COROS: A Multi-Agent Software Architecture for Cooperative and Autonomous Service Robots

Anis Koubâa, Mohamed-Foued Sriti, Hachemi Bennaceur, Adel Ammar, Yasir Javed, Maram Alajlan, Nada Al-Elaiwi, Mohamed Tounsi and Elhadi Shakshuki

Abstract Building distributed applications for cooperative 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 a single autonomous robot, cooperative multi-robot systems require additional care in the design of software components to ensure communication and coordination between the robotic

A. Koubâa (✉) · Y. Javed · M. Tounsi
Prince Sultan University, Riyadh, Saudi Arabia
e-mail: akoubaa@coins-lab.org

Y. Javed
e-mail: yasir.javed@coins-lab.org

M. Tounsi
e-mail: mtounsi@cis.psu.edu.sa

M.-F. Sriti · H. Bennaceur · A. Ammar · M. Alajlan
College of Computer and Information Sciences, Al Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh, Kingdom of Saudi Arabia
e-mail: mfsriti@ccis.imamu.edu.sa

H. Bennaceur
e-mail: hachemi@ccis.imamu.edu.sa

A. Ammar
e-mail: adel.ammar@ccis.imamu.edu.sa

M. Alajlan
e-mail: maram.ajlan@coins-lab.org

A. Koubâa · Y. Javed · M. Alajlan · N. Al-Elaiwi
COINS Research Group, Riyadh, Saudi Arabia

A. Koubâa
CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal

E. Shakshuki
Acadia University, Wolfville, Canada
e-mail: elhadi.shakshuki@acadiau.ca

N. Al-Elaiwi
King Abdulaziz City for Science and Technology (KACST), Riyadh, Saudi Arabia
e-mail: nalelaiwi@kacst.edu.sa

A. Koubâa and J.R. Martínez-de Dios (eds.), *Cooperative Robots and Sensor Networks 2015*, Studies in Computational Intelligence 604,
DOI 10.1007/978-3-319-18299-5_1

agents. This chapter proposes COROS, a new multi-agent software architecture for cooperative and autonomous service robots with the objective to make easier the design and development of multi-robot applications. We present a high-level conceptual architecture for multi-agent robotics systems that represents a generic framework for cooperative multi-robot applications. Furthermore, we present an instantiation of this generic architecture with an implementation software architecture on top of the Robot Operating System (ROS) middleware. The proposed concrete software architecture follows a component-based approach to ensure modularity, software reuse, extensibility and scalability of the multi-robot operational software. In addition, one major added value of our architecture is that it provides a tangible solution to supporting multi-robot software development for the ROS middleware, as ROS was originally designed for single-robot applications. We also demonstrate a sample of real-world case studies of cooperative and autonomous service robots applications in an office-like environment, including discovery and courier delivery applications.

Keywords Autonomous service robots · Cooperative robots · Robotic software engineering · Multi-agent systems · Robot operating system (ROS)

1 Introduction

The design of efficient software architecture for service robots (e.g. home automation, indoor surveillance, elderly people care, etc.) is an increasingly important issue in robotics research considering the expansion of their market. Indeed, the demand for these service robots is increasing as the International Federation of Robotics reported in its statistics that 3 million service robots for personal and domestic use were sold in 2012, which represents 20 % more than in 2011, increasing sales up to US$1.2 billion [1]. The success of deployment of service robots at public scale is tightly coupled with the efficiency of software design for service robots applications. A first major challenge with respect to robotic software engineering is the heterogeneity of robotic hardware platform in addition to the absence of standards. One common solution to this problem is the use of robotic middlewares that provide abstraction layers to robotic sensors and actuators. There has been several initiatives for robots middlewares including the Player/Stage project [2], Middleware for Robotic Applications (MIRA) cross-platform framework [3], the Mobile Robot Programming Toolkit (MRPT) [4], and the widely-used Robot Operating System (ROS) [5]. While these middlewares are effective in helping applications developers avoiding the programming complexity of low-level hardware components, they do not provide sufficient abstractions to design complex applications for service robots. For that purpose, several high-level robotic software architectures [6–9] were proposed in the literature to provide a conceptual view of software components and their interactions needed for specifying robotic services. Most of the aforementioned works and middlewares

addressed the architectural design for single robot applications, which might present limitations when applied to multi-robot systems. Indeed, multi-robot systems are typically more challenging in terms of software engineering since they exhibit additional requirements as compared to single robot systems, including communication and coordination to accomplish their missions. Some previous works (e.g. [10]) have also proposed system architectures for multi-robot applications.

In this book chapter, we consider the problem of designing software architecture for cooperative multi-robot applications. Indeed, in the context of iroboapp project [11], we have been working on designing intelligent applications for single and multiple robots in two main research directions, namely, (1) global path planning and (2) the multi-robot task allocation (MRTA) problem. In this project, we conducted a detailed study of the performance of a set of meta-heuristics for single and multi-robots path planning, and multi-robot task allocation. This study was completed with the design of several techniques for these problems. Many simulations of these techniques have been carried out to prove their efficiency. One of our objectives of the project is to validate our findings through real-world implementations and experimentation on robots, which led us to design a software architecture for multi-robots applications. We opted for the use of ROS as programming middleware to have an abstraction layer on top of robotic hardware, sensors and actuators. However, we found out that ROS was designed for single robots applications and several challenges were faced to efficiently implement multi-robot applications on top of ROS. This conducted us to re-think about efficient implementation strategies to support multi-robot in the ROS middleware. As an example, ROS is heavily based on concept of topics which represent a particular stream of data that might be associated to a sensor device (e.g. laser data, camera images, …), actuator (e.g. motor status), or any other program-logic (grid map, user-defined data). Topics basically describe the internal status of the robot, and in the case of cooperative robot applications, there is a need to exchange robots' mutual status. However, ROS does not natively support this kind of interaction. In this work, we provide a solution to this problem in two steps. We first analyse the high-level conceptual requirements of multi-agent robotics systems. Then, we present COROS which is a generic software architecture for ROS-enabled robot that promotes the effectiveness of building new distributed robotic applications.

The summary of contributions are four-folded. First, we present and analyze, in Sect. 3, the state-of-the art of software engineering in robotics and the different approaches adopted to build complex software for single and multi-robots applications. Second, in Sect. 4, we propose COROS, a multi-agent software architecture, at both conceptual level and implementation level, that facilitates the design and development of multi-robot applications and promotes software reuse, modularity and extensibility. Third, we demonstrate how COROS can be integrated with ROS middleware. Finally, in Sect. 5 we present two real-world experimental scenarios to demonstrate the effectiveness of COROS architecture in development of real distributed multi-robot applications.

2 Background

2.1 Robotic Operating System (ROS)

In this section, we present a general overview of the basic concepts of ROS framework [5] to provide the required background needed to understand the software architecture proposed in this chapter. This overview is not intended to be comprehensive but just an introduction of the important concepts, and interested readers may refer to [12] for details.

ROS (Robot Operating System) has been developed, by Willow Garage [13] and Stanford University as a part of STAIR [14] project, as a free and open-source robotic middleware for the large-scale development of complex robotic systems.

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 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

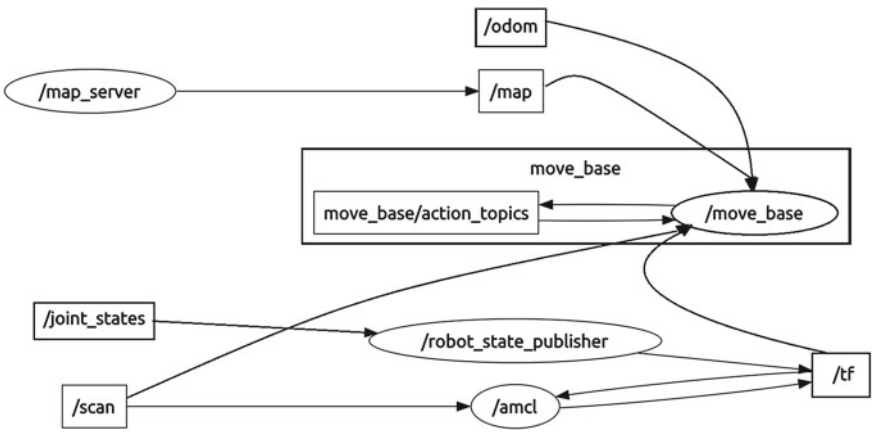

Fig. 1 Example of a ROS computation graph

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.

