Turtlebot at Office: A Service-Oriented Software Architecture for Personal Assistant Robots using ROS

Anis Koubâa **¶‡, Mohamed-Foued Sriti ‖, Yasir Javed ¶, Maram Alajlan ¶††, Basit Qureshi **,
Fatma Ellouze *¶, Abdelrahman Mahmoud §†

¶ Cooperative Networked Intelligent Systems (COINS) Research Group, Saudi Arabia.
**Prince Sultan University, College of Computer and Information Sciences, Saudi Arabia.
†Gaitech International Ltd., Honk Kong
‖ Al-Imam Mohammad Ibn Saud Islamic University, Saudi Arabia.
†† College of Computer and Information Sciences, King Saud University, Riyadh, Saudi Arabia.
§ German University of Cairo, Egypt.
‡ CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal.
* National Engineering Institute ofSfax (ENIS), Tunisia.

akoubaa@coins-lab.org, mfsriti@ccis.imamu.edu.sa, yasir.javed@coins-lab.org, maram.ajlan@coins-lab.org,
qureshi@psu.edu.sa, fatma.ellouze@coins-lab.org, abdelrahman.mahmoud@coins-lab.org

***Abstract*—This paper presents the design of an assistive mobile robot to support people in their everyday activities in office and home environments. The contribution of this paper consists in the design of a modular component-based software architecture that provides different abstraction layers on top of Robot Operating System (ROS) to make easier the design and development of service robots with ROS. The first abstraction layer is the COROS framework composed of complementary software sub-systems providing different interfaces between ROS and the client applications. The second abstraction layer is the integration of Web services into ROS to allow client applications to seamlessly and transparently interact with the robot while hiding all implementation details. The proposed software architecture was validated through a experimental prototype of Turtlebot deployed in University campus. Furthermore, we outline the challenges incurred during experimentation and focus on lessons learned throughout the implementation and deployment.**

I. INTRODUCTION

The tremendous growth in utilization of robots has brought numerous benefits for humans with application to manufacturing, healthcare, mining, deep excavation, space exploration, etc. Use of robots has been a significant factor in improvement of human safety, reduction in maintenance / production costs and improved productivity [1].

It is widely forecasted that service robots would inundate the market reaching record sales in the next 20 years. In its statistical report, The International Federation of Robotics reported sale of 3 million service robots for personal and domestic within 2012. This number represents 20% increase in sales from the previous year accounting to US$ 1.2 billion [2]. It is also forecasted that household robots, in particular, will make a substantial increase of the number of sales from 1.96 million till 15.5 million robots for the period 2013-2016. Nowadays, one of the major challenges in the development of service robots is the lack of software engineering framework to build complex service robots' applications that are modular, reusable, and easily extensible. Most of the available software for service robots is tighly coupled with the robotic platform and lack sufficient abstractions to remain generic for different platforms. Robot Operating System (ROS) is one of the widely used middleware to develop robotics applications and represents an important milestone in the developement of modular software for robots. In fact, it presents different abstractions to hardware, network and operating system such as navigation, motion planning, low-level device control, and message passing. However, the levels of abstractions are still not enough for developing complex and generic applications for mobile robots, in particular if those applications are distributed among several machines, requiring machine-to-machine communication. This paper addresses this gap, and proposes the design of a service-oriented software architecture that contains two layers of software abstractions, including: (*i.*) The `COROS` framework, which consists of several software sub-systems at different levels, namely communication, application logic, robot control, knowledge base, (*ii.*) *ROS Web services interface*, which is designed to expose ROS ecosystem as a Web service following a Service-oriented approach using Representational State Transfer (REST) or Simple Object Access Protocol (SOAP) Web Services. The objective is to provide interfaces for Web developers with no prior background on ROS allowing for a seamless interaction with the service robot. We validate our layered architecture through an implementation on a Turtlebot service robot that we deployed at Prince Sultan University to assist faculty and students.

