Journal of Software Engineering for Robotics 6(1), December 2015, 1-14
ISSN: 2035-3928

ROS As a Service: Web Services for Robot Operating System

Anis Koubaa [1,2,*]

[1] Prince Sultan University, Saudi Arabia
[2] CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal

Abstract—Robot Operating System (ROS) represents nowadays a defacto-standard for the prototyping and development of software for robotics applications. It presents several software abstraction layers to robotic hardware resources allowing software developers to focus more on the software development without having to worry about the low-level hardware programming. However, developing client-side applications requires a full understanding and mastering of ROS and robotics, which is not straightforward for non-ROS users and for beginners. In addition, with the emergence of the Internet-of-Things (IoT) there is a increasing interest in providing Web interfaces that allow users to seamlessly access robots through the Internet. To address those issues, there is a need to add a software abstraction layer on top of ROS to allow a seamless interaction with ROS-enabled robots. In this paper, we leverage the use of Web services to provide new programming abstraction layers to ROS based on SOAP and REST Web services. The innovation of this paper consists in proposing software architecture to expose ROS resources as SOAP and REST Web services. In particular, we present an object-oriented design of software meta-models for the integration of both Web services into ROS and we validate it through a real implementation on a service robot. The experimental validation demonstrates how ROS Web services promote portability, reusability, and interoperability of ROS-enabled robots with client applications. We considered the ROSJAVA client Application Program Interface (API) to implement the proposed meta-models using JAX-RS and JAX-WS Web services APIs. To the best of our knowledge, this is the first work that addresses the integration of Web services into ROS, in particular with the use of ROSJAVA client API, and we believe that it will open new insights towards the emerging concept of Cloud Robotics.

Index Terms—Robot Operating System (ROS), Web services, SOAP, REST, Object-oriented design, Software meta-model.

1 Introduction

The design and development of efficient software for mobile service robots is highly challenging and tedious. It requires cross-disciplinary knowledge from low-level hardware drivers to high-level software abstractions going through the design of complex signal processing algorithms. Fortunately, the emergence of robotics middleware platforms such Robot Operating Systems (ROS) [1], Player [2], OROCOS [3], YARP [4] and others provided one step towards a more affordable approach for robotics software development. In fact, these middlewares typically provide hardware abstraction layers and allow developers to be more dedicated on high-level component-based software development with an emphasis on modularity and reuse. Nowadays, ROS represents the most attractive platform for mobile robot software development and is supported by a large community of users and developers. One major benefit of using ROS is that it provides an abstraction layer to hardware resources (i.e. sensor and actuators) and exposes any physical data as a labeled data stream, called *topic*. This allows software developers to focus more on the high-level development without having to deal with hardware issues. In addition, ROS integrates a comprehensive set of libraries of open-source software components such as openCV for computer vision algorithms, PCL for Point Cloud, gmapping for Simultaneous Localization and Mapping (SLAM), openni for 3D sensing, and supports a wide range of sensors and robotics platforms. Building complex applications with ROS becomes a matter of effectively combining different modules together. This reduces a lot the complexity of robotics software construction; however, the integration of the different software components still requires advanced skills in both robotics and software engineering, and thus remains only limited to experts. This represents one of the limitations that hindered the public and large-scale deployment of service robots because of being

Regular paper – *Manuscript received August 19, 2015; revised October 12, 2015.*

- *This work was supported by the myBot project entitled MyBot: A Personal Assistant Robot Case Study for Elderly People Care under the grant (No.75-34) from King AbdulAziz City for Science and Technology (KACST).*

too much complex to program by non-roboticians experts. In fact, programming client applications that interact with the ROS-enabled robots also requires a deep understanding of the robot internal software architecture and ROS platform due to the lack of abstract interfaces that encapsulate implementation-specific details and hide them from the client applications.

In this paper, we address this problem and we aim at developing a software meta-model to integrate Web services into ROS to provide new types of interfaces that makes easier the interaction with ROS and the development of platform-independent client applications for ROS-enabled robots. The proposed approach leverages the Web services technology by defining a Service-Oriented Architecture (SOA) framework for the integration of Web services into ROS to expose ROS ecosystem resources as Web services. The innovation of this work does not lie in the concept of providing abstractions to robots, as this was addressed in several previous works including [5], [6], [7], but lies in the proposal of an integration process and software meta-models to provide new abstractions to ROS using SOAP and REST Web services. From a practical perspective, the main challenge was the integration of the ROSJAVA client API with the `JAX-WS` and `JAX-RS` Web services' APIs to provide a systematic approach for the implementation of the proposed ROS Web services meta-models. One major non-functional requirement in the implementation is not to rely on third-party Web services engines, like `Glassfish` or `Tomcat`, but using the `JAX-WS` Web services' server API to run and deploy the designed ROS Web services. The objective is to be independent from any particular provider, and to ensure the deployment of Web services on any robotic platform with Java Virtual Machine (JVM) without additional provider-specific engines.

The rest of this paper is organized as follow. Section 2 discusses the state-of-the-art and emphasizes the contribution of this paper as compared to previous works. Section 3 presents a background on the main concepts of ROS. Section 4 presents the ROS Web services contribution; it first outlines the objectives, and then presents the integration process and the software-meta models that integrate SOAP and REST Web services into ROS. An experimental validation that illustrates a case study implementation of the SOAP and REST meta-models for a robot delivery application and the lessons learned are presented in Section 5. Section 6 concludes the paper.

2 Related Works

Recently, there has been a growing interest in integrating robots into the Internet, and in particular to the emerging cloud computing and Internet-of-Things (IoT) paradigms. In 2010, reference [7] is one of the first papers that specified the concept of Robot as a Service (RaaS). Yinong et al. proposed a cloud framework for interacting with robots in the area of service-oriented computing. The authors exploited the SOA to design and implement a prototype of the Robot as a Service (RaaS) cloud computing model, which consists in exposing robotic services and applications, and allows interaction with developers and end-users. The design complies with the common service standards, development platforms, and execution infrastructure, following the Web 2.0 principles.

Since then, there have been several interpretations on how to specify a Robot as a Service using different Internet and cloud computing technologies. In [5], Osentoski et al. proposed the `rosjs` and `rosbridge` middleware. This work represents a milestone in the integration of ROS into the Web and the Internet. The motivation behind `rosbridge` and `rosjs` is mainly two folded: (1) to use commonly available Internet browsers for non-roboticians users to interact with a ROS-enabled robot (2) to provide Web developers with no background on robotics with simple interfaces to develop client applications to control and manipulate ROS-enabled robots. The core concept relies on the design of `rosbridge` as a middleware and an abstraction layer on top of ROS acting as a mediator between the front-end Web applications based on the JavaScript technology (known as `rosjs`) and the back-end ROS ecosystem. `rosbridge` enables a user to interact with ROS topics and services using JavaScript primitives. It also uses JSON serialized messages to interact with the Web interface, which are converted to/from ROS serialized messages to interact with the ROS back-end. This work represents an important milestone towards providing new Human-Robot Interfaces (HRI) opening new horizon towards a more popular deployment and use of mobile robots in the context of education and enabling remote laboratories. In addition, `rosbridge` and `rosjs` are designed for Web applications based on JavaScript, and also provide common interfaces to non-Web clients based on sockets (version 1.0) and Web sockets (version 2.0). The ROS Web services that we present in this paper represents a complementary alternative with respect to `rosbridge` to integrate ROS-enabled robots into the Internet, as we rely on Web services' technologies rather than on JavaScript Web technology to define the abstraction layers on top of ROS. In addition, we do not develop or use any specific middleware, like `rosbridge`, but we simply rely on the Commercial-Off-The-Shell (COTS) Web services' engines (i.e. `Glassfish`, `Tomcat`, in our case `JAX-WS`).

In [8], Kato et al. proposed the Rsi Research cloud, which is a cloud computing platform that offers robotic services through the Internet. It allows for the integration of robot services in the Internet and the composition of robot services. The authors exploited the Robot Service Network Protocol (RSNP) to hide robots complexities and expose robotic services to non-experts in robots. They have tested their proposed architecture by implementing a surveillance camera service. The authors did not present any software architecture and did not use Web/Web service technologies for the integration.