ROS and Multi-Robots: ROS was originally designed for single robot systems, but allows several robot machines or workstation machines to communicate through ROS topics in condition that one of the machines should run the *roscore* node, that is the node that identifies the *ROS Master*. This approach cannot be considered as distributed, although it allows to run several robots, because it relies on a central machine to run the *roscore*, which is prone to the typical single point of failure problem and is not scalable. There are some contributed ROS packages such as *foreign* − *relay* that relays the topic over multiple robots running different ROS Masters. Also a ROS package called *wifi_comm* based on *foreign* − *relay* has been developed allowing peer-to-peer communication between multiple robots. Another effort is the *ros* − *rt* − *wmp* ROS contributed package which aims at replicating whatever ROS topic or service in another computer wirelessly connected with the source without the need of sharing the same *ROS Master*. This approach is more scalable than other approaches but heavily based on a specific routing protocol called RT-WMP [15]. However, all these contributed works are still in their early phases and do not provide comprehensive solutions to develop complex multi-robot systems.

3 Related Works

Till today, there are no standard general purpose guidelines that can be used in the design and development of robotics software, which made development of service robotics system very complicated, inflexible, and increased integration complexity. In fact, robot software developers often experience a sense of frustration when they have to develop from scratch a new application, which is almost the same as several other releases for different projects, because they have not been able to capture and exploit the commonalties. Moreover, current robot programming languages are reaching their limits. They are not flexible and powerful enough to master the challenges imposed by the intended future applications, such as cooperative service robots. Consequently, there is a pressing need to engineer the software development process in order to design reusable robotic software systems and to implement flexible, modular, interoperable code. By using a composition of reusable building blocks, software infrastructures, and unified design techniques, it is possible to reduce the cost and time-to-market of robotic applications while preserving their efficiency, robustness, safety, and reliability [16]. Unfortunately, the adoption of such a software engineering approach by robotics researchers has been slow, impeded by the tradition of individual research groups crafting independent and incompatible solutions to common problems. Moreover, standardization is not yet fully reached, in particular for applications that go beyond industrial robotics, making difficult to realize an effective interoperability of solutions developed for different problems [17]. The complexity of service robots due to high interaction with humans, the integration of numerous sensors and the complexity of components require an easy, efficient and flexible integration and control, that is a better robot system architecture. Besides, the interaction between robot components require a good management at earlier stages for better synchronization. In the literature, there has been several approaches and attempts to address this gap. In what follows, we present a sample of these research efforts.

Component-Based Approaches: A variety of approaches for robotic software development have been proposed, due to the wide range of domains where robots are used, the many forms and functions that a robot can have and perform, and the diversity of robotics researchers [17]. In this scope, references [18, 19] proposed component-based architecture models for robotics middlewares designed for intelligent mobile robots. The advantage of component-based software engineering (CBSE) is that it allows software systems to be created and maintained at lower costs and with increased stability and extensibility through reuse of approved components in flexible software architectures [20], while such important features tend to be neglected in the race toward computational performance. The authors in [18] argued that robotics is particularly well-suited for and in need of component-based approaches. For that purpose, they introduced Orca, an open-source component-based software engineering framework for mobile robotics associated with a repository of free, reusable components for building mobile robotic systems. Several indoor and outdoor projects underway implement robotic systems using Orca components. Nevertheless, Orca does not prescribe a particular architecture. The architecture is rather defined by the

set of components that are chosen and the manner in which they are composed. As pointed out by [21], the CBSE approach has to be complemented with a careful analysis of the application domain, which should lead to the definition of stable data structures and interfaces to effectively achieve component reusability.

Iborra et al. [22] presented a summary of different software practices in the robotic domains, including domain engineering, reference architectures, and component-oriented development, then they focused on model-driven engineering design. They presented a successful implementation named Architectural Framework for Control Units (ACRoSeT) within a European project (EFTCoR) for climbing vehicles, using component-based design paradigm [18, 23, 24]. To overcome limitations of the component-oriented development, the authors implemented a model-driven architecture which proposes the use of models as the principal artifact for software development; a model being a simplified representation of reality that shows only the aspects that are of interest.

Robotic Software Development Frameworks: Calisi et al. [17] identified four key features that a framework for robotic software development must address: concurrency model, information sharing model, support tools, and interoperability. These features have a significant impact on the software development cycle. The observation of these key features led the authors to design OpenRDK, a framework for robotic software components with a multi-threaded multi-processes structure and a blackboard-type inter-module communication and data sharing. This framework supports data exchange among the modules as well as the ability to inspect data and processes and thus to build tools that suitably support debugging. It can be exploited to design the software for a wide class of robotic systems.

In [25], the authors proposed a new design patterns architecture to synchronize the interaction of different robot's components, named Task-State-Pattern. The authors were able to provide an abstract coordination interface and architecture. Such work can ease the job of designers and programmers of the robot systems. In [26], a new object oriented API for robot controls has been suggested which uses advanced concepts like sensor-based motions and multi-robot synchronization. Such an API was provided to realize some advanced motion control concepts, and to overcome some shortages in existing systems, such as in KRL, where multiple robots synchronization is difficult to achieve. The authors suggested a layered vendor-independent software architecture for industrial robot application development; an object-oriented framework which improves a series of shortages of current robotics languages, such as multi-robot cooperation and sensor-guided tasks.

In the SoftRobot research project [26], a consortium of academic and industrial partners analyzed the requirements of current and future applications of industrial robots. Based on this analysis, they developed a software architecture that enables object-oriented software development for industrial robot systems using general-purpose programming languages. This architecture allows specifying real-time critical operations of robots and tools, including advanced concepts like sensor-based motions and multi-robot synchronization.

Westhoff and Zhang [6] proposed a software architecture framework of multi-modal service robots based upon the Roblet-Technology which is evaluated on the service robot TASER of the TAMS Institute at the University of Hamburg. The objective of the authors was to enable the programmer to easily develop advanced applications. The framework was designed for networked applications by providing a layer that encapsulates network programming. Also, it supports component-oriented approach to easily integrate the existing solutions. Developing an application based on this software framework can be easily transferred to other robotic systems, and it enables the building of high-level applications easily. As the framework hides the network programming, it allows to upgrade the programming and testing of applications in service robotics.

A*STAR Social Robotics Laboratory (ASORO) developed a software architecture framework for service robots (SAFSR) [27] that can be configured and programmed using the Extensible Markup Language (XML) scripting language. Their framework consists of several key independent software modules, which are grouped into four layers: cognitive layer, execution and control layer, modality layer and device layer. The objective of [27] was to make SAFSR architecture flexible, extensible, and maintainable. SAFSR has been successfully applied to the development of three different service robots. After exhibiting the three service robots in various events and including them in several technical challenges, the authors concluded that these robots successfully demonstrated the features of the SAFSR; reducing time and complexity required to design and implement intelligent service robotics in different service domains.

Finally, Luzzana [28] proposed an integration between SCA (Service Component Architecture) and ROS that allows the developers to fully exploit the benefits of both approaches while mitigating their deficiencies. A bridge software component connects a ROS-based subsystem with a SCA-based subsystem in such a way that this integration is transparent to each other. The effectiveness of the proposed approach was demonstrated by applying it to a new mobile robot with a distributed computation architecture that enables the developers to experiment several different design approaches. In our work, we also aim at providing an abstraction layer to network programming to make easier the development of distributed robotics applications. The added value of the present work is that we propose a multi-agent component-based approach to build abstraction layers on top of the ROS middleware.

Software Architecture for Multi-Robots Systems DeLoach et al. [29] observed that while many recent agent-oriented architectures have been developed, there have been few attempts at applying high-level approaches to cooperative robotics systems design. Whereas using multi-agent approaches for cooperative robotics may provide some of the missing elements evidenced in many cooperative robotic applications, such as generality, adaptive organization, and fault tolerance. Hence, DeLoach et al. [29] applied the Multiagent Systems Engineering (MaSE) methodology to design high-level cooperative behaviors between autonomous and heterogeneous robots for search and rescue applications. MaSE [30] is a general purpose seven-step process

associated with a detailed sequence of inter-related graphically based models, which guides a system developer from an initial system specification through implementation of heterogeneous multiagent systems. It provides a top-down approach to building cooperative robotic systems instead of the behavior-based bottom up approach employed in traditional robotic implementations [31]. Besides, it was designed to be independent of any particular multiagent system architecture, programming language, or communication framework, which makes it fitting to implement cooperative robot applications. Nevertheless, contrary to standard robotic architectures, DeLoach et al. [29] focused on designing only high-level cooperative behaviors because they assumed that the low-level behaviors common to mobile robots already exist in libraries.

