FP7-ICT-2013-10

RAPP System Architecture

Internal Document – Technical Report

Authors: Alexandros Gkiokas, Fotis E. Psomopoulos, Manos Tsardoulias

Revision: 4

Date of preparation: 08/02/2014

Period covering: 01/12/2013 – 17/02/2014

Start date of project: 01/12/2013

Duration: 36 months

Project coordinator:

Prof. Pericles A. Mitkas

Centre for Research and Technology Hellas

Informatics Technologies Institute

Institution/Company	**Reviewer**	**Date**	**Version**
CERTH/ITI	Manos Tsardoulias	20/12/2013	0.95
Ortelio Ltd	Alexandros Gkiokas	21/12/2013	1.0
Ortelio Ltd	Alexandros Gkiokas	02/02/2014	2.0
Ortelio Ltd	Alexandros Gkiokas	08/02/2014	3.0

Table of Contents

Illustration Index

Introduction

We're merging the technical reports of CERTH/ITI and Ortelio Ltd, as we appear to be in total accordance with respect to the overall RAPP system architecture.

I am also appending a few key notes, relevant to WUT comments, and a few further explanations with respect to earlier document (02/02/2014 Technical Report) which may have been confusing or misunderstood.

Our main goal is to enable a cloud-based approach for RAPP and robotics, not because its an upcoming trend, or because the technology is now ripe for such an architecture, but because we believe it will provide a very interesting alternative to conventional robotic controllers, and can expand the RAPP abilities in a seamless and scaling manner.

This document has been rewritten, starting from an abstract and general introduction, and progresses to more technical details later on.

Architecture

We're considering a segmented architecture, where the **store** is on the cloud, and the **client** is on the robot. In that sense, the **host** is the controller executing on the robot, and the **guest** is the controller executing on the cloud. We consider both **controllers** as a singular entity, separated only by a network socket. Together, they make up the **RAPP**.

In Addition to the **RAPP**, we consider the **Improvement Centre**, also residing in the cloud, which can perform a variety of ML or AI technologies, either when requested by a RAPP, or a standalone process.

During the kick-off meeting in Thessaloniki, Professor Cezary Zieliński described a robotic controller as a finite-state machine, thereby setting the theoretical principle upon which the aforementioned design is based upon. In that sense, The entire complex of RAPP client (the main controller on the robot), the RAPP (the application running on both client and store), as well as the Improvement Centre, are part of that FSM.

We make the distinction and differentiation for practicality, as well as because the actual computational abilities of the proposed robotic platforms do not allow for intense algorithms, nor do they have vast amounts of memory.

Whereas how this can be implemented is a different discussion altogether, what follows is a brief and general description of how we envision such a system, and how we would prefer to develop it and implement it.

Entities

We briefly describe the entities to be found in the RAPP System.

Store

The store as an entity, is simply the complex of processes and data residing on the cloud. Whereas the actual cloud may be a single server, a cluster of servers, or a virtual machine on a commercial cloud, we shall consider **our part** of the RAPP cloud, as the store.

Application

The robotics application platform application, (abbreviated RAPP, albeit that may be confusing) is simple an instance of an application developed by anyone who wishes to develop a RAPP, and distributed by the RAPP platform, after submission. This application is of a distributed nature, it shall always execute on the robot (host) but may as well execute on the cloud (guest). In Zieliński's terms, this is a *behaviour*, or a s*et of behaviours.*

Improvement Centre

This is a RAPP independent module (or complex of modules) which may operate independently, or upon request from a RAPP. It is thus not a part of a RAPP, but may be accessed by the RAPP API. In realistic scenarios, we expect the RIC, to be a collection of machine learning or A.I. processes, accessed via the API, using socket calls.

Users

We briefly describe the users that may control or communicate with a RAPP (and not necessarily with a physical robot).

Developer

This user simply develops a RAPP. That user submits the source code, which is then transformed into a RAPP.

Robot User

This is the actual physical robot user. In most cases, we assume him or her to be the test subject of the inclusion experiments, although it could in theory be any robot owner, user or tester. This user is described by the actual physical interaction with the robot, and thus the RAPP.

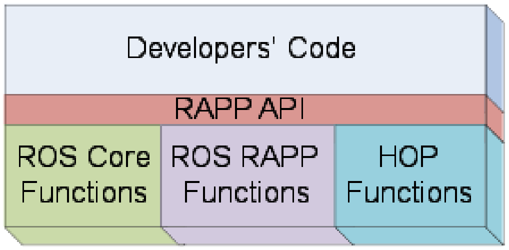
Remote User

This is a user who may monitor the interaction of the RAPP with its physical user remotely, either in real time or later on.

API

What bridges all the entities and users, is in fact the RAPP API, which will be developed as part of this project. Whereas specific technical specifications need to be discussed and resolved, the overall idea is that the API will enable the cloud-nature of the RAPP platform, provide developers a way to develop such cloud-robotic controllers.

The design of RAPP API is the integrating part of the whole design. It will allow for the cross-robot development of binaries, and will provide a high-level abstraction of the base functions via a RAPP wrapper.



