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

D2.5.2 RAPP Communication and Event Processing

Deliverable status:	Final
File Name:	RAPP_D2.5.2_V2.0_30112015.pdf
Due Date:	November 30, 2015
Submission Date:	November 30, 2015
Dissemination Level:	Public
Task Leader:	2 - INRIA
Author:	Vincent Prunet, Ludovic Courtes

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
INGEMA	Fundacion Instituto Gerontologico Matia - Ingema	Spain
AUTH	Aristotle University of Thessaloniki	Greece

Revision Control

VERSION	AUTHOR	DATE	STATUS
0.1	Vincent Prunet (Inria)	December 02, 2014	Initial Draft
0.5	Vincent Prunet (Inria)	January 16, 2015	Sections Network, ROS, HOP, Application Management, Security
0.6	Vincent Prunet (Inria)	January 18, 2015	Section Rowe (edited from Ludovic Courtes report)
0.9	Vincent Prunet (Inria)	February 17, 2015	Full document ready for review
0.9	Fotis Psomopoulos (CERTH)	February 25, 2015	Review
1.0	Vincent Prunet (Inria)	March 10, 2015	Final (Deliverable 2.5.1)
1.5	Vincent Prunet(Inria)	November 15, 2015	2.5.2 Draft
2.0	Emmanouil Tsardoulias (CERTH/ITI)	November 18, 2015	Review & final corrections

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
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System Protocol
DNS-SD	DNS Service Discovery Protocol
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
IoT	Internet of Things
LAN	Local Area Network
MDNS	Multicast DNS Protocol
NAT	Network Address Translation
ROS	Robot Operating System
URL	Uniform Resource Locator
WAN	Wide Area Network
WLAN	Wireless LAN (WIFI)

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).

The document discusses requirements and proposes solutions for the integration of robotic and web application frameworks, the use of the HOP framework to support dynamic downloading and execution of RAPP applications on the robot, and the security model for the applications.
The first section of the document lists technical requirements (network requirements, application distribution with an embedded part and a cloud part, dynamic installation of additional applications on the robot, integration of web communications with ROS communications, security).
The other sections detail proposed solutions to these requirements.
Several API have been specified to implement the solutions. Also, proof of concept code has been developed to demonstrate distributed communications and application downloading from the RAPP store. References to the API and POC code are given in the annex.

The updated document (D2.5.2) includes the following new items:

- Description of additional interactions between software components, in particular the direct connection of Python and C++ clients to Hop.js processes, using Hop.js builtin support of RFC3986. The role of the Hop.js process as the gateway between distributed heterogeneous components is further emphasized by this extension.
- Discussion on robustness and description of design patterns that improve robustness of the distributed applications.
- Description of the security tools available to ensure authentication and privacy(https, token based security).

The Future Work section has been updated.

Introduction

The RAPP DOW describes the corresponding task (T2.5) and the respective attached deliverable as follows:
Task description

> *This task will design the communication and event processing module of RAPP. The component will provide a link between the various functional blocs and the robot's hardware. Its main features will be the following: a) a plug and play protocol stack enabling robots to get an implicit knowledge of nearby resources, b) the management of secure communications with peer devices and remote web services (including the RAPP store), c) the ability to download and execute applications from the RAPP store, d) the management of events to and from sensors/actuators, the data storage component, and user interaction devices, and e) the control of low level robot operations (interfaces to the device dependent firmware). INRIA will be mainly responsible for this task, while WUT will also contribute. The results of T2.5 will be presented in D2.5.1 and D2.5.2. A draft version of D2.5.1 will be available for the first review meeting.*

Deliverable description

> *RAPP Communication and Event Processing: Early report of communication and event processing module of RAPP. The component will provide a link between the various functional blocs and the robot's hardware. [month 15]*

The decision to develop Web aware, distributed, robot applications, as is our intention in the RAPP project, is a challenging prospect. There are multiple constraints:

- Tools that researchers and engineers want to use to be efficient in their development (building on existing pieces of code whenever possible)
- Different tools, protocols, and culture in Robotics and in Web development that require a significant integration effort
- The objective that applications are plug and play with minimal configuration steps (an absolute requirement for applications and hardware that shall be deployed in the field with limited support staff), taking into account the asymmetrical client server architecture of residential Internet connections
- Security of the applications, a critical requirement given the medical context of some applications and privacy concerns for all of them.

We have tried to identify those constraints, to select existing software frameworks to serve as a backbone for our project, to identify the software interfaces to use to integrate these frameworks, to develop additional software components to fill the gap and integrate heterogeneous components together, and finally to prototype proof of concept code to validate the technical developments on some use cases.

1 Technical Requirements

We provide here the technical requirements linked to the task definition. Then technical solutions are detailed in the following sections.

1.1 Distribution of tasks between the robot and cloud services

Deployed applications shall allow distribution, with one part of the application running embedded on the robot, and the other part of the application running on the RAPP cloud platform. Both the robot and the cloud part of the application are divided into a core agent that is always running and a dynamic agent that can be replaced depending on the robot current mission.
Communications between components (the robot, the RAPP platform, web browsers [see section1.4]) shall be bidirectional and allow quasi real-time operations. All devices shall be kept in sync.