More specifically, Matson and DeLoach [32] proposed an organization-based multi-agent system model (OMAS) to overcome the problem of losing sensor capabilities in robots in dangerous environments. The goal was to build fault tolerant system and architectures that deal with detecting and handling sensor failure and faults, and calibration of sensors to adapt to unknown environmental conditions. The proposed model tolerates faults by managing the available hardware sensors as a group, focusing on managing their entire set of capabilities instead of simple brute force approach to sensor switching in cases of failure. The results show that the OMAS would successfully reorganize when a valid adaptation was possible. Although Matson and DeLoach [32] deals with intra-robotic capability, the organization model can also be applied to multiple robots working as a team where the sensor capabilities of one robot can fail over to another robot in a complete or partial manner.

Focusing on the multi-agent perspective, Silva et al. [33] presented a Gaia-based generic model for a multi-agent system based on mobile robotic platforms along with two distinct models derived from it, one for open space environment and the other one for indoor environment. Gaia [34, 35] is a software engineering methodology that excludes requirements elicitation and implementation focusing on analysis and design of the system, for designing multi-robot applications. This methodology is both general, in that it is applicable to a wide range of multi-agent systems, and comprehensive, in that it deals with both the macro-level and the micro-level aspects of systems. The objective of [33] was to demonstrate how robots applications designers could model their systems faster and simpler by adapting the GAIA methodology. The authors concluded that GAIA methodology provides a high level of abstraction when compared to similar methodologies. Nevertheless, as pointed out by DeLoach et al. [29], GAIA methodology falls short when defining the interactions between agents. For this reason, we did not consider GAIA methodology in our work, but we opted for a component-based generic software architecture for multi-robot application and we demonstrate how to integrate it with ROS middleware.

4 Software Architecture for Cooperative Robots

4.1 Cooperative Multi-Robot Software Requirements

The overall objective in the development of the software architecture is to facilitate the programming of distributed applications for cooperative and autonomous service robots.

As an illustrative scenario, consider a team of multiple autonomous robots deployed in an indoor environment (e.g. office, home) that assist humans in daily activities. Consider also that these robots should coordinate among each other for any new mission to be executed. For example, when a user sends a command to the robots' team to deliver a courier from one office to another, the robots should coordinate so that only the robot with the lowest cost will execute the mission. Although the scenario seems to be simple there are a lot of challenges to be addressed from software engineering perspective.

These challenges can be enumerated in the following functional requirements:

- **Communication**: this is an essential requirement for distributed robotic applications as robots need to communicate to be able to coordinate and exchange information among each other. The main problem in communication is the limited wireless coverage of the robots, and this prevents some robots to be reachable from others. As such, each robot will rely on the partial information it will be getting from the neighbor robots to take decision on mission execution. For example, with limited coverage, two non-neighbor robots may decide to execute the same mission because of lack of communication.
- **Problem Solving**: for any cooperative robots application, a consensus must be achieved among robots for any mission to be executed. As such, each mission represents a new problem that must be solved in a distributed manner in all the robots. In the above example, the problem was to find and select the robot with the lowest cost to execute the task. This requires a specific agent in each robot responsible for finding an effective solution, in a distributed manner, for any new mission shared among the robots.
- **Knowledge base**: In distributed applications, each robot should have its own knowledge base continuously gathering information about the current environment, other robots, and accomplished and unaccomplished tasks. The knowledge base helps the robot taking optimal or good decisions for any new coming mission. This introduces another layer of intelligence, allowing the robots to better coordinate to efficiently execute the tasks. For example, if a robot is known to have failed accomplishing some previously assigned tasks, it can be excluded from coordination on future similar tasks.

On the other hand, in addition to common software engineering requirements namely modularity, extensibility and reuse, we considered the following four key non-functional requirements for the development of the software architecture.

- **Decentralization**: any algorithm or protocol to be implemented must be fully distributed in the sense that any application should not rely on a central agent (or unit) for taking decisions, but decisions should be taken by robots with the partial knowledge they would have about the system. This is a core requirement, as it is not realistic to assume that all robots are fully connected all the time with each other or with a central system, as they are typically dotted with limited range wireless transceivers.
- **Fault-Tolerance**: any application should continue its operation correctly even if some robots become faulty before/during/after the execution of any mission. This is indeed a natural consequence from the aforementioned decentralization requirement, which avoids the single point of failure problem. Indeed, if any robot fails or has its battery deplete, other robots should detect this and should manage to undertake the task previously assigned to the dead robot.
- **Performance measure**: it is important to be able to evaluate how were successful the set of robots to achieve the overall mission. Several criteria could be used for that purpose, some of them may depends on the application. The criterion may vary from the satisfactory behavior to an optimal behavior.
- **Heterogeneity**: In multi-robot systems, it is typical to deploy robots of different types and brands. One of the major requirements of our architecture is to support different types of robots, so that the same code be reused by different types of robots without major changes. This is an important issue to ensure modularity and code reuse.

4.2 Software Architecture for ROS-enabled Cooperative Robots

General Overview In this section, we present our multi-robot software architecture and we demonstrate how it is integrated with ROS. The COROS architecture is based on the concept of multi-agent, where an agent represents an independent entity, typically a robot machine (i.e. Robot Agent) or a monitoring or control workstation (i.e. Monitor Agent). Unless otherwise specified, by default we consider an *agent* as a Robot Agent. Each agent is composed of a set of *components* that build the internal behaviour of the robot and allow it to interact remotely with other agents. In what follows, we present the main components of the software architecture and explain how they satisfy the aforementioned requirements.

Architecture Components Figure 2 shows the component diagram of the software architecture. The software system is decomposed into five subsystems, each of which plays the role of a container of a set of components. These subsystems are:

1. **Communication**: this subsystem was designed to address communication requirement, as it ensures the inter-robot interaction between different agents. It comprises extensible and modular client and server components that enable agents

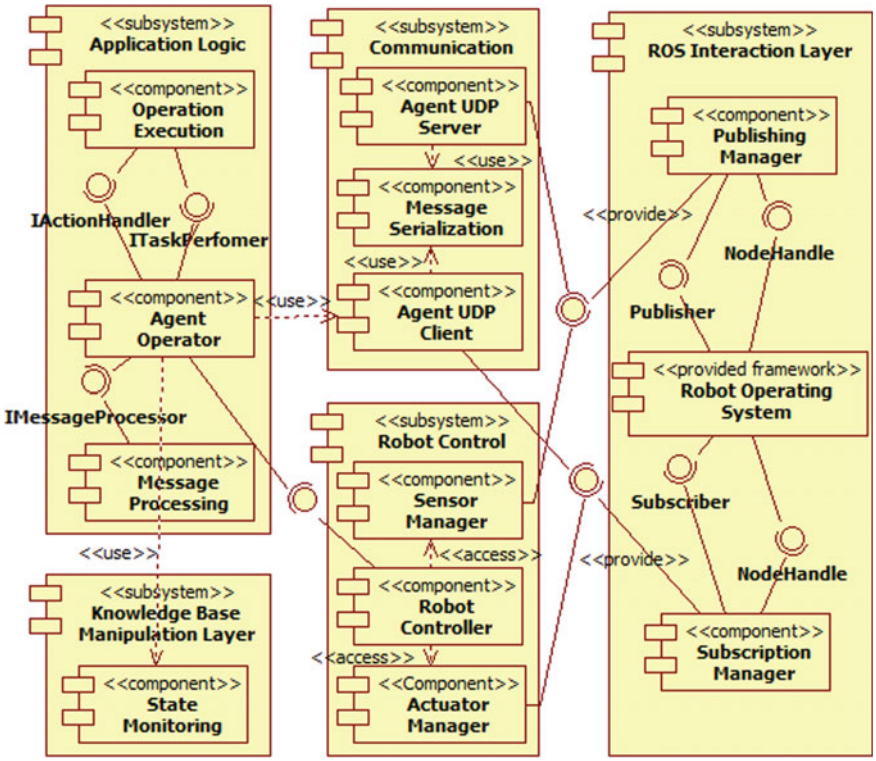

Fig. 2 COROS components

to exchange serialized messages through the network interface using sockets. One major challenge encountered is the incompatibility between socket-based messages and ROS messages. Indeed, as mentioned in Sect. 2, message-passing between nodes in ROS requires a particular message structure that is different from the structure of message received from/send through network interfaces. For that purpose, we designed a new component for message processing which maps any message between ROS and network interfaces. Indeed, any message received through sockets is converted to a ROS-compatible message and vice-versa.