Drawing 1: RAPP API

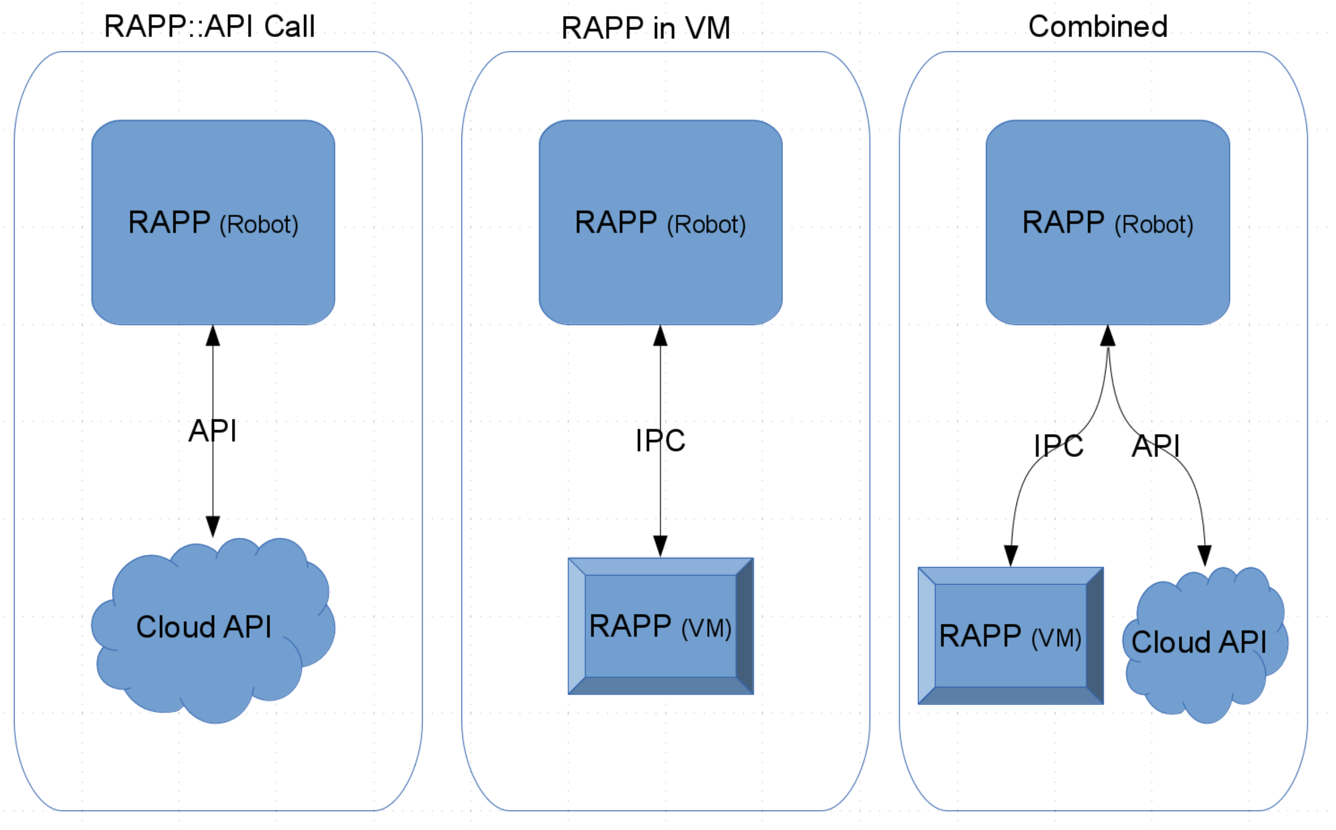
The API *should* provide the functionality of ROS, of HOP and of any RAPP developed functions. Such functions include, but are not limited by, machine learning algorithms, statistical procedures, data analysis etc. In future versions of the RAPP API, these functions will be updated to allow for higher complexity modules. The developer should be aware which parts run on the robot, and which on the cloud, but should not have to dig deep into lower level development of how they intercommunicate, interact or synchronize. We believe the developer should preferably create one application, not two, unless he or she wishes to.
The benefit of distributed robotic controllers can only be fully achieved, if we enable the creation of distributed applications in an easy and convenient manner, so as to attract developers from across disciplines, similar to how smart-phone development rose through the *app* paradigm.
This implies that the API will hide quite a lot from the developers. Maybe even enforce or impose certain restrictions. We are willing to create a closed ecosystem for the development of such applications, if and only if, the gain is greater than those imposed restrictions.
If for example, the developer can quickly create an application for a robot, that uses object recognition, voice recognition, motion planning, navigation, and other complicated and computationally intensive tasks, all thanks to our API, then it is worth imposing those restrictions on certain programming practices.
What this means, is that we do not want the developer to have to deal with the lower-level specifics of such a system. Just as when nowadays developing multi-threaded applications, we (as developers) don't want to be burdened with the specifics of low level threads, similarly, we would want the developer using our API, to simply use high level abstracted function and classes.
However, there exists a big problem with this approach. Contrary to classic distributed applications, the executing platforms are not identical, nor do we want the executing code to be identical. On the opposite, we want the generation of two different and distinct binaries, one for the robot and one for the cloud. Certain parts of the API, similarly to OpenMPI [8], will have to be distributed.

But a distinction of the place-holder of current execution must be made: for example object recognition or actuator control should be explicit. Meaning, the application will run object recognition only on the cloud, and actuator control only on the robot.

What all this means is that the developer will know what is executing (or allocated) where. The developer will be aware that two binaries may be generated, and that specific parts of his or her code will run on different platforms. Whilst we do not want to make it complicated enough for the developer to have to write two apps instead of one, certain issues arise.

There are three alternatives to a unified distributed API:

1. Have the developer create two applications, one for the cloud and one for the robot.
2. Create a Robot API which can use a cloud API calls (similar to RoboEarth [Error: Reference source not found])
3. Do both 1 & 2. Enable a robot API, a cloud API, and the possibility to create cloud-based projects in addition to robot applications.



Drawing 2: RAPP API possibilities

In the first scenario, we create a RAPP::Cloud::API. Assuming we can establish the entirety of cloud-based functionality (meaning, create a full list of what we will enable to run on the cloud) we may be able to do the following:

Instead of having the developer generate an application for the cloud, we will provide a list of functions/classes which can perform certain operations on the cloud. The developer can use those via the API calls.

There exists an advantage to this case. The developer simply creates an application for the robot, which can communicate with the cloud. The disadvantage is also clear. If the developer wishes to do something on the cloud, that is not the way to do it.

This could be the case for newcomers and beginners. In this case, the developer simply uses what is available for him on the cloud.

In the second scenario, we believe to be the case that should be enabled only for advanced developers. It will overcomplicate things for no apparent reason, and add extra workload for the developer. This could be the case where the developer wants to write and run an algorithm only on the cloud. In this case, the developer is responsible for the generation of the cloud RAPP, and the differentiation of it from the robot project. This RAPP will run within a VM that is provided by the RAPP store. The actual VM may reside in the RAPP cloud.

In this sense, the developer is generating a robot application and a cloud application. He then submits both, and we compile and distribute both.

The third scenario is a combination of both 1 & 2. The developer has to develop a RAPP, and he can use the cloud API. He *may optionally chose* to write a *cloud application* in addition to the above, which must be compilable and executable on the cloud.

Knowledge Transference

An arising request from many partners, has been the issue of transferring and sharing knowledge between RAPPs. We consider the duality of the issue, as the RIC may act as a centralised container of knowledge accessible by many RAPPs, or, a RAPP instance, may respond to knowledge queries by other RAPPs or the RIC itself.