The contributions of this paper can be summarized as follows:

- Design of a low-cost service robot Based on the Turtlebot platform and Commercial off the Shelf (COTS) hardware.
- Design of the COROS software architecture as an abstraction layer on top of robots allowing easier developement

978-1-5090-2255-7/16 $31.00 © 2016 IEEE
DOI 10.1109/ICARSC.2016.66

of distributed robots' applications.
- Integration of SOAP and REST Web services into ROS, that provides additional abstraction layers for developers with no prior background on ROS or robotics. Furthermore, we present a software meta-model for the integration of Web services into ROS. To the best of our knowledge, the work presented is ground breaking as far as such integration is concerned.
- Experimentation and deployment of the service robot for the validation of our architecture and discussion of experimental challenges.

The rest of this paper is organized as follows. Section II discusses the state-of-the-art with an emphasis on the contribution of this paper compared to similar works. Section III presents the mechanical design of the service robot. In Section IV, we present the layered software architecture, namely the COROS framework in addition to the Web services integration with ROS. In Section V, we present experimental setup and deployment of the service robot with a discussion on challenges incurred. Section VI concludes the paper and outlines future works.

II. RELATED WORKS

Recently, Human-Robot Interface design is gaining a lot of attention in the context of service robots. Building software architecture and frameworks for service robots has attracted a lot of research work in the literature. Authors in [3] postulate that a robot should be capable of generating meaningful questions regarding the task procedures in real time and should be able to apply the results to modify its task plans. Authors in [4] proposed a development process that defines how reference architectures can be exploited for building robotic applications. They developed the Hyper-Flex software tool-chain for supporting the process of exploiting reference architectures and demonstrated how reference architecture can be used for building complex software systems. In a later work [5], authors extended Hyper-Flex tool-chain focusing on ROS meta-models and ROS-specific tools such as automatic generation of launch files from the models of configured systems. The limitation of these works is the lack of concrete implementations demonstrating instantiation of these processes. In our paper, we present both the architecture and its implementation and deployment on a real service robot.

In [6], authors proposed an architecture for a Domestic Robot targeting elderly users in assisting them to remain autonomous in their homes. The proposed architecture is based on the integration of three middleware frameworks PEIS, MIRA, and ROS. Most of the computation is performed by a large number of ROS nodes; the resulting robot services are exported to the PEIS middleware for seamless integration of the robot into the ambient assisted living system. In [7], authors proposed ROBCO12 which is an Intelligent Modular Service Mobile Robot targeting elderly and disabled person care. It consists of Intelligent onboard multilevel control system, ROS or/and Microsoft Robotics Studio based software for the control. ROBCO12 can provide many services to the users including personnel recognition, person following and elderly person fall detection. Unfortunately the proposed system provides no interface for easy programming of autonomous and semi-autonomous tasks.

With regards to integration of Web services in ROS, rosjs and rosbridge `rosjs` have recently been proposed. Both these frameworks essentially cater to (1) allowing common web browsers to exploit users to interact with ROS enabled robots; (2) to provide Web developers lacking expertise in robotics with simple interfaces to develop client applications allowing control and manipulation of ROS-enabled robots. In [8], the authors proposed `ROStful` by extending `rosbridge` to support REST Web services and developed a lighweight Web server that exposes ROS topics, services and actions through RESTful Web services. In [8], the authors did not provide an architecture or meta-model for the integration of REST into ROS.

To this end, we focus on this lack of attention and address these issues in the proposed work. The architecture proposed in this paper essentially decouples networking, from application logic and robot control. Each component can be defined and implemented independently of others and can be reused as appropriate in different complex applications and scenarios.