The network architecture shall be fully operational in a typical WLAN/LAN/WAN Internet configuration (the major constraint being that all communications shall be initiated by LAN devices (the robot, the web browser) and never by remote devices (the RAPP platform, the RAPP store).

1.2 Dynamic robot configuration according to robot capabilities and run time environment

Each robot is unique due to multiple characteristics, such as its hardware, the resources available in the robot and around the robot, and the intended use of the robot.
Manually configuring a robot for a specific mission would be error-prone and would require a high level of expertise from the person in charge of the configuration. Instead, it is desirable that a software configuration mechanism allows for the automatic discovery of available resources, and the automatic installation of the software required by the robot to perform its mission.

1.3 Communication between software components

RAPP is a heterogeneous framework composed of:

- the robot firmware and robotic control applications, often written in C, C++ or Python, and many of them based on the ROS (Robotic Operating System) framework
- web applications, distributed on the robot and on the cloud platform, based on web protocols.

The RAPP framework shall allow many to many, quasi real-time, communications between distributed software components hosted on the robot or the RAPP platform, and provide a flexible integration between native, ROS and web components.

1.4 Operations from a web agent

Some applications need to provide access to the robot and RAPP platform from web browser clients, to allow the user to configure the robot and the application, to send real-time control commands to the robot, to view data that cannot be conveniently displayed by the robot (graphs, images), to browse through activity monitoring data stored on the RAPP platform.

1.5 Data storage

Persistent storage shall be accessible for configuration data, program files, and operational data (images, activity data), both on the robot and on the cloud platform.

1.6 Robustness

Most applications should survive to temporary network outages, ensuring that critical services do not depend on an active network connection with the RAPP platform or can be suspended safely. For those applications that do not depend on RAPP remote services for their real time capabilities, smooth operation should be ensured even in network free environments.

1.7 Security and privacy

The framework shall ensure that communications between components are trusted.

Applications shall control and filter the access to user sensitive data. User data are private and their full access is reserved to authorized persons.
Core applications shall control the access to sensitive functions on the robot and platform. Not all RAPPs shall be granted full access to the robot firmware and to platform services.

2 Network Layer and Protocols

Most RAPP communications are based on the Internet Protocol (IP) suite. Various transmission layers are used: WLAN, ethernet LAN, DSL/fibre/radio WAN access, Internet).
Additional transmission technologies may be used to connect peripherals and small objects to the robot, such as infrared signaling, bluetooth, and IoT (Internet of Things) protocols.

It is assumed that:

- The robot operates in a closed area where a secure WLAN is available, with DHCP.
- The robot has a WLAN interface, already configured to use the provided WLAN service.
- The Internet gateway device provides Internet connection (optional if the cloud platform is located on the LAN).
- The RAPP platform URL is fixed, and the platform can be reached by the robot.

Communications between the robot and the cloud platform are always initiated by the robot. This asymmetric communication establishment is compatible with the architecture of today's Internet (Residential Gateway routers perform a public/private address translation with NAT, preventing WAN clients to connect to LAN resources). Once a connection is established, it allows for bi-directional low-latency communications between the robot, the RAPP platform and other components (web browsers).

Selected protocols:

- ROS protocol suite (intra robot): this is the standard protocol for inter component communication: contribute/reuse components.
- HTTP based protocols including websockets (robot to platform, robot/platform to control devices and third party services): this is the standard protocol suite for flexible interconnection of distributed software components in modern architectures.

3 Application Layer

RApp applications are distributed across the network: part of the application runs on the robot and part of it runs in the cloud.
There are many good reasons to distribute the application:

- Add computation power, and virtually unlimited data storage capabilities to the robot, without local energy considerations, such as additional batteries (wireless communications cost energy, but much less than what would cost embedded computation)
- Benefit from third party cloud services. The robot is not only connected to a remote service, but also to the whole Internet.
- Increase security. Third party software providers (software applications, web clients) do not come directly to the robot, but rather to the cloud application. Requests from untrusted third parties are filtered, and processed in the cloud. When necessary, trusted applications get a security token from the cloud application to securely connect to and control the robot.

The installation of a RApp on the robot triggers the installation and launch of a sister application in the cloud. Again, the application is retrieved from the RAPP store, and then installed on the RAPP platform.

Two core frameworks have been selected to implement the distributed architecture:

- HOP, a web based framework, to provide robot to cloud, robot/cloud to browser communications, and to run scripts both on the robot and the cloud platform.
- ROS, a robotic framework, primarily to allow for intra robot communications and to support specialized computing agents running either on the robot or on the cloud platform.

A gateway has been developed to integrate both frameworks, and a client C library (rowe) has been specified and implemented to allow other programs (typically pieces of firmware code) to integrate with the RAPP framework.