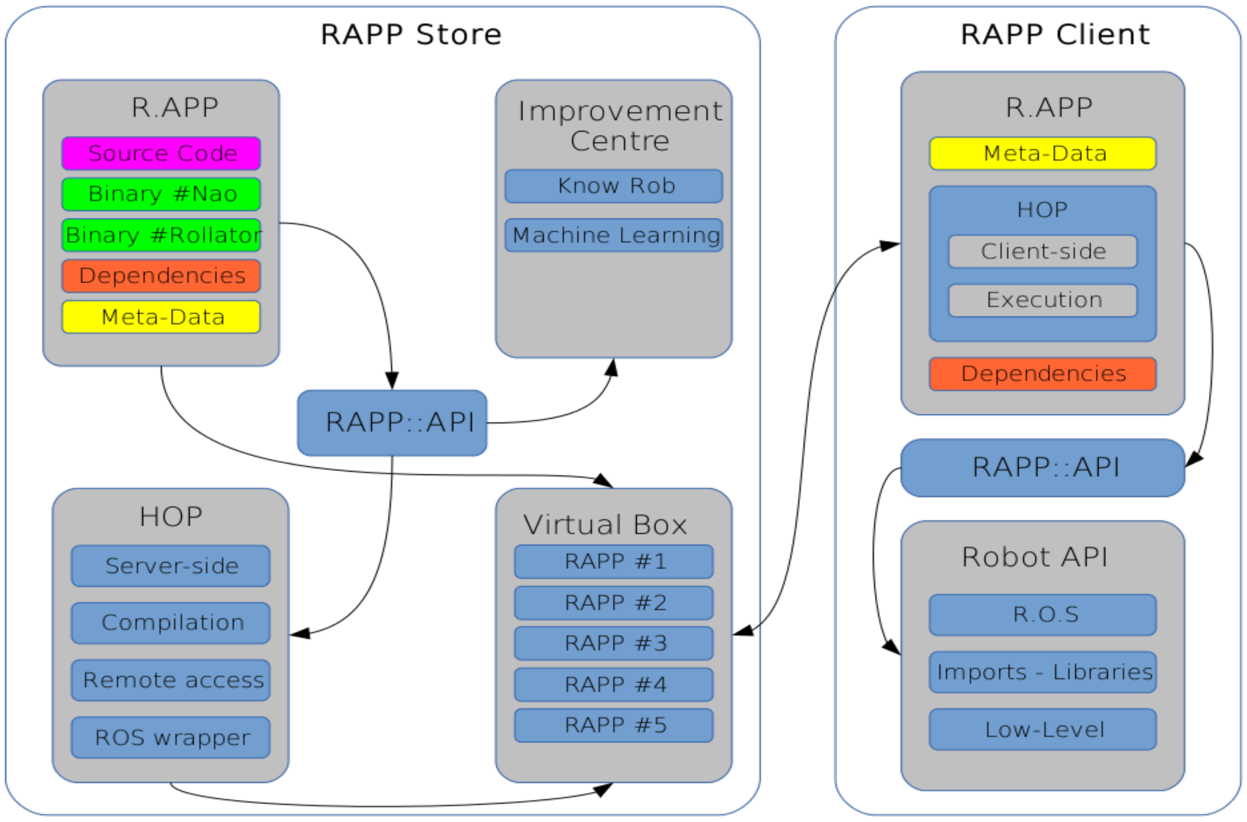
Whereas the first case, would see the RIC acting as the knowledge container, the second case would see a peer-to-peer scenario of knowledge transfer. We **assume** that both cases are implemented through the API as calls or objects.

1. All communication between RAPP platform and the RAPP application will be performed using the RAPP provided API. Under no circumstances we will provide low level access to modules on the platform.
2. The data storage should be designed keeping all ethical and security issues in mind.
3. Virtual Machines (VMs) will be provided by the RAPP platform, both as a means of extending the abilities of the RAPP applications and as independent computational modules
4. RAPP application updates can be either binary updates (i.e. the whole application will be replaced by a newer version) or knowledge updates (i.e. the robot knowledge ontology will be updated)

KR and transference are also discussed later on (RAPP Ontologies).

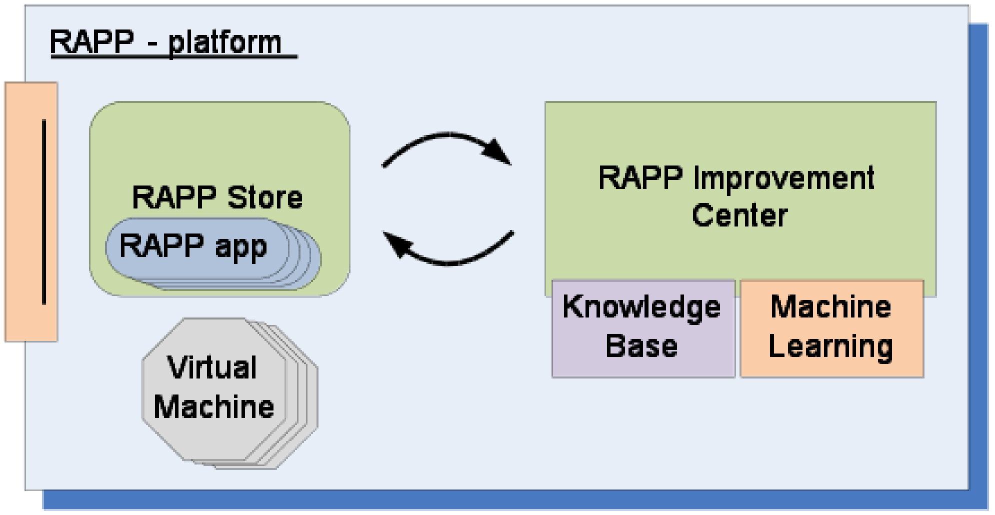
RAPP Platform

Below is the proposed platform outline for both the **store** and **client**.



Drawing 3: RAPP platform

Whereas the actual implementation will be discussed later on, it is important to note two key entities in the above diagram.



Drawing 4 RAPP platform

RAPP Client (Robot)

The client, is assumed to be (programmatically) the master controller, that downloads, installs and executes RAPPs (behaviours). It is also the controller via which human-robot interactions are delegated to the RAPPs.

As such, it needs to be carefully designed, implemented and tested. A concern about autonomy of the robot was raised, with respect to controlling or not controlling the client remotely. In theory we would like this controller to be as autonomous as possible (that is fully autonomous). In practice however, certain operations may have to be discussed upon in the second meeting.

As already mentioned, the *RAPP client* should be the proxy software that enables the installation, distribution, authentication, validation, patching and updating of applications, as well as (and most importantly) the execution or initialisation of a *RAPP.*

Distribution

Distributing *RAPPs* is a task which involves authenticating as a user with the *RAPP store*, obtaining dependencies required for executing a *RAPP*, and ascertaining whether a *RAPP* obtained via the internet, is indeed the same binary file. We consider distribution as a transfer of files, and as such, we do not need to over-engineer the process. A form of local database (e.g. SQLite) could be used to index version of the *RAPP,* all the relevant meta-data, and as such, assist with updates and patches.

Installation

The installation process is somewhat more complicated than the distribution. It is our understanding that installation should be a one-click process. There exist two possible scenarios this could be achieved, each one associated with certain issues.

Remote Installation

In this scenario, the robot user has to obtain the *RAPP* by using a computer, laptop or even smartphone. Some form of synchronisation mechanism will be required by the *RAPP client*, in order to obtain an updated version of this user's *RAPP* list (the *RAPPs* he or she has installed on the robot). The problem with this approach is that the *RAPP client* is responsible with synchronizing via an intermediary device (e.g. the robot has to acquire a *RAPP* dictated by another device). This scenario is in our opinion easier to implement, as the actual process could take place in form of a web-portal, whilst the *RAPP client* is responsible for updating the currently available *RAPPs* on the robot.

A catalogue of the web-portal could display *RAPPs* in a fashion similar to *Apple's APP store*. User interaction with the web-portal should be very simple and easy, so simple that even technical illiterate people would be able to use it.

Local Installation

In this scenario, the *RAPP client* software instance installed on the robot, is the controller which the robot user interacts with in order to obtain new *RAPPs*. In this sense, the robot user interacts directly with the *RAPP client,* using vocal commands.

This approach brings certain difficulties to the task. It implies that the *RAPP client* will be able to understand human commands, alert the user via speech synthesis of the available *RAPPs*, and respond to commands for obtaining a *RAPP*. Whilst this approach is closer to the concept of 1-click installation, and does not require any technical abilities from the robot user, it is at the same time, much harder to develop and implement.

