Funded by the 7th Framework Programme
of the European Union

Project Acronym:	RAPP
Project Full Title:	Robotic Applications for Delivering Smart User Empowering Applications
Call Identifier:	FP7-ICT-2013-10
Grant Agreement:	610947
Funding Scheme:	Collaborative Project
Project Duration:	36 months
Starting Date:	01/12/2013

D3.5.2 Final RAPP API

Deliverable status:	Final
File Name:	RAPP_D3.5.2_V1.0_30112015.pdf
Due Date:	30 Nov, 2015
Submission Date:	30 Nov, 2015
Dissemination Level:	Public
Task Leader:	5 - Ortelio
Author:	Alexandros Gkiokas

The RAPP project consortium is composed of:

CERTH	Centre for Research and Technology Hellas	Greece
INRIA	Institut National de Recherche en Informatique et en Automatique	France
WUT	Politechnika Warszawska	Poland
SO	Sigma Orionis SA	France
Ortelio	Ortelio LTD	United Kingdom
ORMYLIA	Idryma Ormylia	Greece
MATIA	Fundacion Instituto Gerontologico Matia - Ingema	Spain
AUTH	Aristotle University of Thessaloniki	Greece

Project Abstract

The RAPP project will provide an open-source software platform to support the creation and delivery of Robotic Applications (RApps), which, in turn, are expected to increase the versatility and utility of robots. These applications will enable robots to provide physical assistance to people at risk of exclusion, especially the elderly, to function as a companion or to adopt the role of a friendly tutor for people who want to partake in the electronic feast but don't know where to start.

The RAPP partnership counts on seven partners in five European countries (Greece, France, United Kingdom, Spain and Poland), including research institutes, universities, industries and SMEs, all pioneers in the fields of Assistive Robotics, Machine Learning and Data Analysis, Motion Planning and Image Recognition, Software Development and Integration, and Excluded People. RAPP partners are committed to identify the best ways to train and adapt robots to serve and assist people with special needs.

To achieve these goals, over three years, the RAPP project will implement the following actions:

- Provide an infrastructure for developers of robotic applications, so they can easily build and include machine learning and personalization techniques to their applications.
- Create a repository, from which robots can download Robotic Applications (RApps) and upload useful monitoring information.
- Develop a methodology for knowledge representation and reasoning in robotics and automation, which will allow unambiguous knowledge transfer and reuse among groups of humans, robots, and other artificial systems.
- Create RApps based on adaptation to individuals and taking into account the special needs of elderly people, while respecting their autonomy and privacy.
- Validate this approach by deploying appropriate pilot cases to demonstrate the use of robots for health and motion monitoring, and for assisting technologically illiterate people or people with mild memory loss.

The RAPP project will help to enable and promote the adoption of small home robots and service robots as companions to our lives. RAPP partners are committed to identify the best ways to train and adapt robots to serve and assist people with special needs. Eventually, our aspired success will be to open and widen a new 'inclusion market' segment in Europe.

Table of Contents

List of Abbreviations

Abbreviation	Definition
API	**Application Programming Interface**
ISO	**International Organization for Standardization**
ROS	**Robot Operating System**

Executive summary

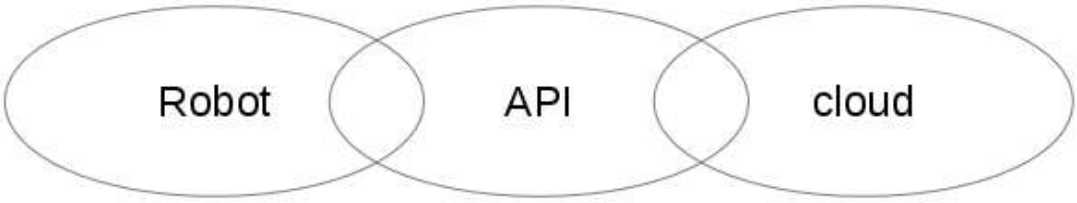
The present document is a deliverable of the RAPP project, funded by the European Commission's Directorate-General for Communications Networks, Content & Technology (DG CONNECT), under its 7th EU Framework Programme for Research and Technological Development (FP7).

Deliverable named "D3.5.2 Final RAPP API" is a prototype. The code of the RAPP API is available here: https://github.com/rapp-project/rapp-api. This document is a brief description of the RAPP API prototype used by developers, in order to enable abstract high-level development and programming of Robot applications.