III. ROBOT DESIGN

We focus on three main design requirements:

- **Cost-effectiveness:** Cost effectiveness would be a key parameter for wider acceptance of the platform.
- **ROS-enabled design:** Nowadays ROS has become a defacto standard in the development of robotics applications. We opted for the use of a robot platform supporting ROS to take dull benefit of the open source libraries in developing mobile robots applications, such as navigation services, image processing, drivers, etc.
- **Commercial-off-the-shell hardware:** We plan to use commonly available and affordable off-the-shelf hardware to enable affordable and extensibe design for end-users.

We opted for the use of the Turtlebot 2 robot as a base platform for the design of our service robot as it fulfills all the above requirements. However, the software architecture that we propose in this paper can be applied to any type of ROS-enabled robot thanks to the abstraction layers we designed for robot control and that will be presented in Section IV.A.

We extended the Turtlebot platform by adding a robotic arm to perform grasping and manipulation tasks. We have used the low-cost PhantomX Pincher Arm by Trossens Robotics, commonly used with the Turtlebot robot. The arm has five degrees of freedom and an Arduino- based microcontroller and it is fully supported by ROS. We wrote a tutorial in [9] on how to get started with PhantomX Pincher Arm and also explains the configuration needed to update the robot URDF description to integrate the robotic arm. To control the gripper, we integrated an Arduino ultrasonic sensor and Arduino RFID reader to the robotic arm [9]. The interconnection of the Arduino sensor with ROS was performed using the `arduino-rosserial`

package. The ultrasonic is used to automatically open the gripper when the object to be grasped is put in proximity to the range sensor situated close to the gripper. A push button was programmed to close the gripper when pressed. In the same way, the Arduino RFID was programmed to open the gripper if the RFID tag is recognized. This allows to open the gripper for authorized users, for example those who are destintation of a courier, or the cafe waiter to take the money from the gripper.

In addition, we added and configured the robot with the Hokuyo laser scanner URG-LX4, which has a maximum range of 4 meters and an beam opening of 240 degrees, better than the 57 degrees of the Kinect laser scanner. A tutorial on how to add the Hokuyo laser range finder to a Turtlebot robot can ne found in [9].

For human-robot interaction, we integrated a 10-inch Samsung tablet on the top of the robot through an Android interface. A user can also interact with the robot using an Android smartphone with the same interface of the Tablet. For that purpose, we have developed a full suite of applications that allow a user to communicate, control and interact with the robot. We will present these interfaces in **Section IV**. For computer vision applications, we have also added an Asus Live Pro 3d sensor, similar to the Kinect, and it was placed on the top plate of the robot to observe people and objects while navigating. The Kinect 3d sensor was placed on the lowest place at a height of 25 cm, which is basically used to avoid short obstacles. The Asus 3D sensor has two main advantages over the Kinect sensor (1) it has a much lighter weight, (2) it is powered by only the 5V USB port with no need for additional external power supply. One major challenge faced in the design of the body is that the Kobuki base has a maximum payload of 5 Kg on hard floor and 4 Kg on carpet including all accessories added on its top. Indeed, the robot had difficulties to move if a heavy payload is put on top of it. We have considered this physical constraint when we configured the additional hardware added to the robot.

IV. SOFTWARE ARCHITECTURE

Building distributed applications for service robots systems is a very challenging task from software engineering perspective. Indeed, apart from the complexity of designing software components for the control of the service robot, additional care is required in the design of reusable interfaces and components to alleviate the complexity of development. To achieve this objective, we designed a software architecture that provides two abstraction layers on top of ROS to make easier the development of distributed applications for service robots. It includes two major layers, namely: (1) COROS [10], which is a component-based software architecture that provides a first abstraction layer on top of ROS composed of modular components to develop cooperative and distributed applications, (2) ROS Web services is the second abstraction layer that allow client applications to seamlessly and transparently interact with the robot while hiding all implementation details. Services are made public to client applications, which can be invoked by a Web Service client. The advantage of our architecture is that it decouples networking, from application logic , and robot control. Each component can be defined and implemented independently of others and can be reused as appropriate in different complex applications. In what follows, we present the main features.