3.1 The HOP framework

HOP is a framework and a programming language, based on web standards, which aims to provide robot to cloud, robot/cloud to browser communications, and to run scripts both on the robot and the cloud platform. An initial description of HOP has been presented in section 3.2.12 of RAPP deliverable 3.1 "*State-of-the-art Report*".

A HOP process is basically a fully programmable web server, able to serve files over HTTP, to handle complex HTTP requests involving algorithmic data processing on local (file system, database) or network (ROS, web services) resources, and to provide browser clients with HTML contents and dynamically generated JavaScript code to run on the browser side. HOP natively supports websockets, which are heavily used in the RAPP project for communication between the ROS nodes and the robot firmware components.

A HOP process is an OS process with the following capabilities:

- DNS-SD advertising to let peer devices discover the HOP services
- Dynamic load, run time compilation and execution of HOP scripts
- Access to the underlying operating system services (process control, sockets, file system)
- A built in HTTP server, which enables the script to provide HOP services to peer processes
- A built in HTTP client service, to invoke web services and peer hop services through HTTP
- Support of web sockets, which can be used to provide various protocol API to communicate with other processes
- Control of hardware/software resources by linking and wrapping additional libraries. For example, HOP provides APIs to use SQLite libraries, or Phidget robotic sensors and actuators from a HOP script.

HOP scripts can be written either in Scheme (the core development language for HOP, together with C) or in JavaScript. Specifically, the HOP JavaScript front end is compatible with Node.js and supports most of Node.js API and the Node module system. All RAPP applications for HOP use the JavaScript syntax.

HOP provides specific extensions on top of JavaScript:

- *Service invocation*. A HOP process can invoke a service provided by another HOP process. Service arguments and the service response are automatically encoded/decoded as standard HTML requests. Services can also be invoked from off-the-shelf web browsers running JavaScript programs generated by the HOP server they are connected to. Service invocation Is either non blocking (asynchronous) or blocking, at the choice of the caller.
- *Web service invocation*. A HOP process can invoke services provided by third party web servers, due to the built-in API which can generate service URLs and retrieve service responses.
- *Service implementation*. A service implementation is defined within a HOP process as a custom JavaScript function, which is automatically invoked by the HOP run time library whenever the corresponding HTTP request is received by the server. Service handling can be either synchronous or asynchronous. This is an important aspect of HOP, as it allows HOP to act as a proxy and get the response from another HOP server or from ROS.

- *Web Service implementation*. In addition to proprietary HOP services, a HOP process may define Web Services that can be invoked from third party client processes. Services may be either synchronous or asynchronous. HOP supports the RFC3986 syntax for service invocation, third party client may use either GET or POST methods to invoke services. Both the application/x-www-form-url encoded and multipart/form-data Content-Type options are supported. This feature has been used in the RAPP architecture to enable the direct interaction of C++ and Python client programs with the HOP server.
- *Broadcast APIs*. A HOP server process can broadcast events to dynamically registered clients (HOP processes or browser clients) using a simple API that hides the complexity of server to client communications and event routing. The mechanism is client initiated, fully interoperable with firewalls and NAT traversal.
- *Dynamic construction of programs for browser clients*. The client HTML and JavaScript code are generated from the server application. This feature is very useful to implement sophisticated user interfaces and client applications running on standard web browsers.
- *Multi-threaded HOP processes*. In general, a thread is allocated for each low-level system/communication task, but JavaScript programs can also be split into multiple threads using the *Worker* abstraction. A HOP process using workers decides which worker implements which service, then service invocations are automatically routed to the thread providing the service. Workers do not share memory, except for internal system resources, which can be manipulated safely by the developer through a number of APIs. For example, the handler for an asynchronous service implementation can be passed from one worker to another as a value, allowing for very flexible and highly responsive data processing patterns.

3.2 ROS

ROS is a framework for robotic applications. Within RAPP, ROS is used to allow for intra robot communications and to support specialized computing agents running either on the robot or on the cloud platform. ROS has already been presented in section 3.2.1 of RAPP deliverable 3.1 “*State-of-the-art Report*”.

The ROS protocol is fully supported in the RAPP architecture, and used both on the robot and on the cloud platform.

A typical ROS enabled robot (NAO) implements a single embedded ROS graph. The ROS graph is composed of a set of ROS nodes, including the ROS master node that contains the directory of all nodes participating to the ROS graph. Each ROS node implements some of the functions of the robot, including drivers to control the robot hardware. Within the robot, ROS nodes communicate with each other using standard ROS mechanisms, such as ROS topics (spontaneous broadcast of messages by a node to interested peers) and ROS services (a node may invoke a service implemented on another node). The communication with cloud ROS services and other services is performed through the ROS/HOP gateway.

The RAPP cloud platform also hosts one or several HOP graphs. The attached ROS nodes may provide common services to the robots connected to the cloud platform (shared nodes), or may be dedicated to a single robot (dedicated nodes). Shared nodes are stateless, whereas dedicated nodes retain some knowledge of the robot state. Connection between the embedded graph and the cloud platform graphs is achieved through the ROS / HOP gateway and HOP to HOP service invocations.