Introduction

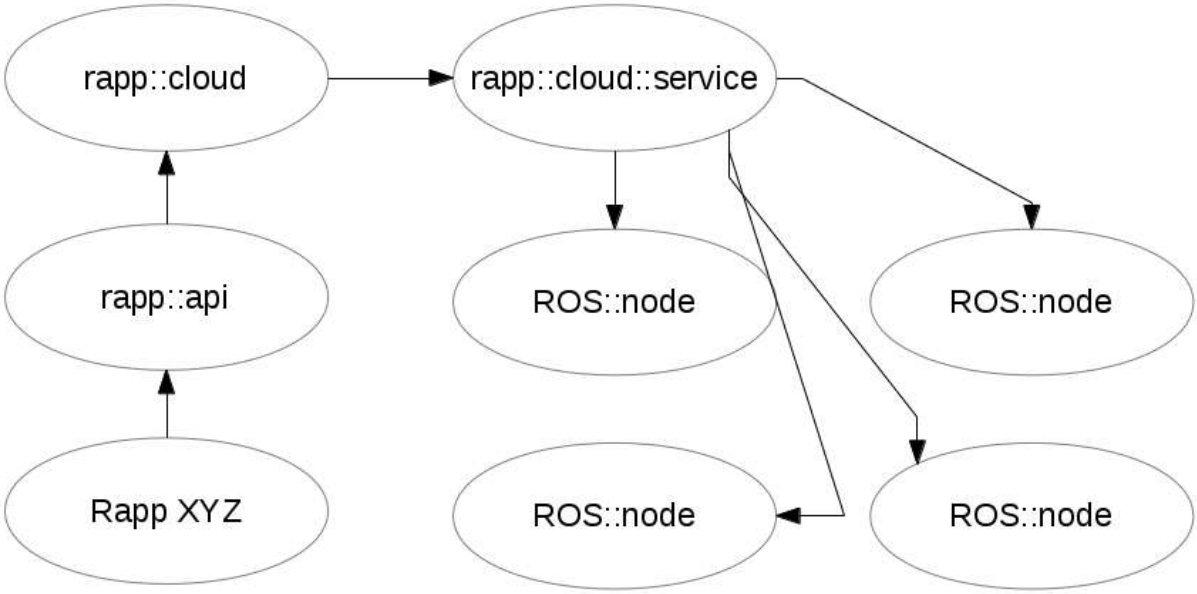
The following document is a brief description of the API used by developers, in order to enable abstract high-level development and programming of Robot applications. The languages in which the API will be implemented are: C++11 (the ISO approved standard - August 2011) and JavaScript. Future API may be implemented in Python, but at the moment of writing this, the norm in robotics development is C++ and ROS, and thus, we focus on the unofficial standards.

Each side of the RAPP platform, is represented either by the appropriate namespace (in C++) or object class (in JavaScript). For ease of use, we describe the C++ namespaces, which are directly translatable to the JavaScript equivalents.



1 Cloud Services

Services on the cloud run by using the *Robot Operating System* (ROS) and other frameworks (e.g., OpenCV or OpenNI). Services have a single ROS Node, serving as the request handler (via TCP/UDP) which then serves each request to the corresponding ROS node (e.g., face detection, database handling, etc.).



1.1 Available Services

rapp::cloud::available_services( )

This class requests from the cloud a list of available services. The response is simply a list of of words.

list<string>

1.2 Face Detection

rapp::cloud::faceDetector (rapp::object::image, callback)

This class is constructed using an image as a parameter, and a callback function. Since this is an asynchronous operation, there is no guarantee of when the cloud service will respond, therefore, it may take place in a thread or some other parallel mechanism. The image is sent to the cloud platform, and it is processed by ROS and OpenCV, in order to determine faces. The callback receives as a parameter an array/vector of faces detected:

list<rapp::object::face>

Which are then available as rectangle coordinates, with an optional text label.

1.3 Fetch Personal Data

rapp::cloud::fetchPersonalData (string, callback)

This class is constructed using a user name, and a callback function. Since this is an asynchronous operation, there is no guarantee of when the cloud service will respond. The cloud platform queries a user database, and then retrieves results, which are serialised using the JSON format.