In our earlier work [9], we proposed RoboWeb, a SOAP-based service-oriented architecture that virtualizes robotic hardware and software resources and exposes them as services through the Web. RoboWeb consists in the integration of different Web services technologies with the (ROS) middleware to allow for different levels of abstractions (multi-layer architecture). It was designed to develop a remote lab which allows remote researchers and students to access and monitor robots that belong to the framework. We validated our proposed architecture by developing a web interface that allows non-technical users to access, control and monitor robots that belong to the framework. The ROS Web services proposed in this paper differ from the RoboWeb system in several fronts. In fact, the SOAP Web service in RoboWeb was not integrated inside ROS, but was developed externally to the ROS ecosystem to provide a Web service interface to the robot resources. However, in this paper, we integrate Web services into ROS through an object-oriented design of software meta-models. In addition, REST Web services were not considered in RoboWeb. Furthermore, the implementation of the ROS Web services' meta-models relies on `ROSJAVA` technologies and, `JAX-WS` and `JAX-RS` Web services APIs, which represents an additional difference with respect to [9]. In fact, the meta-models we propose in this paper are more generic, modular, embedded into ROS, and are based on completely different technologies, namely `ROSJAVA`.

Some other recent approaches focused on REST Web services to provide Web interfaces to robots and to ROS.

In [10], Souza et al. presented an approach for adopting REST Web services for networked robotics. The motivation was to overcome the limitations of RPC-based communications, like the absence of multiple operations over a single HTTP transaction and support for asynchronous operations. They evaluated their approach on the FastSlam algorithm and compared it to the RPC approach. Our work differs from the two latter in that we propose SOAP and REST Web services for the ROS middleware and provide a modular software meta-model in a unified framework that makes this integration more effective, with the objective to promote Software as a Service for future cloud robotics.

Later, in [11], Safaripour et al. proposed a RESTfull architecture for enabling rapid development of companion robot applications. The architecture consists of two layers: (*i.*) an *Application layer*, that includes the core logic and is composed of the client applications, (*ii.*) a *Service layer* that defines the RESTfull Web services, which is also divided into high-level and low-level layers, to increase the flexibility and maintainability of the system. A good contribution of this work is the proposal of REST interfaces and resources to define REST services for robots. They validate their proposal through an implementation on a Lego robot. The work in [11] contributed to the design of REST interfaces, but SOAP was not considered. SOAP Web services have the advantage of providing a formal and standard description of services, using the Web Services Description Language (WSDL) XML-based standard. In addition, the REST Web services are not designed for ROS but for generic robots, although the concept can also be specified for ROS.

In [6], Kehoe recently developed `ROStful`, an open-source implementation of a lightweight Web server that operates over `rosbridge` to support REST Web services. The lightweight Web server exposes ROS topics, services and actions through RESTful Web services. The motivation of the `ROStful` work is to integrate Software as a Service concept into robotics. The proposed REST server makes use of JSON serialization of `rosbridge`. However, it differs from `rosbridge` in that the latter relies on Web sockets whereas `ROStful` is based on REST Web services. In [6], there is no software architecture or a process description that demonstrates how the integration of REST Web services was performed, but rather an implementation of a `rosbridge`-compatible Web server to allow for processing REST requests over `rosbridge`. Our work differs from [6] by proposing a unified framework for both SOAP and REST Web services, and that is completely independent from `rosbridge`. In addition, we use `ROSJAVA`, `JAX-WS` and `JAX-RS` as enabling technologies for the integration of Web services into ROS.

The Unmanned Aeriel Vehicles (UAVs) area also attracted recent research works on SOA and REST Web services. In [12] Mahmoud and Mohamed proposed a SOA model for collaborative UAVs. A mapping between cloud computing resources and UAVs' resources was presented. Furthermore, essential services (e.g. mission organization service, broker service, ground station commands, etc.) and customized services (e.g. sensing services, actuation services, image/data analysis, etc.) were proposed. The paper only provides high-level description of system architecture, components and services without any specific detail on how these could be implemented on a real system. In addition, the proposed architecture is too generic so that it can applied to any type of robots and not specific to UAVs. In [13], the same authors elaborated more on their previous work [12] and designed a RESTful web services model by following a Resource-Oriented Architecture (ROA) approach to represent the resources and services of UAVs. In addition, a broker that dispatches mission requests to available UAVs was proposed. The broker is responsible for managing the UAVs, their missions and their interactions with the client. A small prototype was implemented on an Adruino board that emulates a UAV and its resources. The limitation of this work is that the cloud is used only to visualize the resources of the UAVs, and is not used for performing extensive computations. In addition, the prototype implementation is very limited as it does not demonstrate a sufficient proof of concept on real drones or robots, but on a simple Arduino broad. Thus the feasibility of the approach was not effectively demonstrated.

In [14], Mahmoud et al. extended their previous work with a more elaborated implementation and performance evaluation study. The experimental testbed for emulated Arduino-based on-broad UAV system were used, and several sensors were used including temperature and humidity, ultrasonic for distance measurements. For the Internet connectivity, an Adafruit

CC3000 Wi-Fi board was used to connect the Arduino to the Internet. RESTful web services were defined and implemented for manipulating each type of sensor through a Web interface. The performance evaluation study shows the response times of the HTTP requests to UAV resources were in the order of 266 ms for the different types of sensors. However, the experimental systems remains limited in terms of studying the scalability issues and also applied to a small local network.

Other recent works adopted a more generic SOA approach for accessing ROS-enabled robots without using Web services. In [15], Brugali et al. proposed the Task Component Architecture (TCA), which is an object-oriented software architecture for the seamless integration of the Service Component Architecture (SCA) into robotic software control systems to execute asynchronous tasks. The authors also integrated TCA with ROS using a `ROSProxyNode` developed with `ROSJAVA`. The TCA wrapper layer of ROS nodes allows a complete management of ROS resources (i.e. nodes, topics, and services). Our work differs from [15] in that we use Web services technologies for enabling SOA in ROS, whereas [15] leveraged the SCA for defining a service-oriented interfaces to robots and ROS.

This paper proposes a new *service-oriented* approach by providing generic Web services interfaces to external clients to seamlessly interact with ROS. The main contribution consists in the design of software meta-models for adding a new abstraction layer to ROS using SOAP and REST Web Services. We propose a new layer referred to as *ROS Web Services* to expose the whole ROS ecosystem as a *service* to client applications. We develop a SOAP Web service API (ros-ws) and a REST Web service API (ros-rs), which are the two major architectural models for SOA. ROS Web services allows any client application on any platform to interact with ROS simply by invoking the Web Services in exactly the same way as invoking traditional Web services. In fact, Web services are platform-independent, and supported by most of commonly used programming languages and operating systems. We also discuss the main real-world challenges and practical considerations that must be taken into account for a successful integration of Web services into ROS ecosystem, as this integration is not trivial.

3 Robot Operating System (ROS)

In this section, we present a general overview of the basic concepts of ROS framework [1] to provide the required background needed to understand the software architecture proposed in this paper.

ROS acts as a meta-operating system for robots as it provides hardware abstraction, low-level device control, inter-processes message-passing and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. The main advantage of ROS is that it allows manipulating sensor data

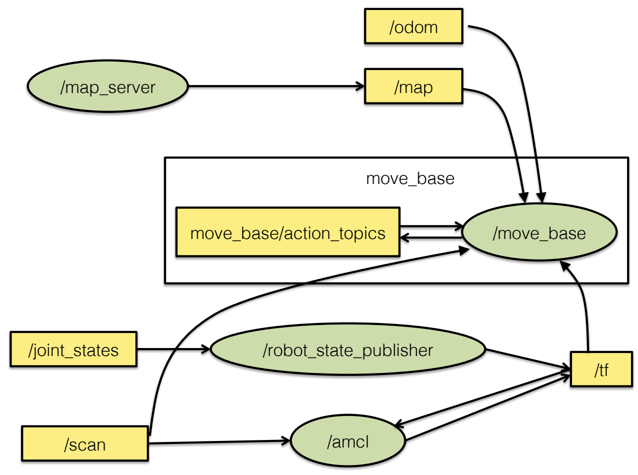


Fig. 1. Example of a ROS Computation Graph. An ellipse represents a node, and a rectangle represents a topic

of the robot as a labeled abstract data stream, called topic, without having to deal with hardware drivers. This makes the programming of robots much easier for software developers as they do not have to deal with hardware drivers and interfaces. It is useful to mention that ROS is not a real-time framework, though it is possible to integrate it with real-time code.