3.3 ROSBridge and the ROS / HOP gateway

The ROS / HOP gateway establishes bi-directional communications between a ROS graph and HOP. An API has been developed within HOP to support the ROS protocol: a HOP process is viewed from ROS as a collection of ROS nodes (emitting ROS topics, receiving topics, invoking or providing ROS services). The API implementation is done using the ROSBridge protocol (JSON over websocket protocol supported as a ROS extension).

The ROS/HOP gateway is used locally on the robot to enable HOP scripts to control the robot behavior. It also provides a means to link, through HOP, different ROS graphs. Thus embedded ROS nodes can communicate with remote ROS nodes hosted on the cloud platform. The use of HOP as a proxy on the robot and on the cloud platform provides a convenient solution to the following problems:

- *Cooperation of several ROS graphs (and master nodes)*: one on the robot, the other ones on the cloud platform, potentially serving several robots)
- *Asymmetrical network connections*. The robot is likely to run on a LAN, connected to the open Internet through a home router that lets the robot open a connection to a remote server, but that generally prevents network servers to open a connection to the robot, which stands behind a firewall and NAT device. HOP allows for robot initiated, bidirectional flows.
- *Security*. An appropriate security policy can be implemented on HOP processes, thus protecting data and accesses to the robot and to the platform.

3.4 ROWE API

Most RAPP software components are either implemented as HOP processes or as ROS Nodes. However, some platforms, and existing robotic/sensor firmware do not natively support ROS. Implementing a full ROS graph on such platforms would require a significant porting effort of ROS to the target software environment, if at all possible given the footprint constraints of such environments. Also existing firmware library code would need to be rewritten to fit the ROS API and event model.

We have considered two alternatives: either link the firmware library code to a HOP process or specify a lightweight C client library offering simple message passing capabilities to arbitrary C programs, allowing these programs to communicate with HOP over TCP/IP with little programing effort.
The first option (link the firmware to HOP) had been previously experimented for another project[1]. Performances are optimal, but flexibility is low, since any change to a function prototype must be reflected in the HOP binding of the library and requires advanced technical skills from the developer. Also, this option prevents distribution of tasks among multiple devices (a peripheral device that runs only firmware code and communicates with a main device running HOP).
The second option (a HOP client communication library) decouples the C firmware program from HOP, and supports distribution. It is by far more flexible and we have decided to specify and implement such a library within the RAPP project. An additional benefit from this design is the ability to develop in parallel the firmware and the RAPP core application, to replace the actual firmware/hardware by a firmware simulator, and to remotely connect to the firmware (the firmware is embedded on the hardware, and the RAPP application runs on the developer's workstation, dramatically improving comfort for the developer).

3.4.1 Design Goals

We have set out a number of design goals for ROWE:

1. Robotics software using ROWE is going to send status updates to other RAPP components at a possibly high rate (for instance, the speed and location of the robot, similar to ROS "topics"), and it must receive and process requests from other components in a timely fashion (such requests may include an emergency stop, for instance, similar to ROS services.) Thus, ROWE must guarantee low latency and high throughput.
2. Programs using ROWE are meant to be connected primarily with HOP programs, so ROWE must be a "native speaker" of the Web protocols and formats.

[1] Cable-Driven Robots with Wireless Control Capability for Pedagogical Illustration in Science, Alexandre dit Sandretto J., Nicolas C., Inria, Control Architecture of Robots (CAR 2013)

3. It must be possible using ROWE to exchange typed and structured messages, such as strings, numbers, records, lists, and so on.
4. The application programming interface (API) of ROWE should match the programming style and expectations of low-level robotics developers. In practical terms, this means that it should be as little disruptive as possible.

Being a native of the Web means that ROWE's transport layer should be based on HTTP, which also has the advantage of being usually allowed through firewalls. Of course using HTTP alone to exchange messages, for instance in a ReST fashion, would incur too much overhead: HTTP connections would regularly need to be instantiated, which unacceptably increases latency, and HTTP GET requests may incur too much bandwidth overhead.

For that reasons, we chose to use WebSockets as the transport layer. WebSockets is an HTTP extension that provides a reliable, bidirectional communication channel that can be used to transfer arbitrary payloads, similar to TCP.

JSON (JavaScript Object Notation) came up as the obvious choice for the message format. It meets our requirements as a mechanism to encode structured and typed messages, it is the natural way to represent data in HOP programs, and has efficient parsers and serializers.

The last design goal is more subjective. In our view, matching the programming style of low-level robotics developers directly relates to specific requirements. First, the API should be usable in single-threaded user programs. Additionally, it should not expose a full-blown event loop framework as commonly found in object-oriented libraries such as Glib. Those frameworks are generally complex, and they impose inversion of control (IoC) through a heavy use of callbacks. This in turn essentially forces developers to write in continuation-passing style (CPS), which is both verbose and difficult to work with. Instead, we want to allow a direct programming style. This has been the main choice driving the design of the programming interface.