The callback receives as a parameter an array/vector of strings which represent the the columns from the database row of the user:

list<string>

1.4 QR Detection

rapp::cloud::qrDetector (rapp::object::image, callback)

This class is constructed the same way as the face detector. An image is passed by value, together with a callback. It will run asynchronously, and once completed, a response from the cloud will indicate a QR object, as a URL, ID, Tag, or other similar encoded information.

1.5 Set De-noise Profile

rapp::cloud::setDenoiseProfile(rapp::object::audio, audio_source, user)

This class handler sets the de-noising profile for a specific user, by capturing an audio ambient noise. The parameters specify for which user what audio should be used. There is no return type or reply from the cloud service.

1.6 Speech to text (CMU Sphinx)

rapp::cloud::speechToText (rapp::object::audio, callback, audio_source, grammar, words, sentences, user)

This class is constructed with a WAV or OGG file which has already been recorded. The function requires use of other parameters, such as *user, audio source, grammar, sentences, words.* Those parameters relate to the service call functionality: *audio source* is the type of audio encoding, *grammar* is a JSGF-form grammar used by CMU sphinx, *sentences* are full sentences matched by the speech recognition process, and *words* are tokens which will be looked up.

Those are asynchronous operations, and therefore the exists no guarantee of when the robot will receive a response. The callback of the first constructor type which uses a WAV file, once executed, will receive as a parameter, an array/vector of strings.

list<string>

The response is JSON serialised, and once parsed, it indicates the words detected, along with a respective confidence level (probability).

The callback of the second constructor type which uses an audio stream, will be executed numerous times, depending on how many words have been detected. Each time a word is detected, the callback is invoked, and receives a list of recognised strings.

It is the developer's responsibility to anticipate and handle the asynchronous nature of such processing. Furthermore, the developer is required to have a basic understanding of how speech recognition works, and in particular how CMU sphinx functions.

1.7 Speech Recognition (Google)

rapp::cloud::speechToText(rapp::object::audio, audio_source, user, language)

Similar to how CMU-Sphinx-based speech recognition works, this call is driven by Google speech, and is more simple. It uses an Open dictionary, and as such it relies only on providing an audio file, the type of audio encoding, the user token and the language. The response JSON when parsed provides a vector of recognised words.

list<string>

1.8 Text to Speech

rapp::cloud::textToSpeech(text, language)

This class handlers generation of audio, for the specified *text* parameter, using the *language* parameter. The cloud serves the appropriate audio file, which is the return type:

rapp::object:wav

1.9 Ontology Subclass of

rapp::cloud::ontologySubclassOf (query, callback)

This class handles ontology subclass detection, and is constructed by querying the cloud Ontology database. The operation is asynchronous, and thus the callback will executed once a response is received by the cloud platform. The callback receives as parameter, an array/vector of strings.

list<string>

Each string, represents a subclass of the original query.

1.10 Ontology Superclass Of

rapp::cloud::ontologySuperclassOf (query, callback)

This class handle ontology super class detection, and is constructed by querying the cloud Ontology database. The operation is asynchronous, and thus the callback will executed once a response is received by the cloud platform. The callback receives as parameter, an array/vector of strings.

list<string>

1.11 Ontology is SubSuperclass Of

rapp::cloud::ontologyIsSubSuperclassOf (query, callback)

This class handles relation between a subclass and a super-class, and is constructed by querying the cloud Ontology database. The operation is asynchronous, and thus the callback will executed once a response is received by the cloud platform. The callback receives as return, a Boolean value.

list<string>

1.12 *Cognitive Test Chooser*

rapp::cloud::cognitiveTestChoose(user, type)

This class handler allows the developer to chose a basic *cognitive test* for a specific user. Parameter *type* can be: *Arithmetic, Awareness,* or *Reasoning.* The return type is a JSON string, *which the developer must parse and handle appropriately.* The JSON contains: questions, possible answers, correct answers, the test instance and test type, test subtypes and any encountered errors.

1.13 Record Cognitive Test Performance

rapp::cloud::recordCognitiveTestPerformance(user, test_instance, score)

This class handles storing the performance of a user's cognitive test scores. It is used so that the progression of a user's cognitive abilities may be monitored over time. The parameter *score* will be stored, and will be acknowledged by being the *return value* of the call.

score

1.14 Object recognition

rapp::cloud::objectDetect (rapp::object::image, callback)

This class is constructed by using an image, and a callback copied by value. The image parameter is sent to the cloud platform, which will process it, and try to detect known objects. The entire process takes place asynchronously, and there exists no guarantee on when the operation will complete. The callback is executed upon receiving a reply from the cloud platform. The callback receives as parameter, an array/vector of detected objects models.

list<rapp::object::model2D>

The list contains each object model found, together with its coordinates in the 2D plane, or an optional 3D mesh, attributes, labels, and other relational characteristics detected. The reply is serialised as JSON from the cloud platform, and parsed on the robot.