2. **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 *PublishingManager* and *SubscriptionManager* without having to directly interact with ROS. These components will use the *NodeHandle* object, needed to create any ROS node, of the class that uses them to publish or subscribe to topics.

These components define (*key*, *value*) map data structures to manage topics to be published or subscribed, where each topic is identified by a unique key in the map. This key is used to as topic reference to publish or subscribe to any topic. ROS services are also supported in the same way.

3. **Robot Control**: this subsystem also adds a second 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 abstraction 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 to management of heterogeneous robots as they adhere to a common component model. Any robot type can be easily configured to provide the interfaces provided by the robot controller components.
4. **Application Logic**: this subsystem addresses the problem solving requirements; it encapsulates all of the components needed to implement a complete multi-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. As mentioned above, the message processor is used to provide a mapping between Socket-based messages and ROS messages, and make message serialization/de-serialization upon sending through/receiving from network interfaces. For the execution of a task (e.g. moving to a certain location), the Agent Operator interact with the Operation Execution component through two possible interfaces: (1) the Task Performer interface (ITaskPerformer) or (2) the Action Handler interface (IActionHandler).
5. **Knowledge Base Manipulation Layer**: This subsystem aims at satisfying knowledge base requirement and maintain an 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 gather 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.

Notice that in the architecture of Fig. 2, there is no direct link, association or interface between the Client and the Server, or between the Robot Control components. Indeed, the communication between these two components is ensured by ROS topics in the case of local (inter-process) interactions, and by the UDP protocol in the case

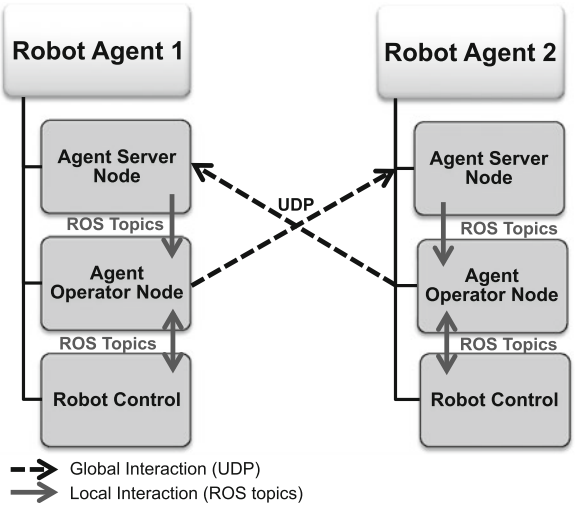

Fig. 3 Global and local interactions

of global (external) interactions. Figure 3 illustrates the interaction model between two robotic agents. In this figure, it is noted that the Client is not explicitly shown as the Agent Operator uses the Agent UDP Client component in a way that it extends all of its functionalities. Indeed, the Agent Operator sends network messages through the UDP Client Component.

The Agent UDP Server is a component that ensures the reception of messages through network interfaces using UDP sockets. This Agent UDP Server receives messages from any agent communicating through a specified application port. When a message is received, the Agent UDP Server forwards it to the Agent Operator through the ROS topic publishing mechanism. As a central component, the Agent Operator should subscribed to any message topic published by the Agent UDP Server, and then makes call to other interfaces from different components (e.g. *IMessageProcessor*, *ITaskPerformer*, and *IActionHandler*), when instantiated for a specific application, to define the logic of the underlying application.

Implementation Classes Figure 4 illustrates the Class Diagram that presents the detailed structure of each aforementioned subsystem, in addition to the relationships between classes. In what follows, we present some important points about the implemented classes and technical choices.

In the class diagram of Fig. 4, the five subsystems are depicted and they are interpreted as logic containers (i.e. doesn't exist at the file system level) of a set of implementation packages. At the file system level, we identify *packages* that are represented by separate folders, each of which represents either a part of a component (monitoring and state), a full component (processing, execution, operator, robot, sensors, and actuators), or two components or more (core). Physically, classes are embedded in their respective packages.

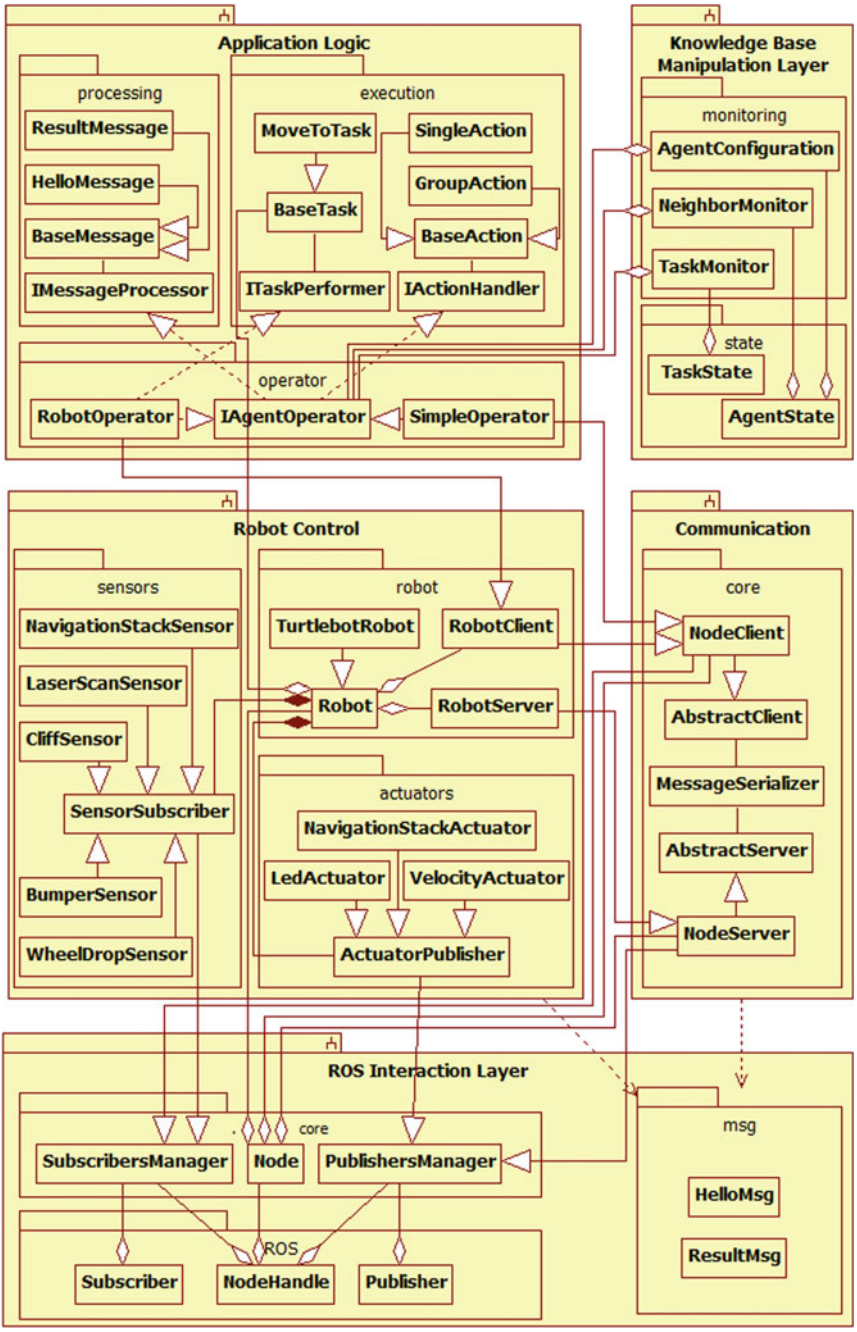

Fig. 4 COROS main classes

Table 1 COROS implementation packages usage

Subsystem	Package	Usage
ROS interaction layer	ROS	Refers to the (non-implemented, but) imported ROS library classes
	core	Simplifies the interaction between multi-robot application and ROS framework
	msg	Folder, container of ROS primitive message files (structure of messages used to publish in ROS topics)
Communication	core	Enables client/server communication using UDP protocol (different agents), client/server communication using ROS topics (same agent), and message serialization
Robot control	robot	Provides access to real robot properties, sensors and actuators. Also, enables client/server accessing robot functionalities
	sensors	Simplifies handling sensors topics subscription
	actuators	Simplifies handling actuators topics publishing
Knowledge base manipulation stack	state	Stores information about agents and tasks
	monitoring	Manages configuration about an agent and monitors agents and tasks states
Application logic	processing	Provides an interface and a base class for processing messages
	execution	Provides interfaces and base classes for handling actions and performing tasks
	operator	Concatenates the processing and execution provided interfaces