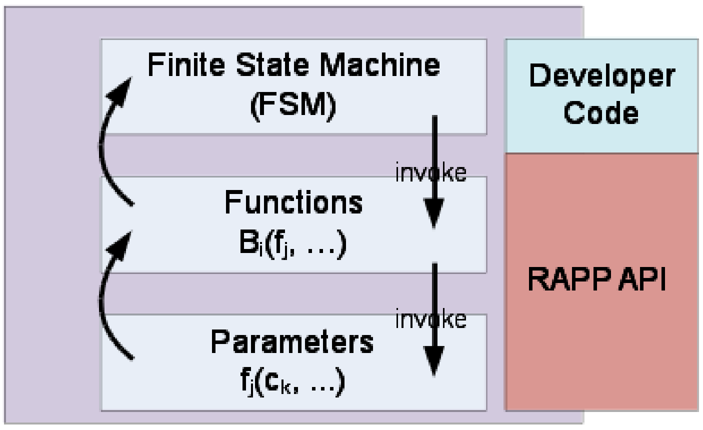
It assumes that the robot will be able to receive voice commands and output sound. The actual voice recognition and speech synthesis process does not have to take place on the robot. Meaning that the *RAPP client* can simply act as a proxy, relaying the installation process to the *RAPP store.*

R.APP (Application)

Each application resides on the robot, and is either installed on demand, or permanently stored on the robot, and invoked by the RAPP client. However, a R.APP, may have a cloud counterpart. This is a module, provided by the RAPP developer, which is to execute on the cloud, and not the robot. Whilst the API provides that functionality, what the developer chooses to do, is his or her own accord. For security and privacy issues, the RAPP executing on the cloud, will be sand-boxed in a virtual machine and possibly encrypted.

The R.APP, may also access the RIC, either locally (e.g. cloud-RAPP calling the RIC) or remotely (e.g. robot-RAPP calling the RIC). All calls are wrapped by the RAPP API, and can be either asynchronous non-blocking network calls, or synchronous blocking network calls. The only limitation in that sense, is the bandwidth capability connecting the two counterparts, and the amount of data transferred.

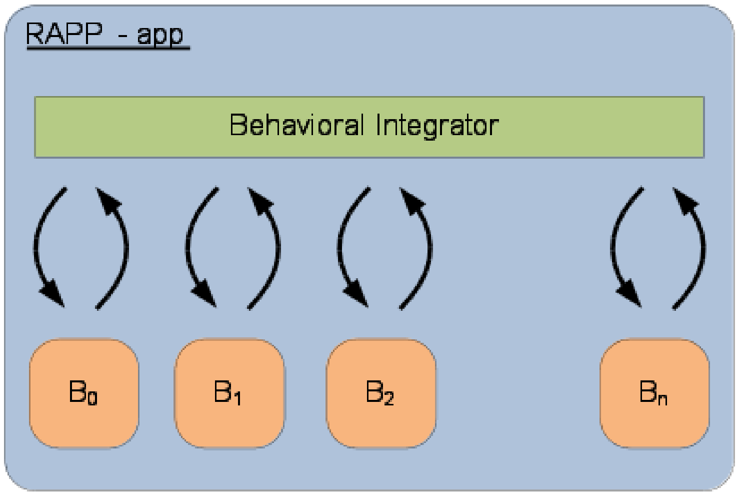
The core of the RAPP project is the RAPP application. It is crucial to define what a RAPP application is going to be, based on the target robot architectures. At this point, the two architectures available are NAO and Smart Rollator.
The RAPP API will provide the functions for the main modules of a robot (i.e. vision, movement, etc), and the Developers’ code will utilize the functions towards the final goal.



Finite State Machine (FSM)
invoke
Functions $B_i(f_j, \ldots)$
invoke
Parameters $f_j(c_k, \ldots)$
Developer Code
RAPP API

Drawing 5: RAPP Application

The Developer shall provide the detailed code for the behavioral integrator. We assume that the developer will be responsible for creating those behaviors/functions as RAPPs. Original design (described above in *Drawing 5*) by M.Tsardoulias depicts the developer code independent of the actual FSM.
The implication here, is that the RAPP created by the developer is a **behavior**, using existing **functions**, which exist in the RAPP::API.
M.Tsardoulias describes this as a behavioral integrator (*Drawing 6*).



Drawing 6: Behavioural Integrator

Source code

Whilst the exact programming or scripting language used for developing applications remains an open issue (currently HOP framework utilises *JavaScript* and *Scheme,* but we would like to see if C++ can also be used), the store is expected to archive and maintain the actual source code submitted by the developer. Doing so can enable security checks, patching, and offline compilation. Furthermore, programming principles could potentially be enforced through analysis of the submitted source code (as opposed to submission of binary files).

The developer may submit changes, updates or patches, which should automatically update the compiled version of the *RAPP* and propagate to robots-clients which have installed the specific *RAPP.*

Binaries

Compilation of source code submitted by a developer, is done by the HOP framework. The HOP framework may perform compilation for the native code run on the server-side, and on-the-fly compilation of the code running on the client. As such, the binary file for each *RAPP* is a tuple of server-code and client-code. The *RAPP store* has to distribute the client-code for specific *RAPP* to each robot which has installed it, validating authenticity and updating or patching when necessary. In the event that a robot has a different CPU architecture or requires a different form of binary, HOP is responsible for compiling it. Generation of binaries (compilation) is performed by the *RAPP store*, upon submission of a new *RAPP* (source code) by the developer, in an automated fashion. Should compilation fail, the *RAPP* will be deemed erroneous, and rejected, whilst notifying the developer of the event.

Dependencies

An open issue that has already been discussed during the kick-off meeting, but needs a consolidated agreement, is *what imports will be allowed* in the *RAPP::API*. The reason for this is that the *RAPP store* will have to ensure that all dependencies with respect to the *RAPP::API* are satisfied by all clients (robots) and for all applications (*RAPPS*). Doing so, will provide a homogeneous execution environment, where dependency issues should not be a problem. Dependency checking and satisfaction could either be performed statically (by having the *RAPP client* install **all** available libraries, imports and headers) or dynamically (by having the *RAPP client* obtain specific required libraries, headers or imports, associated with a specific *RAPP*).

Whereas the actual dependencies may be a lower-level import (static, dynamic or shared library or headers), abstracting accessibility to them through the *RAPP::API* should make their use transparent to the developer. The *RAPP client* should ensure all dependencies are satisfied when installing a new *RAPP* and when executing an existing *RAPP.*

In addition to this, we need to agree on how libraries are loaded from each *RAPP*. Will the *RAPP* installed on a robot be statically linked (and thus not need to transfer dependency libraries to the robot) or will they be dynamically loaded?

Meta-Data

Each *RAPP* should come tied with certain meta-data. That could be (but not limited to), an *SHA512* sum of the binary, a unique *ID* for the *RAPP*, an automatically allocated application version, a developer id, a list of dependencies for given *RAPP*, a user authentication key, and an *x.509* digital certificate used for secure communication and application validation. Should any additional information somehow related to the *RAPP* installed be included, the Meta-data (in form of an XML or JSON array) would be the place to include it.