ROS relies on the concept of computational graph, which represents the network of ROS processes (potentially distributed across machines). An example of a simplified computation graph is illustrated in Fig. 1.

A process in ROS is called a *node*, which is responsible for performing computations and processing data collected from sensors. As illustrated in Fig. 1, a ROS system is typically composed of several nodes (i.e. processes), where each node processes a certain data. For example, $move\_base$ is a node that controls the robot navigation, $amcl$ is another node responsible for the localization of the robot, and map_server is a node that provides the map of the environment to other processes of the system. Nodes are able to communicate through message-passing, where a *message* is a data structure with different typed-fields. This communication between nodes is only possible thanks to a central node, referred to as *ROS Master*, which acts as a name server providing name registration and lookup for all components of a ROS computation graph (e.g. nodes), and store relevant data about the running system in a central repository called *Parameter Server*.

ROS supports two main communication models between nodes:

- The $publish/subscribe$ model: in this model nodes exchange *topics*, which represents a particular flow on data.

One node or several nodes may act as a publisher(s) of a particular topic, and several nodes may subscribe to that topic, through the *ROS Master*. Subscriber and publisher nodes do not need to know about the existence between other because the interaction is based on the topic name and made through the *ROS Master*. For example, in Fig. 1, the map_server is the publisher of the topic $/map$, which is consumed by the subscriber node $move\_base$, which uses the map for navigation purposes. $/scan$ represents the flow of data received from the laser range finder, also used by $move\_base$ node to avoid obstacles. $/odom$ represents the control information used by $move\_base$ to control robot motion.

- The $request/reply$ model: in this model, one node acts as a server that offers the service under a certain name, and receives and processes requests from other nodes acting as clients. Services are defined by a pair of message structures: one message for the request, and one message for the reply. Services are not represented in the ROS computation graph.

4 ROS As A Service: Integration of Web Services into ROS

4.1 Objectives

ROS As A Service (RoAAS) was designed to make an abstraction of ROS resources, including topics, services and actions for developers who do not have prior background on robots or on ROS. We retain three major advantages behind exposing ROS as a service following an SOA approach:

- *Promote public use of robots:* First, non-roboticians programmers, like Web and mobile applications developers, will be able to develop applications that communicate with ROS-enabled robots through a service layer. Web services are a possible instance of the service layer that can be used as a mediator between client applications and the ROS ecosystem. In addition, users will have the capability of directly interacting with robots through the Internet, and this will enlarge the community of users and applications' developers.
- *Robot-Cloud integration*: A second advantage is the integration of the robots into the cloud; The robots can expose their resources as services through the cloud that allows users to virtually access these resources to perform some actions or monitor the robots' status. For example, in case of surveillance applications, an end-user may want to send a set of UAVs for a mission and control their missions remotely through the cloud. The internal status of the UAVs and their observations will be provided as services to the end-user by the cloud back-end.
- *Standard and unified interfaces*: it will be possible to develop unified and standard client applications that seamlessly interact with heterogeneous robots provided that they share the same service-oriented interfaces, independently of the implementation details. Web services are known to be quite effective in meeting this requirement.

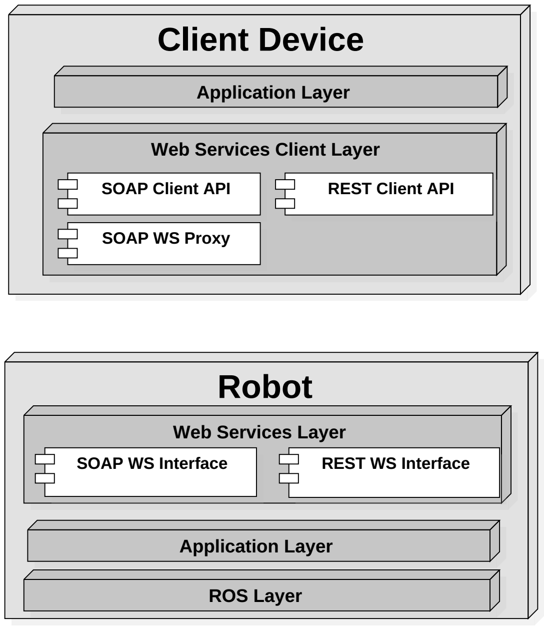


Fig. 2. Deployment Diagram of ROS Web Services

To address these objectives, we propose to use Web services as an additional abstraction layer on top of ROS. We design generic software meta-models for the integration of SOAP and REST Web services into ROS, which represent the two fundamental architectural models for SOA. In addition, we propose a systematic approach to develop concrete implementations of the meta-models for SOAP Web services (ros-ws) and a REST Web services (ros-rs). ROS Web services allow any client application on any platform to interact with ROS simply by invoking the ROS Web services in exactly the same way as invoking traditional Web services. In fact, Web services present the advantage of being platform-independent, and also supported by most of commonly used programming languages and operating systems. In what follows, we present the architecture of ROS Web services (ROS-WS) and explain how to integrate them into ROS.

4.2 System Architecture

Figure 2 depicts the deployment diagram of ROS Web services and illutrates the integration of the Web services' layers into the ROS-enabled service robot and the client device.

The Web services represent an intermediate layer that allows seamless interaction between client applications and ROS ecosystem in the service robot. In our architecture, we propose to define both SOAP and REST Web services to offer client applications' developers a greater flexibility to choose the most appropriate Web service interface for their application. In fact, on the one hand SOAP provides a well-defined contract-based specification of the services using the Web Services Description Language (WSDL) and a standardized message exchange protocol using the SOAP envelop. On the other hand, REST is a lighter weight Web service solution with no formal service specification and provides the ability to the client to interact with the robot through the basic HTTP protocol from Web browsers. The choice of the interface is relative to the client objectives and requirements. The reader may refer to [16] for a thorough discussion and analysis about the comparison between REST and SOAP Web services.

The Web service layer in the robot side must ensure the exposure of ROS topics, services and actions as Web services to the clients. As the the Web service layer is an intermediate layer between ROS and the clients, it should be able to (*i.*) subscribe to and publish any ROS topic, action or service (*ii.*) deliver the ROS messages to clients subscribing to a particular topic, and (*iii.*) forward messages received from the client acting as publishers of ROS messages to ROS. The same should also hold for ROS service invocation. This means that for any ROS topic, action or service that must be exposed, the Web service should instantiate a ROS subscriber and/or a publisher for the topic or the service of interest. Then, Web methods and interfaces should be defined according to which functionalities the robot would like to expose and execute. We present more details about the software architecture of underlying technologies in the next sub-section.

4.3 Software Architecture

4.3.1 Technology Design Choices

To integrate Web services into ROS, we faced the challenge of choosing the most appropriate technology to build the software system and design its architecture. We have opted for the use of Java as a Web service programming language, as it provides a native and advanced support of SOAP and REST Web services, although they are programming-language-independent and platform-independent. However, Java EE provides standard APIs for SOAP and REST Web Services, known as JAX-WS and JAX-RS specifications, respectively. Python also provides REST Web service support, but much less than Java for SOAP Web services.

This choice wouldn't have been possible without the fortunate existence of the still-in-progress ROSJAVA, which is a Java API that defines a ROS client library that allows ROS developers to write ROS programs in the Java language and interact with the ROS Master. It has to be noted that ROS mainly relies on its C++ (roscpp) and Pyhton (rospy) client APIs for developing ROS programs. ROSJAVA is relatively recent and its purpose was mainly to extend ROS capabilities to be integrated into mobile applications, through the `android_core` API that extends ROSJAVA to write ROS client programs for Android devices.

In this paper, we took advantage of all these capabilities to develop Web Services' interfaces using the powerful features of Java EE in combination with ROSJAVA that allows us to integrate Web Services with ROS in an elegant fashion. To the best of our knowledge this is the first work that proposes Web Services integration into ROS.

4.3.2 Software Meta-Models

The UML class diagrams of SOAP Web Services (`ros-ws`) and REST Web Services (`ros-rs`) for ROS are presented in Figures 3 and 4.

