 Unauthorized Error). This is because the RAPP Platform uses token-based application authentication mechanisms to authenticate incoming

client requests. You will have to pass a valid token to the **request headers**. By default, the RAPP Platform database includes a user `rapp` and the token is **rapp_token**. Pass that token value to the `Accept-Token` field of the request header:

“‘bash $ curl -H "Accept-Token: rapp_token" –data "a=100&b=20" `http://localhost:9001/hop/add_two_ints` “‘

The output should now be similar to:

“‘bash {sum_result: 120, user_ontology_alias: "THE_USER_ONTOLOGY_ALIAS", error: ""} “‘

Section 3: Update the RAPP Platform API

First, read on `How to write the API for a HOP service`, if you have not done so already.

The AddTwoInts Cloud Message class is identical to the **simple-example** presented `here`, with the addition of `user_ontology_alias` property to the `AddTwoInts.Response` class:

“‘python from RappCloud.Objects import (File, Payload)

from Cloud import (CloudMsg, CloudRequest, CloudResponse)

class AddTwoInts(CloudMsg): """ Add Two Integers Exqample CloudMsg object"""

class Request(CloudRequest): """ Add Two Integers Cloud Request object. AddTwoInts.Request """ def **init**(self, ∗∗kwargs): """! Constructor

Parameters

∗∗*kwargs*	- Keyword arguments. Apply values to the request attributes. • a • b"

Number #1

self.a = 0

Number #2

self.b = 0

Apply passed keyword arguments to the Request object.

super(AddTwoInts.Request, self).__init__(∗∗kwargs)

```
    def make_payload(self):
        """ Create and return the Payload of the Request. """
        return Payload(a=self.a, b=self.b)

    def make_files(self):
        """ Create and return Array of File objects of the Request. """
        return []

    class Response(CloudResponse):
        """ Add Two Integers Cloud Response object. AddTwoInts.Response """
        def __init__(self, **kwargs):
            """!
            Constructor
```

```
            @param **kwargs - Keyword arguments. Apply values to the request attributes.
            - @ref error
            - @ref sum_result
            """
            ## Error message
            self.error = ''
            ## The sum result of numbers a and b
            self.sum_result = 0
            ## User's ontology alias
            self.user_ontology_alias
            ## Apply passed keyword arguments to the Request object.
            super(AddTwoInts.Response, self).__init__(**kwargs)


    def __init__(self, **kwargs):
        """!
        Constructor

        @param **kwargs - Keyword arguments. Apply values to the request attributes.
            - @ref Request.a
            - @ref Request.b
        """

        # Create and hold the Request object for this CloudMsg
        self.req = AddTwoInts.Request()
        # Create and hold the Response object for this CloudMsg
        self.resp = AddTwoInts.Response()
        super(AddTwoInts, self).__init__(svcname='add_two_ints', **kwargs)
```

‘‘‘

Similar to the simple-example, append the following line of code in the `RappCloud/CloudMsgs/__init__.py` file:

‘‘‘python from AddTwoInts import AddTwoInts ‘‘‘

Now everything is in place to call the newly created RAPP Platform Service, using the python implementation of the rapp-platform-api. An example is presented below:

‘‘‘python from RappCloud.CloudMsgs import AddTwoInts from RappCloud import RappPlatformService

svcClient = RappPlatformService(persistent=True, timeout=1000) msg = AddTwoInts(a=5, b=4)

response svcClient.call(msg)

print response.sum_result

9

print response.user_ontology_alias

"THE_USER_ONTOLOGY_ALIAS"

‘‘‘

Chapter 4

A-full-RAPP-Platform-service-creation-example

In this example, a basic guide will be presented that will cover the basics of creating an example RAPP Platform service. We assume that you have successfully installed the RAPP Platform and run the tests, if not please see our installation guide located at

https://github.com/rapp-project/rapp-platform/wiki/How-can-I-set-up-the-RAPP-Platform-in-my-PC%3F

It is addressed to the novice user and as such, experienced users may find it too detailed or need to skip over some sections. This tutorial assumes some basic ROS knowledge. Users not familiar with ROS are advised to first complete relevant ROS tutorials that can be found at

http://wiki.ros.org/ROS/Tutorials

By the end of the tutorial you should be able to call the service itself and obtain meaningful results. What you will learn:

- How to create a ROS service in the RAPP Platform, how to launch it and how to call it and obtain its response.
- How to create a HOP service for the ROS service that you created. The HOP service will be exposed on the network and will allow you to call the service over an http web request.

Section 1: Creating a ROS RAPP Platform example service

In this section we will cover how to create a basic ROS RAPP Platform service. Let us assume that the service we should to create takes as input two integer number, adds them together and returns the result.

Create a ROS Node

The first thing to do is to create a new ROS Node which will be responsible for the service we want to implement. In order to create a new ROS node, navigate within the rapp-platform directory and create a new package:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/ $ catkin_create_pkg rapp_example_service std_msgs rospy rapp_platform_ros_communications “‘

This command creates a new package named rapp_example_service and adds the its dependencies upon the std-_msgs, rospy and rapp_platform_ros_communications packages. You can now navigate within the newly created package, and create a directory named cfg:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_example_service/ $ mkdir cfg $ cd cfg $ touch rapp_example_service_params.yaml $ echo ’rapp_example_service_topic: /rapp/rapp_example_-service/add_two_integers’ >> rapp_example_service_params.yaml “‘

We declared was the name of the service within the .yaml file. Now we will navigate back into the rapp_example_-service package and create the launch dir.

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_example_service $ mkdir launch $ cd launch $ touch rapp_example_service.launch “‘

The content of `rapp_example_service.launch` follows:

“‘xml <launch> <node name="rapp_example_service_node" pkg="rapp_example_service" type="rapp_example_service_main.py" output="screen"> <rosparam file="$(find rapp_example_service)/cfg/rapp_example_service_params.yaml" command="load"> </launch> “‘

This file declares the launcher of the service.

Now, we will create the source code files according to the coding standards of the RAPP project.

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_example_service/src $ touch rapp_example_service_main.py “‘

The content of `rapp_example_service_main.py` follows

“‘python #!/usr/bin/env python

import rospy from rapp_example_service import ExampleService

if **name** == "__main__": rospy.init_node('ExampleService') ExampleServicesNode = ExampleService() rospy.spin() “‘

We have declared the main function of the ROS node and launched the ROS node with the rospy.spin() command. Now we will create the `rapp_example_service.py` below:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_example_service/src $ touch rapp_example_service.py “‘

The content of `rapp_example_service.py` follows:

“‘python #!/usr/bin/env python

import rospy from add_two_integers import AddTwoIntegers from rapp_platform_ros_communications.srv import (tutorialExampleServiceSrv, tutorialExampleServiceSrvRequest, tutorialExampleServiceSrvResponse)

class ExampleService:

```
def __init__(self):
    self.serv_topic = rospy.get_param("rapp_example_service_topic")
    if(not self.serv_topic):
        rospy.logerror("rapp_example_service_topic")
    self.serv=rospy.Service(self.serv_topic, tutorialExampleServiceSrv, self.tutorialExampleServiceDataHandler

def tutorialExampleServiceDataHandler(self,req):
    res = tutorialExampleServiceSrvResponse()
    it = AddTwoIntegers()
    res=it.addTwoIntegersFunction(req)
    return res
```

“‘

In this file, we initially declare python dependencies, one of them is the AddTwoIntegers class which we will define in a new file next. As you can see, we have imported an srv message files that needs to be created and declared in the rapp_platform_ros_communications package. We will cover this shortly. We also imported the name of the service from the .yaml file located in the cfg folder by the rospy.get_param() command. The return message of the service is handled by the DataHandler function we created, named tutorialExampleServiceDataHandler.

The last file we need to create is the `add_two_integers.py`.

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_example_service/src $ touch add_two_integers.py “‘

The content of `add_two_integers.py` follows:

“‘python #!/usr/bin/env python

import rospy

from rapp_platform_ros_communications.srv import (tutorialExampleServiceSrv, tutorialExampleServiceSrvRequest, tutorialExampleServiceSrvResponse)

class AddTwoIntegers:

```
def addTwoIntegersFunction(self,req):
    res = tutorialExampleServiceSrvResponse()
    res.additionResult=req.a+req.b
    return res
```

“‘

In this file, initially we define the imports again and following we define the AddTwoIntegers class and within it a simple function named addTwoIntegersFunction that accepts the service requirements as input and returns the service response. We initially construct the response named res, and we assign to the addition value the addition of the parameters a and b.

Create the service srv file

The srv file defines what the service will accept as input and what it will return as output. These must be declared in a special file, called the service’s srv file. RAPP Platform conveniently places all ROS service srv files, as well as ROS service message files in the

```
/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_platform_ros_communications/
```

package. Let us assume that the service we should to create takes as input two integer number, adds them together and returns the result. First we need to create the .srv file declaring the inputs and outputs of the service as shown:

“‘bash $ cd /rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_platform_ros_communications/srv $ mkdir ExampleServices $ cd ExampleServices $ touch tutorialExampleServiceSrv.srv “‘

The content of `tutorialExampleServiceSrv.srv` follows:

“‘yaml Header header int32 a

int32 b

int32 additionResult “‘

Toe declare the .srv file in the package, open the `/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform_platform_ros_communications/rapp_platform_ros_communications/CMakeLists.txt` file and add the following line within the `add_service_files()` block,

“‘ ExampleServices/tutorialExampleServiceSrv.srv “‘

The whole file now should look like this:

“‘cmake cmake_minimum_required(VERSION 2.8.3) project(rapp_platform_ros_communications) set(ROS_BUILD_TYPE Release)

find_package(catkin REQUIRED COMPONENTS message_generation message_runtime std_msgs geometry_msgs nav_msgs)

Declare ROS messages, services and actions

Generate messages in the ’msg’ folder

add_message_files(FILES StringArrayMsg.msg CognitiveExercisePerformanceRecordsMsg.msg MailMsg.msg NewsStoryMsg.msg WeatherForecastMsg.msg ArrayCognitiveExercisePerformanceRecordsMsg.msg CognitiveExercisesMsg.msg)

Generate services in the 'srv' folder

add_service_files(FILES

/ExampleServices/tutorialExampleServiceSrv.srv

/CognitiveExercise/testSelectorSrv.srv /CognitiveExercise/recordUserCognitiveTestPerformanceSrv.srv /CognitiveExercise/cognitiveTestCreatorSrv.srv /CognitiveExercise/userScoresForAllCategoriesSrv.srv /CognitiveExercise/userScoreHistoryForAllCategoriesSrv.srv /CognitiveExercise/returnTestsOfTypeSubtypeDifficultySrv.srv

/HumanDetection/HumanDetectionRosSrv.srv

/CaffeWrapper/imageClassificationSrv.srv /CaffeWrapper/ontologyClassBridgeSrv.srv /CaffeWrapper/registerImageToOntologySrv.srv

/DbWrapper/checkIfUserExistsSrv.srv /DbWrapper/getUserOntologyAliasSrv.srv /DbWrapper/getUserLanguageSrv.srv /DbWrapper/registerUserOntologyAliasSrv.srv /DbWrapper/getUserPasswordSrv.srv /DbWrapper/getUsernameAssociatedWithApplicationTokenSrv.srv /DbWrapper/createNewPlatformUserSrv.srv /DbWrapper/createNewApplicationTokenSrv.srv /DbWrapper/checkActiveApplicationTokenSrv.srv /DbWrapper/checkActiveRobotSessionSrv.srv /DbWrapper/addStoreTokenToDeviceSrv.srv /DbWrapper/validateUserRoleSrv.srv /DbWrapper/validateExistingPlatformDeviceTokenSrv.srv /DbWrapper/removePlatformUserSrv.srv /DbWrapper/createNewCloudAgentServiceSrv.srv /DbWrapper/createNewCloudAgentSrv.srv /DbWrapper/getCloudAgentServiceTypeAndHostPortSrv.srv

/OntologyWrapper/createInstanceSrv.srv /OntologyWrapper/ontologySubSuperClassesOfSrv.srv /OntologyWrapper/ontologyIsSubSuperClassOfSrv.srv /OntologyWrapper/ontologyLoadDumpSrv.srv /OntologyWrapper/ontologyInstancesOfSrv.srv /OntologyWrapper/assertRetractAttributeSrv.srv /OntologyWrapper/returnUserInstancesOfClassSrv.srv /OntologyWrapper/createOntologyAliasSrv.srv /OntologyWrapper/userPerformanceCognitveTestsSrv.srv /OntologyWrapper/createCognitiveExerciseTestSrv.srv /OntologyWrapper/cognitiveTestsOfTypeSrv.srv /OntologyWrapper/recordUserPerformanceCognitiveTestsSrv.srv /OntologyWrapper/clearUserPerformanceCognitveTestsSrv.srv /OntologyWrapper/registerImageObjectToOntologySrv.srv /OntologyWrapper/retractUserOntologyAliasSrv.srv

/FaceDetection/FaceDetectionRosSrv.srv

/NewsExplorer/NewsExplorerSrv.srv /Geolocator/GeolocatorSrv.srv

/WeatherReporter/WeatherReporterCurrentSrv.srv /WeatherReporter/WeatherReporterForecastSrv.srv

/QrDetection/QrDetectionRosSrv.srv

/Email/SendEmailSrv.srv /Email/ReceiveEmailSrv.srv

/SpeechDetectionGoogleWrapper/SpeechToTextSrv.srv

/SpeechDetectionSphinx4Wrapper/SpeechRecognitionSphinx4Srv.srv /SpeechDetectionSphinx4Wrapper/SpeechRecognitionSphinx4ConfigureSrv.srv /SpeechDetectionSphinx4Wrapper/SpeechRecognitionSphinx4TotalSrv.srv

/AudioProcessing/AudioProcessingDetectSilenceSrv.srv /AudioProcessing/AudioProcessingTransformAudioSrv.srv

/TextToSpeechEspeak/TextToSpeechSrv.srv

/HazardDetection/LightCheckRosSrv.srv /HazardDetection/DoorCheckRosSrv.srv

/PathPlanning/PathPlanningRosSrv.srv /Costmap2d/Costmap2dRosSrv.srv /PathPlanning/MapServer/MapServerGetMapRosSrv.srv /PathPlanning/MapServer/MapServerUploadMapRosSrv.srv

/ApplicationAuthentication/UserTokenAuthenticationSrv.srv /ApplicationAuthentication/UserLoginSrv.srv /ApplicationAuthentication/AddNewUserFromStoreSrv.srv /ApplicationAuthentication/AddNewUserFromPlatformSrv.srv)

Generate added messages and services with any dependencies listed here

generate_messages(DEPENDENCIES std_msgs # Or other packages containing msgs geometry_msgs nav_msgs
)

catkin specific configuration

The catkin_package macro generates cmake config files for your package

Declare things to be passed to dependent projects

INCLUDE_DIRS: uncomment this if you package contains header files

LIBRARIES: libraries you create in this project that dependent projects also need

CATKIN_DEPENDS: catkin_packages dependent projects also need

DEPENDS: system dependencies of this project that dependent projects also need

catkin_package(

INCLUDE_DIRS include

LIBRARIES rapp_platform_ros_communications

CATKIN_DEPENDS other_catkin_pkg

CATKIN_DEPENDS message_generation message_runtime std_msgs geometry_msgs) “‘

This line will declare the srv file we created and stage it to be compiled.

Now we need to recompile the package. We will navigate to the root RAPP Platform catkin workspace directory and compile the code as shown below:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/ $ catkin_make –pkg rapp_platform_ros_communications “‘

this will recompile only the specific package. Sometimes it is preferable to recompile the whole project, in that case please delete the folders build and devel and compile the whole project, as shown below:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/ $ rm -rf ./build ./devel $ catkin_make “‘

Launch and manually test the service

In order to test that our ROS service works:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/ $ roslaunch rapp_example_service rapp_example_service.-launch “‘

With our service launched, use a different terminal and type:

“‘bash $ rosservice call /rapp/rapp_example_service/add_two_integers “‘

and immediately press tab twice in order to auto complete the service requirements. Please assign values on the parameters a and b and hit enter. Voila! You can see the result in the additionResult parameter.

Section 2: Create the HOP web service

Read on How-to-create-a-HOP-service-for-a-ROS-service, if you have not done so already.

Head to the `rapp_web_services` directory of the `rapp-platform` and create the source file for the Web Service implementation:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_web_services/services $ mkdir example_web_service $ cd rapp_example_web_service $ touch svc.js “‘

Open the **svc.js** file with your favorite editor and implement the **client-response** and **ros-msg** objects. The Web--Service response message inlcude the `sum_result` and `error` properties. The ROS Service request message has two integer properties, `a` and `b`

“‘js var clientRes = function(sum_result, error) { sum_result = sum_result || 0; error = error || ”; return { sum_result: sum_result, error: error } }

var rosReqMsg = function(a, b) { a = a || 0; b = b || 0; return { a: a, b: b } } “‘

The rapp_example_service ROS Service url path is `/rapp/rapp_example_service/add_two_integers`

“‘js var rosSrvUrlPath = "/rapp/rapp_example_service/add_two_integers" “‘

Next you need to implement the WebService **onrequest** callback function. This is the function that will be called as long as a request for the rapp_example_web_service arrives.

“‘js function svcImpl(req, resp, ros) { // Get values of ’a’ and ’b’ from request body. var numA = req.body.a; var numB = req.body.b;

// Create the rosMsg var rosMsg = new rosReqMsg(numA, numB);

/***

- ROS-Service response callback.

function callback(rosResponse){ // Get the sum result value from ROS Service response message. var response = clientRes(rosResponse.additionResult); // Return the sum result, of numbers ’a’ and ’b’, to the client. resp.sendJson(response); }

/***

- ROS-Service onerror callback.

function onerror(e){ // Respond a "Server Error". HTTP Error 501 - Internal Server Error resp.sendServerError(); }

/***

- Call ROS-Service.

ros.callService(rosSrvUrlPath, rosMsg, {success: callback, fail: onerror}); } “‘

Export the service onrequest callback function (svcImpl):

“‘js module.exports = svcImpl; “‘

Add the `rapp_example_web_service` entry in `services.json` file:

“‘json "rapp_example_web_service": { "launch": true, "anonymous": false, "name": "rapp_example_web_service", "url_name": "add_two_ints", "namespace": "", "ros_connection": true, "timeout": 45000 } “‘

The Web Service will listen at `http://localhost:9001/hop/add_two_ints` as defined by the `url_name` value.

You can set the url path for the Web Service to be `rapp_example_web_service/add_two_ints` by setting the url_name and namespace respectively:

“‘json "rapp_example_web_service": { "launch": true, "anonymous": false, "name": "rapp_example_web_service", "url_name": "add_two_ints", "namespace": "rapp_example_web_service", "ros_connection": true, "timeout": 45000 } “‘

We have also set this Web Service to timeout after 45 seconds (`timeout`). This is critical when WebService-to--ROS communcation bridge breaks!