A. COROS

We reused and extended our COROS achitecture defined in [10], by developing new modules for the service robot application logic, and also a new message serializer to effectively handle communication between heterogeneous platforms. In what follows, we describe the architecture and enhancements. COROS consists of five layers illustrated in Figure 1 that shows the component diagram of the software architecture. The software system is decomposed into five subsystems (or

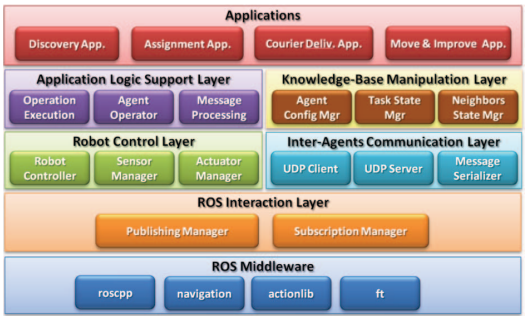

Fig. 1. COROS Software Architecture

layers), each of which plays the role of a container of a set of components. These subsystems are:

- **Communication:** this subsystem was designed to ensure the interaction between the robot and other machines, which can be robots or user devices. It comprises extensible and modular client and server components that enable agents to exchange serialized messages through the network interface using sockets. In [10], COROS was used for communication between homogenous robotic machines, so we used the C++ Boost message serialization to ensure communication between two robots. However, in this paper, the service robot interacts with different types of devices (e.g. Android device) making C++ Boost no longer feasible due to incompatibility and hetergoneity of communicating systems. For this purpose, we developed a generic module that converts ROS messages to JSON formatted messages and vice-versa, based on the `rospy_message_converter` package in ROS.
- **ROS Interaction Layer:** this subsystem adds a lightweight layer on top of ROS allowing a seamless inter-process interaction between ROS nodes (processes) defined in the architecture. The main role of this layer is to provide a simple and efficient way to manage the subscribers and the publishers to ROS topics and services. Any node can publish or subscribe to a new topic using

both components Publishing Manager and Subscription Manager without having to directly interact with ROS.

- **Robot Control:** this subsystem adds another layer on top of ROS providing a bridge between the local software agents and the physical robots. The role of this layer is to manage the robot configuration and its state. The `Robot Controller` component provides an abstract model for any ROS-enabled robot. Indeed, this component provides several interfaces for controlling and monitoring robots' states such as location, published and subscribed topics, provided and used services, etc. This enables to make easier the management of heterogeneous robots as they adhere to a common component model. Any robot type can be easily configured to interact with the interfaces provided by the robot controller components. Currently, we have configured this component for the Turtlebot robot.
- **Application Logic:** this subsystem addresses the problem solving requirements; it encapsulates all of the components needed to implement a complete service robot application. Any new application should reuse and configure the software components to define its proper behaviour. The `Agent Operator` is the main component of the Application Logic subsystem as it implements the actual behaviour of the applications. This means that every type of received message (through `Agent Server` Component) triggers the execution of an appropriate function as specified by the application. The `Agent Operator` uses the Communication subsystem to exchange information with other robotic agents in the environment.
- **Knowledge Base Manipulation Layer:** This subsystem aims at satisfying knowledge base requirements and maintains up-to-date information about the robot status and its environment. Currently, we did not use a specific formal language either for knowledge representation or reasoning in this subsystem. This in reality related to the nature of the service Robots applications, when each robot gathers the captured information from its environment to accomplish a specific task. Usually, the majority of gathered information becomes obsolete from an execution to another. Based on its unique component State Monitoring, this subsystem provides to others with a useful information and services such as allowing the agent to monitor and control its local state, other agents states, the state of the different tasks, and the information about the agent initial configuration.

In the context of MyBot project, we have implemented four applications using COROS, including (1), Discovery application, (2) Courier Delivery application, (3) Coffee delivery application, and (4) people guidance application. For more details about the COROS framework, the reader is referred to our book chapter [10].