RAPP Cloud Synchronisation

How do the different executing parts run? If the code dictates that an object is to be recognized, is that execution synchronous? Meaning, does the application on the robot block until object recognition has finished on the cloud?
Or is it asynchronous? Meaning that the application on the robot will continue to run, and will be notified asynchronously, when the object recognition has finished.
Obviously, this adds a new dimension of complexity for the developer, and it is for this exact reason that the API must be simple and elegant, so simple in fact, that the developer can deal with his or her application work-flow and difficulty, and not with the lower level mechanics.
A logic assumption, is that whatever execution takes place on the cloud will be asynchronous, because we simply don't want to have a robot sitting and waiting for something to run on the cloud.
The direct opposite however, is that whatever execution takes place on the robot will most likely be synchronous. We do not want different actions on the robot's actuators for example, but sequential ones, although asynchronous operation should be possible. Similarly, in the exception where the developer decides to wait for the completion of an operation on the cloud, a synchronous operation should also be possible.

The second and more dubious issue with synchronization is memory sharing. For those familiar with CUDA, this is a very tricky issue. For example, what if the developer creates a global array, such that this array exists in both the cloud and robot. Will that array be always the same in both devices? What happens if it changes on the robot, but in the cloud? Perhaps, much similar to CUDA, such a global memory scheme should not be enabled, perhaps the simplest solution would be to create either a *robot* array, or a *cloud* array, and then transfer one from another.

This approach, however, consumes the developer's time, where he or she now has to deal with transferring and synchronizing arrays, instead of developing the application.
Eventually this is unavoidable. A distributed application entails a certain degree of complexity, not found in familiar single device applications, similarly to how a multi-threaded application is more complex than a single threaded one, but with greater benefits.

Coming back to the closed ecosystem proposed earlier, it may be unavoidable to not enforce certain policies. For example, for synchronization, we may eventually have to design of the distributed functionality, and simply impose it on the developers through the API (possibly hidden away as compiled code within a library, and not open in a header for them to change).

The API should be simple, minimal and elegant. If for example, using C++, Java or Python, encapsulate everything in simple, easy to understand and use classes. In particular, if using C++, we could produce a template (header only library) for the parts that need not to be hidden but require abstraction, and a library (shared, dynamic or static) for the parts that need to be hidden but not necessarily abstracted (in the form of a series of interfaces – *Abstract Base Classes*).
Wrapping around other libraries may also be required (ROS, RoboEarth, etc) or in certain circumstances we could extend them (minimal amount of work). For example, RAPP::API could provide seamless interaction with a *KnowRob* instance running in our cloud, via a wrapper around the actual JSON calls to *KnowRob*.

Execution of a RAPP

The robot user should be able to vocally command the *RAPP client* to initialise a *RAPP*. This assumes that the *RAPP client* is always running on the robot, and ready to respond to recognized commands. Alternatively, the *RAPP client* could be started via bluetooth, from a paired device.

If remote *RAPP* execution is required, then a form of book-keeping of active robots will be required. In this scenario, the *RAPP store* is aware of which *RAPP clients* are currently online and running (much like a centralised messenger), and can instruct a particular *RAPP client*, to execute a specific *RAPP*. The advantage of this approach, other than remote execution of *RAPPs*, is that the *RAPP* could be started by any device which has a web-browser and can connect to the *RAPP store*.

However, both above scenarios, assume that the *RAPP client* is somehow started on the robot, running and accepting connections (or listening through the *RAPP store* for instructions). How the *RAPP client* is started is a different issue altogether. At this point, we have to assume that either the *RAPP client* is started manual (via a bluetooth nearby device, or a tethered computer). Alternatively, the *RAPP client* should be designed to run as a background service (Unix Daemon) running upon the initialisation of the robot's operating system, and constantly polling for commands, either from the *RAPP store*, or via a voice interface. In this sense, the *RAPP client,* is not merely an application but a service.

Upon execution of a *RAPP*, the *RAPP client* has to:

- Verify the *RAPP* integrity and validity
- Check with the *RAPP store* for updates or patches for said *RAPP* and apply them if needed
- Ensure all *RAPP* dependencies are met, or otherwise obtain dependencies from *RAPP store*
- Initiate the *RAPP* as a client process on the robot
- Connect to the *RAPP store* and securely introduce the *RAPP process* (on the robot) with the *RAPP* counterpart running in the VM (introduce socket to socket).

Upon execution of a *RAPP*, the *RAPP store* has to:

- Send any updates or patches to the *RAPP client* starting up
- Ensure user and *RAPP* authenticity against its own database
- Allocate and execute a new *RAPP* process in the VM

• Notify the *RAPP client* of the socket where the *RAPP* instance is running on the cloud (introduce socket to socket).

RAPP Security

Below is a brief description of what we consider essential security requirements for the *RAPP store*.

Each *RAPP* should be allocated the following:

- An *SHA512* hash, produced using the compiled binary. If the hashes don't match either data corruption or malware injection may have taken place.
- A Unique hash generated *RAPP store*, identifying the vendor (developer)
- *RAPP* instance (specific *RAPP UID* for specific user) in the VM
- Generic *RAPP* ID (across all *RAPPS* for all users), same as the VM UID
- *RAPP* version, used for updating or patching
- A unique locked VM directory, used only by the specific *RAPP instance* for specific *RAPP user*. The directory should be accessible only by the specific *RAPP* instance, and could be possibly encrypted, being decrypt-able only by the *user's* password

Each *RAPP user* should be allocated the following:

- A unique password, used for accessing the web-portal and starting the RAPP client remotely
- A unique salted hash, used for identifying the user within the RAPP store
- An x.509 certificate, generated and validated by the RAPP store, used for bidirectional secure socket (TLS/SSL) communication between the RAPP store and RAPP client.

RAPP Store

Finally, apart from the RIC and the VMs, the store, acts as a, store! Meaning, it indexes a collection of RAPPs which may be downloaded by RAPP users on their robots. This would see a complex of processes and databases, where submitted source code is transformed into a RAPP, and then distributed.