For simplicity, we will configure this WebService to be launched under an already existed Web Worker thread (**main-1**).

The `workers.json` file contains Web Workers entries. Add the `rapp_example_web_service` service under the `main-1` worker:

“‘json "main-1": { "launch": true, "path": "workers/main1.js", "services": [... "rapp_example_web_service"] } “‘

The newly implemented `rapp_example_web_service` Web Service is ready to be launched. Launch the RAPP Platform Web Server:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_web_services $ pm2 start server.yaml “‘

If you dont want to launch the Web Server using pm2 process manager, just execute the `run.sh` script in the same directory:

“‘bash $ ./run.sh “‘

You will notice the following output from the logs:

“‘bash info: [] Registered worker service {`http://localhost:9001/hop/add_two_ints`} under worker thread {main-1} info: [] { worker: ’main-1’, path: ’/hop/add_two_ints’, url: ’`http://localhost:9001/hop/add_two_ints’,` frame: [Function] } “‘

All set! The RAPP Platform accepts requests for the `rapp_example_web_service` at `http://rapp-platform-local:9001/hop/add_two_ints`

You can test it using `curl` from commandline:

“‘bash $ curl –data "a=100&b=20" `http://localhost:9001/hop/add_two_ints` “‘

Notice that the RAPP Platform will return `Authentication Failure` (HTTP 401 Unauthorized Error). This is because the RAPP Platform uses token-based application authentication mechanisms to authenticate incoming client requests. You will have to pass a valid token to the **request headers**. By default, the RAPP Platform database includes a user `rapp` and the token is **rapp_token**. Pass that token value to the `Accept-Token` field of the request header:

“‘bash $ curl -H "Accept-Token: rapp_token" –data "a=100&b=20" `http://localhost:9001/hop/add_two_ints` “‘

The output should now be:

“‘bash {sum_result: 120, error: ”} “‘

Section 3: Update the RAPP Platform API

First, read on `How to write the API for a HOP service`, if you have not done so already.

Head to the `RappCloud/CloudMsgs` directory of the RappCloud module and create a file named **AddTwoInts.py**

“‘bash $ cd ∼/rapp_platform/rapp_platform_catkin_ws/src/rapp-api/python/RappCloud/CloudMsgs $ touch AddTwoInts.py “‘

The contents of the **AddTwoInts.py** source file should be:

“‘python from RappCloud.Objects import (File, Payload)

from Cloud import (CloudMsg, CloudRequest, CloudResponse)

class AddTwoInts(CloudMsg): """ Add Two Integers Exqample CloudMsg object"""

class Request(CloudRequest): """ Add Two Integers Cloud Request object. AddTwoInts.Request """ def **init**(self, ∗∗kwargs): """! Constructor

Parameters

∗∗*kwargs*	- Keyword arguments. Apply values to the request attributes. • a • b"

Number #1

self.a = 0

Number #2

self.b = 0

Apply passed keyword arguments to the Request object.

super(AddTwoInts.Request, self).__init__(∗∗kwargs)

```
    def make_payload(self):
        """ Create and return the Payload of the Request. """
        return Payload(a=self.a, b=self.b)

    def make_files(self):
        """ Create and return Array of File objects of the Request. """
        return []

    class Response(CloudResponse):
        """ Add Two Integers Cloud Response object. AddTwoInts.Response """
        def __init__(self, **kwargs):
            """!
            Constructor

            @param **kwargs - Keyword arguments. Apply values to the request attributes.
            - @ref error
            - @ref sum_result
            """
            ## Error message
            self.error = ''
            ## The sum result of numbers a and b
            self.sum_result = 0
            ## Apply passed keyword arguments to the Request object.
            super(AddTwoInts.Response, self).__init__(**kwargs)


    def __init__(self, **kwargs):
        """!
        Constructor

        @param **kwargs - Keyword arguments. Apply values to the request attributes.
            - @ref Request.a
            - @ref Request.b
        """

        # Create and hold the Request object for this CloudMsg
        self.req = AddTwoInts.Request()
        # Create and hold the Response object for this CloudMsg
        self.resp = AddTwoInts.Response()
        super(AddTwoInts, self).__init__(svcname='add_two_ints', **kwargs)
```

‘‘‘

Finally append the following line of code in the `RappCloud/CloudMsgs/__init__.py` file:

“‘python from AddTwoInts import AddTwoInts “‘

Now everything is in place to call the newly created RAPP Platform Service, using the python implementation of the rapp-platform-api. An example is presented below:

“‘python from RappCloud.CloudMsgs import AddTwoInts from RappCloud import RappPlatformService

svcClient = RappPlatformService(persistent=True, timeout=1000) msg = AddTwoInts(a=5, b=4)

response svcClient.call(msg)

print response.sum_result

9

“‘

Chapter 5

Create-a-new-RAPP-user

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

It requires the RAPP Platform to be launched. “‘shell . ∼/rapp_platoform/rapp-platform-scripts/deploy/deploy_rapp-_ros.sh “‘

Then execute: “‘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
- password
- language (can be ommited) “‘

This will create the new user, log him in and return the session token. This should be saved in the proper robot folder as specified by the `RAPP-API documentation`

Chapter 6

(novice) Create-a-remote-robotic-application-for-NAO-in-Python-(novice)

This tutorial has a goal to introduce a novice programmer (who is not necessarily a roboticist) into the RApps concept (where RApps stands for Robotic Applications). Here, a simple application will be created for the NAO robot using the Python programming language. This application will be executed remotely in a PC (and not in the actual robot), in order to avoid any advanced in-robot modifications.

As always, the preparation steps are much more than the actual robotic application programming, so lets begin with this.

Preparation steps

For this tutorial we will use the following tools:

- the NaoQi Python libraries v2.1.4
- the `rapp-robots-api` Github repository

Of course the standard prerequisites are a functional installation of Ubuntu 14.04, an editor and a terminal.

NaoQi Python libraries setup

Since the application is going to be remotely deployed (not directly in the robot), the NaoQi Python libraries are needed in order to achieve communication from the PC to the NAO robot. If you are the owner of a NAO robot, you can download this package following `these instructions`. For this tutorial we will download the `Python 2.7 SDK 2.1.4 Linux 64` from Aldebaran's `software resources webpage`, after logging in with our credentials.

The downloaded file is `pynaoqi-python2.7-2.1.4.13-linux64.tar.gz` and lets assume that it was downloaded in `$HOME`. Next, we create a dedicated folder for our project, move the file there and untar it:

“‘ cd ∼ mkdir rapp_nao mv ∼/pynaoqi-python2.7-2.1.4.13-linux64.tar.gz ∼/rapp_nao cd ∼/rapp_nao tar -xvf pynaoqi-python2.7-2.1.4.13-linux64.tar.gz “‘

Next, the PYTHONPATH environmental variable must be updated with the location of the NaoQi libraries:

“‘ echo 'export PYTHONPATH=$PYTHONPATH:∼/rapp_nao/pynaoqi-python2.7-2.1.4.13-linux64' >> ∼/.bashrc source ∼/.bashrc “‘

Now, the NaoQi is ready for utilization. The next step is to setup the `rapp-robots-api` Python libraries.

RAPP Robots API libraries setup

The first step is to clone the appropriate GitHub repository:

“‘ cd ∼/rapp_nao git clone `https://github.com/rapp-project/rapp-robots-api.git` “‘

Similarly to the NaoQi Python libraries, the `PYTHONPATH` variable has to be updated:

“‘ echo ’export PYTHONPATH=$PYTHONPATH:∼/rapp_nao/rapp-robots-api/python/abstract_classes’ >> ∼/.bashrc echo ’export PYTHONPATH=$PYTHONPATH:∼/rapp_nao/rapp-robots-api/python/implementations/nao-_v4_naoqi2.1.4’ >> ∼/.bashrc source ∼/.bashrc “‘

The last step to configure the `rapp-robots-api` is to declare the NAO IP. It should be stated that the PC and NAO **must** be in the same LAN. In order to find the NAO IP just press once its chest button while in operation and the robot will dictate it.

The IP must be declared in the first line of `this` file, thus if the IP is `192.168.0.101` the `nao_connectivity` file located under `∼/rapp_nao/rapp-robots-api/python/implementations/nao_v4_-naoqi2.1.4/nao_connectivity` should contain:

“‘ 192.168.0.101 9559 “‘

Now all tools are in place to write our simple NAO Python application.

Writing a simple application

Let’s create a Python file for our application and give it execution rights:

“‘ cd ∼/rapp_nao mkdir rapps && cd rapps touch simple_app.py chmod +x simple_app.py “‘

The first step is to check if everything is in place. Write the following in the `simple_app.py` file:

“‘python #!/usr/bin/env python from rapp_robot_api import RappRobot rh = RappRobot() rh.audio.speak("Hello there!") “‘

If everything was set-up correctly the NAO robot should talk and say "Hello there!" to you. If not, one of the aforementioned instructions was not performed correctly (or if it they all were, please submit a bug to correct this tutorial!).

Now for the real application, you can use any of the documented API calls that exist `here`. Insert the following in the `simple_app.py` file:

“‘python #!/usr/bin/env python

Import the RAPP Robot API

from rapp_robot_api import RappRobot

Create an object in order to call the desired functions

rh = RappRobot()

Adjust the NAO master volume and ask for instructions. The valid commands are ’stand’ and ’sit’ and NAO waits for 5 seconds

rh.audio.setVolume(50) rh.audio.speak("Hello there! What do you want me to do? I can sit or get up.") res = rh.-audio.speechDetection([’sit’, ’get up’], 5) print res word = ” inner_word = ” if res[’error’] == None: word = res[’word’]

Check which command was dictated by the human

if word == ’sit’:

The motors must be enabled for NAO to move

rh.motion.enableMotors()

NAO sits with 75% of its maximum speed

rh.humanoid_motion.goToPosture('Sit', 0.75) elif word == 'get up':

The motors must be enabled for NAO to move

rh.motion.enableMotors()

NAO stands with 75% of its maximum speed

rh.humanoid_motion.goToPosture('Stand', 0.75) else:

No command was dictated or the command was not understood

pass

Ask the human what movement to do: move the hands or the head?

rh.audio.speak("Do you want me to move my arms or my head?") res = rh.audio.speechDetection(['arms', 'head'], 5) print res if res['error'] == None: word = res['word']

rh.motion.enableMotors() if word == 'arms': rh.audio.speak("Do you want me to open the left or right hand?") res = rh.audio.speechDetection(['left', 'right'], 5) print res if res['error'] == None: inner_word = res['word'] if inner_word == 'left': rh.humanoid_motion.openHand('Left') elif inner_word == 'right': rh.humanoid_motion.openHand('Right') else: pass

rh.audio.speak("I will close my hands now") rh.humanoid_motion.closeHand('Right') rh.humanoid_motion.closeHand('Left') elif word == 'head': rh.audio.speak("Do you want me to turn my head left or right?") res = rh.audio.speechDetection(['left', 'right'], 5) print res if res['error'] == None: inner_word = res['word']

The head moves by 0.4 rads left or right with 50% of its maximum speed

if inner_word == 'left': rh.humanoid_motion.setJointAngles(['HeadYaw'], [0.4], 0.5) elif inner_word == 'right': rh.humanoid_motion.setJointAngles(['HeadYaw'], [-0.4], 0.5) else: pass

rh.audio.speak("I will look straight now") rh.humanoid_motion.setJointAngles(['HeadYaw'], [0], 0.5) else: pass

rh.audio.speak("And now I will sit down and sleep!") rh.humanoid_motion.goToPosture('Sit', 0.7) rh.motion.disableMotors() "'

As you may have noticed the API calls are robot-agnostic, meaning that the developer (you) can create applications without having to specify on which robot they will be executed. If a specific function is not implemented in a robot that will execute this application, the specific command will simply have no effect.

Finally, the last step is to execute the RApp:

"' python ∼/rapp_nao/rapps/simple_app.py "'

This application can be found here.

Chapter 7

(novice) Create-an-in-robot-application-for-NAO-in--Python-(novice)

This tutorial has a goal to introduce a novice programmer (who is not necessarily a roboticist) into the RApps concept (where RApps stands for Robotic Applications). Here, a simple application will be created for the NAO robot using the Python programming language. This application will be executed in-robot, thus a real NAO robot is necessary.

Preparation steps

For this tutorial we will use the following tools:

- A real NAO robot
- the `rapp-robots-api` Github repository

Of course the standard prerequisites are a functional installation of Ubuntu 14.04, an editor and a terminal.

RAPP Robots API libraries setup

The first step is to clone the appropriate GitHub repository in your PC:

“‘bash mkdir ∼/rapp_nao cd ∼/rapp_nao git clone https://github.com/rapp-project/rapp-robots-api.-git “‘

The next step is to transfer the RAPP Python libraries to the NAO robot. This will be done via `scp`, assuming that the NAO robot's IP is `192.168.0.101` and username and password are `nao`:

“‘bash cd ∼/rapp_nao/ tar -zcvf rapp_api.tar.gz rapp-robots-api/ scp rapp_api.tar.gz nao@192.168.0.101-:∼/rapp_api.tar.gz “‘

Now connect in NAO via ssh by `ssh nao@192.168.0.101` giving `nao` as password. Then untar the API:

“‘bash tar -xvf rapp_api.tar.gz rm rapp_api.tar.gz “‘

The next step is to update the `PYTHONPATH` variable. Since NAO has Gentoo as OS, we will modify the `bash_-profile` file:

“‘bash echo ’export PYTHONPATH=$PYTHONPATH:∼/rapp-robots-api/python/abstract_classes’ >> /home/nao/.bash-_profile echo ’export PYTHONPATH=$PYTHONPATH:∼/rapp-robots-api/python/implementations/nao_v4_-naoqi2.1.4’ >> /home/nao/.bash_profile source ∼/.bash_profile “‘

The last step to configure the `rapp-robots-api` is to declare the NAO IP. Since the robot API is in-robot, the IP must be localhost: `127.0.0.1`.

The IP must be declared in the first line of `this` file, thus the `nao_connectivity` file located under `/home/nao/rapp-robots-api/python/implementations/nao_v4_naoqi2.1.4/nao-_connectivity` should contain:

“‘ 127.0.0.1 9559 “‘

Now all tools are in place to write our simple NAO Python application.

Writing a simple application

Let’s create a Python file for our application and give it execution rights:

“‘bash mkdir ∼/rapp_nao cd /home/nao/rapp_nao mkdir rapps && cd rapps touch simple_app.py chmod +x simple-_app.py “‘

The first step is to check if everything is in place. Write the following in the `simple_app.py` file:

“‘python #!/usr/bin/env python from rapp_robot_api import RappRobot rh = RappRobot() rh.audio.speak("Hello there!") “‘

If everything was set-up correctly the NAO robot should talk and say "Hello there!" to you. If not, one of the aforementioned instructions was not performed correctly (or if it they all were, please submit a bug to correct this tutorial!).

Now for the real application, you can use any of the documented API calls that exist here. Insert the following in the `simple_app.py` file:

“‘python #!/usr/bin/env python

Import the RAPP Robot API

from rapp_robot_api import RappRobot

Create an object in order to call the desired functions

rh = RappRobot()

Adjust the NAO master volume and ask for instructions. The valid commands are ’stand’ and ’sit’ and NAO waits for 5 seconds

rh.audio.setVolume(50) rh.audio.speak("Hello there! What do you want me to do? I can sit or get up.") res = rh.-audio.speechDetection([’sit’, ’get up’], 5) print res word = ” inner_word = ” if res[’error’] == None: word = res[’word’]

Check which command was dictated by the human

if word == ’sit’:

The motors must be enabled for NAO to move

rh.motion.enableMotors()

NAO sits with 75% of its maximum speed

rh.humanoid_motion.goToPosture(’Sit’, 0.75) elif word == ’get up’:

The motors must be enabled for NAO to move

rh.motion.enableMotors()

NAO stands with 75% of its maximum speed

rh.humanoid_motion.goToPosture('Stand', 0.75) else:

No command was dictated or the command was not understood

pass

Ask the human what movement to do: move the hands or the head?

rh.audio.speak("Do you want me to move my arms or my head?") res = rh.audio.speechDetection(['arms', 'head'], 5) print res if res['error'] == None: word = res['word']

rh.motion.enableMotors() if word == 'arms': rh.audio.speak("Do you want me to open the left or right hand?") res = rh.audio.speechDetection(['left', 'right'], 5) print res if res['error'] == None: inner_word = res['word'] if inner_word == 'left': rh.humanoid_motion.openHand('Left') elif inner_word == 'right': rh.humanoid_motion.openHand('Right') else: pass

rh.audio.speak("I will close my hands now") rh.humanoid_motion.closeHand('Right') rh.humanoid_motion.closeHand('Left') elif word == 'head': rh.audio.speak("Do you want me to turn my head left or right?") res = rh.audio.speechDetection(['left', 'right'], 5) print res if res['error'] == None: inner_word = res['word']

The head moves by 0.4 rads left or right with 50% of its maximum speed

if inner_word == 'left': rh.humanoid_motion.setJointAngles(['HeadYaw'], [0.4], 0.5) elif inner_word == 'right': rh.humanoid_motion.setJointAngles(['HeadYaw'], [-0.4], 0.5) else: pass

rh.audio.speak("I will look straight now") rh.humanoid_motion.setJointAngles(['HeadYaw'], [0], 0.5) else: pass

rh.audio.speak("And now I will sit down and sleep!") rh.humanoid_motion.goToPosture('Sit', 0.7) rh.motion.disableMotors() "'

As you may have noticed the API calls are robot-agnostic, meaning that the developer (you) can create applications without having to specify on which robot they will be executed. If a specific function is not implemented in a robot that will execute this application, the specific command will simply have no effect.

Finally, the last step is to execute the RApp:

"'bash python /home/nao/rapp_nao/rapps/simple_app.py "'

Chapter 8

!-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 Web services from your PC or or your robot following this tutorial.