B. ROS Web Services

1) Objectives: The objective of desiging ROS Web services is to expose ROS as a Service to the client applications, providing an additional abstraction layer of ROS resources including topics, services and actions for developers with no prio knowledge on robots or on ROS. There are three main benefits coming from exposing ROS as a service, namely:

- *Fostering public usage of robots:* By exposing the complex ROS ecosystem through Web services interfaces to client applications, Web and mobile developers with no background on robotics can easily interact with the robots through the Internet through Web service invocation. This enables a wider usage of robots at public scale.
- *Integration with the cloud*: Web services and Service Oriented Architecture (SOA) are major components of today's cloud as they allow virtualization of resources. Therefore, embedding Web services into ROS allows for the integration of ROS-enabled robots with the cloud so that users can virtually access the robots' resources through the cloud to either control or monitor the robots status.
- *Standard interfaces*: Web services allows for providing standard interfaces to robotics resources so that it will be possible for client application to interact with heterogenous robots if they are have the same Web services abstractions, independently from implementation details.

To address these objectives, we propose to use Web services as an additional abstraction layer on top of ROS. We develop a SOAP Web Service implementation (ros-ws) and a REST Web Service implementation (ros-rs), which represent the two fundamental architectural models for SOA. 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.

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.

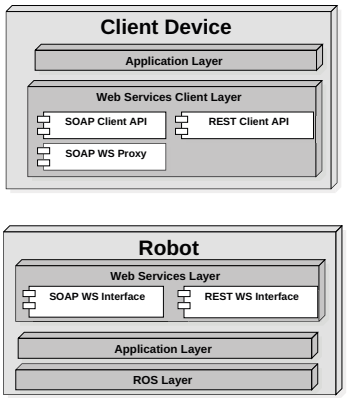

Fig. 2. Deployment Diagram of ROS Web Services

The Web services can be seen as a middleware that allows seamless interaction between client applications and ROS ecosystem in the service robot. Our architecture encompasses both SOAP and REST Web services to provode felxible alter-

native to client applications to interact with ROS ecosystem. In particular, the Web service layer allows a user to subscribe to or publish any ROS topic, action or service, and thus delivering ROS messages to client subscribed to a particular topic.

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 use of 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.

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. 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.

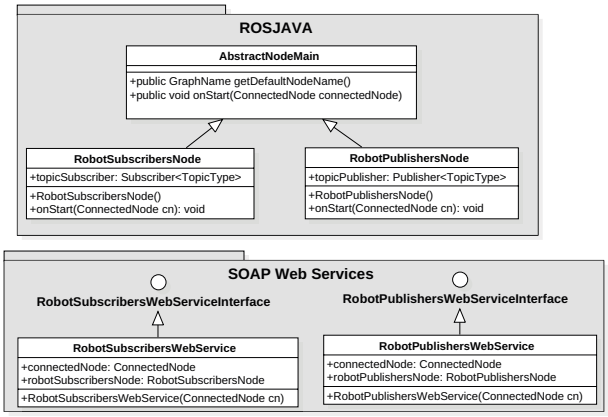

Fig. 3. UML Class Diagram of SOAP Web Services for ROS

The UML class diagram for REST also follows a similar software decomposition. Basically, we distinguish two main packages in Figure 3:

rosjava package: The `rosjava` package contains classes written in ROSJAVA with no Web services functionality. This is an abstraction layer that will be used to link ROS ecosystem

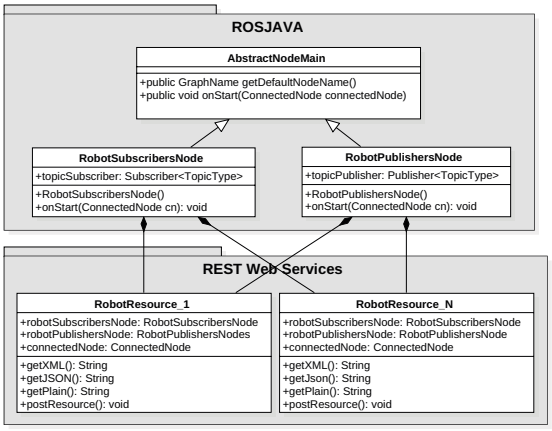

Fig. 4. UML Class Diagram of REST Web Services for ROS

to the Web services that are be defined in the SOAP and REST Web services packages. 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.