The UML class diagrams in Figures 3 and 4 can be considered as generic *meta-models* that can be instantiated for any type of robots. We used the generic term `Robot` as a prefix in the name of the classes, but it can be replaced with the name of a concrete robot when instantiated for a real robot. There are mainly three software packages: (*i.*) ROSJAVA, (*ii.*) SOAP Web services, and (*iii.*) REST Web services.

ROSJAVA package: The `ROSJAVA` package contains classes purely written in ROSJAVA and does not include any Web service functionality. The objective of the classes in this package is to create publishers and subscribers of all ROS topics and ROS services that will later be exposes as services. This is a major requirement as it was explained in Section 4.2. `ROSJAVA` package contains the class `AbstractNodeMain` class that represents the superclass of any ROS node to be created with ROSJAVA. In other words, any ROSJAVA node should extend the ROSJAVA superclass `AbstractNodeMain` and override the method `onStart()` that represents the entry point of the node. Note that `AbstractNodeMain` is an abstract class defined in the `ROSJAVA` client API. In our meta-model, we create one class that represents the ROS node to handle the topic and service subscribers called `RobotSubscribersNode`, and another class for the node `RobotPublishersNode` that handles publishing all topics and services of interest. These two classes will be reused in the classes defining the Web services and will provide a bridge between the ROS ecosystem and the Web services interfaces to be defined in the SOAP Web services package.

SOAP Web services package: The SOAP Web services package, that we call `ros-ws`, contains classes that expose the ROS topics and services specified above as a Web service. The Web services are defined through generic Java interfaces for both subscribers and publishers Web services, namely `RobotSubscribersWebServiceInterface` and `RobotPublishersWebServiceInterface`. These Java interfaces are used to define the contract of the SOAP Web services and help in generating a WSDL document independent of the implementation details. These interfaces are implemented by the Web Services concrete classes `RobotSubscribersWebService` and `RobotPublishersWebService`. These classes provide the implementation of the Web methods that

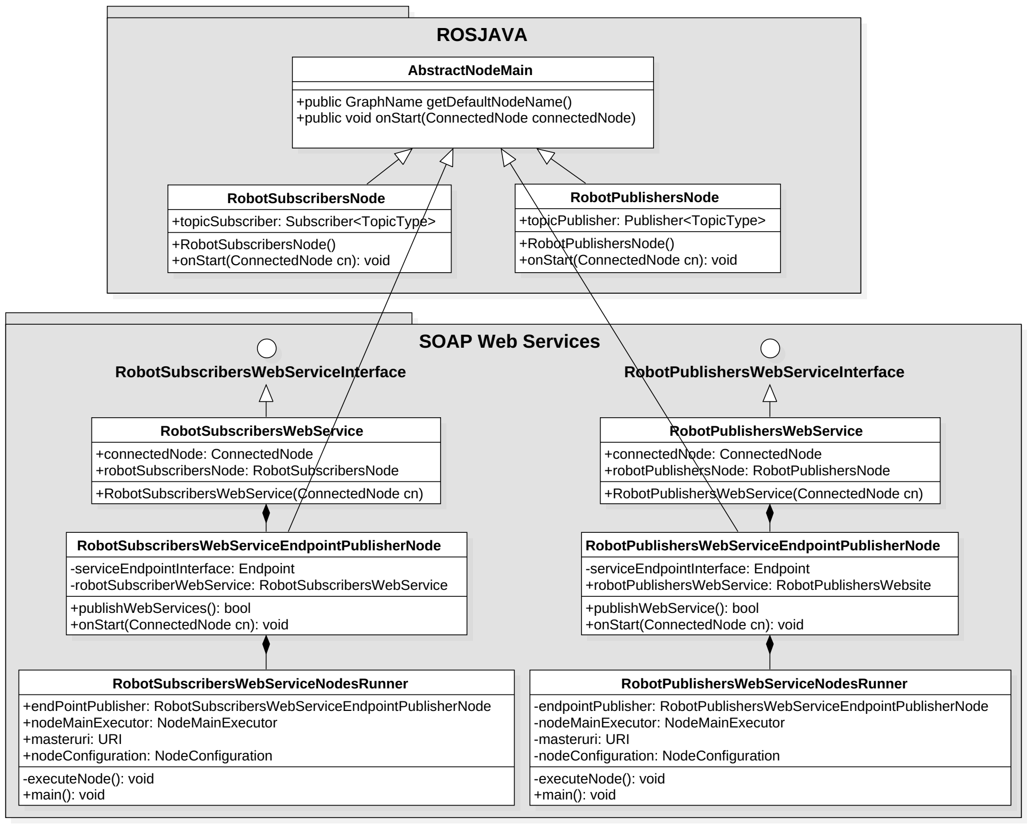

ROSJAVA
AbstractNodeMain
+public GraphName getDefaultNodeName()
+public void onStart(ConnectedNode connectedNode)
RobotSubscribersNode
+topicSubscriber: Subscriber<TopicType>
+RobotSubscribersNode()
+onStart(ConnectedNode cn): void
RobotPublishersNode
+topicPublisher: Publisher<TopicType>
+RobotPublishersNode()
+onStart(ConnectedNode cn): void
SOAP Web Services
RobotSubscribersWebServiceInterface
RobotPublishersWebServiceInterface
RobotSubscribersWebService
+connectedNode: ConnectedNode
+robotSubscribersNode: RobotSubscribersNode
+RobotSubscribersWebService(ConnectedNode cn)
RobotPublishersWebService
+connectedNode: ConnectedNode
+robotPublishersNode: RobotPublishersNode
+RobotPublishersWebService(ConnectedNode cn)
RobotSubscribersWebServiceEndpointPublisherNode
-serviceEndpointInterface: Endpoint
-robotSubscriberWebService: RobotSubscribersWebService
+publishWebServices(): bool
+onStart(ConnectedNode cn): void
RobotPublishersWebServiceEndpointPublisherNode
-serviceEndpointInterface: Endpoint
+robotPublishersWebService: RobotPublishersWebsite
+publishWebService(): bool
+onStart(ConnectedNode cn): void
RobotSubscribersWebServiceNodesRunner
+endPointPublisher: RobotSubscribersWebServiceEndpointPublisherNode
+nodeMainExecutor: NodeMainExecutor
+masteruri: URI
+nodeConfiguration: NodeConfiguration
-executeNode(): void
+main(): void
RobotPublishersWebServiceNodesRunner
-endpointPublisher: RobotPublishersWebServiceEndpointPublisherNode
-nodeMainExecutor: NodeMainExecutor
-masteruri: URI
-nodeConfiguration: NodeConfiguration
-executeNode(): void
+main(): void

Fig. 3. UML Class Diagram of SOAP Web Services for ROS

are exposed to and invoked by the Web services' clients. A Web method acts as a *mediator* that interacts directly with the ROS ecosystem through the instanciated objects of the classes `RobotSubscribersNode` and `RobotPublishersNode`, and then they return the result of the service execution to the Web service client that invoked the method. Once the Web services are defined, they must be published over the Internet to be accessible to client applications. In Java, the typical way to publish a Web service, without an integrated Web services platform (such as `Glassfish` or `Tomcat`), is performed through the `Endpoint` class with its static method `publish(String address, Object implementor)`, where `implementor` is an instance of the Web service class created above. This is performed in the classes `RobotSubscribersWSEndpointPublisherNode` and `RobotPublishersWSEndpointPublisherNode`. These classes act as Web services' proxies that expose the Web services to the client, and at the same time a ROS node as they extend the class `AbstractNodeMain`. One trick is the fact that in the `onStart(ConnectedNode cn)` methods of the Endpoint publishers, the `ConnectedNode cn` is passed as a parameter to the constructors of `RobotSubscribersWebService` and `RobotPublishersWebService`, which in turn will use this `ConnectedNode` reference object to execute the `onStart(ConnectedNode cn)` methods of `RobotSubscribersNode` and `RobotPublishersNode`, needed for these nodes to launch the ROS subscribers and publishers and to communicate with the `ROSMaster`. This is a kind of a cascade reference for the object `ConnectedNode cn` to have a reference to only one node in ROS responsible for (*i.*) interacting with ROS and also (*ii.*) for publishing the Web services. This is an important trick in the integration of SOAP Web services with ROS using ROSJAVA framework. Finally, we define two runner classes `RobotSubscribersWebServiceNodesRunner` and `RobotPublishersWebServiceNodesRunner`, which is typical in ROSJAVA environment, to execute the `Endpoint` publishers' nodes with the `main()` method as the entry point (not the `onStart()` method like in ROS nodes). It is still possible to define the `main()` method in the `Endpoint` publisher nodes, but from software engineering design perspective the seperation allows for better code modularity and reuse.