Chapter 9

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

RAPP Web services

- General Information
- RAPP HOP Web services description

RAPP Nodes

- RAPP Application Authentication
- RAPP Audio Processing
- RAPP Cognitive Exercise
- RAPP Caffe Wrapper
- RAPP Email
- RAPP Face Detection
- RAPP Geolocator
- RAPP Hazard Detection
- RAPP Human Detection
- RAPP Knowrob wrapper
- RAPP MySQL wrapper
- RAPP News Explorer
- RAPP Path planner

- RAPP Platform (metapackage))
- RAPP Platform Launchers
- RAPP Platform ROS Communications
- RAPP QR Detection
- RAPP Speech Detection (Google API)
- RAPP Speech Detection (Sphinx4)
- RAPP Testing Tools
- RAPP Text-to-speech using Espeak & Mbrola
- RAPP Utilities
- RAPP Weather Reporter

RAPP Nodes under development

- RAPP Object Recognition
- RAPP Face Recognition

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

Full tutorials

RAPP Platform

- A full RAPP Platform service creation example: Simple addition (easy)
- A full RAPP Platform service creation example: Simple addition & RAPP Platform service invocation & testing (advanced)

Robotic applications

- Remote application for NAO in Python: Move by speech (easy))
- In-robot application for NAO in Python: Move by speech (easy))
- Remote application for NAO in Python: Detect and track QR tags (normal))
- Remote application for NAO in Python: Use ROS & TLD tracker to approach arbitrary objects (hard))
- In-robot application for NAO in Python: Create a cognitive game (hard))

Cloud agents

- TODO: [Create a simple custom Cloud-agent based service: Addition]()

Troubleshooting

- RAPP Improvement Center (RIC)
- RAPP Platform Web Server

Chapter 10

HOP-Services---General-Information

RAPP Improvement Center (RIC) nodes, can be utilized by robots via the RAPP Platform Web Services. The service layer has been developed using HOP. That consists of a web server implementation, an http/https server, and the web services developed in Hop.js framework. Web services run within server-side workers (web workers). A worker can include more than one web service. We consider server-side workers to be forked processes, thus allowing concurrent execution.

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.

Web service responses are asynchronous http responses. This way we allow robot platforms to request for cloud services while performing other tasks.

Robot platforms can access RAPP Platform services using HTTP POST requests, as most of the services require an arbitrary amount of data to be sent to the Cloud for processing, like image and audio data files. HOP Web Server has been configured to accept application/x-www-urlencoded and $*$multipart/form-data form submissions.

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

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-comm-layers.png]]

Currently, HOP-Server is configured to act as an HTTP/HTTPS Web Server (**Does not accept proxy connections**).

Deploy HOP Web Server

First, go the rapp_web_services directory:

“‘shell $ cd ${HOME}/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_web_services “‘

Install required dependencies:

“‘shell $ npm install “‘

For initiating HOP Web Server (and HOP Web Services), a grunt task exists:

“‘shell $ grunt init-hop “‘

HOP Web Server can also be deployed through the deploy_web_services.sh script

Note: Do not change HOP Server configuration parameters, unless you know what you are doing!!

Chapter 11

? 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 12

?
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. NOT YET SUPPORTED

Invoke this service from your favourite web browser:

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

Chapter 13

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

The setup scripts are located in the `rapp-platform-scripts` in the folder `/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
- installs Caffe
- installs the authentication system

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.
- `12_caffe_setup.sh`: Installs Caffe along with the bvlc reference models.
- `13_authentication_setup.sh`: Installs the authentication management system.

Chapter 14

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

This is quite easy! Just follow the instructions below which describe how to launch the deploy scripts. These scripts are located in the `rapp-platform-scripts` repository in the folder `/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_web_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_web_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 web_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 15

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

As previously stated here we provide and maintain API(s) for the following programming languages:

- Python
- JavaScript
- C++

For simplicity, we will describe the procedure of calling a HOP Web Service using the **python rapp-platform-api**.

We assume that you have previously read on How-to-write-the-API-for-a-HOP-Service and implemented the respective Cloud Message class.

Usage of the Python implementation of the rapp-platform-api is also described here

Chapter 16

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

RAPP Web Services are implemented on top of the `hop.js` framework.

> 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. 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.

A framework has been developed, build ontop of hop.js, for easily implementing Web Services for the RAPP Platform. Zero knowledge of hop.js is required. Additionally, Web Services are fully parametrized through single configuration files:

- `services.json`
- `workers.json`

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

RAPP Web Services framework

For easily implementing and launching Web Services for the RAPP Platform, the RAPP Web Services framework has been developed. It consists of a `WebService` implementation. build ontop of hop.js and an engine that allows to assign specific Web Services to Worker threads (Web Workers).

Web Service implementation

Web Services are implemented in a single function implementation:

“‘js function svcImpl(req, resp, ros){ // Web service implementation here. } “‘

This is the onrequest callback function to feed to the engine and will be called as soon as a request arrives.

> The **req** (request) and **res** (response) objects are passed so you can access the request properties and craft and return responses. Additionally a **ros** object is passed that allows connections to ROS thought the rosbridge-websocket transport layer.

The `req` object has the following properties:

- `header`: Request header.

“‘js console.log(req.header)

{ host: 'localhost:9001', content-length: 679, accept-encoding: 'gzip, deflate', accept: '*/*', user-agent: 'rapp-platform-api/python', accept-token: 'rapp_token', connection: 'keep-alive', content-type: 'multipart/form-data; boundary=595d1046de7f4c958ca662c37140215a' }

‘‘‘

- `socket`: Connection socket

‘‘‘js console.log(req.socket)

{ hostname: 'localhost', hostAddress: '127.0.0.1', localAddress: '127.0.0.1', port: 43661 }

‘‘‘

- `body`: Request body

‘‘‘js console.log(req.body)

{ fast: true }

‘‘‘

- `files`: In case of uploading files, this field contains the paths to the uploaded files. Access the files by name using dot notation. For example a service receives a single-file in fieldname `single_file` and an array of files in fieldname `file_array`:

‘‘‘js console.log(req.files)

{ single_file: ["PATH"], file_array: ["PATH_FILE_1", "PATH_FILE_2"] }

‘‘‘

Note: Note that even if it is a **single-file**, the `single_file` property of `req.files` (`req.files.single_file`) is an Array.

- `username`: This is the **username** of the client that requested access to the RAPP Platform resources (Services). It is automatically applied to the `req` object, after appliance of the `RAPP Authentication` on request arrival. Note that before execution of the onrequest callback function, we apply authentication to the request. If the authentication is not successful, an **HTTP 401 Unauthorized** error is returned to the client.

‘‘‘js console.log(req.username)

"RAPP_USER"

‘‘‘

The `resp` object has the following properties (methods):

- sendJson(obj): Send an application/json response

‘‘‘js function svcImpl(req, resp, ros){ ...

var response = {error: ''}; resp.sendJson(response); } ‘‘‘

- `sendServerError()`: Respond with **HTTP 500 Internal Server Error**

“‘js function svcImpl(req, resp, ros){ ...

resp.sendServerError(); } “‘

- `sendUnauthorized()`: Respond with **HTTP 401 Unauthorized Client Error**

“‘js function svcImpl(req, resp, ros){ ...

resp.sendUnauthorized(); } “‘

Web Service configuration and registration.

Web Services are fully parametrized through the `services.json` file. This file includes Web Services to be launched (along with the Web Service parameters), that was previously declared in the `workers.json` file.

Below is the face_detection entry:

“‘js "face_detection": { "launch": true, "anonymous": false, "name": "face_detection", "url_name": "face_detection", "namespace": "", "ros_connection": true, "timeout": 45000 } “‘

Web Service configuration parameters:

- `launch` (Boolean): If true this Web Service will be launched.
- `anonymous` (Boolean): If true, this service will be anonymous, which means that it will be assigned a random url path.
- `name` (String): The service name.
- `urlname` (String): The service urlname. Service name can be different from the urlname.
- `namespace` (String): Namespace for the urlname to append as a prefix to the service url name. For example, a service with **urlname="faca_detection"** and **namespace="computervision"** will be translated to ∗∗/computervision/face_detection∗∗
- `timeout` (String): Request timeout value.
- `ros_connection` (Boolean): If true, a ros object that allowes for calls to the ROS Services will be passed to the onrequest callback function.

Run a Web Service within an existing Web Worker

Web services run within server-side workers (Web Workers). A worker can include more than one web service. We consider server-side workers to be forked processes, thus allowing concurrent execution.

To run a Web Service within a Web Worker, just specify the service name in the **services** field of the worker in the `worker.json` file.

For example, the **weather_report** worker holds the **weather_report_current** and **weather_report_forecast** Web Services:

“‘js "weather_report": { "launch": true, "path": "workers/weather_report.js", "services": ["weather_report_forecast", "weather_report_current"] } “‘

Web Worker configuration parameters:

- `launch` (Boolean): Weather to launch the Web Worker or not.
- `path` (String): Path to the Web Worker source file. Relative to the **rapp_web_services** directory
- `services`: (Array): Services to launch under the Web Worker thread.

Where to store Web Service implementation source file(s) and how to launch it.

Source files are stored under the `services` directory, of the `rapp_web_services` package.

Web Services are automatically loaded from single .js files, as node.js modules. Make sure you export the Web Service implementation function:

“‘js function svcImpl(req, resp, ros){ // Web service implementation here. }

...

module.exports = svcImpl;

“‘

Complete Web Service Implementation Example - Face Detection

The following example illustrates the implementation of a WebService that connects to the Face-Detection RAPP--Platform backend Service

ROS Service Message: “‘bash

Contains info about time and reference

Header header

The image’s filename to perform face detection

string imageFilepath

Flag to define if a fast detection if desired

bool fast

Container for detected face positions

geometry_msgs/PointStamped[] faces_up_left geometry_msgs/PointStamped[] faces_down_right string error “‘

and the Face-Detection ROS Service url path is: `/rapp/rapp_face_detection/detect_faces`

Web Service Request:

- `file`: Image file.
- `fast` (Bool): If true, detection will take less time but it will be less accurate.

Web Service Response:

- `faces` (Array): Detected faces.
- `error` (String): Error message.

First, create the face_detection `svc.js` file:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_web_services/services $ mkdir face_-detection && cd face_detection $ touch svc.js “‘

Open the svc.js file with your favorite editor:

“‘bash $ vim svc.js “‘

Implement the structure of the `client-response` and `ros-msg` objects:

“‘js var clientRes = function(faces, error) { return { faces: [], error: ” } }

var rosReqMsg = function(imageFilepath, fast) { return { imageFilepath: ”, fast: false } } “‘

Assign ROS Service url path to a global variable:

“‘js var rosSrvUrlPath = "/rapp/rapp_face_detection/detect_faces"; “‘

Next, implement the service onrequest callback function:

“‘js function svcImpl(req, resp, ros) { // If no image file received, return to client with an error if(! req.files.file){ // Create a client response object response = new client_res(); response.error = "No image file received"; // Send response (application/json) resp.sendJson(response); return; }

// Create a ROS Service Request Message and fill values from client request var rosMsg = new rosReqMsg(); rosMsg.imageFilename = req.files.file[0]; rosMsg.fast = req.body.fast;

/***

- ROS-Service response callback. ∗/ function callback(data){ // Delete image file from the Platform cache directory. fs.exists(_filepath, function(exists){ if(exists){ fs.unlink(_filepath) } }) // Parse rosbridge message and craft client response var response = parseRosbridgeMsg(data); resp.sendJson(response); }

/***

- ROS-Service onerror callback. ∗/ function onerror(e){ // Delete image file from the Platform cache directory. fs.exists(_filepath, function(exists){ if(exists){ fs.unlink(_filepath) } }) // Respond a "Server Error". HTTP Error 501 - Internal Server Error resp.sendServerError(); }

/***

- Call ROS-Service. ∗/ ros.callService(rosSrvUrlPath, rosMsg, {success: callback, fail: onerror}); }

function parseRosbridgeMsg(rosbridge_msg) { var faces_up_left = rosbridge_msg.faces_up_left; var faces_down-_right = rosbridge_msg.faces_down_right; var error = rosbridge_msg.error; var numFaces = faces_up_left.length;

// Create a new response object var response = new client_res();

if(error){ // If ROS Service responded with an error response.error = error; return response; }

for (var ii = 0; ii < numFaces; ii++) { var face = { up_left_point: {x: 0, y:0}, down_right_point: {x: 0, y: 0} };

face.up_left_point.x = faces_up_left[ii].point.x; face.up_left_point.y = faces_up_left[ii].point.y; face.down_right_-point.x = faces_down_right[ii].point.x; face.down_right_point.y = faces_down_right[ii].point.y; response.faces.push(face); }

return response; } “‘

Export the service onrequest callback function (`svcImpl`):

“‘js module.exports = svcImpl “‘

Next you will need to create the Web Worker to launch the Web Service:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_web_services/workers $ touch face_-detection.js “‘

Open the **face_detection.js** file with your favorite editor:

“‘bash $ vim face_detection.js “‘

Import the workerUtils module:

“‘js var path = require(’path’);

var ENV = require(path.join(__dirname, '../..', 'env.js'));

// Include it even if not used!!! Sets properties to the thread's global scope. var workerUtils = require(path.join(ENV.PATHS.INCLUDE_DIR, 'common', 'worker_utils.js')); "'

Next, you will have to set the worker name and call to launch all services registred to this Web Worker:

"'js // Set worker thread name under the global scope. (WORKER.name) workerUtils.setWorkerName('face_detection');

// Launch all services assigned to this worker thread. // Search in workers.json config file for assigned web services. workerUtils.launchSvcAll(); "'

We need to tell the *run-engine* to launch the, newly implemented, Web Worker.

The `workers.json` file containes Web Workers entries. It is located under:

"'bash ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_web_services/config/services "'

Append the following entry in the workers.json file:

"'js "face_detection": { "launch": true, "path": "workers/face_detection.js", "services": ["face_detection"] } "'

Finally append the following *web-service* entry in the services.json file (under the same directory):

"'js "face_detection": { "launch": true, "anonymous": false, "name": "face_detection", "url_name": "face_detection", "namespace": "", "authentication": true, "ros_connection": true, "timeout": 45000 } "'

Now the RAPP Platform is ready to receive requests for the newly created face_detection service.

"'bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_web_services $ pm2 start server.yaml "'

You will notice the following output from the logs:

"'bash info: [Service Handler] Registered worker service {http://rapp-platform-local:9001/hop/face_detection} under worker thread {face_detection} info: [Service Handler] { worker: 'face_detection', path: '/hop/face_detection', url: 'http://rapp-platform-local:9001/hop/face_detection', frame: [Function] } "'

Further study

You can check on already implemented Web Services here.

The RAPP WebService code API is documented [here]().

Documentation of the RAPP Web Services package can be found here.

Chapter 17

? 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 18

? 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_test.py` will be used as a reference.

Each integration test must have the following characteristics:

- Each test class is written as a seperate python source file (.py).
- Each test inherits from `unittest.TestCase` class.
- The `Python RAPP Platform API` is used to call RAPP Platform Services.

If you are not familiar with the Python unit testing framework (unittest), start reading from `there` as the rapp_-testing_tools is build ontop of it. Also, basic knowledge of using the `python-rapp-platform-api` is required.

So lets create a test class for testing the integration behaviour of the Face-Detection RAPP Platform service/functionality.

Step 1: Create the source file for the test class

Copy the `template_test.py` file and name it `face_detection_tests.py` under the `default_tests` directory

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_testing_tools/scripts/default_tests $ cp template_test.py face_detection_tests.py “‘

Rename the test class to `FaceDetectionTests`:

“‘python class FaceDetectionTests(unittest.TestCase):

```
def setUp(self):
    self.ch = RappPlatformAPI()

    rospack = rospkg.RosPack()
    self.pkgDir = rospack.get_path('rapp_testing_tools')

def test_templateTest(self):
    self.assertEqual(1, 1)
```

“‘

Note that the `setUp()` method instantiates a RappPlatformAPI() object for calling RAPP Platform Services.

Step 2: Implement a test case

As the `FaceDetectionTests` class inherits from `unittest.TestCase`, individual tests are defined with methods whose names start with the letters `test`

We will implement a test of performing face-detection on a single image file (Lenna.png). We assume that the image file was previously stored under the `test_data` directory. So lets create a member method named `test_lenna` This method will be responsible for loading the image file, call the RAPP Platform Service, through the API, and evaluate the results using `unittest` assertions

“‘python class FaceDetectionTests(unittest.TestCase):

```
def setUp(self):
    self.ch = RappPlatformAPI()

    rospack = rospkg.RosPack()
    self.pkgDir = rospack.get_path('rapp_testing_tools')

def test_lenna(self):
    self.assertEqual(1, 1)
    response = self.ch.faceDetection(imagepath)

    valid_faces = [{
        'up_left_point': {'y': 201.0, 'x': 213.0},
        'down_right_point': {'y': 378.0, 'x': 390.0}
    }]

    self.assertEqual(response['error'], u'')
    self.assertEqual(response['faces'], valid_faces)
```

“‘

The first assertion, ‘self.assertEqual(response['error'], u”)‘ evaluates that no error was reported from the RAPP Platform The second assertion evaluates the response from the FaceDetector.

The complete face_detection_tests.py source file is:

“‘python from os import path import timeit import unittest import rospkg

path = os.path.dirname(os.path.realpath(**file**))

from RappCloud import RappPlatformAPI

class FaceDetectionTests(unittest.TestCase):

```
def setUp(self):
    self.ch = RappPlatformAPI()

    rospack = rospkg.RosPack()
    self.pkgDir = rospack.get_path('rapp_testing_tools')

def test_lenna(self):
    self.assertEqual(1, 1)
    response = self.ch.faceDetection(imagepath)

    valid_faces = [{
        'up_left_point': {'y': 201.0, 'x': 213.0},
        'down_right_point': {'y': 378.0, 'x': 390.0}
    }]

    self.assertEqual(response['error'], u'')
    self.assertEqual(response['faces'], valid_faces)
```

if **name** == "__main__": unittest.main() “‘

Step 3: Executing the Face Detection test case

Head to the `scripts` directory and execute the `rapp_run_test.py` script, giving as input argument the `face_detection_tests.py` file to execute:

“‘bash $ cd ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_testing_tools/scripts $ ./rapp_run_-test.py -i default_tests/face_detection_tests.py “‘

On successful test execution, the output should be:

“‘bash

RAPP Platfrom Tests

- Parameters: – Number of Executions for each given test: [1] – Serial execution
- Tests to Execute: 1] face_detection_tests x1

Running face_detection_tests... Ran 1 tests in 0.383s Success “‘

If an error occures on the RAPP Platform, the returned error message will be reported to the console output. For example, if the RAPP Platform Web Server was not previously launched properly, we should get a "Connection" error on the assertion of the `error` property of the response:

“‘bash

RAPP Platfrom Tests

- Parameters: – Number of Executions for each given test: [1] – Serial execution
- Tests to Execute: 1] face_detection_tests x1

Running face_detection_tests... Ran 1 test in 0.078s Failed

FAIL: test_lenna (main.FaceDetectionTests)

Traceback (most recent call last): File "/home/rappuser/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_testing_tools/scripts/default_tests/face_detection_tests.py", line 61, in test_lenna self.assert-Equal(response['error'], u”) AssertionError: 'Connection Error' != u”

Ran 1 test in 0.078s

FAILED (failures=1)

“‘

Chapter 19

? 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 API 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-Platform-API Overview - Usage

API users are able to select from 2 API implementations:

- **High level API**
- **Advanced API**

The first one allow API users to easily call RAPP PLatform Services through simple function calls. The second one is for advanced usage, delivered for expert developers. This is an object-oriented implementation. As we will later describe, the advanced API usage allow creation of Cloud Messages. Both Platform requests and responses are described by static objects.