SOAP Web services `ros-ws` package: This package contains classes that wrap the exposed ROS functionalities as SOAP Web services. 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 are exposed to and invoked by the Web services' clients.

REST Web services `ros-rs` package: This package 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. Typical `HTTP` operations namely `GET`, `POST`, `PUT`, and `DELETE` are used to access the exposes resources. The output can be in any user-defined format including plain text, JSON or XML.

V. EXPERIMENTATION AND DEPLOYMENT

We implemented several services and functionalities in our service robot using the COROS architecture and Web services. We deployed this service robot at Prince Sultan University to deliver courier between offices and also to bring coffee from the central cafe of the University among other applications implemented. In what follows, we present the main implemented services, and the challenges and lessons

learnt from the deployment experience.

A. Murphy's Law in Deployment

In this section, we present the challenges encountered in setting up and configuring the service robot to operate in the real environemnt.

Building a large map is not trivial : We have deployed the robot in the corridors of the CCIS first floor in Prince Sultan University. For the navigation of the robot, it is required to feed the navigation stack in ROS with the map of the environment where the robot has to navigate. Thus, we have built a map of the CCIS first floor of a dimension 150*80 m^2. Building the map of such a large environment was quite challenging and tedious. The process completely fails with the Turtlebot robot due to the high inaccuracy of its odometry and low resolution of its Kinect-based laser scanner. Attempts with the Hokuyo 04LX laser scanner were not much better. The problem of building the large map was overcome with the use of the PeopleBot robot equiped with a SICK LMS 500 laser scanner with 30 meters of maximum range and high resolution. The resulting map of the whole floor has a resolution of 3200*1680 pixels.

Unraveling the depolyment challenges: This map is quite huge and was unpractical to use with a Turtlebot. On a real robot, we noticed that considering the large size of the map, the navigation of the robot was oscillating and jerky. This is because the update of the map consumes most of the robot processing resources and thus the robot misses its control loops very often. Although, we used a laptop with higher processing capabilities (MAC Book Air with core i5, 4GB RAM), the problem persisted. We tried different configurations of the navigation stack by reducing the update rate of the map, and tuning the navigation stack parameters, but the navigation was always an issue using a large scale map.

Another problem with the large-scale deployment was the WiFi coverage problem. The robot must always be connected to a WiFi spot to be connected to users end devices and the Internet. In a first attempt, we wanted to use the wireless local network at CCIS PSU that covers the whole area of interest. However, this was not possible because the switches were configured to block the broadcast traffic, and thus the discovery messages (refer to next subsection) sent by the robot to get discovered by the user devices were blocked. We have thus used a battery-powered wireless router that is attached to the robot, which solves partially our problem, as the robot will always remain in the range of this WiFi spot plugged to it.

So, to overcome all these problems, we opted for making the deployment into a more restricted area of the CCIS first floor using a partial map of environment, which is presented in **Figure 7**. The smaller map has a resolution of 837*477 pixels and provided a fuild navigation of the service robot, either in simulations or real experiments.