Tables 1 and 2 briefly describe the COROS implemented packages and classes, respectively. The objective of COROS implementation architecture (i.e. packages and classes) is to provide the robotic software developer with a set of reusable classes, that can be effectively and easily used/extended for developing new applications, which help speeding up the implementation process, and promoting the code reusability.

There are four major types of classes to be used for the development of any new application: (1) server (*NodeServer* and *RobotServer*), (2) operator (*SimpleOperator* and *RobotOperator*), (3) operation (*BaseAction* and *BaseTask*), and (4) message (*BaseMessage*). When developing a new application, the software developer needs to identify the agents composing the application and their different roles. According to the role of an agent, the choice of the server and the operator is based on whether to interact with a robot (i.e. *RobotServer* and *RobotOperator*) or not (i.e. *NodeServer* and *SimpleOperator*). Similarly, the choice of operations to be implemented is based on whether the operation involves an actuator (the operation is called *Task*), such as robot wheel motors, or not (the operation is called *Action*). To ensure code reuse, classes of type *Task*, *Action*, or *Message* should be derived

Table 2 COROS main classes description

Subsystem	Package	Class	Description
ROS interaction layer	core	Node	Encapsulates the NodeHandle. Aggregated by classes who need to communicate with ROS topics mechanism
		PublishersManager	Manages several ROS publishers using a single NodeHandle. Aggregated or extended by classes who need to create different topics' publishers and publish messages
		SubscribersManager	Manages several ROS subscribers using a single NodeHandle. Aggregated or extended by classes who need to create different topics' subscribers and receive messages
Communication	core	AbstractServer	Binds a UDP socket on a specified port and receives and deserializes messages
		AbstractClient	Serializes and sends or broadcasts messages using UDP protocol
		NodeServer	Enhances the UDP server with ROS capabilities (namely: publishing messages)
		NodeClient	Enhances the UDP client with ROS capabilities (namely: subscribing to topics)
		MessageSerializer	Provides serialization/deserialization for client and server
Robot control	robot	Robot	Represents a real robot with its properties and allows access to all robots sensors and actuators
		Turtlebot	Represents the Turtlebot robot, its properties and the needed sensor subscriptions and actuator publishing
		RobotServer	Enhances a ROS enabled UDP server to access to robot functionalities
		RobotClient	Enhances a ROS enabled UDP client to access to robot functionalities
	sensors	SensorSubscriber	Simplifies handling sensors topics subscription
		LaserScanSensor	Provides subscription to Laser Scan sensor captured data
	actuators	ActuatorPublisher	Simplifies handling actuators topics publishing
		NavigationStack Actuator	Allows publishing Navigation Stack data

(continued)

Table 2 (continued)

Subsystem	Package	Class	Description
Knowledge base manipulation stack	state	AgentState	Stores information about an agent (its ID, status, role, etc.)
		TaskState	Stores information about a task (its ID, status, performed by, etc.)
	monitoring	Agent-Configuration	Manages agent configuration information and state
		NeighborMonitor	Gathers information about the state of neighbor agents
		TaskMonitor	Gathers information about the state of tasks
Application logic	processing	IMessage-Processor	Interface, to be implemented by classes need to process messages
		BaseMessage	Base class for all application related messages
		HelloMessage	A message class to be used for announcing agent information to its neighbors
		ResultMessage	A message class to be used for announcing agent task end
	execution	IActionHandler	Interface, to be implemented by classes need to handle actions
		BaseAction	Base class for all application related actions
		SingleAction	Base class for all application related single actions; which execute once
		GroupAction	Base class for all application related group actions; which execute several times and save the context between different executions
		ITaskPerformer	Interface, to be implemented by classes need to perform tasks
		BaseTask	Base class for all application related tasks
		MoveToTask	A sample class that can be used as base class for tasks need to make the robot moving to a specific target
	operator	IAgentOperator	Interface, concatenates the methods of IMessageProcessor and IActionHandler and realizes some of them
		SimpleOperator	Extends NodeClient and realizes IAgentOperator to provide a client class capable to communicate using UDP and ROS topics, to process messages, and to handle actions
		RobotOperator	Extends NodeClient and realizes IAgentOperator and ITaskPerformer to provide a client class capable to communicate using UDP and ROS topics, to process messages, to handle actions, and to perform robot tasks

from *BaseTask*, *BaseAction*, and *BaseMessage*, respectively. With respect to the *Message* class instantiation, the implemented class should define a member function facilitating the message serialization. Another approach would be to encode any message as a string in the JSON format.

Conversion Between Application Messages and ROS Messages As mentioned above, there are two types of communications: (1) The communication between agents, carried out by the UDP protocol by using serialized messages, referred to as *Application Messages*, which is typically derived from *BaseMessage* class, and (2) communication between processes (ROS nodes) of a single agent referred to as inter-process communication and is carried out by *ROS Messages*. Once an agent receives an application message from its UDP interface, it must forward this received message to ROS system for processing it and performs required actions and/or tasks. For that purpose, we have developed a conversion between ROS Messages and Application Messages and vice-versa. To illustrate this through an example, consider the following ROS message structure used for the discovery application that allow neighbor agents to discover each other (refer to Sect. 5.1 for details about the application).

```
int32 message_code  # the code designating the needed operation
int32 agent_id      # the ID of the agent
string agent_ip     # the IP address of agent's machine
int32 agent_port    # the port number of agent's machine
string agent_role   # the role played by an agent in the app.
string agent_status # the current status of the agent
float64 timestamp   # the timestamp of the last status
```

At compilation time, ROS system auto-generates C++ header files corresponding to this ROS message structure. This ROS message will have an equivalent serialized Application Message. The idea of serialization is the simplest way to define a standard UDP client and server since the size of a string variable is easily computable, but is not the case to compute the size of the instance of a class. For that purpose, Boost Serialization[1] library were the most stable and best fit our need. The equivalent serialized application message is the following:

```
template<class Archive>
void DiscoveryMessage::serialize(Archive & ar,
  const unsigned int version){
    ar & message_code;
    ar & id;
    ar & ip;
    ar & port;
    ar & role;
    ar & status;
    ar & timestamp;
}
```

[1] http://www.boost.org/doc/libs/1_55_0/libs/serialization/doc/index.html.