3.4.2 Programming Interface

The bulk of ROWE's programming interface has purposefully been kept minimal and simple. In ROWE version 1, connections are modeled by an endpoint. A ROWE program can only be connected to one peer at a time (see Section Conclusion for a discussion and desired changes to this approach.) A ROWE program can be an HTTP server:

```
struct rowe_enpoint *endpoint;
endpoint = rowe_open_local_endpoint (8080);
```

or it can be an HTTP client:

```
endpoint = rowe_open_remote_endpoint ("hop.inria.fr", 8080);
```

The API to send and receive messages is the same regardless of whether the program is a server or a client. Messages are JSON objects, as implemented by the JSON-C library[2] sent to an endpoint using the `rowe_send` function. Callers can specify a time-to-live (TTL) for the message: if no peer was connected after the TTL has expired, the message is discarded and not sent. This is useful for periodic messages such as updates on the robot's status, akin to ROS topics: an old update is not valuable and can be discarded, allowing the peer to get fresher updates instead.

```
extern int rowe_send (const struct rowe_endpoint *endpoint,
      const struct json_object *obj,
      long ttl);
```

The `rowe_send` function is synchronous and blocks until the message has either been sent, or has been discarded. Alternately, the `rowe_async_send` function is non-blocking, and may typically be used when sending messages containing status updates.

2 The JSON-C library: https://github.com/json-c/json-c/wiki

Similarly, the `rowe_receive` functions blocks until a message is received or the user-specified timeout has expired, and returns a pointer to a json_object structure or NULL. Programs may also perform remote procedure calls (RPCs), using the `rowe_invoke` function:

```
extern struct json_object *
rowe_invoke (const struct rowe_endpoint *endpoint,
     struct json_object *obj, long timeout);
```

The function sends the given JSON object, which denotes a procedure invocation, blocks until a reply has been received, and returns it, unless the given timeout has expired. The actual format of the JSON object representing the procedure call is at the user's discretion. An example JSON-formatted service invocation may look like this:

```
{
  "service": "add-two-numbers",
  "a": 38,
  "b": 4
}
```

Lastly, ROWE programs can reply to RPCs, using the `rowe_reply` function or using `rowe_async_reply`, its non-blocking counterpart. To facilitate the creation of JSON objects representing key/value associations, the `rowe_message` convenience function is provided. For instance, the message shown above may be instantiated with the following call:

```
struct json_object *invocation;
invocation = rowe_message ("service",
   json_type_string, "add-two-numbers",
   "a", json_type_int, 38,
   "b", json_type_int, 4, NULL);
```

3.4.3 *Implementation*

The implementation of the above API goes along the following lines.

3.4.3.1 Service Thread

ROWE builds upon the JSON-C and libwebsockets libraries. Since it does not expose an event loop interface, the actual event loop runs in a dedicated service thread, which is spawned when the endpoint is opened. The service thread polls for connection requests and for "in" and "out" events on open connections.

The service thread adds incoming messages on a queue that is checked by functions such as `rowe_receive`. When `rowe_send` and similar functions are called from the user thread, they add the given message to an outgoing message queue, which the service thread checks when the connection is ready to accept outgoing messages. When the service thread accesses the outgoing message queue, it deletes any messages whose TTL has expired.

The `rowe_async_send` function is the simplest: it just adds a message to the outgoing message queue and returns immediately. Conversely, the `rowe_send` and `rowe_receive` functions need to synchronize with the service thread. `rowe_receive` checks for message in the incoming message queue; when the message queue is empty, it waits on a condition variable associated with it. `rowe_send` works by passing a notification object, which essentially bundles together a condition variable and a return value, which the service thread notifies when the message is discarded due to TTL expiration, or once it has been sent.

3.4.3.2 Remote Procedure Calls

RPC replies need special treatment: when `rowe_invoke` is used, unrelated messages may be received after the invocation message has been sent and before the reply has been received; yet, `rowe_invoke` must return the RPC reply, not another message that happened to be received first.
To address that, ROWE takes several steps. First, it requires invocation messages to be JSON objects (key/value associations) and, upon invocation, it automatically adds them a message_id entry whose value is a unique identifier, allowing the invocation to be distinguished from other invocations of the same remote procedure. RPC replies must also be JSON objects, and they must have a in_reply_to entry whose value is the message_id of a previous invocation message.
Additionally, ROWE maintains a table that allows it to match RPCs with replies, and to wake up the user thread that is waiting in `rowe_invoke`. The table is essentially a list of pairs of message_id values and corresponding notification object that allows the user thread, which may be waiting in `rowe_invoke`, to be woken up.

3.4.4 Usage

ROWE is the selected protocol for communication between the ANG-MED rollator firmware and the embedded HOP process that governs the rollator behaviour. Within WP5, INRIA has specified ROWE

3.4.5 Summary

The ROWE library provides a simple programming interface for connected components. It is well suited for the RAPP project where it allows low-level robotics software to communicate with HOP programs using Web protocols, and with good performance, notably on low-end embedded ARM-based devices.
As of this writing, version 2 of ROWE is being developed. The main goal is to allow users to distinguish between an endpoint and an established connection, and to support connections with multiple peers.
The ROWE library is free software, available from https://github.com/rapp-project/rowe and from ftp://ftp-sop.inria.fr/indes/rapp/rowe/.