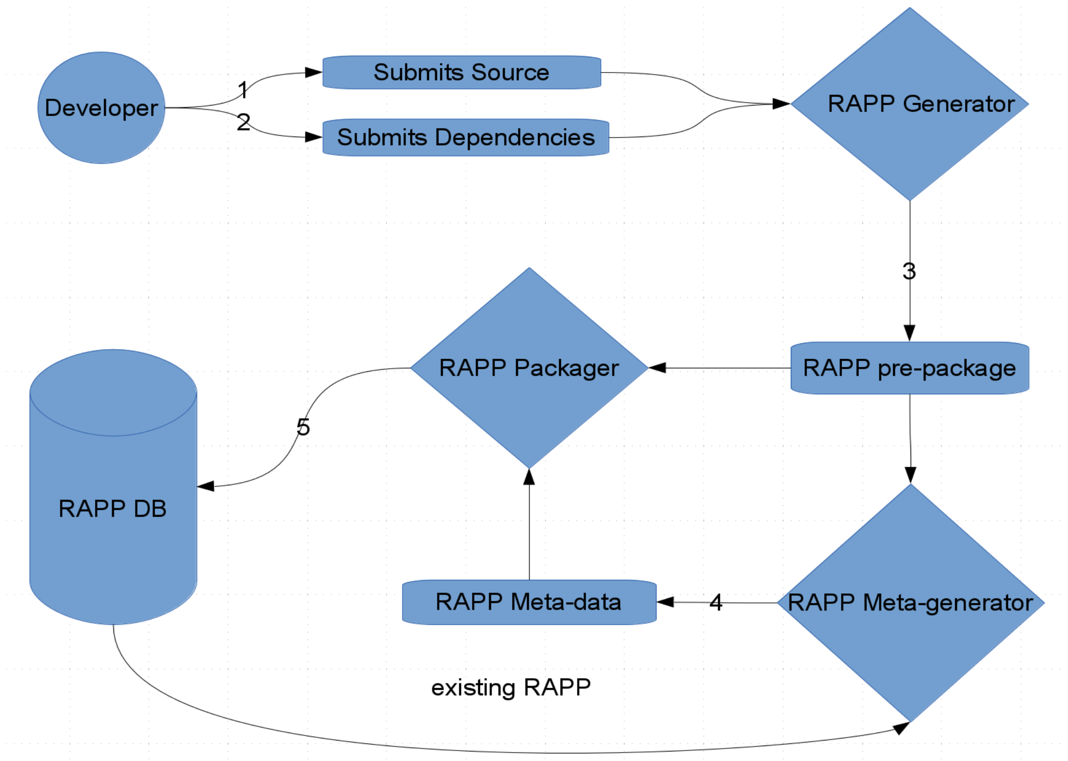
Contrary to the initial document submitted by Manos Tsardoulias, we do not consider the Store to be only the front-end, but the entire cloud-platform, whereas the RAPP Platform is in fact the combination of the Store & Client (*Drawing 1*). Whilst this is simply a matter of interpretation, we should at some point, agree on names and abbreviations used in the technical part of the project.

In effect, the **Store** is responsible for:

1. Processing new RAPPs
2. Indexing & Distributing RAPPs
3. Housing & Executing cloud-based RAPPs (within Virtual Machines)
4. Housing the RIC & Delegating RIC API calls

Store Process Overview

This is the overall picture (*Drawing 7*) of how a RAPP is created. Please note that when we mention the *RAPP store,* we in effect mean the *cloud or server*, upon which all the following processes take place. The RAPP store is not only the web-portal to which developers submit projects, or from which users download *RAPPs*.



Drawing 7: Store Processes

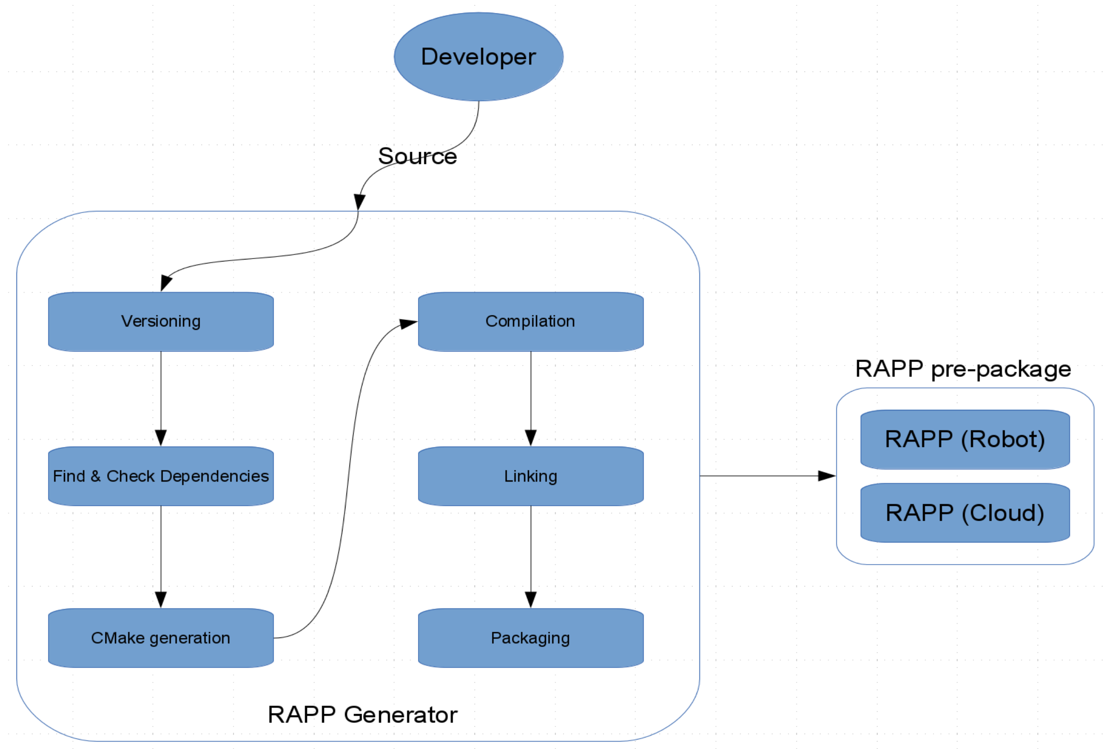
The process described can be broken into four distinct parts (one not shown).

1. Developer submits his or her RAPP, by providing entire source code and dependencies
2. Generator creates a project pre-package
3. Meta-Generator creates the project meta-data
4. Packager puts everything together and stores them

It is important to remember that a project may contain one binary (for the robot) or two binaries (one for the robot and one for the VM/cloud). Furthermore, a package may be an update of a previous package. Also, depending on the programming language used for the API, a dependency check (libraries and headers) may be required from the Generator, as both the Generator (in order to compile and link) as well as the executing platform(s) will have to satisfy those dependencies.

RAPP Generator

Below is an overview of how a RAPP is generated in an automated fashion by the *RAPP Store.* As before, depending on the API language used, Instead of compiling and linking (C/C++) we may be looking at packaging (Java/Python). This process has to version the project, check and detect dependencies (albeit they may be provided by the developer, a heuristic reassurance may be needed), generate in an automated fashion a CMake file, which will in turn compile and link the project (or projects if one is developed for the cloud).