4.3 Guidelines for Developing New Applications

In what follows, we provide a simple methodology that guides the robotic software developer in developing its own application using the COROS architecture. Indeed, the main goal behind proposing COROS architecture is to provide the main building blocks needed to develop distributed multi-robot applications favoring the cooperation between ROS-enabled robots. We define two main phases for the development of new applications:

1. **The Design Phase**
   (a) Determine how many agents (and their roles, e.g.: robot, monitor, etc.) are needed for the application.
   (b) Define the application message types and their structure.
   (c) Define ROS topics and messages files with their contents.
   (d) Design operations (actions and/or tasks) to process application messages.
   (e) Design ROS callback functions (subscriber side) to process ROS messages.
   (f) Determine addition information for the configuration of an agent (this is done only in the case of the standard provided configuration structure (AgentConfiguration class) does not suffice).
2. **The Implementation Phase**
   (a) Implement the application-specific configuration class (if any) as subclass of *AgentConfiguration*.
   (b) Create *ROSmessage* files according to ROS specific format.
   (c) Create *Applicationmessage* classes (subclass of *BaseMessage*) and implement serialization functions.
   (d) Create action classes (subclass of *BaseAction*) and task classes (subclass of *BaseTask*), then override their respective *execute()* functions.
   (e) Create a server class as a subclass of *NodeServer* or *RobotServer*, according to the agent role, for each agent by redefining their constructors, overriding the initialization methods *init()* for creating publishers of the specified topics, and overriding the *forward()* method for deserializing received application messages and publishing them as ROS messages to the operator.
   (f) Create an operator class as a subclass of *SimpleOperator* or *Robot-Operator*, according to the agent role, for each agent by redefining the constructor, and implementing *init()* method for creating subscribers to the specified topics, and the callback methods related to each topic subscription for message processing.

5 Experimental Validation

In what follows, we present two experimental application scenarios to validate COROS architectural concepts and to demonstrate how it is effective in building new distributed robotic applications while ensuring component reuse and extensibility.

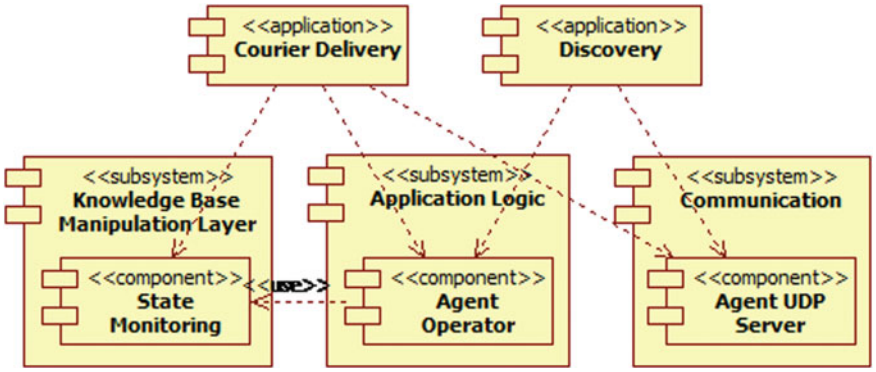

Fig. 5 Discovery and courier delivery applications in COROS architecture

The documentation and source code of COROS are available in [36]. The first scenario describes a discovery protocol which allows agents to discover their neighborhood. The second scenario describes a courier delivery from an office to another. Figure 5 depicts the different subsystems directly used by these two applications. The remaining subsystems are also used but indirectly.

5.1 Scenario 1: Discovery Protocol

Overview: In this section, we present the discovery protocol as an application that allows robots to discover their neighborhood, using the COROS architectural concepts. The reason behind implementing the discovery protocol is because it is needed by typical distributed robotic applications, since a robot basically needs to have information about its neighbors to be able cooperate with them in accomplishing missions. We build the discovery protocol as an independent ROS software component that can be used by any other ROS application.

The discovery protocol is simple. Each robot periodically sends a *HELLO* message carrying out information needed to identify a robot. Once a robot receives a *HELLO* message from a neighbor robot, it adds it to its neighbor list. The timestamp field allows to keep track of the last *HELLO* message update from a neighbor robot and can be used to discard robots with outdated record, so that to maintain the neighborhood list up-to-date when robots are moving.

Figure 6 presents the classes and components reused to implement the discovery application based on COROS architecture.

Implementation: As depicted in Fig. 6, the discovery package comprises two main classes: (1) The *DiscoveryOperator* is a class, which maps to an *AgentOperator* component in the COROS architecture, and that inherits from the *SimpleOperator* class providing all interfaces for processing any incoming message according to discovery application logic. For each received message, the robot will update its neighborhood

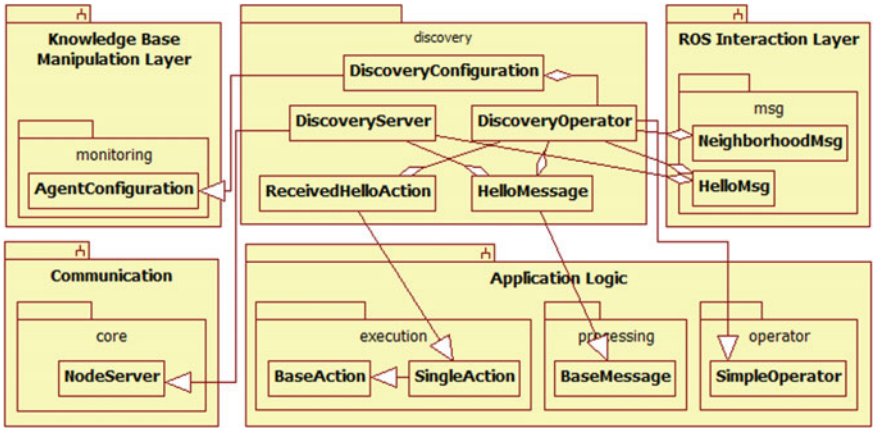

Fig. 6 Discovery application class diagram

table and publishes it as a *NeighborhoodMsg* topic which can be used by any ROS node in the robot. For example, in a bidding-auctionning process, a robot may need to contact its neighbor robots to bid on a certain task. (2) The *DiscoveryServer* is responsible for handling network communication between robots, by sending/receiving message to/from other robots. Received applications messages are decoded in the server class and forwarded as *HelloMsg* ROS messages to the *DiscoveryOperator*, which processes the message and executes the action accordingly, as explained above.

Discussion: The COROS architectural components and classes enabled us to easily implement the discovery application as a new ROS component that can be used by any other ROS node (i.e. process) that requires information about neighbor robots. Indeed, the discovery application can be executed as a set of ROS nodes, i.e. *DiscoveryServer* and *DiscoveryOperator*, that can be easily configured in a ROS launch file, taking as parameters the *RobotID*, the port number of the discovery application, the robot IP address and the frequency of *HELLO* messages. The discovery application updates the neighborhood table of the robot and publishes this table as *robot_id/neighborhood_list* topic that can be used by any ROS node by subscribing to this topic. Figure 7 depicts an example of execution of the discovery protocol in a robot and the *robot*1*/neighborhood_list* published topic. In this example, only one robot with ID equal to 1 is discovered in the neighborhood.

5.2 Scenario 2: Courier Delivery Application

Overview: In this section, we present the courier delivery application to test a more advanced scenario that involves cooperation between a team of robots and to demonstrate how COROS architecture is effective in building such distributed

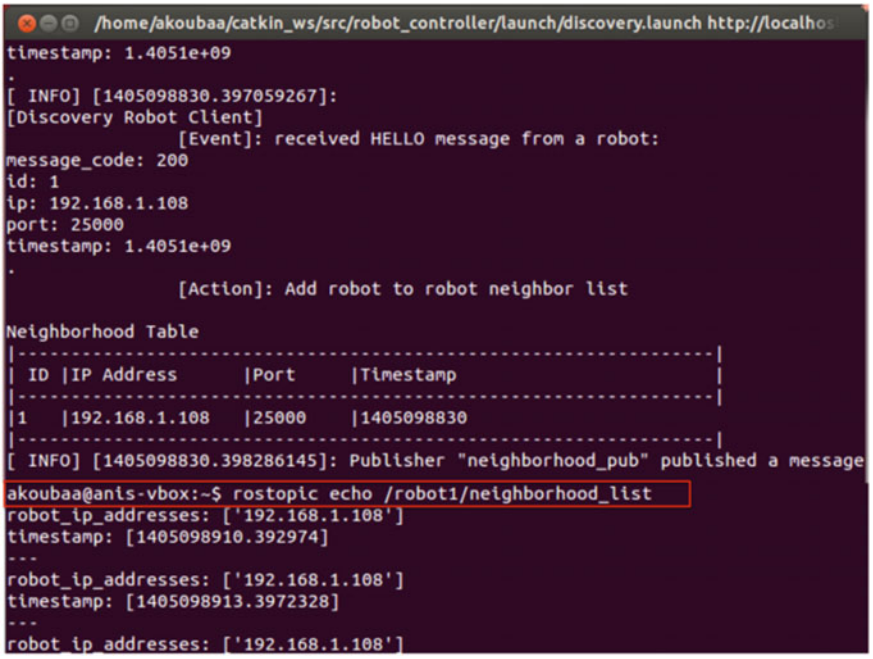

Fig. 7 Discovery protocol published topic and execution

applications. We choosed the courier delivery application as a proof-of-concept of the COROS architecture and its implementation.