3.5 Event processing

A RAPP application needs to deal with several kinds of events.

3.5.1 ROS events

ROS nodes are typically developed using ROS API (C++, Python). The API provides event handling routines, which will not be further detailed here.

ROS is also supported in HOP (through the the ROSBridge server front end that translates ROS messages to JSON over web sockets). The ROSBridge.js HOP module has been developed to provide asynchronous processing capabilities for ROS messages.
The principles of the ROSBridge.js module for ROS services implemented in HOP, are the following:

- To register a HOP service for each ROS service provided by HOP
- To invoke the HOP service asynchronously whenever the ROS service request is received and to send back the result as a ROS message, with the proper request ID.

HOP may also invoke external ROS services asynchronously. The caller provides the service name, service arguments, and a callback HOP function that is called by the HOP built-in event handler when HOP gets the service response from ROSBridge. Support for ROS topics is done the same way.

3.5.2 ROWE events

Like ROS events, the event handling is done in a different way in HOP or in a C program using the ROWE library. Handling ROWE events in HOP is done in a very similar way as ROS messages. Both protocols use web sockets and JSON objects to carry the payload.

Handling ROWE events in a C program is done internally by the ROWE library (which runs its own service thread). It is the responsibility of the main program to perform a lookup on the event queue when appropriate. This design choice lets the robotic component developer decides how often to look for incoming ROWE events. The caller may perform a non blocking poll (0 timeout), or decide to wait until a message or a timeout occurs (thus controlling the main loop frequency, while still being responsive to external events).

3.5.3 HOP events

HOP events consist in service invocations (HTTP requests), event broadcast (from server to registered clients), web socket messages (including ROS and ROWE ones), worker events between multiple HOP threads running concurrently, and other events corresponding to the user interface (mouse, keyboard, window open/close on web clients) , the server OS (file system, etc), and server peripherals (robotic components drivers).

All these events are handled consistently by the implicit HOP event queue manager, providing automatic routing of events to the HOP worker in charge of the resource, and letting the developer register HOP event handlers. The overall programming logic is, similarly to Node.js and other JavaScript engines, to process each incoming event completely (possibly deferring the processing by invoking an asynchronous service or worker process) then to wait for the next event. The architecture ensures that the system is responsive enough to control the robot high level behaviours and to provide real-time feedback to the end user. In fact, the system is even capable to handle most robotic control loops (although such tasks will generally be delegated to ROS nodes and to the robot firmware.

Event broadcasting is a HOP specific API very similar to ROS topics. Clients may register their interest on receiving specific server events. Then the HOP server automatically sends the corresponding events to all interested clients. This feature is particularly useful to synchronize web browser clients with the HOP server they are attached to, and provide a real-time view of the robot state.

3.6 Web clients

Many robotic applications need a web user interface to:

- configure the robot and cloud services (such as the robot users database),
- serve during the robot operation (input control commands, receive output messages and contents that cannot be rendered by the robot, e.g. images),
- browse activity data stored on the cloud platform.

Web clients display html pages and run JavaScript code, both generated by and downloaded from the HOP server(s) the client is attached to. Web clients are technically extensions of the RApps running on the robot and on the platform.
Web clients initially connect to the HOP web server hosting the Rapp through a standard http request on a well-known URL (we assume that the URL for a RApp application on the platform is known). Since the robot will get a dynamic IP address from the DHCP server of the hosting LAN, we do not assume prior knowledge of the robot URL.
Connections to the robot can go different ways:

- Some robots may not provide web services to browsers, and delegate to the platform all user interaction. This is the preferred architecture for resource constrained robots.
- Some robots may provide the platform with a dynamic URL to allow client connections to the robot. The platform is in charge of authenticating the web user and sending to the browser the URL of the robot service together with a security token, or simply redirecting the first client request to the Robot embedded web server and let the robot authenticate the user. This last option is currently by ANG-MED: the robot connects to the platform to advertise its dynamic IP address, web browsers do the same to retrieve the Robot URL and establish a direct link to the robot. This method can also be used when the robot has got a static IP address or if the robot operates its own private wireless network.

- The last option is for the robot application to advertise its URL using multicast DNS and DNS service discovery. This technique requires that the web client supports DNS-SD discovery natively or through a helper application running on the client computer. Experiments with this technique have shown that mobile devices such as tablets and smartphones do not provide enough support to multicast DNS and DNS service discovery in their native web browsers, native applications must be used to discover the robot then launch the browser on the advertised services. Since it was not our goal to develop a native mobile app within the project, we have decided not to rely on MDNS DNS-SD for RAPP discovery.