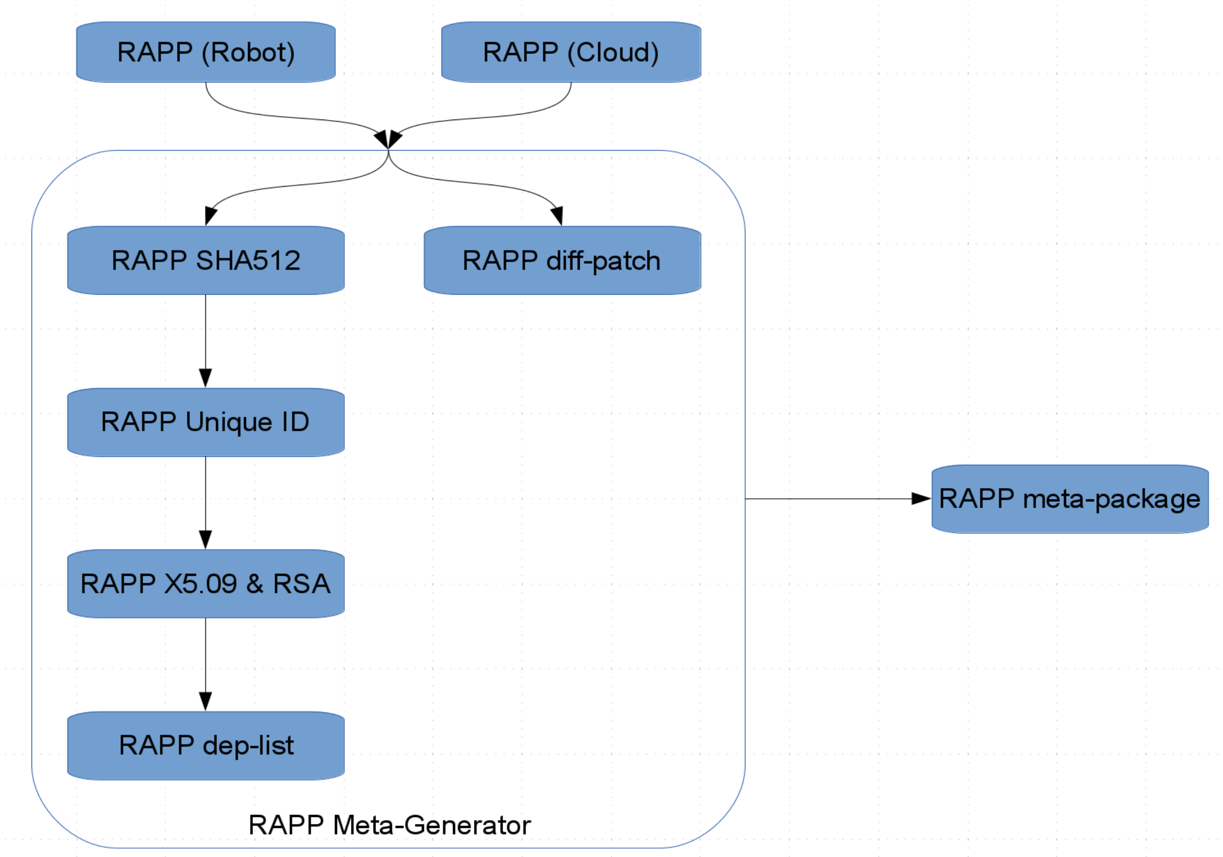
Drawing 8: RAPP Generation

At this point, we may additionally chose to generate a package (*.deb, *.rpm, *.yum, etc.) and distribute it instead. Doing so offers the extra advantage that a *RAPP* when installed, can use *checkinstall* to verify that the *RAPP*'s dependencies are satisfied on the platform where it will be executing.

It is also important to note, that a lot of the above functionality already exists in the **HOP** framework. As such, it could be highly beneficial to avoid having to re-invent the wheel, and simply employ HOP, or extend it for our purposes if needed.

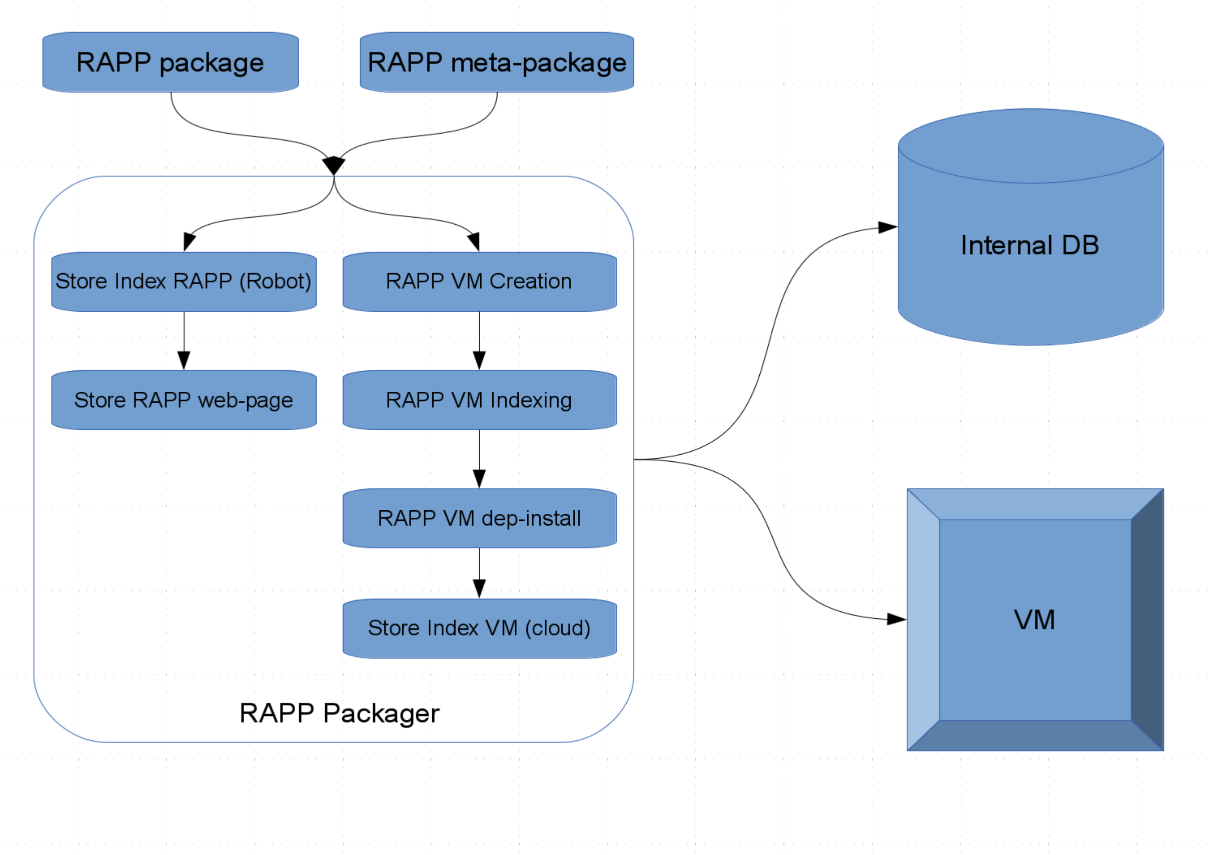
RAPP Meta-Generator

The Meta-Generator creates all the meta-data related to a specific RAPP project (and not just a package as they may be two). As described earlier (Meta-Data) a series of security and indexing related tasks are to be performed. Calculating a SHA512 checksum using the RAPP binary, generating a unique id for the *RAPP* (different from the unique id generated for a RAPP instance, an allocated RAPP for a specific user), creating a x5.09 certificate for this particular *RAPP* to use over TCP/IP, generating an RSA key for encrypting and securing the sandbox/directory within the VM, or encrypting RAPP user data, as well as a list of all dependencies needed by this particular RAPP. Additionally, in the event this is an update to an existing RAPP, we may chose to generate a diff patch, instead of transmitting the entire RAPP all over again.



Drawing 9: RAPP Meta-Generator

RAPP Packager



Drawing 10: RAPP Packaging

The final process, is packaging the RAPP, and storing it (the RAPP for the robot on the store for distribution, the RAPP for the cloud, within a VM, for execution) as well as indexing all packages for bookkeeping.