REST Web services package: In contrast to SOAP Web services, which contains two main classes for publishers and subscribers, The REST Web services package, that we call

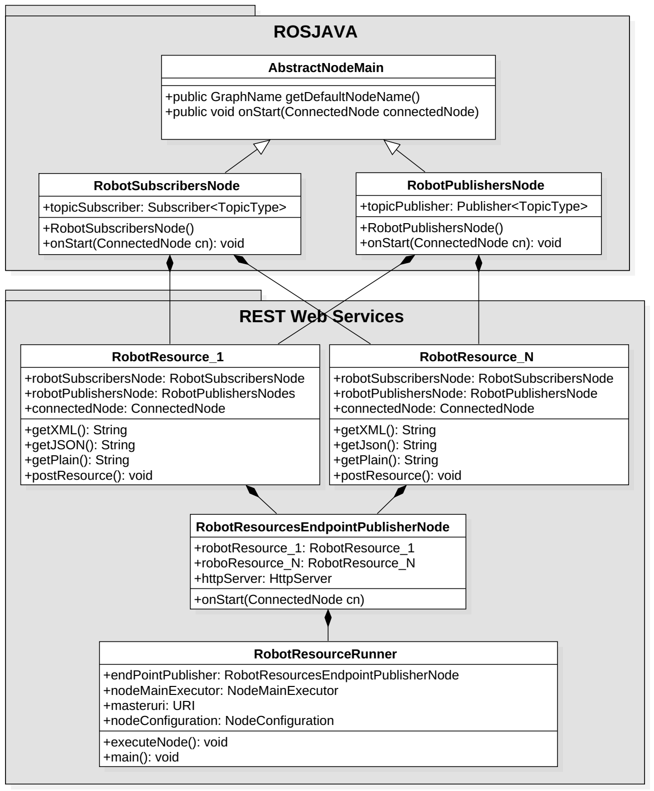

ROSJAVA
AbstractNodeMain
+public GraphName getDefaultNodeName()
+public void onStart(ConnectedNode connectedNode)
RobotSubscribersNode
+topicSubscriber: Subscriber<TopicType>
+RobotSubscribersNode()
+onStart(ConnectedNode cn): void
RobotPublishersNode
+topicPublisher: Publisher<TopicType>
+RobotPublishersNode()
+onStart(ConnectedNode cn): void
REST Web Services
RobotResource_1
+robotSubscribersNode: RobotSubscribersNode
+robotPublishersNode: RobotPublishersNodes
+connectedNode: ConnectedNode
+getXML(): String
+getJSON(): String
+getPlain(): String
+postResource(): void
RobotResource_N
+robotSubscribersNode: RobotSubscribersNode
+robotPublishersNode: RobotPublishersNode
+connectedNode: ConnectedNode
+getXML(): String
+getJson(): String
+getPlain(): String
+postResource(): void
RobotResourcesEndpointPublisherNode
+robotResource_1: RobotResource_1
+roboResource_N: RobotResource_N
+httpServer: HttpServer
+onStart(ConnectedNode cn)
RobotResourceRunner
+endPointPublisher: RobotResourcesEndpointPublisherNode
+nodeMainExecutor: NodeMainExecutor
+masteruri: URI
+nodeConfiguration: NodeConfiguration
+executeNode(): void
+main(): void

Fig. 4. UML Class Diagram of REST Web Services for ROS

`(ros-rs)`, defines a class for each particular resource. Any ROS resource (topic, action or service) that must be exposed as a REST Web service must be defined into a specific class that represents this particular resource. For this reason, there will be as many classes as the number of resources that must be exposed to public. Each resource will be callable through the four traditional `HTTP` methods, namely, `GET`, `POST`, `PUT`, and `DELETE`. The operations related to each of these methods is left to developer's own choice based on the application logic he would like to implement. For each method, it is possible to specify a particular format for returning the result to the calling client, namely as a plain text, or JSON format or XML format, which provides multiple ways to exploit the data at the client side. This is illustrated in the resource classes `RobotResource_i`, where i is the index of the resource. To expose these resources over `HTTP`, an `HttpServer` object is created in the class `RobotResourceEndpointPublisherNode`, which launches the HTTP server and makes reference to the resources created in the resources' classes. As such, these resources will be accessible through the REST Web service model using the HTTP protocol as a transport protocol, and resources are accessed according to the specified path in the resource class. Finally, the `RobotResourceRunner` will run the `RobotResourceEndpointPublisherNode` node to execute the HTTP server and start receiving requests.

An important point in the design of REST Web services is the design of resources paths and names. Although there is no formal specification that governs the design of the resources' paths and names, a natural design choice in the context of ROS would be to use the representative and unified naming structure for typical topics and services to define the path to a resource of interest. In ROS, the velocity topic can have different names such as `cmd_vel` or `cmd_vel_mux` or other more complex names. It is not appropriate to use the topic name as is to define the resource name, as the topic name is subject to changes, whereas a good design choice is to keep the name of the resource fixed as a unified interface to the client application independently of the implementation details. For that purpose, it will be more appropriate to use a specific keyword such as `/velocity` to represent a `velocity` resource, independently of the real name of the topic, which is implementation dependent. This allows for a better reuse of the REST resources through standard interfaces for heterogeneous robots. In Table 1, we propose a sample of possible REST resources for some common topics, services and actions in ROS. For example, `/robot/pose` allows a user to retrieve the pose of the robot by making an HTTP request using the `GET` method. Other actions and topics may also use the other HTTP methods such as `PUT`, `POST` and `DELETE` to perform specific operation, such as for instance DELETE can be used to cancel a goal sent to the robot, while it is executing it.

In the next section, we validate our architecture through an experimental deployment of REST and SOAP web services into a Turtlebot service robot.

5 Experimental Validation

In this section, we present two illustrative examples that demonstrates how to integrate Web services into ROS in a real-world application of a service robot. The application consists of a service robot based on the Turtlebot 2 platform that we developed in the context of the MyBot funded project [17]. The service robot prototyped to provide services to faculty members and students in Prince Sultan University (PSU). We have used the Robot Operating System (ROS) development framework for developing software modules and applications of the service robot. A video demonstration showing the deployment of courier delivery application (without using Web services) is presented in the following video [18]. A video demonstration that presents the integration of SOAP Web service into ROS is available on this YouTube link [19].

5.1 SOAP Web Services for Delivery Application

The use case consists in sending a command to the robot to perform a courier/coffee delivery between offices and from the Cafe of the University, respectively. We will only focus on the Web services interfaces developed and their integration into ROS, and we will not describe how the back-end application was implemented in ROS. The reader may refer to [20] for more details about the back-end implementation. Listing 1 presents an instantiation of the class RobotPublishersNode as TurtlebotPublishersNode.

This class advertises the ROS topic /DeliveryRequestMsg/from_json. When a message of type DeliveryRequestMsg is published on the topic /DeliveryRequestMsg/from_json, the robot will start executing the delivery mission. Depending on the mission code, the robot will deliver either a coffee or a courier. Now, the question how to send this request through a Web service client? So, the objective is to expose the publisher of the topic /DeliveryRequestMsg/from_json as a Web service so that it can be invoked by a Web service client to send a request to the robot for courier or coffee delivery.