Once the web client gets the URL of the platform (or the robot), it can invoke HOP services from the server, and can also open web sockets to receive messages through the HOP broadcast services or raw web sockets opened by the client to the HOP server. The ability to receive messages from the server in real-time ensures that the web client is kept in sync with the server and robot state, thus allowing the web client to monitor the robot and control its operation.

3.7 Persistent Data Storage

We identify the following Data Storage requirements:

- Applications on the robot have access to the underlying file system.
- Applications on the cloud platform have access to a file system and to database servers.
- Data can be pushed back and forth between the robot and the cloud platform, according to the application needs.

For example, the embedded application can store activity data locally and backup the data file to the RAPP platform once the user session is completed. On the platform, the data are kept in a database for additional processing and browsing.
Finally, configuration data for a given user can be retrieved from the cloud platform and installed on the robot for instant use.

3.8 Robustness

Ensuring robustness within a distributed architecture requires that each component can survive to temporary network outages or application failures. Some applications will switch to a temporary degraded mode, until critical remote services are on again. Some other applications may be able to continue operation using locally cached resources, and reconnect to the platform when the network is up again, just like a calendar app on a smartphone that is still usable while disconnected from its remote server.
Both ANG-MED and NAO RAPP applications are designed to handle connection errors with the RAPP platform. In addition, ANG-MED manages an embedded cache on the robot containing the profiles of recent users of the robot. ANG-MED stores locally activity data, at least until they can be pushed to the RAPP platform. The typical pattern is:

IF *the connection to the platform is ON, synchronize with the platform.*
ELSE *run using cached resources, and store data locally.*
ATTEMPT *a reconnection after 1, 2, 4, … 2^n seconds (with a maximum sleep time currently set to 30 seconds).*
WHEN *the connection is on again, ask for fresh data and send local history.*

A similar algorithm is performed on the ROWE connection between the ANG-MED RAPP application and the firmware.
The efficient implementation of these patterns necessitate that the RAPP application never blocks while trying to connect to the platform or to the firmware, and thus does never change state when a message is sent, but rather when the message is acknowledged by the peer.
Also, it is important for the RAPP application to try to synchronize with the peer when the connection is up again, by querying about the changes that may have happened while the connection was down. For example, when the

connection is established between the ANG-MED RAPP application and the firmware, the RAPP application requests the firmware to send up to date information about the current user, current caregiver, and robot state, since these parameters may have changed while the connection was down.
The firmware and RAPP application processes are monitored by a launch script that automatically restarts any of these processes following the detection of a failure.

4 Application Management

One of the key features brought by the RAPP project is the ability to select at run time which application will run concurrently on the robot and on the cloud platform, and trigger the installation and execution of this selected application. In the following section we describe a network boot proof-of-concept application that has been developed using HOP to demonstrate how a robot connects to the RAPP store, gets a list of candidate applications suitable for the robot, then selects, downloads and runs the appropriate application according to the user wishes.

4.1 Automatic discovery, robot personalization

Robots get their software from the RAPP store, depending on their capabilities and their environment.

The standard process of configuring a robot is based on a local configuration file which contains robot specific information (model, name, serial number, supported hardware peripherals) and the path to the RAPP store. Some other configuration data may be synthetized at boot time by a configuration script which purpose is to detect the robot hardware and nearby resources (such as cameras, sensors, spread in the environment). Nearby resources can be detected using standard discovery protocols such as mDNS, DNS-SD or the UPNP discovery protocol. All these protocols are supported by HOP.

The netboot POC simply reads a configuration file, which contains the robot identification data and the RAPP store URL. The robot information is consequently analyzed by the bootloader and store agent (which the robot connects to) and triggers the download and execution of the core RApp from the RAPP store. The core RApp is able to interoperate with the robot firmware (including optional components), remote objects and servers. The core RApp is also able to identify the robot user and context of use, and then download from the RAPP store and launch dynamic RApps according to user needs. In the POC, the core RApp provides a web user interface to present available RApps and let the user make his choice. A similar process runs concurrently on the RAPP platform.

Once a user has selected a RApp to execute, the application is downloaded to the robot, and the cloud part of the application is installed on the cloud platform.

4.2 Application download over an Internet connection

We provide several ways to enable the core RApp (on the robot) to download dynamic RApps from the RAPP store.

The standard technique is based on the HOP-hz packaging tools. An HOP-hz package is an archive file that may contain HOP executable files together with data files, shell scripts, and ROS executables. The HOP-hz API allows the core RApp to download a package and install a package to the robot, then to execute program files.

An alternative technique is based on the JavaScript require clause (node.js) which has been extended in HOP to require remote JavaScript modules by specifying the script URL instead of the module name. This technique is reserved to the installation of JavaScript or JSON files. If the application requires additional files (binaries, resources), these may be downloaded using the HOP file transfer API.

The proof of concept demo makes use of the “URL require” feature.
The definitive implementation of the application download service in RAPP consists in downloading a compressed archive file containing the application code and resource files, installing and executing the file on the robot. The file format is compatible with NPM (the Node.js package manager), the standard packaging tool for JavaScript applications.