Apart from indexing the RAPP (robot binary) with a description, version, etc, and displaying it on the actual store web-front, if there exists a RAPP that is to be run in a VM, the following actions must take place:

1. Check if a VM exists for this RAPP
2. If not, create it (clone an image of a standardized Debian system, using a VBoxHeadless)
3. Index the VM for the RAPP, allocate directories and Database within the VM
4. Ensure all dependencies are satisfied within the VM, so that the RAPP (cloud) can execute
5. Index the VM and all related details to the master RAPP record

After this process, the RAPP is now ready to be displayed on the RAPP web-store, and downloaded on to robots.

Virtual Boxing the *RAPPs*

Proposed system architecture employs Virtual Machines in order to execute the *RAPPs* in the *RAPP store.* Although there exists a variety of emulators, simulators, hypervisors and virtual machines, we propose the employment of *Virtual Box* [10] instances for this purpose. Alternatively, a *chrooted* environment could be used.

Using a *VirtualBox* for executing *RAPPs* offers certain advantages. However a distinction is required to be made, as there are different scenarios under which a *VirtualBox* can be used.

It is important to note that the *RIC* data and processes reside outside those VM's (possibly in a VM of their own) and are accessible via local sockets, provided by the *RAPP::API* only. Enabling A.I. processes based on actual hardware devices (such as NVIDIA CUDA) requires that process to run **outside** a VM, as it needs direct access to the hardware. If for example, an Artificial Neural Network requires to run in the RIC, it should be accessible via the RAPP::API, either from within a cloud-controller or a robot-controller.

Virtual Box per RAPP Group

First examined scenario is the one where each group of *RAPPs* has been allocated one VM. In this case, all instances of the same *RAPP* execute under the same isolated environment, as a different unique processes, and store their data under separate directories. Each *RAPP* is controlled by the *RAPP store* upon receiving a notification from a robot that a *RAPP* has been initialised. The group of identical *RAPPs* (belonging to different users*,* originating from different robots) could in theory share data with each other at the *RAPP store* level, by sharing a common database within the VM. Data sharing can be enabled either by the *RAPP::API (*by wrapping around MySQL calls to the VM Database instance, or otherwise).

Isolating the *RAPPs* on a per-group case, offers security, resilience (i.e. if a *RAPP* crashes, it won't compromise the entire *RAPP store*), in a controlled, sandboxed environment. The *RAPP store* can control incoming and outgoing network connections at the hardware (VirtualBox) level, and as such, the *RAPP store policies* are enforced transparently, and not on a per-application basis (e.g. if we wish to not allow distribution of private data obtained by the *RAPP*, we simply disallow outdoing or incoming network connections initiated by the *RAPP* within the VM).

Furthermore, each *RAPP* instance can be allocated a unique sandboxed directory, and store temporary, cached or persistent data, in the form of a local file or ram-disk. Also, each *RAPP* can be allocated a certain amount of CPU time and RAM memory as a processes spawned by the *RAPP store* within that VM. Connections should be monitored by the *RAPP store*, and security could be enforced in both the TCP/IP layer by using TLS/SLL, and on the disk, by employing encrypted containers (e.g. the entire VM could reside in an encrypted file container or filesystem). In the event of a *RAPP* crash, graceful termination could be possible, providing debug information to the developer and the administrator.

Virtual Box per RAPP Instance

Similar to the above scenario, a unique VM is allocated per each RAPP instance (application per user). This is a somewhat extreme scenario, which could become a performance bottleneck, but could enable high restrictions and security if required. We advise against it, as it could potentially degrade performance.

Universal Virtual Box

At the other side of the spectrum, under this scenario only one VM is used for all *RAPPs* available in the *RAPP store.* The sand-boxing benefits are still applicable, but security is more lax.

RAPP Ontologies

The RAPP architecture will utilize two main ontologies. The first ontology (name?) will focus on the communication process between RAPP platform and RAPP applications. The second ontology (name?) will be used as the knowledge representation of the robots. RAPP will utilize existing robot ontologies, and extend them in order to allow for the differences required for the specific use cases.
The state of the art robot ontology repository that is officially supported by ROS is KnowRob [1]. KnowRob contains four groups of OWL ontologies: Base ontologies, Semantic Robot Description Language, Computable definitions and Semantic environment maps [2]. Additionally KnowRob has a very detailed documentation that will boost the ontology manipulation [3][5].
There is a possibility of extending KnowRob, as other extensions are already "officially" included [4]. Finally other possible ontology sources are ORO-Ontology [6] and the one used in RobotML Modeling Platform [7].

Appendices

A. Technical Work group minutes

The following is an abstract of the Technical Work group minutes from the RAPP kickoff meeting at Thessaloniki, Greece (12/12/2013).

- *Critical points in discussion:*
- *Both NAO and Ang use Linux, so the RAPPs will not be OS-cross platform but Robot-cross platform*
    - *Linux will be used, as ROS supports only Unix OSs.*
- *Something is needed to handle the RAPPs in the actual robot. A wrapper / FSM? This must be downloaded before an actual RAPP is installed on robot.*
- *The RIC (RAPP Improvement Center) idea is introduced. The RAPP Store will be the frontend and RIC the backend where heavy computations should take place.*
- *The RAPPs downloaded will be executables. The path from the developer to the robot is the following:*
    - *Developer writes code (C++/Python?)*
    - *Developer commits code to RAPP Store*
    - *Automatic build from RIC and executable creation for different platforms*
    - *Binaries provided by tools such as apt-get*
    - *Used downloads and installs RAPP in his robot*
- *For using HOP in the robots we must have a web-server, because HOP suggests web-distributed applications.*
- *INRIA will support ROS in the following 6 months.*
- *Maybe we will have different binaries for different robots*
- *The behavior described by WUT is basic robotic actions: Find an object in image, move your arm etc.*
- *The developer will provide the Plans that use the behaviors to give the robot a functionality. So a RAPP will be a plan (an FSM maybe). RAPPs could use directly ROS functionalities.*
- *RIC includes the ontology data.*
- *Proper API must be provided so that when the robot does not have knowledge about a certain object, the RIC will be contacted and the specific ontology (behavior or parameter) will be downloaded. Ontologies must be translated to parameters.*
- *The developer must know all about RIC in order to use it in his/hers RAPP.*
- *Applications could work in a distributed manner: Part on-robot, part in a virtual machine in RIC.*
- *Aggregate data should be collected for RIC and processed in order for the knowledge exchange to work. For example: In the memory ball, the robot learns that keys are usually placed on the table. This statistical information could be passed and used to other robots as well.*
- *ROS and HOP will be used simultaneously on all robots. This helps in RAPP's generalization.*

B. Abbreviations

Term	Abbreviation	Definition
RAPP application	R.APP	
RAPP platform	RAPP	
RAPP API	RAPP::API	
RAPP wrapper	RAPP-FSM	
RAPP Improvement Center	RIC	
RAPP Store	RAPP Store	

References

1. http://www.knowrob.org
2. http://www.knowrob.org/ontologies
3. http://www.knowrob.org/doc/knowrob_taxonomy
4. http://knowrob.org/kb/unr_actions.owl
5. http://www.knowrob.org/doc
6. http://www.openrobots.org/wiki/oro-ontology
7. http://europe.bourges.univ-orleans.fr
8. http://www.open-mpi.org
9. http://www.roboearth.org
10. http://www.virtualbox.org