According to our architecture in Figure 3, we first declare a Java interface for the Web Service, and this represents a good practice in Web Services design so as to separate the implementation of the Web service from its interface (i.e. description). The Web service interface is presented in Listing 2. The interface contains one Web method exposed to the Web service clients. The method ExecuteDeliveryRequest(...), which takes a DeliveryRequestMessage as a parameter and is responsible for executing the request for delivering a coffee or courier to the user, depending on the mission code. It is clear that this interface provides a clear description of the service provided by the service robot, and hides implementation details from the end-users. All what clients' applications need to know is the service description, also known as the contract, that defines the methods that can be invoked.

```
package com.rosjava;


public class TurtlebotPublishersNode extends
    AbstractNodeMain{

    public static Publisher<mybot_delivery_app.
        DeliveryRequestMsg>
        delivery_message_publisher;

    public GraphName getDefaultNodeName() {
    return GraphName.of("rosjava/
        turtlebot_publisher_node");
  }

  public void onStart(final ConnectedNode
      connectedNode) {
    delivery_message_publisher = connectedNode.
        newPublisher("/DeliveryRequestMsg/from_json"
        , mybot_delivery_app.DeliveryRequestMsg.
        _TYPE);
  }
}
```

Listing 1. TurtlebotPublishersNode Class

```
package org.ros-ws;

@WebService (serviceName="TurtlebotOfficeService",
    name="TurtlebotOffice",
        targetNamespace="http://ros-ws.org")
public interface TurtlebotPublishersInterface {

    @WebMethod (action="execute_delivery",
        operationName="executeDelivery")
    public void ExecuteDeliveryRequest(
        DeliveryRequestMessage msg);

}
```

Listing 2. TurtlebotPublishersInterface Interface

Now, in the back-end of the Web service layer, we define the implementation of the Web service interface as illustrated in Listing 3. Observe that the constructor of the class takes a ConnectedNode object as a parameter to initialize the connectedNode attribute. This connectedNode object is responsible for advertising the topics' publishers, and publishing messages in the ROS ecosystem. The ConnectedNode object passed as a parameter in the constructor is effectively created in the Endpoint publisher class in Listing 4, which publishes the Web service. In Listing 3, the ConnectedNode reference is passed to the onStart() method that is used to advertise a new topic publisher, as in Listing 1. As such, this single node reference is used across all the layers of Web services and ROS. This is the key of the Web services intergration into ROS, as mentioned earlier.

TABLE 1
Examples of REST Resources for ROS Topics, Actions and Services

Resources	Description	HTTP Action	ROS Topic Type
/robot/pose/	manages the pose of the robot	GET: retrieve the pose of the robot	**nav_msgs/Odometry**
/robot/velocity/	manages the velocity of the robot	GET: retrieve the angular and linear velocities of the robots PUT: update the velocity of the robot	**geometry_msgs/Twist**
/robot/velocity/linear/	handle the linear velocity component of the robot		**geometry_msgs/Vector3**
/robot/sensors/	list of available sensors in the robot	GET: retrieve the list of all sensors in the robot	
/robot/navigation/services/	list of all services provided by the navigation stack	GET retrieve the list of all services of the navigation stack	
/robot/navigation/action/goal/	handle the action of sending/cancelling a goal location	GET: get current goal location sent to the navigation stack of the robot PUT: send a goal location to the navigation stack of the robot DELETE: cancel the goal the location execution	**move_base_msgs/MoveBaseActionGoal**
/robot/navigation/service/plan	handle the generation of a plan in the navigation stack	GET: retrieve a plan to a given pose from the navigation stack	**nav_msgs/GetPlan**

```
package org.ros-ws;

@WebService(endpointInterface = "org.ros-ws.
    TurtlebotPublishersInterface")
public class TurtlebotPublishersWebService
    implements TurtlebotPublishersInterface{

    static ConnectedNode connectedNode;
    static TurtlebotPublishers turtlebotPublishers
        = new TurtlebotPublishers();

    public TurtlebotPublishersWebService(
        ConnectedNode cn) {
        connectedNode = cn;
    turtlebotPublishers.onStart(cn);
    }

    public void ExecuteDeliveryRequest(
        DeliveryRequestMessage
        deliveryRequestMessage) {

    DeliveryRequestMsg rosMessage =
      TurtlebotPublishers.
          delivery_message_publisher.newMessage();

    /** update deliveryRequestMessage **/
    rosMessage.setDepotY(deliveryRequestMessage.
        getDepot_y());
    rosMessage.setDepotX(deliveryRequestMessage.
        getDepot_x());
    rosMessage.setDestinationX(
        deliveryRequestMessage.getDestination_x());
    rosMessage.setDestinationY(
        deliveryRequestMessage.getDestination_y());
    rosMessage.setSourceX(deliveryRequestMessage.
        getSource_x());
    rosMessage.setSourceY(deliveryRequestMessage.
        getSource_y());
    rosMessage.setMessageCode(
        deliveryRequestMessage.getCode());
    /** publish the message to the "/
        DeliveryRequestMsg/from_json" topic **/
    TurtlebotPublishers.delivery_message_publisher.
        publish(rosMessage);
    }
}
```

Listing 3. TurtlebotPublishersWebService Class

When the Web service clients invokes the Web method `ExecuteDeliveryRequest(...)`, it will create a ROS message, and publish it so that it is processed by the application logic subscribing to this topic; in this case, it executes either a delivery of a coffee or a courier, depending on mission code. Finally, a node runner is created to launch the Endpoint publisher node (Listing not presented for conciseness).

```
public class
    TurtlebotPublishersWebServiceEndpointPublisherNode
     extends AbstractNodeMain {

    @Override
    public GraphName getDefaultNodeName() {
        return GraphName.of("rosjava/
            turtlesim_publishers/endpoint_publisher"
            );
    }

    @Override
    public void onStart(final ConnectedNode
        connectedNode) {
        Endpoint sei = Endpoint.publish("http://"+
            Turtlebot.IPAddress+":5555/
            turtlesim_publisher_ws",
            new TurtlebotPublishersWebService(
                connectedNode)); //pass
                ConnectedNode reference to the
                constructor
    }
}
```

Listing 4. TurtlebotPublishersWebServiceEndpointPublisherNode Class

5.2 REST Web Services for Delivery Application

Listing 5 presents an implementation of a resource class for the delivery application. The constructor initiates the `turtlebotPublishers` object using the

ConnectedNode object passed as parameter in the constructor of the Enpoint publisher class that effectively creates the node reference, as depicted in Listing 6.

```
package org.mybot.turtlebot.rest;

/**import section **/

@Path("robot/app/delivery/")
public class DeliveryTopicResource {

    public static TurtlebotPublishers
        turtlebotPublishers;
    static ConnectedNode connectedNode ;

    public DeliveryTopicResource(ConnectedNode
        connectedNode){
        turtlebotPublishers = new
            TurtlebotPublishers();
    turtlebotPublishers.onStart(connectedNode);
    }

    @PUT
    @Path("code/{code}/depot_x/{depot_x}/depot_y/{
        depot_y}/dest_x/{dest_x}/dest_y/{dest_y}/
        source_x/{source_x}/source_y/{source_y}")
    public void executeDelivery (
            @PathParam("code") int code,
            @PathParam("depot_x") double depot_x,
            @PathParam("depot_y") double depot_y,
            @PathParam("dest_x") double dest_x,
            @PathParam("dest_y") double dest_y,
            @PathParam("source_x") double source_x,
            @PathParam("source_y") double source_y){
        DeliveryRequestMsg rosMessage =
            TurtlebotPublishers.
            delivery_message_publisher.newMessage();
        rosMessage.setDepotX(depot_x);
        rosMessage.setDepotY(depot_y);
        rosMessage.setDestinationX(dest_x);
        rosMessage.setDestinationY(dest_y);
        rosMessage.setSourceX(source_x);
        rosMessage.setSourceY(source_y);
        rosMessage.setMessageCode(code);
        System.out.println("\n\n\n\n\n\n Message
            received \n\n\n\n");
        TurtlebotPublishers.
            delivery_message_publisher.publish(
            rosMessage);
    }

}
```

Listing 5. DeliveryTopicResource Class

In Listing 5, the ExecuteDelivery(...) Web method executes when the Web service receives an HTTP request using the PUT operation on the following URI /robot/app/delivery/, added to this the list of parameters passed to the Web method to prepare the delivery request message according to user request, and publishes it through the deliver_message_publisher of the TurtlebotPublishers class. As such, the application logic embedded in the robot will receive the request and makes the robot execute it.

```
public class TurtlebotResourcesEndpointPublisherNode
     extends AbstractNodeMain{
    @Override
    public GraphName getDefaultNodeName() {
            return GraphName.of("rosjava/
                resources_publisher/
                rest_endpoint_publisher_node");
    }

    @Override
    public void onStart(final ConnectedNode
        connectedNode){
    try {
        DeliveryTopicResource deliveryTopicResource
            =
            new DeliveryTopicResource(connectedNode)
            ;
        HttpServer httpServer =
            HttpServerFactory.create(
                "http://"+TurtleSim.IPAddress+"
                :7003/",
                new ClassNamesResourceConfig(
                    DeliveryTopicResource.class )
            );
        httpServer.start();
    } catch (Exception e) {
            e.printStackTrace();
    }
    }
}
```

Listing 6. TurtlebotResourcesEndpointPublisherNode Class

5.3 Web Service Client

It is clear from these two illustrative examples that SOAP and REST Web services represent an effective approach to expose ROS resource to end-users in a seamless fashion. The clients apps can then make abstractions of ROS and the specific implementation details of the back-end service robot applications, and mainly focus on the services exposes by Web service interfaces. Listing 7 presents a simple example of a possible client application that invokes the ExecuteRequest().