Note: The High Level API actually wraps the Advanced API.

Advanced API usage

`RappPlatformService` is the RAPP term for an established connection to the RAPP-Platform Services, over the www (World-Wide-Web). Each Platform Service has it's own unique Response and Request message.

The `RappPlatformService` class is used to establish connections to the RAPP-Platform Web-Services, while `CloudMsg` objects include:

- `Request` object. RAPP-Platform Service specific Request message
- `Response` object. RAPP-Platform Service specific Response message

“‘python from RappCloud import RappPlatformService

svcClient = RappPlatformService(persistent=True, timeout=30000) “‘

By Default it connects to the localhost, assuming that the RAPP Platform has been setup on the local machine. The constructor of the `RappPlatformService` class allow to specify the RAPP Platform parameters to connect to.

“‘python from RappCloud import RappPlatformService

svcClient = RappPlatformService(address=’RAPP_PLATFORM_IPv4_ADDRESS’, port=’RAPP_PLATFORM_PORT_NUMBER’, protocol=’http’) “‘

`RappPlatformService` object constructor allow to set:

- persistent (Boolean): Force peristent connections. **Defaults to True**
- timeout (Integer): Client timeout value. This is the timeout value waiting for a response from the RAPP Platform. **Defaults to infinity**
- address (String): The RAPP Platform IPv4 address to connect to. **Defaults to ’localhost’**
- port (String): The RAPP Platform listening port. **Defaults to "9001"**
- protocol (String): The configured application protocol for the RAPP Platform. Valid values are "∗∗https∗∗" and "∗∗http∗∗". **Defaults to "http"**

The `persistent` and `timeout` properties of a `RappPlatformService` object are public members and can be set using the **dot** (.) notation:

“‘py svcClient = RappPlatformService() svcClient.persistent = True svcClient.timeout = 30000 “‘

`CloudMsg` objects are feed to the `RappPlatformService` object to specific RAPP-Platform Services. `CloudMsg` classes can be imported from the CloudMsgs submodule of the RappCloud module:

“‘py from RappCloud.CloudMsgs import FaceDetection “‘

The above line of code is used as an example of importing the `FaceDetection` CloudMsg class.

A complete description on available CloudMsg classes as long as their Request and Response message classes is available here

CloudMsg objects hold a Request and a Response object:

“‘py from RappCloud.CloudMsgs import FaceDetection faceDetectMsg = FaceDetection()

reqObj = faceDetectMsg.req respObj = faceDetectMsg.resp “‘

Request and Response objects of a CloudMsg can be serialized to a dictionary:

“‘py reqDict = faceDetectMsg.req.serialize() print reqDict

{fast: False, imageFilePath: ”}

respDict = faceDetectMsg.resp.serialize() print respDict

{faces: [], error: ”}

“‘

CloudMsg Request property values can be set through the `req` property of the CloudMsg object. or as keyword arguments to the constructor of a CloudMsg:

“‘py from RappCloud.CloudMsgs import FaceDetection

msg = FaceDetection(imageFilepath=’/tmp/face-sample.png’) print msg.req.serialize()

{fast: False, imageFilepath: ’/tmp/face-sample.png’}

msg.req.fast = True print msg.req.serialize()

{fast: True, imageFilepath: ’/tmp/face-sample.png’}

“‘

`RappPlatfomrService` objects have a `.call()` method for calling RAPP-Platform Services:

“‘py class RappPlatformService: ...

def call(self, msg=None): ... return self.resp

... “‘

The `.call()` method returns the Response object.

“‘py svcClient= RappPlatformService() msg = FaceDetection() msg.req.fast = True msg.req.imageFilepath = ’/tmp/face-sample.png’

response = svcClient.call(msg) print response.faces print response.error

“‘

CloudMsg objects are passed as argument to the `.call()` method of the `RappPlatformService` object:

“‘py svcClient= RappPlatformService() msg = FaceDetection(imageFilepath=’/tmp/face-sample.png’) response = svcClient.call(msg) “‘

`CloudMsg` objects can also be passed to the constructor of the `RappPlatformService` class:

“‘py faceMsg = FaceDetection(imageFilepath=’/tmp/face-sample.png’) svcClient= RappPlatformService(msg=faceMsg, timeout=15000) response = svcClient.call() “‘

Note: Calling several different RAPP-Platform Services is done by passing the service specific Cloud Message objects to the `.call()` of the `RappPlatformService` object.

The following example creates a `FaceDetection` and a `QrDetection` CloudMsg to call both the Face--Detection and Qr-Detection RAPP-Platform Services.

“‘py from RappCloud import RappPlatformService from RappCloud.CloudMsgs import (FaceDetection, Qr-Detection)

svcClient = RappPlatformService(timeout=1000) faceMsg = FaceDetection(fast=True, imageFilepath=’/tmp/face-sample.png’) qrMsg = QrDetection() qrMsg.req.imageFilepath = ’/tmp/qr-sample.png’

fdResp = svcClient.call(faceMsg) print "Found %s Faces" len(fdResp.faces)

qrResp = svcClient.call(qrMsg) print "Found %s QRs: %s" %(len(qrResp.qr_centers), qrResp.qr_messages)

“‘

High Level API usage

Like previously mentioned, API users can also use the High Level implementation of the RAPP Platform API. Benefits from using this implementation is lack of knowledge of how Cloud Messages and RappPlatformService are used. Calls to the RAPP Platform are done through simple function calls, under the RappPlatformAPI module.

Below is an example of performing a query to the ontologyi, hosted on the RAPP Platform, using the High Level API implementation:

“‘python from RappCloud import RappPlatformAPI ch = RappPlatformAPI()

response = ch.ontologySubclasses("Oven")

print response

{’results’: [u’http://knowrob.org/kb/knowrob.owl#Microwave-Oven’, u’http://knowrob.org/kb/knowrob.owl#RegularOven’, u’http-://knowrob.org/kb/knowrob.owl#ToasterOven’], ’error’: u”}

“‘

The RappPlatformAPI usage and calls are fully documented here, also with examples of usage.

Example - Implementing the FaceDetection API call.

Lets say we want to implement the `FaceDetection` Cloud Message.

The face detection RAPP Platform Web Service has a Request and Response object

Web-Service Request

- `fast` (Boolean): If true, detection will take less time but it will be less accurate
- `file` (File): Image file.

Web-Service Response

- `faces` (Array): An array of the detected faces coordinates (point2D), on the image frame.
- `error` (String): Error message.

Start by creating the python source file for the FaceDeteciton Cloud Message implementation. Head to the `Rapp-Cloud/CloudMsgs` directory of the RappCloud module and create a file named **FaceDetection.py**

Cloud Messages classes inherit from the `CloudMsg` class and `Request` and `Response` classes inherit from `CloudRequest` and `CloudResponse` classes respectively. So first import those classes and write the structure of the `FaceDetection` Cloud Message:

“‘python from Cloud import (CloudMsg, CloudRequest, CloudResponse)

class FaceDetection(CloudMsg):

```
class Request(CloudRequest):
    def __init__(self, **kwargs):
        pass


class Response(CloudResponse):
    def __init__(self, **kwargs):
        pass


def __init__(self, **kwargs):
    pass
```

“‘

Add the appropriate properties to the Request and Response classes. Property names can differ from the Web--Service request and response property names. Mapping from implemented property names to actual request payload will be studied later on:

“‘python class Request(CloudRequest):

```
def __init__(self, **kwargs):
    ## File path to the image file
    self.imageFilepath = ''
    ## If true, detection will take less time but it will be less accurate
    self.fast = False
    ## Apply keyword arguments to the Request object.
    super(FaceDetection.Request, self).__init__(**kwargs)
```

class Response(CloudResponse):

```
def __init__(self, **kwargs):
    ## Error message
    self.error = ''
    ## Detected faces. Array of face objects.
    self.faces = []
    ## Apply keyword arguments to the Request object.
    super(FaceDetection.Response, self).__init__(**kwargs)
```

‘‘‘

Notice calling `CloudResponse` and `CloudRequest` construcors.

Remember that a `Request` class must implement two member methods, `make_payload()` and `make_-files`. For the payload it’s the `fast` property and imageFilepath is a file.

‘‘‘python from RappCloud.Objects import (Payload, File)

class Request(CloudRequest):

```
def __init__(self, **kwargs):
    ## File path to the image file
    self.imageFilepath = ''
    ## If true, detection will take less time but it will be less accurate
    self.fast = False
    ## Apply keyword arguments to the Request object.
    super(FaceDetection.Request, self).__init__(**kwargs)

def make_payload(self):
    return Payload(fast=self.fast)

def make_files(self):
    return [File(filepath=self.path, postfield="file")]
```

‘‘‘

Next, you have to instantiate a `Request` and `Response` objects for the `FaceDetection` class to hold:

‘‘‘python class FaceDetection(CloudMsg): ...

def **init**(self, ∗∗kwargs):

Create and hold the Request object for this CloudMsg

self.req = FaceDetection.Request()

Create and hold the Response object for this CloudMsg

self.resp = FaceDetection.Response() ‘‘‘

Each RAPP Platform Web Service has a unique service name resolving to a url name/path. The service name for the Face Detection RAPP Platform Web Service is:

‘‘‘shell face_detection ‘‘‘

`CloudMsg` constructor takes as input the service name:

‘‘‘python class FaceDetection(CloudMsg): ...

def **init**(self, ∗∗kwargs):

Create and hold the Request object for this CloudMsg

self.req = FaceDetection.Request()

Create and hold the Response object for this CloudMsg

self.resp = FaceDetection.Response() super(FaceDetection, self).__init__(svcname=’face_detection’, ∗∗kwargs) ‘‘‘

Note: Dont forget to document the code using **doxygen**

Below is the complete FaceDetection.py file

“‘python from RappCloud.Objects import (File, Payload)

from Cloud import (CloudMsg, CloudRequest, CloudResponse)

class FaceDetection(CloudMsg): """ Face Detection CloudMsg object"""

class Request(CloudRequest): """ Face Detection Cloud Request object. FaceDetection.Request """ def **init**(self, ∗∗kwargs): """! Constructor

Parameters

∗∗*kwargs*	- Keyword arguments. Apply values to the request attributes. • imageFilepath • fast"

File path to the image to load. This is the image to perform

face-detection on.

self.imageFilepath = ”

If true, detection will take less time but it will be less

accurate

self.fast = False

Apply passed keyword arguments to the Request object.

super(FaceDetection.Request, self).__init__(∗∗kwargs)

```
    def make_payload(self):
        """ Create and return the Payload of the Request. """
        return Payload(fast=self.fast)

    def make_files(self):
        """ Create and return Array of File objects of the Request. """
        return [File(self.imageFilepath, postfield='file')]

    class Response(CloudResponse):
        """ Face Detection Cloud Response object. FaceDetection.Response """
        def __init__(self, **kwargs):
            """!
            Constructor

            @param **kwargs - Keyword arguments. Apply values to the request attributes.
            - @ref error
            - @ref faces
            """
            ## Error message
            self.error = ''
            ## Detected faces. Array of face objects. TODO create face object.
            self.faces = []
            ## Apply passed keyword arguments to the Request object.
            super(FaceDetection.Response, self).__init__(**kwargs)
```

```
    def __init__(self, **kwargs):
        """!
        Constructor

        @param **kwargs - Keyword arguments. Apply values to the request attributes.
            - @ref Request.fast
            - @ref Request.imageFilepath
        """

        # Create and hold the Request object for this CloudMsg
        self.req = FaceDetection.Request()
        # Create and hold the Response object for this CloudMsg
        self.resp = FaceDetection.Response()
        super(FaceDetection, self).__init__(svcname='face_detection', **kwargs)
```

“‘

Finally append the following line of code in the `RappCloud/CloudMsgs/__init__.py` file:

“‘python from FaceDetection import FaceDetection “‘

Now everything is in place to call the newly created Face detection RAPP Platform Service, using the python implementation of the rapp-platform-api. An example is presented below:

“‘python from RappCloud.CloudMsgs import FaceDetection from RappCloud import RappPlatformService

svcClient = RappPlatformService(persistent=True, timeout=30000) msg = FaceDetection(imageFilepath="PATH", fast=True)

response svcClient.call(msg)

if response.error: print "An error has occured: %s" response.error else: print response.faces “‘

If you want to also include it in the High Level API implementation, you will have to modify the `RappPlatform-Api.py` file.

First import the `FaceDetection` Cloud Message, that was previously implemented:

“‘py ...

from CloudMsgs import FaceDetection “‘

Next, we must implemend the `faceDetection` method for the RappPlatformAPI class. Input arguments to the method are:

- imageFilepath: Path to the image file to perform face detection on.
- fast: Force fast detection. If true, detection takes less time but it will be less accurate

The output is a python `dict` of the response fields:

- faces: Detected faces
- error: Error message, if one occures.

The `faceDetection` method implementation must be as presented below

“‘py ...

class RappPlatformAPI(): """ RAPP Platform simple API implementation """ def **init**(self): self.svc_caller = Rapp-PlatformService()

...

def faceDetection(self, imageFilepath, fast = False): """! Face detection API service call. imageFilepath: string

Parameters

imageFilepath	Path to the image file. fast: bool
fast	Perform fast detection. If true, detection will take less time but it will be less accurate. : dict

Returns

: Returns a dictionary of the service call response. {'faces': [], 'error': ''} """ msg = FaceDetection() try: msg.-req.imageFilepath = imageFilepath response = self.svc_caller.call(msg) except Exception as e: response = FaceDetection.Response(error=str(e)) return { 'faces': response.faces, 'error': response.error } "'

Note: Make sure to launch both the back-end and the, listening for requests, HOP Web Server, of the RAPP Platform, before executing the above example. Here you can find instructions on how to launch the RAPP Platform.

Chapter 20

? 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 `155.207.19.229` and you can invoke any HOP service using the following url:

155.207.19.229:9001/hop/HOP_SRV_NAME

You can find the HOP service names here.

Chapter 21

? 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 22

(hard) In-robot-application-for-NAO-in-Python:--Create-a-cognitive-game-(hard)

This is an example application for Cognitive Exercises.

In this tutorial we will focus on a step-by-step guidance through implementing the Cogntive Exercises RApp (Robotic Application).

The complete implemnention of the CognitiveExercises RApp can be found here

Preparation steps

For this tutorial we will use the following tools:

- A real NAO robot
- The rapp-robots-api for commanding NAO Robot.
- The Python rapp-platform-api as we will need to call several RAPP Platform Services.

Of course the standard prerequisites are an editor and a terminal.

RAPP Robots API libraries setup

The first step is to clone the appropriate GitHub repository in your PC:

“‘bash $ mkdir ∼/rapp_nao $ cd ∼/rapp_nao $ git clone https://github.com/rapp-project/rapp-robyyots-api.-git “‘

The next step is to transfer the RAPP Python libraries to the NAO robot. This will be done via scp, assuming that the NAO robot’s IP is 192.168.0.101 and username and password are nao:

“‘bash $ cd ∼/rapp_nao/ $ tar -zcvf rapp_robots_api.tar.gz rapp-robots-api/ $ scp rapp_robots_api.tar.gz nao@192.168.0.101:/home/nao “‘

Now connect in NAO via ssh by ssh nao@192.168.0.101 giving nao as password. Then untar the API:

“‘bash $ tar -xvf rapp_robots_api.tar.gz $ rm rapp_robots_api.tar.gz “‘

The next step is to update the PYTHONPATH variable. Since NAO has Gentoo as OS, we will modify the bash_-profile file:

“‘bash $ echo ’export PYTHONPATH=$PYTHONPATH:/home/nao/rapp-robots-api/python/abstract_classes’ >> /home/nao/.bash_profile $ echo ’export PYTHONPATH=$PYTHONPATH:/home/nao/rapp-robots-api/python/implementations/nao-_v4_naoqi2.1.4’ >> /home/nao/.bash_profile $ source ∼/.bash_profile “‘

The last step to configure the rapp-robots-api is to declare the NAO IP. Since the robot API is in-robot, the IP must be localhost: 127.0.0.1.

The IP must be declared in the first line of `this` file, thus the `nao_connectivity` file located under `/home/nao/rapp-robots-api/python/implementations/nao_v4_naoqi2.1.4/nao_connectivity` should contain:

“‘ 127.0.0.1 9559 “‘

RAPP Platform API (Python) libraries setup

The first step is to clone the `rapp-api` GitHub repository in your PC:

“‘bash $ cd ∼/rapp_nao $ git clone `https://github.com/rapp-project/rapp-api.git` “‘

The next step is to transfer the Python libraries to the NAO robot. This will be done via `scp`, assuming that the NAO robot's IP is `192.168.0.101` and username and password are `nao`:

“‘bash $ cd ∼/rapp_nao/ $ tar -zcvf rapp_api.tar.gz rapp-api/ $ scp rapp_api.tar.gz `nao@192.168.0.101`:/home/nao “‘

Now connect in NAO via ssh by `ssh nao@192.168.0.101` giving `nao` as password. Then untar the API:

“‘bash $ tar -xvf rapp_api.tar.gz $ rm rapp_api.tar.gz “‘

The next step is to update the `PYTHONPATH` variable. Since NAO has Gentoo as OS, we will modify the `bash_profile` file:

“‘bash $ echo ’export PYTHONPATH=$PYTHONPATH:/home/nao/rapp-api/python’ >> /home/nao/.bash_profile $ source ∼/.bash_profile “‘

Writing the Cognitive Exercises RApp

We will work on the host machine while implementing the RApp and then we are going to transfer the application source files to the NAO Robot and execute.

Let's create a package for the application:

“‘bash $ mkdir -p ∼/rapp_nao/rapps/cognitive_exercises $ cd ∼/rapp_nao/rapps/cognitive_exercises $ mkdir rapps && cd rapps $ touch cognitive_exercises.py $ chmod +x cognitive_exercises.py “‘

Import the required components into the code:

“‘python #!/usr/bin/env python

Import the RAPP Robot API

from rapp_robot_api import RappRobot

Create an object in order to call the desired functions

rh = RappRobot()

from RappCloud import RappPlatformService from RappCloud.CloudMsgs import (CognitiveExerciseSelect, CognitiveRecordPerformance, SpeechRecognitionSphinx, SetNoiseProfile) “‘

Chapter 23

RAPP-Application-Authentication

RAPP application authentication allows the various RAPP Platform users to authenticate in order to gain access to the RAPP Platform services.

ROS Services

Add new user from Platform

Add new user using Platform credentials.