2 Robot API

The API for the RAPP platform also includes certain generic and abstract services, which provide a *high-level* interface to *low-level robot-specific* implementations. This is accomplished by interface abstraction, and is described by the Abstract Base Class (ABC) interface, in combination with the *Factory Method-Pattern*, where the interface is defined by generality, but the sub-classing defines instantiations and implementations. Alternatively, it can also be further implemented using the *private implementation* (pimpl) idiom.

By doing so, we can enable multi-robot and cross-platform, for all robots that support C++, JavaScript or Python. The Robot API deals with three distinct categories: *Communication, Navigation* and *Vision.* We separate those functions from the cloud, albeit they *may* rely on the cloud API. The described interface below is an abstract interface (enforced by the *ABC* in C++) which can be implemented different for each robot implementation library.

2.1 Communication

rapp::robot::communication (argc, argv [])

Create a communication module, using the typical (optional) unix arguments. This is a ABC that enables auditory communication of the robot with the user.

2.1.1 Play Audio

rapp::robot::communication::playAudio(file, position, volume, balance, loop)

This class method will make a robot play an audio file. The parameters are self-explanatory: the robot will play an audio file (note, not a rapp::object::audio) from given position, at given volume and balance, and if *loop* is set to true, it will keep repeating the same audio file. Return value is Boolean.

2.1.2 Text to Speech

rapp::robot::communication::textToSpeech(string, language)

Same as the cloud method, this communication class method will make the robot's speakers create speech from the parameter *string* for given *language.* Return type is *Boolean*.

2.1.3 Word Spotting

rapp::robot::communication::wordSpotting(dictionary[], size)

This communication method will try to spot a word in a dictionary, and return the actual *string* recognised.

2.1.4 Capture Audio

rapp::robot::communication::captureAudio(time)

This communication method will capture audio from the robot's microphones, and will return a *string* designating the file and path location of where the recorded audio file is.

2.1.5 Capture Audio (overloaded)

rapp::robot::communication::captureAudio(file, time, energy)

An overloaded version of the previous method, will capture audio designated by the *file* parameter, for given *time* using specified *microphone energy.*

2.2 Navigation

rapp::robot::navigation (argc, argv [])

The constructor for a navigation module.

2.2.1 Move to

rapp::robot::navigation::moveTo(x, y, theta)

Make robot move to position at (*x,y*) with orientation *theta*.

2.2.2 Move Velocity

rapp::robot::navigation::moveVel(x, y, theta)

Make robot move with linear velocity (*x, y*) and angular velocity *theta.*

2.2.3 Move Stop

rapp::robot::navigation:moveStop()

Make robot to stop moving.

2.2.4 Move Joint

rapp::robot::navigation::moveJoint(joint, angle)

Make the robot's *joint* move to specified *angle.*

2.2.5 Take Predefined Posture

rapp::robot::navigation::takePredifinedPosture(pose)

Make robot take a predefined *pose.* Those poses are respective to robot implementation of the interface and the actual physical robot and model.

2.2.6 Look at Point

rapp::robot::navigation::lookAtPoint(x, y, z)

Make the robot's head move so it looks at a given point *(x, y, z)* in reference to the start position.

2.2.7 GetTransform

cv::Mat rapp::robot::Navigation::getTransform (std::string chainName, int space)

Return transposition matrix of frame *chainName* with respect to frame *space*.

2.2.8 Rest

bool rapp::robot::Navigation::rest ()

Make robot sit and disable motor stiffness. Return True on function success request.

2.2.9 MoveAlongPath

bool rapp::robot::Navigation::moveAlongPath(rapp::objects::Path path)

Make robot move along the specified path. Return True on function success request.

2.2.10 SetGlobalPose

bool rapp::robot::Navigation::setGlobalPose(rapp::objects::Pose pose)

Set the robot position with respect to the world frame and return True on function success request.

2.2.11 GetRobotPose

rapp::objects::Pose rapp::robot::Navigation::getRobotPose()

Return robot position with respect to the world frame.