Observe that the client application is completely unaware of the ROS ecosystem and the details of the implementation. It mainly consists in invoking through the Service Enpoint Interface (SEI) the Web method ExecuteDeliveryRequest(...) to make the robot perform the requested mission, in this case to deliver a coffee, from the Cafe in location (60.63, 37.51) to the office in location (73.02, 41.35). The developer makes a complete abstraction of ROS and is able to command the robot through available service interfaces. Finally, we note that the response time of REST Web services is slightly faster than SOAP Web services. In fact, we evaluated the response time of a request sent through REST and SOAP clients to the Web services in a localhost machine, and we found out that the average response time is about 1.5 msec in for REST requests whereas around 2.5 msec for SOAP requests. This confirms that REST is a lighter weight than SOAP in terms of deployment.

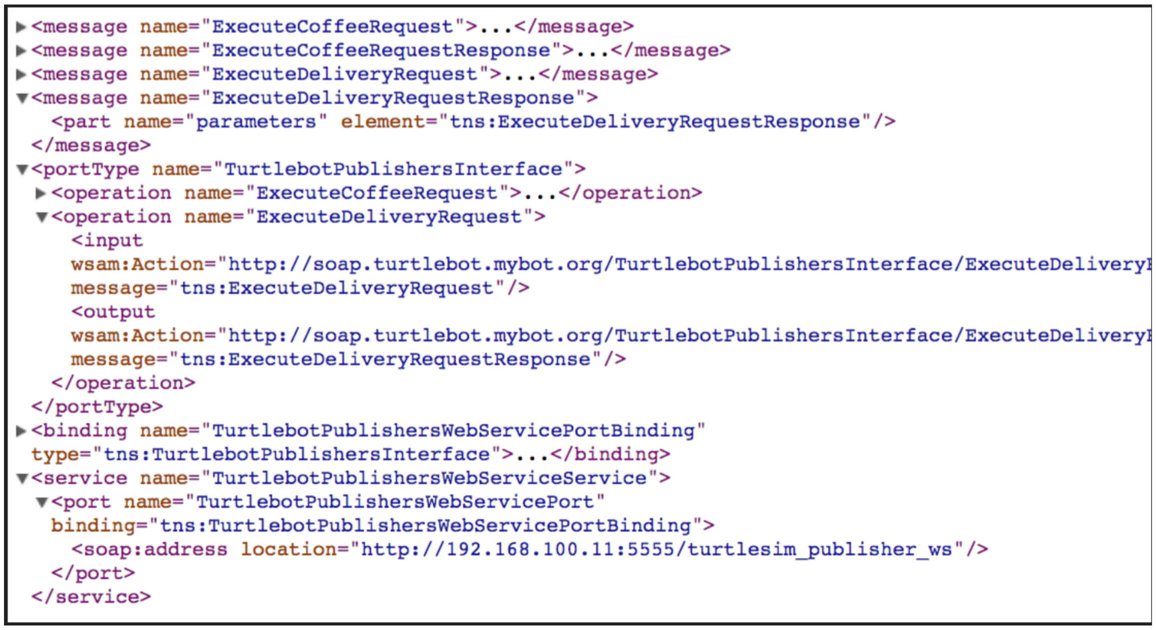
```
public class TurtlebotPublishersWebServiceClient {

static final  int COURIER_DELIVERY_MESSAGE_CODE=3;
static final  int COFFEE_DELIVERY_MESSAGE_CODE=4;

public static void main(String []args){

URL url = new URL("http://"+Turtlebot.IPAddress+"
    :5555/turtlesim_publisher_ws?wsdl");

    //qualified name of the service
    QName qname = new QName ("http://soap.turtlebot.
        mybot.org/",
      "TurtlebotPublishersWebServiceService");

    //create a factory for the service
    Service service = Service.create(url, qname);

    //extract the endpoint interface, the service "
        port"
    TurtlebotPublishersInterface sei = service.
        getPort(TurtlebotPublishersInterface.class);

    DeliveryRequestMessage msg = new
        DeliveryRequestMessage();
    msg.setMessage_code(COFFEE_DELIVERY_MESSAGE_CODE
        );
    msg.setSource_x(76.02);
    msg.setSource_y(41.35);
    msg.setDestination_x(60.633);
    msg.setDestination_y(37.51);
    msg.setDepot_x(68.269);
    msg.setDepot_y(38.718);

    sei.ExecuteDeliveryRequest(
        deliveryRequestMessage);
  }
}
```

Listing 7. TurtlebotPublishersWebServiceClient Class

5.4 Discussion and lessons learned

The Robot As A Service (RAAS) research area is nowadays an exciting yet challenging field with a lot of promising potentials in the near future. In fact, this concept is rather broad and open to many interpretations and concrete implementations. This paper presented a new approach that leverages the use of Web services to expose robots's resources as services, with a particular focus on ROS. Several previous and ongoing works (i.e.[5], [9], [10], [15]) have investigated the RAAS vision from different facets. The emergence of this research trend is the result of the convergence of two major technologies in the today's IT world, namely the IoT and cloud computing. The aforementioned efforts, including this work, aim at exposing future robotic systems to end-users through the IoT. The use of SOA and in particular Web services, as in this paper, is indeed a promising approach for achieving this objective. We retain the following lessons from this paper:

- Web services present an effective approach for providing abstractions to ROS resources. On the one hand, it is possible with the proposed ROS Web services to provide an abstract specification of ROS-enabled robot services by taking advantage of the WSDL standard for SOAP Web services (refer to Figure 5) , and the WADL standard for REST Web services. On the other hand, these abstract services' descriptions effectively hide the implementation details from non-robotician users and developers. For example, it is clear from the Web service client application example that the robot shows-up as a list of services, including the coffee request and delivery request services that can be remotely invoked, without having to know how these services are actually implemented. This is illustrated in Figure 5 that depicts the WSDL document of the SOAP Web service corresponding to the delivery application. According to this WSDL, and from the end-user perspective, the robot is rather a service provider of two services provided by the `TurtlebotPublisherIterface`, namely the `ExecuteCoffeeRequest` and the `ExecuteDeliveryRequest`. Of course, more services can be added based on the user requirement. We note that this generic service interface can be implemented differently on different robots, but from the end-user perspective, the service (or the contract) remains the same independently of the robotic platform and the service implementation. This is an important feature for dealing with robot heterogeneity when developing client applications or when deploying robots into the cloud.
- The ROSJAVA client API of ROS is a major player towards the integration of ROS into the Internet through the use of the powerful feature of Java programming language. In fact, it also allowed ROS to extend its capabilities to be supported by Android devices, typically relying on Java. However, Python remains an other strong candidate to achieve comparable integration capabilities to Java, as it greatly supports Internet applications development, is better supported by ROS, and also provides native Web services APIs.
- The ROS Web services abstraction would be quite useful for enabling future cloud robotics, as they provide a SOA solution that would facilitate the integration of robots into the cloud. In fact, SOA and resources' virtualization represent a key component of today's emerging clouds. ROS Web services can allow the control of these robots through abstract cloud interfaces based on Web services, as it is the case nowadays with general-purpose computing and storage resources.
- The proposed SOAP and REST meta-models provide an abstract description of services interfaces of ROS ecosystem. However, this approach can be generalized by following a model-driven development approach (like in [21]) that allows for the code auto-generation. To achieve this, one possible alternative is to follow a contract-based approach by starting with the definition of the WSDL for SOAP Web services and the WADL for REST



```
▶<message name="ExecuteCoffeeRequest">...</message>
▶<message name="ExecuteCoffeeRequestResponse">...</message>
▶<message name="ExecuteDeliveryRequest">...</message>
▼<message name="ExecuteDeliveryRequestResponse">
   <part name="parameters" element="tns:ExecuteDeliveryRequestResponse"/>
 </message>
▼<portType name="TurtlebotPublishersInterface">
  ▶<operation name="ExecuteCoffeeRequest">...</operation>
  ▼<operation name="ExecuteDeliveryRequest">
     <input
     wsam:Action="http://soap.turtlebot.mybot.org/TurtlebotPublishersInterface/ExecuteDeliveryI
     message="tns:ExecuteDeliveryRequest"/>
     <output
     wsam:Action="http://soap.turtlebot.mybot.org/TurtlebotPublishersInterface/ExecuteDeliveryI
     message="tns:ExecuteDeliveryRequestResponse"/>
   </operation>
 </portType>
▶<binding name="TurtlebotPublishersWebServicePortBinding"
 type="tns:TurtlebotPublishersInterface">...</binding>
▼<service name="TurtlebotPublishersWebServiceService">
  ▼<port name="TurtlebotPublishersWebServicePort"
   binding="tns:TurtlebotPublishersWebServicePortBinding">
     <soap:address location="http://192.168.100.11:5555/turtlesim_publisher_ws"/>
   </port>
 </service>
```