Service URL: `/rapp/rapp_application_authentication/add_new_user_from_platfrom`

Service type:

AddNewUserFromPlatformSrv.srv “‘ string creator_username string creator_password string new_user_username string new_user_password

New user’s language

string language

string error

Suggested username if provided already exists

string suggested_username “‘

Add new user from Store

Add new user using Store credentials.

Service URL: `/rapp/rapp_application_authentication/add_new_user_from_store`

Service type:

AddNewUserFromStoreSrv.srv “‘

New user's username

string username

New user's password

string password

Creator device token from RAPP Store

string device_token

New user's language

string language

string error

Suggested username if provided already exists

string suggested_username "'

Authenticate Token

Authenticates an active application token.

Service URL: `/rapp/rapp_application_authentication/authenticate_token`

Service type:

UserTokenAuthenticationSrv.srv "'

string token

string error

The token corresponding username

string username "'

Login

Allows a user to login useing the RAPP Platform credentials.

Service URL: `/rapp/rapp_application_authentication/login`

Service type:

UserLoginSrv.srv "' string username string password

The device from which a user tries to login

string device_token

string error string token “‘

Login from Store

Allows a user to login useing the RAPP Store token.

Service URL: `/rapp/rapp_application_authentication/login_from_store`

Service type:

UserLoginSrv.srv “‘ string username string password

The RAPP Store token

string device_token

string error string token “‘

Launchers

Standard launcher

Launches the **rapp application authentication** node and can be invoked using: “‘bash roslaunch rapp_application-_authentication_manager application_authentication.launch “‘

Web Services

Login User

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

Input/Output

“‘ Input = { username: ”, password: ”, device_token: ” } Output = { token: ”, error: ” } “‘

Register user from Platform

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

Input/Output

“‘ Input = { creator_username: ”, creator_password: ”, new_user_username: ”, new_user_password: ”, language: ” } Output = { suggested_username: ”, error: ” } “‘

Register user from Store

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

Input/Output

“‘ Input = { username: ”, password: ”, device_token: ”, language: ” } Output = { suggested_username: ”, error: ” } “‘

Chapter 24

RAPP-Architecture

The overall design of the RAPP Architecture is depicted in the following figure (click for larger resolution).

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 RAPP 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 they are 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 Web services exposed by RIC. The Web services can be invoked by the RAPP Platform 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 includes 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 ecosystem. It uptakes the tasks of downloading / updating the RApps and providing robot-aware 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 ROS. 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 utilizes 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 upon its submission.

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 behaviors 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 presented in the image below and all functionalities are analysed in their respective wiki pages.

[[images/rapp_platform_class_diagram.png]]

Chapter 25

RAPP-Audio-Processing

#Methodology

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 Google and Sphinx4 speech recognition modules are functional when the input is captured from a headset, the same does not apply with 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.

Once the service is invoked, 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 hard gating in the time domain, of the signal based on the RMS metric. 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 "'

#Web 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 = { "file": "THE_AUDIO_FILE" "audio_source": "nao_ogg, nao_wav_1_ch, nao_wav_4_ch" } Output = { "error": "Possible error" } "'

The full documentation exists here

Chapter 26

RAPP-Caffe-Wrapper

#Methodology The RAPP Caffe Wrapper node contains the services responsible for object identification via the Caffe deep learning framework and the assistive services for translating caffe classes to ontolgoy classes and registering the object images to the ontology.

ROS Services

/rapp/rapp_caffe_wrapper/register_image_to_ontology

Image classification

This service will classify an image using the Caffe deep learning framework and if requested it will translate its caffe class to ontology class and register it to the ontology.

Service URL: `/rapp/rapp_caffe_wrapper/get_ontology_class_equivalent`

Service type: “‘bash #the url of the image string objectFileUrl #the username of the user calling the service string username #true if the image will be registered to the ontology

bool registerToOntology

#the value of the ontology entry if the image was registered string ontologyNameOfImage #the caffe object class of the image string objectClass bool success

possible error

string error #tracen information string[] trace

“‘

Get ontology class equivalent

This service will return the ontology class equivalent of a caffe object class.

Service URL: `/rapp/rapp_caffe_wrapper/get_ontology_class_equivalent`

Service type: “‘bash #the caffe object class of the iamge

string caffeClass

#true if a corresponding object exists in the ontology bool existsInOntology #the ontology equivalent class of the caffe class string ontologyClass bool success

possible error

string error #tracen information string[] trace

‘‘‘

Register image to ontology

This service will register an image to the ontology along with the caffe and ontology class of the depicted object.

Service URL: `/rapp/rapp_caffe_wrapper/get_ontology_class_equivalent`

Service type: “‘bash #the caffe object class of the iamge

string caffeClass

#true if a corresponding object exists in the ontology bool existsInOntology #the ontology equivalent class of the caffe class string ontologyClass bool success

possible error

string error #tracen information string[] trace

‘‘‘

Launchers

Standard launcher

Launches the **rapp_caffe_wrapper** node and can be launched using “‘ roslaunch rapp_caffe_wrapper rapp_caffe_wrapper_functional_tests.launch “‘

HOP services

Object-Recognition-Caffe

Service-Url

“‘ /hop/object_recognition_caffe “‘

Service request arguments

“‘js { file: ” } “‘

- **file**: Path to the uploaded file (**image file**), stored by hop-server. This is the form-data name to attach the file to.

Service response

application/json response.

“‘js { object_class: ”, error: ” } “‘

- **object_class** (String): Recognized object class.
- **error** (String): Error message, if one occures. (String)

Chapter 27

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 28

RAPP-Email

RAPP Email provides an interface to allow users to handle their email accounts. It provides two services; fetch user's emails and send new email. The receive email service supports the IMAP protocol and the send mail the classic SMTP protocol. RAPP Email *does* **not** *implement an email server* and requires the user to have his own email provider who allows connections to the IMAP and SMTP servers.

ROS Services

Fetch Emails

Service URL: `/rapp/rapp_email/receive_email`

Service type:

ReceiveEmailSrv.srv "'bash

The user email username

string email

The user email password

string password

The email server's imap address, i.e. 'imap.gmail.com'

string server

The email server imap port

string port

Define which mails the users requests.

Values: ALL, UNSEEN(DEFAULT)

string requestedEmailStatus

Emails since date (unix timestamp)

uint64 fromDate

Emails until date (unix timestamp)

uint64 toDate

Number of requested emails

uint16 numberOfEmails

Response

0 success, -1 failure

int8 status

The requested emails

rapp_platform_ros_communications/MailMsg[] emails “‘
MailMsg.msg “‘

Path to the email body file

string bodyPath
string subject string sender string[] receivers string dateTime

Paths to the email’s attachments

string[] attachmentPaths “‘

Send Emails

Service URL: `/rapp/rapp_email/send_email`
Service type:
SendEmailSrv.srv “‘bash

The user email username

string userEmail

The user email password

string password

The email server's smtp address, i.e. 'smtp.gmail.com'

string server

The email server smtp port

string port

Email addresses of the recipients

string[] recipients

The email body

string body

The email subject

string subject

File paths of the attachments

string[] files

Response

0 success, -1 failure

int8 status "'

Launchers

Standard launcher

Launches the **rapp email** node and can be invoked using: “‘bash roslaunch rapp_email email.launch “‘

HOP Services

Send Email

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

Input/Output

“‘ Input = { file_uri: ”, email: ”, passwd: ”, server: ”, port: ”, recipients: [], body: ”, subject: ” } Output = { error: ” } “‘

Receive Email

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

Input/Output

“‘ Input = { email: ”, passwd: ”, server: ”, port: ”, email_status: ”, from_date: 0, to_date: 0, num_emails: 0 } Output = { emails: [], error: ” } “‘

Chapter 29

RAPP-Face-Detection

#Methodology

In the RAPP case, the face detection functionality is implemented in the form of a C++ developed ROS node, interfaced by a Web service. The Web 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 Web 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 Web service calls the ROS service via the ROS Bridge, the ROS node makes the necessary computations and a response is delivered.

In the current implementation the `fast` input variable exists. If `fast = False`, the face detection algorithm searches for faces both profile and en face and then cross-checks the results, aiming to provide the most precise result.

On the other hand, if `fast = True` only profile faces are checked without further checking, thus the response if faster but not that precise.

#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

Flag to define if a fast detection is desired

bool fast

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 “‘

#Web services

URL

`localhost:9001/hop/face_detection`

Input / Output

“‘ Input = { "file": “THE_ACTUAL_IMAGE_DATA” "fast": true } Output = { “faces”: [{ “up_left_point” : {x: 10, y: 30}, “down_right_point” : {x: 100, y: 200} }, { . . . }] } “‘

The full documentation exists here

Chapter 30

RAPP-Geolocator

RAPP Geolocator allows a user to get information about his location using his IP address. It is mainly considered an intermediate node that provides information to other nodes such as `RAPP Weather Reporter`. The availability of geolocator services that rely on third party APIs such as IP-API is restricted according to the APIs' rules and limitations. Thus, service call failures may exist.

Currently supported geolocators:

- `IP-API`

ROS Service

Locate

Service URL: `/rapp/rapp_geolocator/locate`

Service Type:

GeolocatorSrv.srv

‘‘‘ string ip

string geolocator

string error

string city string country string countryCode float32 latitude float32 longtitude string regionName string timezone string zip ‘‘‘ **Available geolocator values:**

- ” (uses a default geolocator)
- ’ip-api’

Launchers

Standard launcher

Launches the rapp geolocator node and can be invoked by executing:

```
roslaunch rapp_geolocator geolocator.launch
```

HOP Services

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

Input/Output

“‘ Input = { ipaddr: ”, engine: ” } “‘ **Available engine values:**

- ” (uses a default engine)
- ’ip-api’

“‘ Output = { city: ”, country: ”, country_code: ”, latitude: 0.0, longtitude: 0.0, region: ”, timezone: ”, zip: ”, error: ” } “‘

Chapter 31

RAPP-Hazard-Detection

In the RAPP case, the hazard 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 hazards has to be checked. The second step is for the HOP service to locally save the input image. At the same time, the hazard_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.

Light checking

Sample image (taken at exposure of 1ms) present a lamp that is switched on (see figure). Final result for image is computed by comparison of average brightness of eight surrounding regions (red) with central region (blue). In the case of looking at the lamp central region will be much brighter than surrounding (light left switched on). In case of looking through the door into other room central column will be brighter than border ones for light left switched on.

[[images/hazard_detection/lamp_on_small.jpg]]

Door checking

Solution is based on a single image (see figure), that is acquired in a way that a bottom part of the door, with (possibly) both right and left frame is visible. By analyzing vertical and horizontal lines decision about door opening angle is taken. If horizontal lines are almost parallel to each other, doors are closed. If the lines between frames (vertical) have different angle to those to the left and right of the frame, doors are treated as opened.

[[images/hazard_detection/door_1.png]]

#ROS Services

Light checking

Service URL: `/rapp/rapp_hazard_detection/light_check`

Service type: “‘bash

Contains info about time and reference

Header header

The image's filename to perform light checking

string imageFilename

Light level in the center of the provided image

int32 light_level string error “‘

Door checking

Service URL: `/rapp/rapp_hazard_detection/light_check`

Service type: “‘bash

Contains info about time and reference

Header header

The image's filename to perform door checking

string imageFilename

Estimated door opening angle

int32 door_angle string error “‘

#Launchers

Standard launcher

Launches the **hazard_detection** node and can be launched using “‘ roslaunch rapp_hazard_detection hazard_-detection.launch “‘

#Web services

Light checking

URL

`localhost:9001/hop/hazard_detection_light_check`

Input / Output

“‘ Input = { "file": “THE_ACTUAL_IMAGE_DATA” } Output = { "light_level": 50, "error": "" } “‘

Door checking

URL

`localhost:9001/hop/hazard_detection_door_check`

Input / Output

“‘ Input = { "file": "THE_ACTUAL_IMAGE_DATA" } Output = { "door_angle": 50, "error": "" } “‘

The full documentation exists here and here

Chapter 32

RAPP Platform Web Services

Synopsis

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

Platform Services	Description
Computer Vision	
`Face-Detection`	Detect faces on an image frame
`Qr-Detection`	Detect and recognize Qr-Codes on an image frame
`Hazard-Detection-Door-Check`	Detect open-doors (hazard) on an image frame
`Hazard-Detection-Light-Check`	Detect lights-on (hazard) on an image frame
`Human-Detection`	Detect human existance on on an image frame
`Object-Recognition-Caffe`	Recognize objects on an image frame using caffe framework
Speech Recognition	
`Set-Noise-Profile`	Set user's noise profile. Used to apply denoising on speech-recognition
`Speech-Detection-Sphinx4`	Performs speech-detection using the Platform integrated Sphinx4 engine
`Speech-Detection-Google`	Performs speech-detection using the Platform integrated Google engine
Ontology Queries	
`Ontology-SubclassesOf`	Perform Ontology, subclasses-of, query
`Ontology-SuperClassesOf`	Perform Ontology, superclasses-of, query
`Ontology-Is-SubSuperClass`	Perform Ontology, is-subsuperclass-of, query
Cognitive Exercises	
`Cognitive-Test-Selector`	Returns a Cognitive Exercise literal that describes the test

Cognitive-Record-Performance	Record user's performance on a Cognitive Exercise
Cognitive-Get-History	Returns user's history on Cognitive Exercise(s)
Cognitive-Get-Scores	Returns user's performance scores on Cognitive Exercise(s)
Email Support	
Email-Fetch	Fetch user's emails
Email-Send	Send an email, using user's account
Weather Report	
Weather-Report-Forecast	Get detailed information about future weather conditions
Weather-Repost-Current	Get detailed information about current weather conditions
Path Planning	
Path-Planning-Upload-Map	Upload a map to store on the Platform
Path-Planning-Plan-Path-2D	2D Path Planning service
Authentication-Registration	
Login-User	Login existing user
Register-User-From-Platform	Add new platform user using RAPP-Platform credentials
Register-User-From-Store	Add new platform user using RAPP-Store credentials
Other	
Text-To-Speech	Text-to-speech translation on given input plain text
Available-Services	Returns a list of the Platform available services (up-to-date)
Geolocaiton	Get information about client's location
News-Explore	Search for news articles

Service specifications - Request arguments and response objects

The Web Services listen to **POST** requests and can parse contents of the following type (Content-Type):

- application/x-www-form-urlencoded
- multipart/form-data

A lot of services require from the client to upload a file to the server for processing, like Computer-VIsion, Speech--Recognition and more. The *multipart/form-data* content-type is used to call a service that requresres file upload to the server. Otherwise, use *application/x-www-form-urlencoded*.

All data, except files, have to be be send under a field named **json**. In case of using *application/x-www-form-urlencoded* type this will look like:

"'http POST /hop/ontology_subclasses_of HTTP/1.1 Connection: keep-alive ... Content-Type: application/x-www-form-urlencoded

json=%7B%22query%22%3A+%22Oven%22%7D "'

and in case of multipart/form-data:

"'http POST /hop/face_detection HTTP/1.1 Connection: keep-alive ... Content-Type: multipart/form-data; boundary=993ac36c568042aa86582023b7422092

–993ac36c568042aa86582023b7422092 Content-Disposition: form-data; name="json"

{"fast": false} –993ac36c568042aa86582023b7422092 Content-Disposition: form-data; name="file"; filename="-Lenna.jpg"

<file-data-here> ...

‘‘‘

Computer Vision

Face-Detection

Service-Url

‘‘‘ /hop/face_detection ‘‘‘

Service request arguments

‘‘‘js { file: ”, fast: false } ‘‘‘

- **file**: Path to the uploaded file (**image file**), stored by hop-server. This is the form-data name to attach the file to.
- **fast** (Boolean): If true, detection will take less time but it will be less accurate.

Service response

application/json response.

‘‘‘js { faces: [{<face_1>}, ..., {<face_n>}], error: ” } ‘‘‘

- **faces** (Array): Vector with the recognized faces in an image frame.
- **error** (String): Error message, if one occures.

where **face_x** is an object of the following structure:

‘‘‘js face: { up_left_point: {<point>}, down_right_point: {<point>} } ‘‘‘

- **up_left_point** (Object): The up-left point coordinates of the detected face.
- **down_right_point** (Object): The down-right point coordinates of the detected face.

and **point** coordinates are presented in Cartesian Coordinate System as:

‘‘‘js point: { x: <value_int>, y: <value_int> } ‘‘‘

Response Sample:

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

QR-Detection

Service-Url

‘‘‘ /hop/qr_detection ‘‘‘

Service request arguments

‘‘‘js { file: ” } ‘‘‘

- **file**: Path to the uploaded file (**image file**), stored by hop-server. This is the form-data name to attach the file to.

Service response

application/json response.

“‘javascript { qr_centers: [{<point_1>}, ..., {<point_n>}], qr_messages: ["<qr_msg_1>", ..., "<qr_msg_n>"], error: ” } “‘

where **point_n** coordinates are presented in Cartesian Coordinate System as:

“‘js point: { x: <value_int>, y: <value_int> } “‘

- **qr_centers** (Array): Vector of points (x,y) of found QR in the image frame.
- **qr_messages** (Array): Vector that containes message descriptions of found QR in an image frame (Array of Strings).
- **error** (String): Error message, if one occures.

Response Sample:

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

Hazard-Detection-Door-Check

Service-Url

“‘ /hop/hazard_detection_door_check “‘

Service request arguments

“‘js { file: ” } “‘

- **file**: Path to the uploaded file (**image file**), stored by hop-server. This is the form-data name to attach the file to.

Service response

application/json response.

“‘js { door_angle: 0, error: "" } “‘

- **door_angle** (Integer): The angle of the detected door, on the image frame. (Integer)
- **error** (String): Error message, if one occures.

Hazard-Detection-Light-Check

Service-Url

“‘ /hop/hazard_detection_light_check “‘

Service request arguments

“‘js { file: ” } “‘

- **file**: Path to the uploaded file (**image file**), stored by hop-server. This is the form-data name to attach the file to.

Service response

application/json response.

“‘js { light_level: 0, error: "" } “‘

- **light_level** (Integer): The, detected on the iimage frame, light level. (Integer)
- **error** (String): Error message, if one occures. (String)

Human-Detection

Service-Url

“‘ /hop/human_detection “‘

Service request arguments

“‘js { file: ” } “‘

- **file**: Path to the uploaded file (**image file**), stored by hop-server. This is the form-data name to attach the file to.

Service response

application/json response.