B. Discovery Service

The discovery protocol is required to allow the client applications to auto-detect the robot(s) in its neighborhood

Fig. 5. Discovery Android App

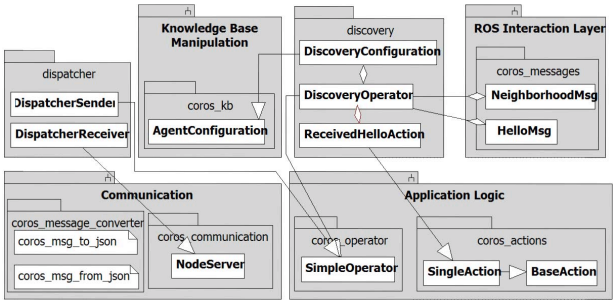

Fig. 6. Discovery Application Class Diagram

to connect to it. Each robot periodically broadcasts a `HELLO` message carrying out information needed to identify a robot, including its IP address. The user will automatically detect the robots in proximity through an Android interface, as illustrated in Figure 5, and thus can select the robot of interest to interact with.

Figure 6 presents the classes and components reused to implement the discovery application based on COROS architecture. The COROS architectural components and classes enabled us to easily implement the discovery application as a new ROS component that can be reused by other ROS applications that requires information about neighbor robots. In fact, we use the same discovery component in a distributed multi-robot applications, where robots had to discover each other, in addition to using it in this service robot to be discovered by the the android client device. In the former application, we used the C++ boost message serialization as the robots use the same boost library and thus are compatible with each other. In the latter case of the service robot, we used our developed JSON serialization modules, which is more convenient for communication between heterogenous devices namely the C++ based robot and Java-based android device.

C. Delivery Application

We developed a use case where the robot assists staff and faculty members to exchange courier between offices or coffee from central cafe to offices. A video demonstrator is presented

in [11]. In this case, we have used two approaches to send the mission (1) using ROS to JSON serialization interface defined in the COROS framework, (2) using the Web Services interfaces (`ros-ws` and `ros-rs`). A video demonstration illustration on how to program and use Web Services interfaces with ROS is presented in [12]. As illustrated in Figure 7 that presents the architecture of the delivery application, the user selects the source and destination offices through the Android client application, then, the coordinates of corresponding locations of the selected offices are fetched from a MySQL database on the cloud based on their names, so that the end-user does not have to write manually the real coordinates on the map, but just selects logical locations. These coordinates are sent through a serialized JSON message to robot, which will be processed by the back-end ROS application, developed using the COROS architecture.

In case of using the Web services interfaces, the ROS Web Service layer will invoke the appropriate Web service method corresponding to the selected mission of the robot, in our case the delivery Web service; and once invoked, the Web method submits the mission to the back-end ROS application that will process the message accordingly.

The back-end ROS application remains listening on the topic `DeliveryRequestMsg/from_json`, which receives ROS messages sent by the publisher of the dispatcher component. The latter deserializes the JSON message incoming from the mobile client application or from the Web service method. Then, it converts the received JSON message to a ROS message published to the `MybotDeliveryOperator`, which is the software agent responsible for processing and executing the query. With respect to Figure 1, the `MybotDeliveryOperator` is an implementation of the `Agent Operator` in the Application Logic Support Layer.

The delivery application was extensively tested in simulation mode and experiments and demonstrated to be correctly operational. In the experimental mode, we though observed that the Turtlebot could not sometimes perform rotation correctly on the smooth ground of our University. This is serious limitation of the Turtlebot platform wheels which slides on ceramic ground.

VI. CONCLUSIONS

In this paper, we propose a layered service-oriented software architecture for service robots using ROS. The architecture comprises (*i.*) the COROS framework that provide abstraction layers to building complex distributed applications, and (*ii.*) Web services layer that expose ROS ecosystem through SOAP and REST Web services. We designed a meta-model for the integration of Web services into ROS.

We are working towards extending our architecture to virtualize the robot access through the cloud. In fact, we aim integrating service robots to the cloud to promote sharing of resources and allow for an easier access and interaction between users and service robots through the cloud. The cloud will present an infrastructure for storage and processing of the service robots.