5 Security and Privacy

5.1 Authentication/Authorization framework

Both the robot and the RAPP cloud platform protect themselves from unauthorized connections by standard authentication/authorization techniques.

Specifically, the robot and the platform mutually authenticate each other before accepting messages. Access to specific services is granted according to the authorization profile attached to the authenticated agent. HOP provides the infrastructure to implement the authentication/authorization framework.

Web access to the cloud platform (or to the robot) is also protected by authentication/authorization measures. This applies for example to caregivers who connect to the cloud platform or to the robot to control exercises, access user records and history. A fine grain authorization policy (per user/ per data/ per caregiver) could be set up in the future if needed.

RAPP prototypes use password based authentication and then token based authorization to access services. Meta service constructors relieve the developer from the burden of managing security at the service level.

5.2 Protection of communications

Communications between the robot and the cloud platform may in some circumstances (if not protected) be listened to or event tampered with by an attacker. We consider the security issues in two different deployment cases:

- *Site Local Deployment*. The robot and platform run on the same LAN, the LAN is trusted with secured wireless connections (WPA2), no Ethernet access granted to attackers, protection against Internet attacks (NAT + firewall in the router). No further protections are required to protect data transmission.
- *Full Deployment*. The LAN is not trusted, or communications between the robot and the platform cross the open Internet. Then communications shall be encrypted end to end using HTTPS or another appropriate encryption technology.

Similar protection measures apply to communications between the robot (cloud platform) and remote components (typically a web browser which a user would use to interact with the robot or cloud platform).

Robot (platform) file systems and internal communications between software components are not exposed to observation/tampering by unauthorized users. Both the robot and the platform must ensure that they do not accept connections from untrusted third parties, and that authenticated third parties connected to the robot or platform can only access to the proper network interfaces and resources, depending on their authorization level. Standard Unix security procedures shall be used to enforce the protection of “internal” resources.

5.3 RApp Sandboxing

Although the RApps initially developed within the project will all be trusted, we shall anticipate the future deployment of applications provided by untrusted third parties and ensure that these applications solely extend the robot capabilities and will neither harm the robot functions nor access to sensitive data.

Typical use cases of untrusted RApps are the deployment of an observation RApp (receive and process data, but do not activate robot functions), or customized control RApp (send high level control messages to the core RApp, knowing that the core RApp filters those messages and protects the robot firmware from undesirable ones).

The proposed solution leverages on Linux security to sandbox these applications and to control their access to resources and API.

Each untrusted dynamic RApp is launched as a separate process, attached to an unprivileged unix user, forbidding access to sensitive resources. RApps developed using HOP will run a dedicated HOP process. The core RApp acts as a trusted firewall to filter access to messages and services used by the untrusted RApp, for example preventing the untrusted RApp to tamper with the ROS infrastructure or the robot firmware.

This architecture pattern can be implemented using HOP since HOP allows fine grain control of service invocations. Finally, the protection against untrusted ROS nodes is not so simple as ROS does not yet provide fine grain control on protocol operation. Therefore we suggest that either untrusted ROS nodes run their own ROS graph in a separate virtual machine (a viable option on the Rapp platform, but not on the embedded side due to resources constraints), or wait for future improvements in ROS security.

6 Future Work and Conclusion

Beyond ANG-MED and the RAPP project, we plan to a second version of the ROWE client library (used on the ANG-MED rollator), allowing the connection to multiple concurrent processes, and providing improved control on the web socket connection status. This development is not required by the use case, and will not be done during the RAPP project, but it is definitely a wish for a number of applications.

Also we aim at developing higher level security functions to address access control requirements, and experiment with the sandbox architecture.

Work conducted on task 2.5 has demonstrated that the RAPP architecture can be implemented efficiently by integrating two complementary frameworks: ROS and HOP, and using extensively the Web application architecture paradigm. The ROWE Client library (C code) has been developed to integrate robotic firmware and other non ROS components with HOP. HOP modules have been developed to support the ROS and Rowe protocols.

Design patterns and examples of use have been developed to demonstrate distributed communications and application downloading from the RAPP store.
Robustness requirements of distributed applications have been addressed and led to the development of dedicated design patterns, using asynchronous communications, temporary caches, and watch dog applications.

7 Annexes

The following code modules are part of the deliverable. All these modules are available on the project Github repository:

- Rowe C client library https://github.com/rapp-project/rowe
- rowe.js HOP module https://github.com/rapp-project/rowe/blob/master/src/rowe.js

- RosBridge.js HOP module https://github.com/rapp-project/hop-utilities/tree/master/hop_samples/rosbridge
- Hop examples of use of the above components https://github.com/rapp-project/hop-utilities/tree/master/hop_samples/demos
- Netboot example of HOP based network boot of the robot, web based user selection of a distributed RApp, download from the RAPP store, installation on the robot and on the RAPP cloud platform and execution. https://github.com/rapp-project/hop-utilities/tree/master/hop_samples/netboot