“‘js { humans: [{<human_1>}, ..., {<human_n>}], error: "" } “‘

- **humans** (Array): Array of detected humans.
- **error** (String): Error message, if one occures. (String)

where **human_x** is an object of the following structure:

“‘js human: { up_left_point: {x: 0, y: 0}, down_right_point: {x: 0, y: 0} } “‘

Each point (x, y) is presented in Cartesian Coordinate System as:

“‘js point2D: { x: <val, y: <val> } “‘

Object-Recognition-Caffe

Service-Url

“‘ /hop/object_recognition_caffe “‘

Service request arguments

“‘js { file: ” } “‘

- **file**: Path to the uploaded file (**image file**), stored by hop-server. This is the form-data name to attach the file to.

Service response

application/json response.

“‘js { object_class: ”, error: ” } “‘

- **object_class** (String): Recognized object class.
- **error** (String): Error message, if one occures. (String)

Speech Recognition

Set-Noise-Profile

Service-Url

“‘ /hop/set_noise_profile “‘

Service request arguments

“‘js { file: ”, audio_source: ” } “‘

- **file**: Path to the uploaded file (**audio file**), stored by hop-server. This is the form-data name to attach the file to.
- **audio_source** (String): A value that presents the <robot>_<encode>_<channels> information for the audio source data. e.g "nao_wav_1_ch".

Service response

application/json response.

“‘javascript { error: ” } “‘

- **error** (String): Error message, if one occures.

Speech-Detection-Sphinx4

Service-Url

“‘ /hop/speech_detection_sphinx4 “‘

Service request arguments

“‘js { file: ”, language: ”, audio_source: ”, words: [], sentences: [], grammar: []} “‘

- **file**: Path to the uploaded file (**audio file**), stored by hop-server. This is the form-data name to attach the file to.
- **language** (String): Language to be used by the speech_detection_sphinx4 module. Currently valid language values are ‘gr’ for Greek and ‘en’ for English.
- **audio_source** (String): A value that presents the <robot>_<encode>_<channels> information for the audio source data. e.g "nao_wav_1_ch".
- **words** (Array): A vector that carries the words to search for into the voice-audio-source.
- **sentences** (Array): The under consideration sentences.
- **grammar** (Array): Grammars to use in speech recognition.

Service response

application/json response.

“‘javascript { words: [], error: ” } “‘

- **words** (Array): A vector with the "words-found"
- **error** (String): Error message, if one occures.

Speech-Detection-Google

Service-Url

“‘ /hop/speech_detection_google “‘

Service request arguments

“‘js { file: ”, audio_source: ”, language: ”} “‘

- **file**: Path to the uploaded file (**audio file**), stored by hop-server. This is the form-data name to attach the file to.
- **language** (String): Language to be used by the speech_detection_sphinx4 module. Currently valid language values are ‘gr’ for Greek and ‘en’ for English.
- **audio_source** (String): A value that presents the {robot}_{encode}_{channels} information for the audio source data. e.g "nao_wav_1_ch".

Service response

application/json response.

“‘javascript { words: [], alternatives: [] error: ” } “‘

- ∗∗’words’∗∗ (Array): A vector that contains the "words-found" with highest confidence.
- **alternatives** (Array): Alternative sentences. e.g. [[’send’, ’mail’], [’send’, ’email’], [’set’, ’mail’]...]
- **error** (String): Error message, if one occures.

Ontology Queries

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

Ontology-SubClasses-Of

Service-Url

“‘ /hop/ontology_subclasses_of “‘

Service request arguments

“‘js { ontology_class: ”, recursive: false } “‘

- **ontology_class** (String): The ontology class.
- **recursive** (Boolean): Recursive search.

Service response

application/json response.

“‘javascript { results: [], error: ” } “‘

- **results** (Array): Query results.
- **error** (String): Error message, if one occures.

“‘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

Service-Url

“‘ /hop/ontology_superclasses_of “‘

Service request arguments

“‘js { ontology_class: ”, recursive: false } “‘

- **ontology_class** (String): The ontology class.
- **recursive** (Boolean): Recursive search.

Service response

application/json response.
“‘javascript { results: [], error: ” } “‘

- **results** (Array): Query results returned from ontology database.
- **error** (String): Error message, if one occures.

Ontology-Is-SubSuperClass-Of

Service-Url

“‘ /hop/ontology_is_subsuperclass_of “‘

Service request arguments

“‘js { parent_class: ”, child_class: ”, recursive: false } “‘

- **parent_class** (String): The parent class name.
- **child_class** (String) The child class name.
- **recursive** (Boolean): Recursive search.

Service response

application/json response.
“‘javascript { result: true, error: ” } “‘

- **result** (Boolena): Success index on ontology-is-subsuperclass-of query.
- **error** (String): Error message, if one occures.

Cognitive Exercises

Cognitive-Test-Selector

Service-Url

“‘ /hop/cognitive_test_chooser “‘

Service request arguments

“‘js { test_type: ”, test_subtype: ”, test_diff: ”, test_index: ” } “‘

- **test_type** (String): Cognitive Exercise test type. Can be one of
  - ’ArithmeticCts’
  - ’AwarenessCts’
  - ’ReasoningCts’
  - ”
- **test_subtype** (String): Use this to force select from this subtype. Defaults to empty string "".
- **test_diff** (String): Use this to force select from this difficulty. Defaults to empty string "".
- **test_index** (String): Use this to force select from this id. Defaults to empty string "".

i.e

“‘js { test_type: ’Arithmetic’, test_subtype: ’BasicArithmetic’, test_diff: ’1’, test_index: ’1’ } “‘

For more information on available exercises have a look here

Service response

application/json response.

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

- **questions** (Array): The exercise set of questions.
- **possib_ans**(Array): The set of answers for each question. vector<vector<string>>
- **correct_ans**(Array): The set of correct answers for each question. vector<string>
- **test_instance** (String): Returned test name. For example, ’ArithmeticCts_askw0Snwk’
- **test_type** (String): Cognitive exercise class/type. Documentation on Cognitive Exercise classes can be found here
- **test_subtype** (String): Cognitive exercise sub-type. Documentation on Subtypes can be found here
- **error** (String): Error message, if one occures.

Cognitive-Record-Performance

Service-Url

“‘ /hop/cognitive_record_performance “‘

Service request arguments

“‘js { test_instance: ”, score: 0 } “‘

- **test_instance** (String): Cognitive Exercise test instance. The full cognitive test entry name as returned by the **cognitive_test_chooser** web service.
- **score** (Integer): User’s performance score on given test entry.

Service response

application/json response.

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

- **performace_entry** (String): User’s cognitive test performance entry in ontology.
- **error** (String): Error message, if one occures.

Cognitive-Get-History

Service-Url

“‘ /hop/cognitive_get_history “‘

Service request arguments

“‘js { from_time: ”, to_time: 0, test_type: ” } “‘

- **test_type** (String): Cognitive Exercise test type. Can be one of [’ArithmeticCts’, ’AwarenessCts’, ’Reasoning-Cts’] or leave empty ("") for all.
- **from_time** (Integer): Unix timestamp.
- **to_time** (Integer): Unix timestamp.

Service response

application/json response.

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

- **records** (Object): Users history records on Cognitive Exerises
- **error** (String): Error message, if one occures.

Cognitive-Get-Scores

Service-Url

“‘ /hop/cognitive_get_scores “‘

Service request arguments

“‘js { up_to_time: 0, test_type: ” } “‘

- **test_type** (String): Cognitive Exercise test type. Can be one of [’ArithmeticCts’, ’AwarenessCts’, ’Reasoning-Cts’] or leave empty ("") for all.
- **up_to_time** (Integer): Unix timestamp. Return scores that have been recorder up to this time value.

Service response

application/json response.

“‘javascript { test_classes: [], scores: [], error: ” } “‘

- **test_classes** (Array): An array of the test classes indexes.
- **scores** (Array): Array of scores. Each array index corresponds to the test class of the **test_classes** property.
- **error** (String): Error message, if one occures.

Email Support

Email-Fetch

Service-Url

“‘ /hop/email_fetch “‘

Service request arguments

“‘js { email: ”, passwd: ”, server: ”, port: ”, email_status: ”, from_date: 0, to_date: 0, num_emails: 0 } “‘

- **email** (String): The user’s email username
- **passwd** (String): The user’s email password
- **server** (String): The email server’s imap address, i.e. ’imap.gmail.com’
- **port** (String): The email server imap port
- **email_status** (String): Define which mails the users requests. Values: ALL, UNSEEN(DEFAULT)
- **from_date** (Integer): Emails since date. Unix timestamp.
- **to_date** (Integer): Emails until date. Unix timestamp.
- **num_emails** (Integer): Number of requested emails

Service response

application/json response.

“‘javascript { emails: [{<emailEntry_1>}, ..., {<emailEntry_n>}], error: ” } “‘

where emailEntry is an object of structure:

“‘js { sender: ”, receivers: [], body: ”, date: ”, attachments: [] } “‘

- **emails** (Array): An array of emailEntry objects.
- **error** (String): Error message, if one occures.

Email-Send

Service-Url

“‘ /hop/email_send “‘

Service request arguments

“‘js { email: ”, passwd: ”, server: ”, port: ”, recipients: [], body: ”, subject: ”, file: ” } “‘

- **email** (String): The user’s email username
- **passwd** (String): The user’s email password
- **server** (String): The email server’s smtp address, i.e. ’smtp.gmail.com’
- **port** (String): The email server imap port
- **recipients** (Array): Email addresses of the recipients
- **body** (String): The email body
- **subject** (String): The email subject
- **file**: File attachment. Single file. **In case of multiple attachments a zip file must be send to this field name.**

Service response

application/json response.

“‘javascript { error: ” } “‘

- **error** (String): Error message, if one occures.

Weather Report

Weather-Report-Current

Service-Url

“‘ /hop/weather_report_current “‘

Service request arguments

“‘js { city: ”, weather_reporter: ”, metric: 0 } “‘

- **city** (String): The desirec city
- **weather_reporter** (String): The weather API to use. Defaults to "yweather" .
- **metric** (Integer): The return value units.

Service response

application/json response.

“‘javascript { weather_current: { date: ”, temperature: ”, weather_description: ”, humidity: ”, visibility: ”, pressure: ”, wind_speed: ”, wind_temperature: ”, wind_direction: ” }, error: ” } “‘

- **date** (String): The date.
- **temperature** (String): The current temerature.
- **weather_description** (String): A brief desctiption of the current weather
- **humidity** (String): The current humidity
- **visibility** (String): The current visibility.
- **pressure** (String): The current pressure.
- **wind_speed** (String): The current speed of the wind.
- **wind_temperature** (String): The current temperature of the wind.
- **wind_direction** (String): The current direction of the wind.

Weather-Report-Forecast

Service-Url

“‘ /hop/weather_report_forecast “‘

Service request arguments

“‘js { city: ”, weather_reporter: ”, metric: 0 } “‘

- **city** (String): The desirec city
- **weather_reporter** (String): The weather API to use. Defaults to "yweather" .
- **metric** (Integer): The return value units.

Service response

application/json response.

“‘javascript { forecast: [{<forecastEntry_1>}, ..., {<forecastEntry_n>}], error: ” } “‘

- **forecast** (Array): Array of **forecastEntry** objects.
- **error** (String): Error message, if one occures.

where forecast entries are forecastEntry objects:

{ high_temp: ”, low_temp: ”, description: ”, date: ” }

Path Planning

Path-Planning-Plan-Path-2D

Service-Url

“‘ /hop/path_planning_plan_path_2d “‘

Service request arguments

“‘js { map_name: ”, robot_type: ”, algorithm: ”, start: {}, goal: {} } “‘

- **map_name** (String): The map name to use.
- **robot_type** (String): The robot type. It is required to determine it’s parameters (footprint etc.)
- **algorithm** {String}: The path planning algorithm to apply.
- **start** {Object}: Start pose of the robot. (ROS-GeometryMsgs/PoseStamped)
- **goal**: Goal pose of the robot. (ROS-GeometryMsgs/PoseStamped)

More information on argument values under the `rapp_path_planning package`

Service response

“‘js { plan_found: 0, path: [], error: ” } “‘

- **plan_found** (String): Plan Status. Can be one of
  - 0 : path cannot be planned.
  - 1 : path found.
  - 2 : wrong map name.
  - 3 : wrong robot type.
  - 4 : wrong algorithm.
- **path**: if plan_found is true, this is an array of waypoints from start to goal, where the first one equals start and the last one equals goal.
- **error** (String): Error message, if one occures.

Path-Planning-Upload-Map

Service-Url

“‘ /hop/path_planning_upload_map “‘

Service request arguments

“‘js { png_file: ”, yaml_file: ”, map_name: ” } “‘

- **png_file**: The map image png file.
- **png_file**: The map description yaml file.
- **map_name**: The map name.

Service response

“‘js { error: ” } “‘

- **error** (String): Error message, if one occures.

Authentication - Registration

Login-User

Service-Url

“‘ /hop/login_user “‘

Service request arguments

“‘js { username: ”, password: ”, device_token } “‘

- **username** (String): Account username.
- **password** (String): Account password.
- **device_token** (String): The device (token) from which a user tries to login.

Service response

application/json response.
“‘javascript { token: ”, error: ” } “‘

- **token** (String): Token used for accessing RAPP-Platform resources.
- **error** (String): Error message, if one occures.

Register-User-From-Platform

Service-Url

“‘ /hop/register_user_from_platform “‘

Service request arguments

“‘js { creator_username: ”, creator_password: ”, new_user_username: ”, new_user_password: ”, language: ” } “‘

- **creator_username** (String): Creator’s (robot-admin) RAPP-Platform account username.
- **creator_password** (String): Creator’s (robot-admin) RAPP-Platform account password.
- **new_user_username** (String): New user’s account username.
- **new_user_password** (String): New user’s account password.
- **language** (String): New user’s language.

Service response

application/json response.

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

- **suggested_username** (String): Suggested username if the provided one already exists.
- **error** (String): Error message, if one occures.

Register-User-From-Store

Service-Url

“‘ /hop/register_user_from_store “‘

Service request arguments

“‘js { username: ”, password: ”, device_token: ”, language: ” } “‘

- **username** (String): New user’s username.
- **password** (String): New user’s password.
- **device_token** (String): Creator device token from RAPP Store.
- **language** (String): New user’s account language.

Service response

application/json response.

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

- **suggested_username** (String): Suggested username if the provided one already exists.
- **error** (String): Error message, if one occures.

Other

Text-To-Speech

Service-Url

“‘ /hop/text_to_speech “‘

Service request arguments

“‘js { text: ”, language: ” } “‘

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

Service response

application/json response.

“‘javascript { payload: <audio_data>, basename: <audio_file_basename>, encoding: <payload_encoding>, error: <error_message> } “‘

- **payload** (String/base64): 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** (String): 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** (String): A static basename for the audio data file, returned by the platform service. Ignore this field. May be usefull in future implementations.
- **error** (String): Error message, if one occures.

Available-Services

"' /hop/available_services "'

Service request arguments

None.

Service response

"'js { services: [], error: " } "'

- **services** (Array): An array of available RAPP Platform Services.
- **error** (String): Error message, if one occures.

Geolocation

Service-Url

"' /hop/geolocation "'

Service request arguments

"'js { ipaddr: ", engine: " } "'

- **ipaddr**: The machine's ipaddr
- **engine**: Engine to use. Defaults to 'ip-api' (Currently the only supported).

Service response

application/json response.

"'js { city: ", country: ", country_code: ", latitude: 0.0, longtitude: 0.0, region: ", timezone: ", zip: ", error: " } "'

- **city**: (String): The city.
- **country** (String): The country.
- **country_code** (String): The country code.
- **latitude**: (Float): The latitude.
- **longtitude** (Float): The longtitude.
- **timezone** (String): The timezone.
- **zip** (String): The zip postal code.
- **error** (String): Error message, if one occures.

News-Explore

Service-Url

“‘ /hop/news_explore “‘

Service request arguments

“‘js { news_engine: ”, keywords: [], exclude_titles: [], region: ”, topic: ”, num_news: 0 } “‘

- **news_engine** (String): The news search engine to use.
- **keywords** (Array): Desired keywords.
- **exclude_titles** (Array): Reject list of previously read articles, in order to avoid duplicates.
- **region** (String): Language/Region.
- **topic** (String): Main topics, i.e. sports, politics, etc.
- **num_news** (Integer): Number of news stories.

Service response

application/json response.

“‘js { news_stories: [{<story_1>}, ..., {<story_n>}], error: ” } “‘

- **news_stories** (Array): Array of **story** objects.
- **error** (String): Error message, if one occures.

where *story* object is:

“‘js { title: ”, content: ”, puplisher: ”, publishedDate: ”, url: ” } “‘

- **title** (String): Article title.
- **content** (String): Article brief content.
- **publisher** (String): Article publisher.
- **publishedDate** (String): Article publication date.
- **url** (String): Article original url.

Chapter 33

RAPP-Human-Detection

#Methodology

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

#ROS Services

Human detection

Service URL: `/rapp/rapp_hazard_detection/light_check`

Service type: “‘bash

Contains info about time and reference

Header header

The image’s filename to perform light checking

string imageFilename

List of bounding box borders, where the humans were detected

geometry_msgs/PointStamped[] humans_up_left geometry_msgs/PointStamped[] humans_down_right

Possible error

string error “‘

#Launchers

Standard launcher

Launches the **human_detection** node and can be launched using “‘ roslaunch rapp_human_detection human_-detection.launch “‘

#Web services

Human detection

URL

`localhost:9001/hop/human_detection`

Input / Output

“‘ Input = { "file": “THE_ACTUAL_IMAGE_DATA” } Output = { "humans": [{ "up_left_point": {x: 0, y: 0}, "down_-right_point": {x: 0, y: 0} }] } “‘

The full documentation exists here

Chapter 34

RAPP-Knowrob-wrapper

#Methodology

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 services of the RAPP Knowrob wrapper are detailed below.

#ROS Services Each service is analyzed below.

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 “‘

Register image object to ontology

This service will register an image, annotated by the rapp_caffe_wrapper to the ontology.

Service URL: `/rapp/rapp_knowrob_wrapper/register_image_object_to_ontology`

Service type: “‘bash

Contains info about time and reference

Header header #the user’s ontology alias string user_ontology_alias #the ontology class of the to be registered object string object_ontology_class #the caffe class of the to be registered object string caffe_class #the path to the image representing the object string image_path #timestamp of the time of registration

int32 timestamp

The results of the query

#the name of the registered object entry string object_entry #possible error string error #trace information string[] trace #true if service call was successful bool success “‘

Retract user ontology alias

This service will remove the ontology alias of a user from the ontology.

Service URL: `/rapp/rapp_knowrob_wrapper/retract_user_ontology_alias`

Service type: “‘bash

Contains info about time and reference

Header header #the user ontology alias to be retracted

string ontology_alias

The results of the query

#possible error string error #trace information string[] trace #true if service call was successful bool success “‘

Clear user cognitive test performance records

This service will remove all cognitive test performance records of a user from the ontology.