2.3 Vision

rapp::robot::vision ( )

The constructor for a vision module. Implementation depends upon robot type and model.

2.3.1 Capture Image

rapp::robot::vision::captureImage(camera, resolution, encoding)

Capture an image using *camera*, at specified *resolution* with *encoding.* Return type is a *rapp::object::image*

2.3.2 Set Camera Parameters

rapp::robot::vision::setCameraParam(camera_id, parameter_id, new_value)

Change camera parameter. Method parameters specify which camera should be updated, the parameter to be updated and the new value. Return type is *Boolean.*

2.3.3 Set Multiple Camera Parameters

rapp::robot::vision::setCameraParams(camera_id, params[int, int])

Change multiple camera parameters. Method parameters specify which camera should be updated, whereas *params* list, is in fact a map of *parameter_id* and *new_value.* Return type is a map of *param_id* and *Boolean* values for each successfully updated parameter.

2.3.4 Face Detect

rapp::robot::vision::faceDetect(image, camera, resolution)

Detect faces in the parameter *image,* using *camera* at specified *resolution.* Return type is a list of *faces.*

list< rapp::object::face >

2.3.5 QR Detect

rapp::robot::vision::qrCodeDetection(image, robotToCameraMatrix, camera_matrix, landmarkTheoreticalSize)

Detect QR codes real-time on the robot. Parameter *image* will be scanned, and parameters *camera_matrix* and *robotToCameraMatrix* will be used to detect changes in rotation, pitch and angle. Parameter *landmarkTheoreticalSize* is a hint to the magnification of landmarks to a theoretical size. Return type is 3D QR code:

rapp::object::QRcode3D

3 Job Scheduling and Service Calling

It is important to explain the nature of asynchronous (non-blocking) versus synchronous (blocking) calling. All *cloud-based service calls* are asynchronous (C++,JavaScript and Python). The developer is given the ability to control the execution of the service calls, called *jobs.* Either a single *job* can be executed asynchronously, and once finished, the result is acquired by a callback, with a signature unique and relevant to the class handler being used, or the developer may opt to use a batch of *jobs*, organised in a group.

In both cases, the underlying mechanisms of the C++ API all depend on the *Boost framework,* which essentially gave rise to the new C++11 standard. The usage of *boost::asio* enables the seamless and cross-platform development under different architectures (amd86, i386, arm, etc) and essentially provides a high-level interface to the user who does not need to be concerned with details, and low level implementations. This philosophy adheres to *Apple's NS objective-c principles,* which dictate that the developers should have to deal only with high level objects, in order to be able to create complex, scalable and robust software.

Other languages such as JavaScript already have the *asynchronous execution is* enabled by default, and the Python implementation takes the same approach. Under JavaScript, the execution is based on AJAX calls, and their callbacks.

4 RAPP Object Classes and Types

As already mentioned, in order to enable and facilitate *Object-Oriented Programming* rather than procedural or functional programming, in both C++ and JavaScript, we encapsulate certain objects in classes, whilst relying on existing types found in both languages. This form of development should enable fast and easy usage of the API, and thus provide additional incentives to users of the RAPP platform.

Object Class
rapp::object::image
rapp::object::face
rapp::object::wav
rapp::object::audio
rapp::object::ogg
rapp::object::headset
rapp::object::qrCode
rapp::object::qrCode3D
rapp::object::model2D

The following table shows the current progress in terms of cloud service handlers for each programming language.

Cloud Service	C++	JavaScript	Python
face detection	✓	✓	✓
qr detection	✓	✓	✓
text to speech	✗	✗	✓
de-noise profile	✓	✓	✓
speech to text (CMU sphinx)	✓	✓	✓
speech to text (Google)	✗	✗	✓
object recognition	✓	✗	✓
available services	✓	✗	✓
ontology sub-classes of	✓	✓	✓

ontology super-classes of	✓	✓	✓
ontology is sub-super-class of	✓	✓	✓
cognitive test chooser	✗	✗	✓
record cognitive test performance	✗	✗	✓

Upcoming version of the RAPP API, will support all service calls shown in the table above in all programming languages.

5 API Unit Testing

The C++ API uses the *boost::test::unit_test* framework in order to ensure that all classes, objects and constructors function as intended. These tests do not examine service call functionality, but only that no undefined behaviour occurs. Similarly, the JavaScript API uses both object (unit) tests as well as cloud (service) tests.