We consider a team of N robots receive a command for delivering a courier from one office to another, coordinate together using a market-based mechanism to elect the robot with the lowest cost to execute the task. The scenario is as follows: A user sends a courier delivery mission as an auction to the team of robot. The auction message includes the location of the two offices, the sender of the courier and the receiver. Each robot receiving the courier delivery task calculates its bid for executing the action and sends the bid to the auctioneer. The bid represents an estimate of the total distance that the robot has to travel between the two offices starting from its location. Once all bids are received, the auctioneer selects the robot with the lowest bid to execution the mission. The winning robot moves to the sender office first to get the courier and then goes to the receiving office for delivery. We have deployed this application in the Computer Science department at Prince Sultan University using two robots and four offices as illustrated in Fig. 8. A map of the environment was established using ROS gmapping package. A video demonstration explaining the scenario is available in the iroboapp project page [37]. In this scenario, the robot with the lowest cost was select and has successfully delivered the courier from one office to another.

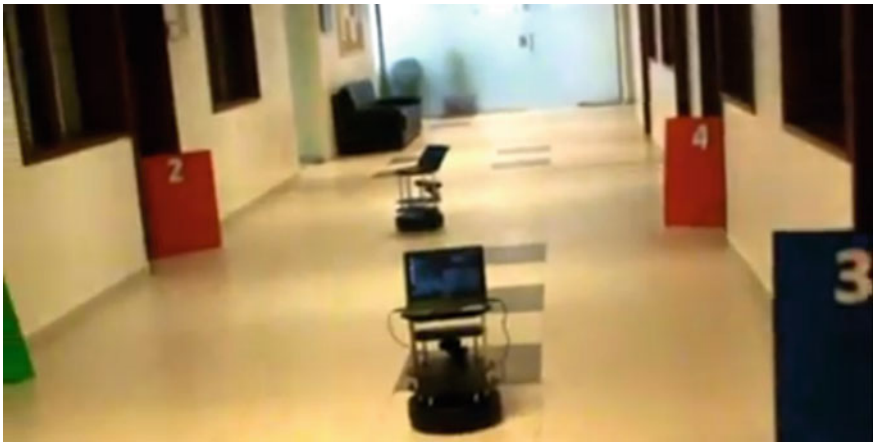

Fig. 8 Courier delivery deployment environment

Implementation: As depicted in Fig. 9, the courier delivery application package is implemented following COROS architectural concepts. The *Communication*, *ROS Interaction Layer*, *RobotControl*, *Application Logic* and *Knowledge Base Manipulation Layer* packages provide the required classes and components for the *courier* package that encodes the behavior of the application.

The courier packages comprises the following main modules: (1) a user module that includes a *CourierUserServer* a subclass of *NodeServer* and *CourierUserOperator* a subclass of *SimpleOperator* that allow an end-user to send mission to the robots and process received bids, respectively, (2) a robot module that includes *CourierRobotServer* a subclass of *RobotServer* and *CourierRobotOperator* a subclass of *RobotOperator* allowing a robot to receive mission orders and process commands. The user and robots exchange different types of messages depending on the type of interaction of the market-based protocol.

The execution of the delivery mission is ensure by the *CourierDeliveryTask* class, which reuses the *MoveToTask* generic class of the *execution* package to define how the robot should execute the mission, which in this case, moving to the sender office, then moving to the receiving office after getting the courier. The *CourierDeliveryTask* uses an instance of the *TurtlebotRobot* needed to have access to all sensor and actuator of the Turtlebot robot used in the experiments. It will be straightforward to use the same code on another type of robot, which just requires to change the type of robot instance to the appropriate one. Of course, it is needed to develop the Robot Control module for any new robot to be used in the COROS framework.

Discussion: Figure 10 depicts the deployment diagram of the courier application. It is clear that the COROS architecture facilitates us the development of the courier

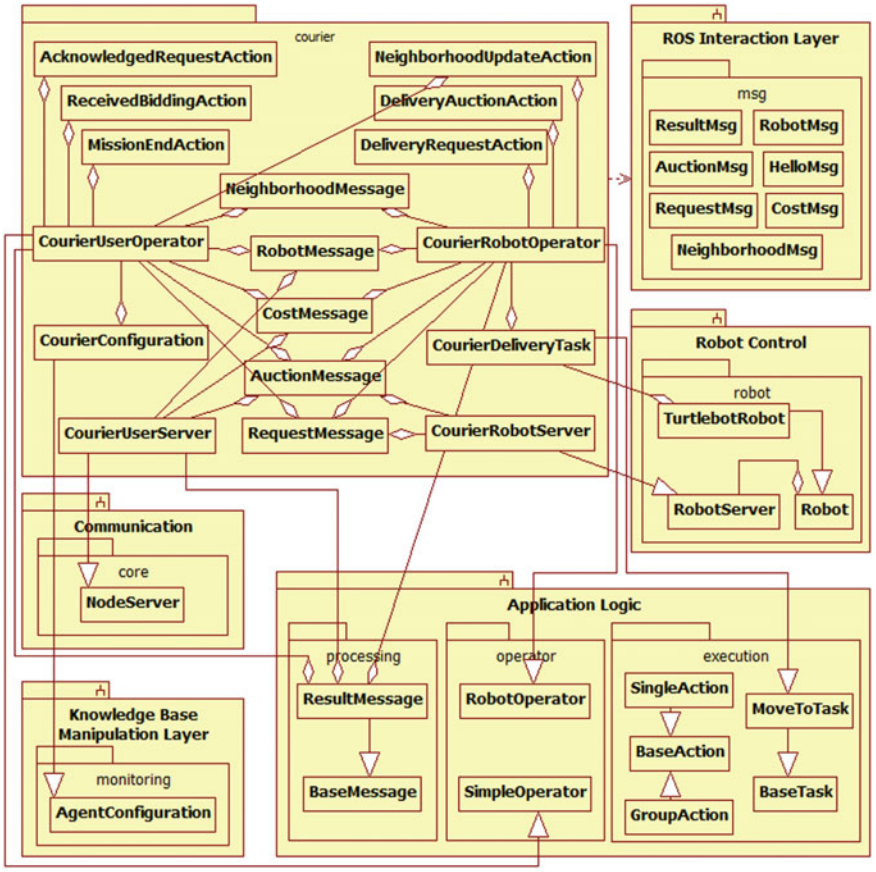

Fig. 9 Courier delivery implemented classes

application that involve both an end-user and a robot, each has its own server and operator components, as depicted in Fig. 10. It can be observed that the Discovery application was used as an independent component of the COROS framework in both the user and robot machines. Adding any other application in the COROS framework is easy and any application can interact with any other using the ROS middleware through ROS topics and services. In addition, all components can be configured using a ROS launch files.

The major advantage of the COROS architecture is that it allows building new distributed robotic applications by instantiating existing generic COROS classes and packages and making abstraction to several low-level implementation (e.g. network communication, ROS primitives) details embedded in these generic packages.