Service URL: `/rapp/rapp_knowrob_wrapper/clear_user_cognitive_tests_performance_records`

Service type: “‘bash

Contains info about time and reference

Header header #the username of the user whose records will be cleared string username #specify clearing the records of this test type onle

string test_type

The results of the query

#possible error string error #trace information string[] trace #true if service call was 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

Ontology-SubClasses-Of

Service-Url

“‘ /hop/ontology_subclasses_of “‘

Service request arguments

“‘js { query: ” } “‘

- **query** (String): The query to the ontology database.

Service response

application/json response.

“‘javascript { results: [], error: ” } “‘

- **results** (Array): Query results.
- **error** (String): Error message, if one occures.

“‘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

Service-Url

“‘ /hop/ontology_superclasses_of “‘

Service request arguments

“‘js { query: ” } “‘

- **query** (String): The query to the ontology database.

Service response

application/json response.
“‘javascript { results: [], error: ” } “‘

- **results** (Array): Query results returned from ontology database.
- **error** (String): Error message, if one occures.

Ontology-Is-SubSuperClass-Of

Service-Url

“‘ /hop/ontology_is_subsuperclass_of “‘

Service request arguments

“‘js { parent_class: ”, child_class: ”, recursive: false } “‘

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

Service response

application/json response.
“‘javascript { result: true, error: ” } “‘

- **result** (Boolena): Success index on ontology-is-subsuperclass-of query.
- **error** (String): Error message, if one occures.

Chapter 35

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 36

RAPP-MySQL-wrapper

#Methodology

The RAPP MySQL wrapper ROS node provides the means to interact with the developed MySQL database by providing a number of ROS services that perform specific, predefined procedures of reading from and writing to the database.

#ROS Services Each service is analyzed below.

Add store token to device service

Service URL: `/rapp/rapp_mysql_wrapper/add_store_token_to_device`

Service type: “‘bash

string store_token

string error “‘

Check active application token service

Service URL: `/rapp/rapp_mysql_wrapper/check_active_application_token`

Service type: “‘bash

string application_token

bool application_token_exists bool success string error string[] trace “‘

Check active robot session service

Service URL: `/rapp/rapp_mysql_wrapper/check_active_robot_session`

Service type: “‘bash string username

string device_token

bool application_token_exists bool success string error string[] trace “‘

Check if user exists service

Service URL: `/rapp/rapp_mysql_wrapper/check_if_user_exists`

Service type: “‘bash

string username

bool user_exists bool success string error string[] trace “‘

Create new application token service

Service URL: `/rapp/rapp_mysql_wrapper/create_new_application_token`

Service type: “‘bash string username string store_token

string application_token

bool success string error string[] trace “‘

Create new cloud agent service

Service URL: `/rapp/rapp_mysql_wrapper/create_new_cloud_agent`

Service type: “‘bash string username string tarball_path string container_identifier string image_identifier

string container_type

bool success string error string[] trace “‘

Create new cloud agent service service

Service URL: `/rapp/rapp_mysql_wrapper/create_new_cloud_agent_service`

Service type: “‘bash string container_identifier string service_name string service_type int16 container_port

int16 host_port

bool success string error string[] trace “‘

Create new platform user service

Service URL: `/rapp/rapp_mysql_wrapper/create_new_platform_user`

Service type: “‘bash string username string password string creator_username

string language

bool success string error string[] trace “‘

Get cloud agent service type and host port service

Service URL: `/rapp/rapp_mysql_wrapper/get_cloud_agent_service_type_and_host_port`

Service type: “‘bash string container_identifier

string service_name

string service_type int16 host_port bool success string error string[] trace “‘

Get user language service

Service URL: `/rapp/rapp_mysql_wrapper/get_user_language`

Service type: “‘bash

string username

string user_language bool success string error string[] trace “‘

Get username associated with application token service

Service URL: `/rapp/rapp_mysql_wrapper/get_username_associated_with_application_token`

Service type: “‘bash

string application_token

string username bool success string error string[] trace “‘

Get user ontology alias service

Service URL: `/rapp/rapp_mysql_wrapper/get_user_ontology_alias`

Service type: “‘bash

string username

string ontology_alias bool success string error string[] trace “‘

Get user password service

Service URL: `/rapp/rapp_mysql_wrapper/get_user_password`

Service type: “‘bash

string username

string password bool success string error string[] trace “‘

Register user ontology alias service

Service URL: `/rapp/rapp_mysql_wrapper/register_user_ontology_alias`

Service type: “‘bash string username

string ontology_alias

bool success string error string[] trace “‘

Remove platform user service

Service URL: `/rapp/rapp_mysql_wrapper/remove_platform_user`

Service type: “‘bash

string username

string password bool success string error string[] trace “‘

Validate existing platform device token service

Service URL: `/rapp/rapp_mysql_wrapper/validate_existing_platform_device_token`

Service type: “‘bash

string device_token

bool device_token_exists bool success string error string[] trace “‘

Validate user role service

Service URL: `/rapp/rapp_mysql_wrapper/validate_user_role`

Service type: “‘bash

string username

string error string[] trace “‘

Launchers

Standard Launcher

Launches the rapp_mysql_wrapper node and can be invoked using `roslaunch rapp_mysql_wrapper mysql_wrapper.launch`

Chapter 37

RAPP-News-Explorer

Rapp News Explorer allows users to search for news articles employing various news search engine APIs. The user can specify the desired news search engine, topics (i.e. sports, economy, headlines, etc.), desired keywords, language/region (can be specified employing `RAPP Geolocator`), etc. The node returns a list of related articles. The availability of news_explorer services that rely on third party APIs such as Google News are restricted according to the APIs' rules and limitations. Thus, service call failures may exist.

Currently supported News Engines:

- Google News (`API`) (discontinued by Google)
- `EventRegistry` (`GitHub wiki`)

EventRegistry does not require login, however it is subjected to `daily access restrictions.` If you have an EventRegistry account, you should provide the credentials in the file `${HOME}/.config/rapp_platform/api_keys/event_registry` in the format:

‘‘‘ username password ‘‘‘

ROS Services

Fetch News

Service URL: `/rapp/rapp_news_explorer/fetch_news`

Service Type:

NewsExplorerSrv.srv ‘‘‘

Request

The news search engine

string newsEngine

Desired keywords

string[] keywords

Reject list of previously read articles, in order to avoid duplicates

string[] excludeTitles

Language/Region

string regionEdition

Main topics, i.e. sports, politics etc

string topic

Number of news stories

int8 storyNum

Response

string error # Error description

List of news articles

rapp_platform_ros_communications/NewsStoryMsg[] stories "' **Available newsEngine values:**

- '' (uses default news engine)
- 'google'
- 'event_registry'

NewsStoryMsg.msg "'

Article title

string title

Article brief content

string content

Article publisher

string publisher

Article publication date

string publishedDate

Article original url

string url “‘

Launchers

Standard launcher

Launches the rapp_news_explorer node and can be invoked by executing:

```
roslaunch rapp_news_explorer news_explorer.launch
```

HOP Services

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

Input/Output

“‘ Input = { news_engine: ”, keywords: [], exclude_titles: [], region: ”, topic: ”, num_news: 25 } “‘ **Available news_-engine values:**

- ” (uses default news engine)
- ’google’
- ’event_registry’

“‘ Output = { news_stories: [], error: ” } “‘

Chapter 38

RAPP-Object-Recognition

Recognition of known objects, implemented in RAPP Platform, is based on feature points extracted from the image. From this reason, only textured objects can be recognized and localized on the image at the moment, but in typical RAPP scenarios there are a lot of objects of this type - food boxes and cans, books, medicines and more everyday things. Object recognition module is able to recognize multiple objects at the same time, many instances od the same type of object in particular. Whole process of object learning and recognition is composed of four services.

Every user using object recognition module owns separate objects database. Recognition module can use all of the models or only selected subset during object detection, and this subset loaded at the moment is called operational memory.

Learn new object

Recognition module is prepared to find specific object instances, learnt beforehand by the user. To learn new objects, picture consisting its view on plain (featureless) background should be used. Best recognition results are achieved, if models are learnt using the same sensor that will be used for detection (i.e. don't use high resolution, high quality object images if recognition will be done using lower quality camera). Every model should have different name. Subsequent calls to learn service with the same object name will overwrite previously learnt model. Model names are used in further calls of load service.

Clear objects

Clears operational memory for selected user. After this operation, object recognition module will always fail to recognize anything.

Load models

Load one or more models to operational memory. This operation should be done at least once before first recognition request. Subsequent calls of this service extends operational memory. To replace operational memory with new models, clear service should be called first.

Find objects

When set of models is loaded to operational memory, user can provide query image to detect objects on. If any object of known type is recognized, its center point in query image, model name and recognition score (certainty) is returned. User can setup search results to contain only limited number of strongest object hypotheses. If more objects are found, weakest are dropped.

[[images/object_recognition/ex_match_0.jpg]]

ROS Services

Learn object

Service URL: `/rapp/rapp_object_recognition/learn_object`

Service type: “‘bash

Path (id) of model image

string fname

Name of the object

string name

User name

string user

Result: 0 - ok, -1 - no models, -2 - no image to analyse

int32 result “‘

Clear models

Service URL: `/rapp/rapp_object_recognition/clear_models`

Service type: “‘bash

User name

string user

Result: 0 - ok, -1 - no models, -2 - no image to analyse

int32 result “‘

Load models

Service URL: `/rapp/rapp_object_recognition/load_models`

Service type: “‘bash

User name

string user

Object names to load

string[] names

Result: 0 - ok, -1 - no models, -2 - no image to analyse

int32 result “‘

Find objects

Service URL: `/rapp/rapp_object_recognition/find_objects`

Service type: “‘bash

Path (id) of query image

string fname

Limit search to N best matches

uint32 limit

User name

string user

List of found objects - names

string[] found_names

List of found objects - centroids in image

geometry_msgs/Point[] found_centers

List of found objects - scores

float64[] found_scores

Result: 0 - ok, -1 - no models, -2 - no image to analyse

int32 result “‘

#Launchers

Standard launcher

Launches the **object_recognition** node and can be launched using “‘ roslaunch rapp_object_recognition object_-detection.launch “‘

#Web services

TBD

Chapter 39

RAPP-Path-planner

List of contents

Components

1. *rapp_path_planning*

Rapp_path_planning is used in the RAPP case to plan path from given pose to given goal. User can costomize the path planning module with following parameters:

- pebuild map - avaliable maps are stored here,
- planning algorithm - **for now, only** dijkstra **is avaliable**,
- robot type - customizes costmap for planning module. **For now only** NAO **is supported**.

2. *rapp_map_server*

Rapp_map_server delivers prebuild maps to rapp_path_planning component. All avaliable maps are contained here.

This component is based on the ROS package: map_server. The rapp_map_server reads .png and .yaml files and publishes map as OccupacyGrid data. RAPP case needs run-time changing map publication, thus the rapp_map_server extands map_server functionality. The rapp_map_server enables user run-time changes of map. It subscribes to ROS parameter: rospy.set_param(nodeName+"/setMap", map_path) and publishes the map specified in map_path. Examplary map changing request is presented below. "'python nodename = rospy.get_name() map_path = "/home/rapp/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_map_server/maps/empty.yaml" rospy.set_param(nodename+/setMap, map_path) A ROS service exists to store new maps in each user's workspace, calledupload_map. Then each application can invoke theplanPath2D"' service, providing the map's name (among others) as input argument.

Package launch

∗**Standard launch**∗

Launches the **path planning** node and can be launched using “‘bash roslaunch rapp_path_planning path_-planning.launch “‘

Prepare new map manually

New map files have to be compatible with the `map_server` package. Detailed description of files and their parameters can be found `here`. Exemplary map files can be found `here`.

1. Prepare PNG file:
   a) Dimension desired area,
   b) Scale measurements by a desired factor. Chosen factor represents the map resolution!
   c) Draw obstacles and walls with black color carefully. Size of obstacles and walls is very important and every pixel makes difference!
   d) Rest of the map should be left white.
   e) Export your image to the .png file. Name of the file is important, so choose wisely!
2. Write configuration file:
   a) Create empty file: `<map_name>.yaml`
   b) Open the file and set following configuration parameters:

“‘

Name of .png file.

image: <png_file_name> # example -> image: test_map.png

The map’s resolution [meters / pixel]

resolution: <resolution> # example -> resolution: 0.1

The map’s origin. 2D pose of the map origin. [x, y, yaw]

origin: <map_origin> # example -> origin: [0.0, 0.0, 0.0]

Whether the occupied / unoccupied pixels must be negated

negate: <negate> # example -> negate: 0

Pixels with occupancy probability greater than this threshold are considered completely occupied.

occupied_thresh:

example -> occupied_thresh: 0.65

Pixels with occupancy probability less than this threshold are considered completely free.

free_thresh:

example -> free_thresh: 0.196 “‘

ROS Services

2D path planning

Service URL: `/rapp/rapp_path_planning/planPath2D`

Service type: “‘bash

Contains name to the desired map

string map_name

Contains type of the robot. It is required to determine it’s parameters (footprint etc.)

string robot_type

Contains path planning algorithm name

string algorithm

Contains start pose of the robot

geometry_msgs/PoseStamped start

Contains goal pose of the robot

geometry_msgs/PoseStamped goal

status of the service

plan_found:

∗ 0 : path cannot be planned.

∗ 1 : path found

∗ 2 : wrong map name

∗ 3 : wrong robot type

∗ 4 : wrong algorithm

uint8 plan_found

error_message : error explanation

string error_message

path : vector of PoseStamped objects

if plan_found is true, this is an array of waypoints from start to goal, where the first one equals start and the last one equals goal

geometry_msgs/PoseStamped[] path "'

map file upload

Service URL: `/rapp/rapp_path_planning/upload_map`

Service type: "'bash

The end user's username, since the uploaded map is personal

string user_name

The map's name. Must be unique for this user

string map_name

The map's resolution

float32 resolution

ROS-specific: The map's origin

float32[] origin

ROS-specific: Whether the occupied / unoccupied pixels must be negated

int16 negate

Occupied threshold

float32 occupied_thresh

Unoccupied threshold

float32 free_thresh

File size for sanity checks

uint32 file_size

The map data

char[] data

byte status “‘ More information on the Occupancy Grid Map representation can be found here

Web services

Path planning 2D

URL

`localhost:9001/hop/path_planning_path_2d`

Input / Output

“‘ Input = { "map_name": “THE_PRESTORED_MAP_NAME”, "robot_type": "Nao", "algorithm": "dijkstra", "start": {x: 0, y: 10}, "goal": {x: 10, y: 0} } Output = { "plan_found": 0, "path": [{x: 0, y: 10}, {x: ...], "error": "" } “‘

Upload map

URL

`localhost:9001/hop/path_planning_upload_map`

Input / Output

“‘ Input = { "png_file": “map.png”, "yaml_file": "map.yaml", "map_name": "simple_map_1" } Output = { "error": "" } “‘

The full documentation exists here and here.

Author

Wojciech Dudek

Chapter 40

(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 41

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 42

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 43

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

QR 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 "'

Web services

URL

`localhost:9001/hop/qr_detection`

Input / Output

“‘ Input = { “file”: “THE_ACTUAL_IMAGE_DATA” } Output = { “qr_centers”: [{x: 100, y: 200}], "qr_messages": ["rapp project qr sample"], "error": "" } “‘

The full documentation exists here

Chapter 44

RAPP-Scripts

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

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-scripts repository in a `zip format`, or by cloning it in your PC using git:

```
git clone https://github.com/rapp-project/rapp-platform-scripts.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.
- 12_caffe_setup.sh: Installs Caffe.
- 13_authentication_setup.sh: Installs authentication related information.

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-scripts/blob/master/setup/4-_rosjava_setup.sh
  - https://github.com/rapp-project/rapp-platform-scripts/blob/master/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!

utillities

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
- create_rapp_user.sh: Create and authenticate a new RAPP User.

Chapter 45

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 “‘

Speech detection Google 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 = { “file”: “THE_AUDIO_FILE” “audio_source”: “nao_ogg, nao_wav_1_ch, nao_wav_4_ch” “language”: “el” } Output = { “words”: [“WORD_1”, “WORD_2”, . . . , “WORD_N”], “alternatives”: [[“WORD_1”, “WORD_2”, . . . , “WORD_N”], [....]], "error": "" } “‘

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”], “alternatives”: [[“I”, “want”, “to” , “use” , “the” , “Skype”], [“I”, “want”, “to” , “use” , “Skype”], [“I”, “want”, “to” , “use” , “the” , “Skype” , “app”]], "error": "" } “‘

The full documentation exists here

Chapter 46

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

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

It should be stated that the language model is created based on the ARPA model for the `sentences` and on the JSGS for the `grammar` parameters, nevertheless only pure sentences are supported (i.e. the advanced JSGF uses cannot be employed). More information on the Sphinx language model can be found here.

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 “‘

Web 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_sphinx4`

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 . . .]” “file”: “AUDIO_FILE_URI” “audio_source”: “nao_ogg, nao_wav_1_ch, nao_wav_4_ch, headset” } 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.
- `file`: 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

The full documentation exists here

Chapter 47

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 Python implementation of the RAPP Platform API,`rapp-platform-api/python`, to request for .

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

- The source code of the test class.
- (**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 is splitted into the following:

- The RAPP Platform Service call(s).
- The validation of the response.

Test classes inherit from Python unit testing framework, `unittest`

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. This script accepts the following arguments passed by the cli:

- [∗∗-i∗∗] : Test filepath, to execute. If empty, all tests under the **default_tests** directory will be executed.

“‘bash $ ./rapp_run_test_.py -i default_tests/face_detection_tests.py “‘

- [∗∗-n∗∗] : Number of times of test(s) execution.

“‘bash $ ./rapp_run_test.py -i default_tests/face_detection_tests.py -n 10 “‘

- [**-t**] : Run the test in threaded mode. Multible invocations can be done, simultaneous. Each test execution is handled by a single thread.

“‘bash $ ./rapp_run_test.py -i default_tests/face_detection_tests.py -n 10 -t “‘

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

Developing integration tests

A template_test.py is located under the `default_tests` directory of the rapp_testing_tools package.

“‘bash $ <path_to_rapp_testing_tools_package>/scripts/default_tests/ “‘

Also, several tests have already been developed, located under the default_tests directory. Those can be used as a reference/examples for developing new tests.

A Section in the `Wiki` exists that guides through developing integration tests for the RAPP Platform.

Chapter 48

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 “‘

It should be mentioned that there are cases where Espeak fails to produce the entirety of the given text as audio (for example if the input is a long sentence in Greek, it would probably create a shorter audio than the desired).

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 “‘

Web services

URL

`localhost:9001/hop/text_to_speech`

Input / Output

“‘ Input = { “text”: “THE_TEXT_TO_BECOME_SPEECH” “language”: “THE_TEXT_LANGUAGE” } Output = { "payload": THE_AUDIO_DATA, "basename": "test.wav", "encoding": "base64", "error": "" } “‘

The full documentation exists here

Chapter 49

RAPP-Utilities

RAPP Utilities provides users with basic functionalities.

Currently, these comprise:

- exceptions
- html_parser
- http_json_parser
- http_request_handler
- utilities

Exceptions

This package provides the RappError exception class.

HTML Parser

This package allows the handling of url encoded strings.

HTTP_JSON Parser

This package allows the handling of JSON HTTP responses.

HTTP request handler

This package allows users to perform HTTP requests.

Utilities

This package provides various utility functions. Currently it provides the `rapp_print` function which is used in the RAPP project to print ROS log messages.

Chapter 50

RAPP-Weather-Reporter

RAPP Weather Reporter allows users to get details about current or future weather conditions. It provides two services; a) current weather conditions and b) weather forecast. The user has to provide his current city, which could be evaluated using `RAPP Geolocator`. The availability of weather services that rely on third party APIs such as YWeahter are restricted according to the APIs' rules and limitations. Thus, service call failures may exist.

Currently supported weather reporters:

- `YWeather`
- `ForecastIO` (default)

In order for the ForecastIO weather reporter to be deployed an API key is required. After obtaining the key via registration, the key should be stored in the following location: "' ${HOME}/.config/rapp_platform/api_keys/forecast.-io "'

ROS Services

Current Weather

Service URL: `/rapp/rapp_weather_reporter/current_weather`

Service Type: "'

The desired city

string city

The weather API

string weather_reporter

The return value units

int8 metric

string error

string date string temperature string weather_description string humidity string visibility string pressure string wind-_speed string wind_temperature string wind_direction “‘ **Available weather_reporter values:**

- ” (uses default weather reporter)
- ’yweather’
- ’forecastio’

Weather Forecast

Service URL: `/rapp/rapp_weather_reporter/weather_forecast`

Service Type: “‘

The desired city

string city

The weather API

string weather_reporter

The return value units

int8 metric

string error

rapp_platform_ros_communications/WeatherForecastMsg[] forecast “‘

Available weather_reporter values:

- ” (uses default weather reporter)
- ’yweather’
- ’forecastio’

WeatherForecastMsg.msg

“‘ string high_temperature string low_temperature string description string date “‘

Launchers

Standard Launcher

Launches the rapp weather reporter node and can be invoked by executing:

```
roslaunch rapp_weather_reporter weather_reporter.launch
```

HOP Services

Current weather

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

Input/Output

“‘ Input = { city: ”, weather_reporter: ”, metric: 0 } “‘

Available weather_reporter values:

- ” (uses default weather reporter)
- ’yweather’
- ’forecastio’

“‘ Output = { date: ”, temperature: ”, weather_description: ”, humidity: ”, visibility: ”, pressure: ”, wind_speed: ”, wind_temperature: ”, wind_direction: ”, error = ” } “‘

Weather forecast

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

Input/Output

“‘ Input = { city: ”, weather_reporter: ”, metric: 0 } “‘

Available weather_reporter values:

- ” (uses default weather reporter)
- ’yweather’
- ’forecastio’

“‘ Output = { forecast: [], error: ” } “‘

Chapter 51

(normal) Remote-application-for-NAO-in-Python:--Detect-and-track-QR-tags-(normal)

The current tutorial describes a more complicated application, as it will employ both the robot-agnostic RAPP robot API and the RAPP Platform API, aiming to implement a simple detection of a QR code, as well as a reaction from NAO. This application will be executed remotely in a PC.

For this tutorial we will use the following tools:

- A real NAO robot
- the NaoQi Python libraries
- the rapp-robots-api Github repository
- the rapp-api Github repository
- the rapp-platform up and running

Of course the standard prerequisites are a functional installation of Ubuntu 14.04, an editor and a terminal.

Preparation steps

NaoQI Python libraries setup

Please check here#naoqi-python-libraries-setup).

RAPP Robots Python API setup

Please check here#rapp-robots-api-libraries-setup).

RAPP Platform API setup

It is assumed that till now you have created the ∼/rapp_nao folder where the NaoQi Python libraries and the RAPP Robots API Python libraries exist. The next step is to clone the RAPP Platform API:

“‘bash cd ∼/rapp_nao git clone https://github.com/rapp-project/rapp-api.git “‘

In order to utilize the RAPP Platform API the PYTHONPATH environmental variable has to be updated:

“‘bash echo ’export PYTHONPATH=$PYTHONPATH:∼/rapp_nao/rapp-api/python’ >> ∼/.bashrc source ∼/.bashrc “‘

It should be stated that if you have already set-up the RAPP Platform in your PC, the RAPP Platform API has been already installed and the previous step is not necessary.

RAPP Platform setup

In order to set-up the RAPP Platform you can follow this tutorial. Have in mind that the RAPP Platform installation is quite time consuming.

Writing the application

Let's create a Python file for our application and give it execution rights:

"'bash cd ∼/rapp_nao mkdir rapps && cd rapps touch qr_app.py chmod +x qr_app.py "'

The robotic application follows:

"'python #!/usr/bin/env python

import time, threading from os.path import expanduser

Import the RAPP Robot API

from rapp_robot_api import RappRobot

Import the RAPP Platform API

from RappCloud import RappPlatformAPI

Create an object in order to call the desired functions

rh = RappRobot() ch = RappPlatformAPI()

Enable the NAO motors for the head to move

rh.motion.enableMotors()

def callback(): print "Callback called!"

Capture an image from the NAO cameras

rh.vision.capturePhoto("/home/nao/qr.jpg", "front", "640x480")

Get the photo to the PC

rh.utilities.moveFileToPC("/home/nao/qr.jpg", expanduser("∼") + "/Pictures/qr.jpg")

Check if QRs exist

imageFilepath = expanduser("∼") + "/Pictures/qr.jpg" res = ch.qrDetection(imageFilepath) print "Call to platform finished"

if len(res['qr_centers']) == 0: # No QR codes were detected print "No QR codes were detected" else: # One or more QR codes detected print "QR code detected" qr_center = res['qr_centers'][0] qr_message = res['qr_messages'][0]

Directions are computed bounded in [-1,1]

dir_x = (qr_center['x'] - (640.0/2.0)) / (640.0 / 2.0) dir_y = (qr_center['y'] - (480.0/2.0)) / (480.0 / 2.0) angle_x = -dir_x ∗ 30.0 / 180.0 ∗ 3.1415 angle_y = dir_y ∗ 23.7 / 180.0 ∗ 3.1415

Get NAO head angles

ans = rh.humanoid_motion.getJointAngles(["HeadYaw", "HeadPitch"]) angles = ans['angles']

Set NAO angles according to the QR center

rh.humanoid_motion.setJointAngles(["HeadYaw", "HeadPitch"], \ [angle_x + angles[0], angle_y + angles[1]], 0.1)

if callback.qr_found == False: rh.audio.speak("I found a QR with message: " + qr_message) rh.audio.speak("I will track it") callback.qr_found = True

Capture an image from the NAO cameras

rh.vision.capturePhoto("/home/nao/qr.jpg", "front", "640x480")

Get the photo to the PC

rh.utilities.moveFileToPC("/home/nao/qr.jpg", expanduser("∼") + "/Pictures/qr.jpg")

threading.Timer(0, callback).start()

callback.qr_found = False callback() "'

In order to execute the application the RAPP Platform must be up and running. This can be done by executing the two deployment scripts:

"' bash ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_scripts/deploy/deploy_rapp_ros.sh "' and "' bash ∼/rapp_platform/rapp-platform-catkin-ws/src/rapp-platform/rapp_scripts/deploy/deploy_web_services.sh "'

Finally, you just have to execute `python ∼/rapp_nao/rapps/qr_app.py` to deploy the application.

Chapter 52

(hard) Remote-application-for-NAO-in-Python:-Use-ROS-&-TLD-tracker-to-approach-arbitrary-objects-(hard)

In the current tutorial we will showcase the way to create a more complex application which utilizes external (out of RAPP) packages. This application will be create as a ROS node, in order to take advantage of other ROS packages.

In this application we will track an object using the NAO front camera and the `OpenTLD` algorithm, as well as `its wrapper` used in the `PANDORA robotics team`.

Thus for this tutorial we will need:

- A real NAO robot
- the NaoQi Python libraries
- the rapp-robots-api GitHub repository
- the OpenTLD GitHub repository
- the PANDORA OpenTLD wrapper
- ROS Indigo installed
- ROS packages for NaoQi

This application's code can be found `here` in the form of a ROS package.

Preparation

Setup the NaoQi Python libraries

Check `here`#naoqi-python-libraries-setup) for details.

Setup the RAPP Robots Python API

Check `here`#rapp-robots-api-libraries-setup) for details.

Install ROS Indigo and ROS packages for NaoQi

The installation procedure can be found `here` for the Ubuntu 14.04 distribution.

After ROS installation, install the NaoQi ROS packages by:

“‘bash sudo apt-get install ros-indigo-naoqi-∗ sudo apt-get install ros-indigo-nao-bringup ros-indigo-nao-apps ros-indigo-usb-cam “‘

Setup OpenTLD and PANDORA TLD ROS packages

Initially, a `catkin_repository` must be created and the ROS packages cloned in it:

“‘bash

Create the proper folders

mkdir -p ∼/rapp_nao/rapps && cd ∼/rapp_nao/rapps mkdir -p rapp_catkin_ws/src && cd rapp_catkin_ws/src

Initialize the Catkin workspace

catkin_init_workspace

Clone the OpenTLD and PANDORA TLD packages

git clone https://github.com/pandora-auth-ros-pkg/pandora_tld.git git clone https://github.com/pandora-auth-ros-pkg/open_tld.git

Build the packages

cd ∼/rapp_nao/rapps/rapp_catkin_ws && catkin_make -j1

Export the proper environmental variables

echo ’source ∼/rapp_nao/rapps/rapp_catkin_ws/devel/setup.bash –extend’ >> ∼/.bashrc source ∼/.bashrc “‘

Next, you must change the input image topic to TLD in order to bind the callback from the NAO camera. The `∼/rapp_nao/rapps/rapp_catkin_ws/src/pandora_tld/config/predator_topics.yaml` must include the following:

“‘yaml predator_alert : /vision/predator_alert input_image_topic: /nao_robot/camera/top/camera/image_raw begin_hunt_topic: /predator/hunt “‘

Creating the application

Initially a ROS package must be created that will facilitate the code. The package will have `rospy` as dependency and will be named `tld_tracker_nao`:

“‘bash cd ∼/rapp_nao/rapps/rapp_catkin_ws/src catkin_create_pkg tld_tracker_nao rospy geometry_msgs sensor_msgs std_msgs mkdir -p tld_tracker_nao/scripts touch tld_tracker_nao/scripts/tracker.py chmod +x tld_tracker_nao/scripts/tracker.py “‘

The contents of `tracker.py` follow:

“‘python #!/usr/bin/env python

Authors:

Chrisa Gouniotou (https://github.com/chrisagou)

Aspa Karanasiou (https://github.com/aspa1)

Manos Tsardoulias (https://github.com/etsardou)

Import the RAPP Robot API

from rapp_robot_api import RappRobot

from geometry_msgs.msg import Polygon from geometry_msgs.msg import Twist

from naoqi_bridge_msgs.msg import JointAnglesWithSpeed

import rospy import sys import time

class NaoTldTracker:

```
def __init__(self):
    self.rh = RappRobot()

    # Use naoqi_brdge to move the head
    self.joint_pub = rospy.Publisher('/joint_angles', JointAnglesWithSpeed, queue_size=1)

    # NAO stands up
    self.rh.motion.enableMotors()
    self.rh.humanoid_motion.goToPosture("Stand", 0.7)

    self.lost_object_counter = 20
    self.lock_motion = False
    self.hunt_initiated = False

    # These are the NAO body velocities. They are automatically published
    # in the self.set_velocities_callback
    self.x_vel = 0
    self.y_vel = 0
    self.theta_vel = 0

    # Subscription to the TLD tracking alerts
    self.predator_sub = rospy.Subscriber("/vision/predator_alert", \
            Polygon, self.track_bounding_box)

    # Timer callback to check if tracking has been lost
    rospy.Timer(rospy.Duration(0.1), self.lost_object_callback)
    # Callback to set the velocities periodically
    rospy.Timer(rospy.Duration(0.1), self.set_velocities_callback)

# Callback of the TLD tracker. Called when the object has been detected
def track_bounding_box(self, polygon):
    self.hunt_initiated = True

    # Set the timeout counter to 2 seconds
    self.lost_object_counter = 20

    # Velocities message initialization
    joint = JointAnglesWithSpeed()
    joint.joint_names.append("HeadYaw")
    joint.joint_names.append("HeadPitch")
    joint.speed = 0.1
    joint.relative = True

    # Get center of detected object and calculate the head turns
    target_x = polygon.points[0].x + 0.5 * polygon.points[1].x
    target_y = polygon.points[0].y + 0.5 * polygon.points[1].y
```

```
    sub_x = target_x - 320 / 2.0
    sub_y = target_y - 240 / 2.0

    var_x = (sub_x / 160.0)
    var_y = (sub_y / 120.0)

    joint.joint_angles.append(-var_x * 0.05)
    joint.joint_angles.append(var_y * 0.05)

    ans = self.rh.humanoid_motion.getJointAngles(['HeadYaw', 'HeadPitch'])
    head_pitch = ans['angles'][1]
    head_yaw = ans['angles'][0]

    # Get the sonar measurements
    sonars = self.rh.sensors.getSonarsMeasurements()

    # Check if NAO is close to an obstacle
    if sonars['sonars']['front_left'] <= 0.3 or sonars['sonars']['front_right'] <= 0.3:
        self.lock_motion = True
        rospy.loginfo("Locked due to sonars")
    # Check if NAOs head looks way too down or up
    elif head_pitch >= 0.4 or head_pitch <= -0.4:
        self.lock_motion = True
        rospy.loginfo("Locked due to head pitch")
    # Else approach the object
    elif self.lock_motion is False:
        self.theta_vel = head_yaw * 0.1
        if -0.2 < head_yaw < 0.2:
            print "Approaching"
            self.x_vel = 0.5
            self.y_vel = 0.0
        else:
            self.x_vel = 0.0
            self.y_vel = 0.0
            print "Centering"
        self.joint_pub.publish(joint)

    # Check the battery levels
    batt = self.rh.sensors.getBatteryLevels()
    battery = batt['levels'][0]
    if battery < 25:
        self.rh.audio.setVolume(100)
        self.rh.audio.speak("My battery is low")
        self.predator_sub.unregister()
        self.rh.humanoid_motion.goToPosture("Sit", 0.7)
        self.rh.motion.disableMotors()
        sys.exit(1)

# Callback invoked every 0.1 seconds to check for lost of object tracking
def lost_object_callback(self, event):
    # Continues only after the user has selected an object
    if self.hunt_initiated:
        self.lost_object_counter -= 1
        # If 2 seconds have passed without tracking activity the robot stops
        if self.lost_object_counter < 0:
            self.lock_motion = True
            self.x_vel = 0.0
            self.y_vel = 0.0
            self.theta_vel = 0.0
            rospy.loginfo("Locked due to 2 seconds of non-tracking")

            self.predator_sub.unregister()

# Callback to periodically (0.1 sec) set velocities, except from the
# case where the robot has locked its motion
def set_velocities_callback(self, event):
    if not self.lock_motion:
        self.rh.motion.moveByVelocity(self.x_vel, self.y_vel, \
                self.theta_vel)
    else:
        self.rh.motion.moveByVelocity(0, 0, 0)
```

The main function

if **name** == "__main__": rospy.init_node('nao_tld_tracker', anonymous = True) nao = NaoTldTracker() rospy.spin() “‘

The next step is to create a ROS launcher to deploy the node:

“‘bash cd ∼/rapp_nao/rapps/rapp_catkin_ws/src/tld_tracker_nao/ mkdir launch && cd launch touch tld_tracker_nao.launch “‘

The `tld_tracker_nao.launch` must contain:

“‘xml <launch>

<arg name="nao_ip" value="$(arg nao_ip)">

<node name="nao_tld_tracker_node" type="tracker.py" pkg="tld_tracker_nao" output="screen"> </launch> “‘

Furthermore, we need another launch file called `nao_bringup.launch`. First we create it:

“‘bash cd ∼/rapp_nao/rapps/rapp_catkin_ws/src/tld_tracker_nao/launch touch nao_bringup.launch “‘

And then we fill it with:

“‘xml <launch>

<arg name="nao_ip"> <arg name="nao_port" default="$(optenv NAO_PORT 9559)">

<node pkg="diagnostic_aggregator" type="aggregator_node" name="diag_agg" clear_params="true"> <rosparam command="load" file="$(find nao_bringup)/config/nao_analysers.yaml"> </node>

<arg name="version" value="V40">

<arg name="nao_ip" value="$(arg nao_ip)">

<arg name="nao_ip" value="$(arg nao_ip)">

<arg name="nao_ip" value="$(arg nao_ip)"> <arg name="nao_port" value="$(arg nao_port)">

<arg name="nao_ip" value="$(arg nao_ip)"> <arg name="resolution" value="1"> <arg name="nao_ip" value="$(arg nao_ip)"> <arg name="source" value="1">

<arg name="nao_ip" value="$(arg nao_ip)"> <arg name="memory_key" value="Device/SubDeviceList/US/Left/Sensor/Value"> <arg name="frame_id" value="LSonar_frame"> <arg name="nao_ip" value="$(arg nao_ip)"> <arg name="memory_key" value="Device/SubDeviceList/US/Right/Sensor/Value"> <arg name="frame_id" value="RSonar_frame">

<arg name="nao_ip" value="$(arg nao_ip)">

</launch> “‘

Finally, the application can be launched by:

“‘bash roslaunch tld_tracker_nao tld_tracker_nao.launch “‘

If everything is correct, a TLD window must be raised, showing live the stream from the NAO front camera. Click on this window, press `r` and select a rectangle with your mouse for the NAO to track. Have in mind that the object-to-track must have sufficient features for the TLD to succeed (e.g. a white wall won't work).

Then, NAO should center the object with its head and start moving towards there. There are four reasons for NAO to stop:

- To loose tracking for 2 seconds
- To have a sonar measurement under 30 cm
- For the head pitch angle to be larger than 0.4 rad or smaller than -0.4 rad (the object is on the floor or very high)
- The battery level to be under 25%

Below you can see an example where NAO tracks an orange!

Chapter 53

RAPP Improvement Center (RIC) related troubleshooting

Chapter 54

RAPP Platform Web Server (RIC) related troubleshooting

Chapter 55

?-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 RAPP Web 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 web 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.