Fig. 5. Excerpt from the WSDL document of the SOAP ROS Web Service of the Delivery Application

Web services then auto-generating the code based on the abstract service description and also the structure of the meta-models. We plan to investigate this approach in the future.

6 Conclusions and Future Works

In this paper, we presented a novel approach for integrating Web services into ROS. The objective is to provide new programming abstractions for robots applications' developers to control service robots through Web interfaces without having to deal with implementation specific details. We developed both SOAP and REST Web services interfaces for exposing ROS resources to client applications so that services of interest can be easily invoked. The new Web services abstraction layer is an enabling technology for the newly emerging paradigm of cloud robotics, where robots uses cloud computing capabilities to either boost its computation resources or also to be exposed as services through the cloud [9]. In fact, we are currently working on extending these Web services interfaces for a seamless interaction between the cloud and the robot, where the robot may act as a client of the cloud if it needs to use its computation resources for computation-greedy applications (e.g. computer vision, SLAM, ...) or it may act as a server behind the cloud that virtualizes the access to the robot through a service interface. In this case, an end-user will just initiate a mission, and the cloud will manage which real robot will execute the mission based on the robots' availability and capability. We are investigating this paradigm in a surveillance context of applications using drones, where drones are exposed as services behind the cloud, which will decide which drone will execute which mission based on user and application requirements. We believe that the work presented in this paper will pave the way towards more concrete implementations and deployment of clouds robotics.

Acknowledgments

This work is supported by the myBot project entitled "MyBot: A Personal Assistant Robot Case Study for Elderly People Care" under the grant number 34-75 from King AbdulAziz City for Science and Technology (KACST). This work is partially supported by Prince Sultan University.

References

[1] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009. 1, 3

[2] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *In Proceedings of the 11th International Conference on Advanced Robotics*, 2003, pp. 317–323. 1

[3] H. Bruyninckx, "Open robot control software: the orocos project," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 3, 2001, pp. 2523–2528 vol.3. 1

[4] D. D. A. P. G. M. L. N. Paul Fitzpatrick, Elena Ceseracciu, "A middle way for robotics middleware," in *Journal of Software Engineering for Robotics*, 2014, pp. 42–49. 1

[5] S. Osentoski, G. Jay, C. Crick, B. Pitzer, C. DuHadway, and O. C. Jenkins, "Robots as web services: Reproducible experimentation and application development using rosjs," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, 2011. 1, 2, 5.4

[6] "Introducing rostful: Ros over restful web services, http://www.ros.org/news/2014/02/introducing-rostful-ros-over-restful-web-services.html," 2015. 1, 2

[7] C. Yinong, D. Zhihui, and G.-A. Marcos, "Robot as a service in cloud computing," in *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*, 2010. 1, 2

[8] K. Yuka, I. Toru, M. Yoshihiko, O. Keiju, U. Miwa, T. Yosuke, and N. Masahiko, “Research and development environments for robot services and its implementation,” in *System Integration (SII), 2011 IEEE/SICE International Symposium on*, 2011. 2

[9] A. Koubaa, “A service-oriented architecture for virtualizing robots in robot-as-a-service clouds,” in *Architecture of Computing Systems ARCS 2014*, ser. Lecture Notes in Computer Science, E. Maehle, K. Rmer, W. Karl, and E. Tovar, Eds. Springer International Publishing, 2014, vol. 8350, pp. 196–208. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-04891-8_17 2, 5.4, 6

[10] R. Souza, F. Pinho, L. Olivi, and E. Cardozo, “A restful platform for networked robotics,” in *Ubiquitous Robots and Ambient Intelligence (URAI), 2013 10th International Conference on*, Oct 2013, pp. 423–428. 2, 5.4

[11] R. Safaripour, F. Khendek, R. Glitho, and F. Belqasmi, “A restfull architecture for enabling rapid development and deployment of companion robot applications,” in *Computing, Networking and Communications (ICNC), 2014 International Conference on*, Feb 2014, pp. 971–976. 2

[12] S. Mahmoud and N. Mohamed, “Collaborative uavs cloud,” in *Unmanned Aircraft Systems (ICUAS), 2014 International Conference on*, May 2014, pp. 365–373. 2

[13] ——, “Broker architecture for collaborative uavs cloud computing,” in *Collaboration Technologies and Systems (CTS), 2015 International Conference on*, June 2015, pp. 212–219. 2

[14] S. Mahmoud, N. Mohamed, and J. Al-Jaroodi, “Integrating uavs into the cloud using the concept of the web of things,” *Journal of Robotics*, vol. 2015, September 2015. 2

[15] D. Brugali, A. Da Fonseca, A. Luzzana, and Y. Maccarana, “Developing service oriented robot control system,” in *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on*, April 2014, pp. 237–242. 2, 5.4

[16] C. Pautasso, O. Zimmermann, and F. Leymann, “Restful web services vs. ”big”’ web services: Making the right architectural decision,” in *Proceedings of the 17th International Conference on World Wide Web*, ser. WWW ’08. New York, NY, USA: ACM, 2008, pp. 805–814. [Online]. Available: http://doi.acm.org/10.1145/1367497.1367606 4.2

[17] “Mybot project, KACST Project Number 34-75,” 2015. 5

[18] “Video demonstration of mybot service robot for courier delivery application, https://www.youtube.com/watch?v=oTLtmX2-ucA,” 2015. 5

[19] “Video demonstration of Web Services integration in ros, https://www.youtube.com/watch?v=WvjY5XjAX7U,” 2015. 5

[20] A. Koubâa, M. Sriti, H. Bennaceur, A. Ammar, Y. Javed, M. Alajlan, N. Al-Elaiwi, M. Tounsi, and E. M. Shakshuki, “COROS: A multi-agent software architecture for cooperative and autonomous service robots,” in *Cooperative Robots and Sensor Networks 2015*, 2015, pp. 3–30. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-18299-5_1 5.1

[21] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, “The brics component model: A model-based development paradigm for complex robotics software systems,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC ’13. New York, NY, USA: ACM, 2013, pp. 1758–1764. [Online]. Available: http://doi.acm.org/10.1145/2480362.2480693 5.4

Dr. Anis Koubaa received his B. Sc. in Telecommunications Engineering from Higher School of Telecommunications (Tunisia), and M. Sc. degrees in Computer Science from University Henri Poincar (France), in 2000 and 2001, respectively, and the Ph. D. degree in Computer Science from the National Polytechnic Institute of Lorraine (France), in 2004. He was a faculty member at Al-Imam University from 2006 to 2012. Currently, he is an associate professor in the Department of Computer Science at Prince Sultan University and research associate in CISTER Research Unit, ISEP-IPP, Portugal. He becomes a Senior Fellow of the Higher Education Academy (SFHEA) in 2015. He has published over 120 refereed journal and conference papers. His research interest covers mobile robots, robotics software engineering, Internet-of-Things, cloud computing and wireless sensor networks. Dr. Anis received the best research award from Al-Imam University in 2010, and the best paper award of the 19th Euromicro Conference in Real-Time Systems (ECRTS) in 2007. He is the head of the ACM Chapter in Prince Sultan University. His H-Index is 26.