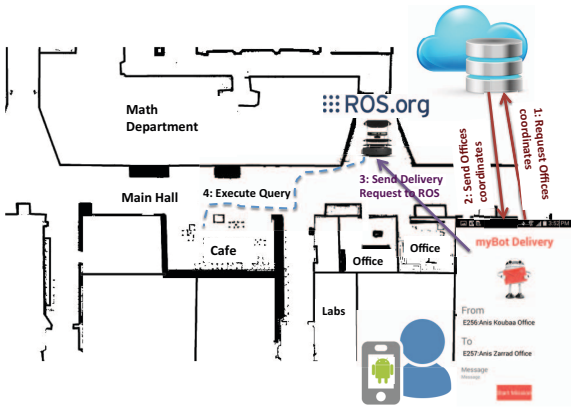

Fig. 7. Courier Delivery Scenario

ACKNOWLEDGMENTS

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

REFERENCES

[1] M. Waibel, M. Beetz, J. Civera, R. D'Andrea, J. Elfring, D. Galvez-Lopez, K. Haussermann, R. Janssen, J. M. M. Montiel, A. Perzylo, B. Schiessle, M. Tenorth, O. Zweigle, and R. van de Molengraft, "RoboEarth," *Robotics Automation Magazine, IEEE*, vol. 18, pp. 69–82, June 2011.

[2] "World Robotics 2013 Service Robots," 2013.

[3] Y. Kim and W. Yoon, "Generating task-oriented interactions of service robots," *Systems, Man, and Cybernetics: Systems, IEEE Transactions on*, vol. 44, pp. 981–994, Aug 2014.

[4] L. Gherardi and D. Brugali, "Modeling and Reusing Robotic Software Architectures: The HyperFlex Toolchain," in *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*, pp. 6414–6420, 2014.

[5] D. Brugali and L. Gherardi, "HyperFlex: a Model Driven Toolchain for Designing and Configuring Software Control Systems for Autonomous Robots," in *Robot Operating System (ROS) - The Complete Reference (to appear)* (A. Koubaa, ed.), vol. 604 of *Studies in Systems, Decision and Control*, Springer International Publishing, 2015.

[6] N. Hendrich, H. Bistry, and J. Zhang, "PEIS, MIRA, and ROS: Three frameworks, one service robot – A tale of Integration," in *Robotics and Biomimetics (ROBIO), 2014 IEEE International Conference on*, pp. 1749–1756, Dec 2014.

[7] N. Chivarov, S. Shivarov, K. Yovchev, D. Chikurtev, and N. Shivarov, "Intelligent Modular Service Mobile Robot ROBCO 12 for Elderly and Disabled Persons Care," in *Robotics in Alpe Adria Danube Region (RAAD), 2014 23rd International Conference on*, pp. 1–6, Sept 2014.

[8] "Introducing ROStful: ROS over RESTful web services, http://www.ros.org/news/2014/02/introducing-rostful-ros-over-restful-web-services.html," 2015.

[9] "ROS Tutorials at COINS Research Group, http://wiki.coins-lab.org/index.php?title=ros_tutorials," 2015.

[10] A. Koubaa, M.-F. Sriti, H. Bennaceur, A. Ammar, Y. Javed, M. Alajlan, N. Al-Elaiwi, M. Tounsi, and E. Shakshuki, "COROS: A Multi-Agent Software Architecture for Cooperative and Autonomous Service Robots," in *Cooperative Robots and Sensor Networks 2015* (A. Koubaa and J. Martinez-de Dios, eds.), vol. 604 of *Studies in Computational Intelligence*, pp. 3–30, Springer International Publishing, 2015.

[11] "MyBot Courier Delivery Demonstrator, https://www.youtube.com/watch?v=oTLtmX2-ucA," 2015.

[12] "MyBot Cafe Delivery using Web Services Interfaces, https://www.youtube.com/watch?v=WvjY5XjAX7U," 2015.