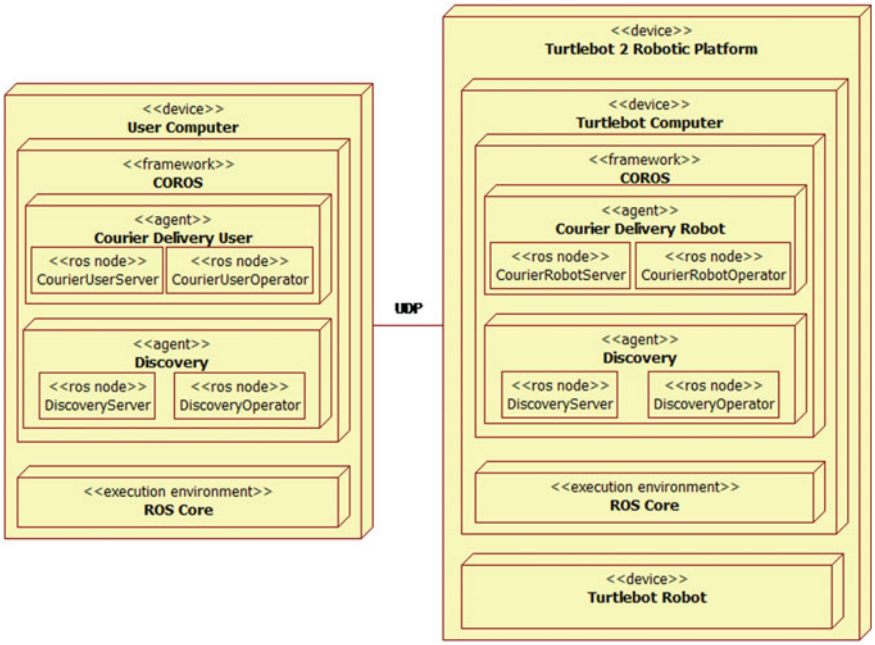

Fig. 10 Courier delivery application deployment based on COROS implementation

6 Conclusion and Future Works

In this chapter, we proposed COROS, a multi-agent software architecture for cooperative service robots applications. At conceptual level, COROS comprises five subsystems including communication, ROS interaction layer, robot control, application logic and knowledge bases. Each system provides an abstraction layer to the robotics software developer that allow to facilitate the design and implementation of new distributed robotic applications. The COROS framework has the advantage of promoting software reuse and modularity through the adoption of a component-based approach in the design of the architecture. The COROS architecture was extensively validated through two distributed applications including the discovery protocol and the courier delivery application in indoor environments. It has been demonstrated how COROS was effective in building these two applications.

Although the COROS framework was proven to be effective, we are planning several extensions for incorporating advanced functionalities. First, we have the objective of contributing to promote COROS to support Rapid Application Development (RAD) in robotic context by providing utilities to automate the creation of new agent servers and operators for any new application instead of editing them manually. Another extension consists in improving and standardizing message serialization. Currently, messages are serialized using the *boost* library and we aim at serialization

message into JSON or ROS format so that it becomes platform-independent. Furthermore, we aim at hiding more network complexity by adding an abstraction layer that allows the application developer to deal only with ROS topic without the need to handle network and message serialization issues.

We believe that COROS provides an important milestone in the design and development of distributed robotic application on top of the ROS middleware.

Acknowledgments This work is supported by the iroboapp project "Design and Analysis of Intelligent Algorithms for Robotic Problems and Applications" under the grant of the National Plan for Sciences, Technology and Innovation (NPSTI), managed by the Science and Technology Unit of Al-Imam Mohamed bin Saud University and by King AbdulAziz Center for Science and Technology (KACST). This work is also 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 National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project FCOMP-01-0124-FEDER-037281 (CISTER); by Prince Sultan University.

References

1. World Robotics 2013 Service Robots: http://www.ifr.org/service-robots/statistics/ (2013)
2. The Player/Stage Project: http://playerstage.sourceforge.net/
3. The MIRA Project: http://www.mira-project.org/
4. Mobile Robot Programming Toolkit (MRPT): http://www.mrpt.org/
5. Robot Operating System (ROS): http://www.ros.org
6. Westhoff, D., Zhang, J.: A unified Robotic software architecture for service Robotics and networks of smart sensors. In: Berns, K., Luksch, T. (eds.) Autonome Mobile Systeme 2007, Informatik Aktuell, pp. 126–132. Springer, Berlin (2007)
7. Kim, M., Kim, S., Park, S., Choi, M.-T., Kim, M., Gomaa, H.: UML-based service robot software development: a case study. In: Proceedings of the 28th International Conference on Software Engineering, ICSE'06, pp. 534–543. ACM, New York, USA (2006). doi:10.1145/1134285.1134360, http://doi.acm.org/10.1145/1134285.1134360
8. Kim, M., Kim, S., Park, S., Choi, M.-T., Kim, M., Gomaa, H.: UML-based service robot software development: a case study. In: Advances in Service Robotics, InTech, pp. 127–148 (2008).doi:10.5772/5947
9. Wojtczyk, M.: A new model to design software architectures for mobile service robots, Dissertation, Technische Universität München, München, Germany (2010)
10. Viguria, A., Maza, I., Ollero, A.: Distributed service-based cooperation in aerial/ground robot teams applied to fire detection and extinguishing missions. Adv. Robot. **24**(1–2), 1–23 (2010)
11. The Iroboapp Project: http://www.iroboapp.org (2014)
12. O'Kane, J.M.: A gentle introduction to ROS, independently published (2013). http://www.cse.sc.edu/jokane/agitr/
13. Wyrobek, K., Berger, E., Van der Loos, H., Salisbury, J.: Towards a personal Robotics development platform: rationale and design of an intrinsically safe personal Robot. In: IEEE International Conference on Robotics and Automation, ICRA 2008, pp. 2165–2170. IEEE (2008)
14. Quigley, M., Berger, E., Ng, A.Y.: Stair: hardware and software architecture. In: AAAI 2007 Robotics Workshop, Vancouver, BC, pp. 31–37 (2007)
15. Tardioli, D.: Real-time communication in wireless Ad-Hoc networks. The RT-WMP Protocol, Ph.D. thesis, Universidad de Zaragoza (October 2010)

16. Brugali, D., Prassler, E.: Software engineering for Robotics [from the guest editors]. IEEE Robot. Autom. Mag. **16**(1), 9–15 (2009)
17. Calisi, D., Censi, A., Iocchi, L., Nardi, D.: Design choices for modular and flexible Robotic software development: the OpenRDK viewpoint. J. Softw. Eng. Robot. **3**(1), 13–27 (2012)
18. Brooks, A., Kaupp, T., Makarenko, A., Williams, S., Oreback, A.: Towards component-based Robotics. In: 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, (IROS 2005), pp. 163–168. IEEE (2005)
19. Lee, T.-Y., Seo, H.-R., Lee, B.-H., Shin, D.-R.: A software component model and middleware architecture for intelligent mobile Robot. In: 2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE), vol. 4, pp. 453–456. IEEE (2010)
20. Hasselbring, W.: Component-Based Software Engineering. Handbook of Software Engineering and Knowledge Engineering
21. Broten, G., Mackay, D., Monckton, S., Collier, J.: The robotics experience. IEEE Robot. Autom. Mag. **16**(1), 46–54 (2009). doi:10.1109/MRA.2008.931632
22. Iborra, A., Caceres, D., Ortiz, F., Franco, J., Palma, P., Alvarez, B.: Design of service robots. IEEE Robot. Autom. Mag. **16**(1), 24–33 (2009)
23. Lau, K.-K., Wang, Z.: Software component models. IEEE Trans. Softw. Eng. **33**(10), 709–724 (2007)
24. Bruyninckx, H.: Open Robot control software: the OROCOS project. In: IEEE International Conference on Robotics and Automation. Proceedings 2001 ICRA, vol. 3, pp. 2523–2528. IEEE (2001)
25. Lütkebohle, I., Philippsen, R., Pradeep, V., Marder-Eppstein, E., Wachsmuth, S.: Generic middleware support for coordinating Robot software components: the task-state-pattern. J. Softw. Eng. Robot. **2**(1), 20–39 (2011)
26. Angerer, A., Hoffmann, A., Schierl, A., Vistein, M., Reif, W.: Robotics API: object-oriented software development for industrial Robots. J. Softw. Eng. Robot. **4**(1), 1–22 (2013)
27. Limbu, D., Tan, Y.-K., Jiang, R., Dung, T.A.: A software architecture framework for service Robots. In: 2011 IEEE International Conference on Robotics and Biomimetics (ROBIO), pp. 1736–1741. doi:10.1109/ROBIO.2011.6181540 (2011)
28. Luzzana, A.: Classification and integration of software component models for Robotics, Ph.D. thesis, Università degli studi di Bergamo (April 2013)
29. DeLoach, S.A., Matson, E.T., Li, Y.: Exploiting agent oriented software engineering in cooperative Robotics search and rescue. Int. J. Pattern Recognit. Artif. Intell. **17**(05), 817–835 (2003)
30. DeLoach, S.A., Wood, M.F., Sparkman, C.H.: Multiagent systems engineering. Int. J. Softw. Eng. Knowl. Eng. **11**(03), 231–258 (2001)
31. Parker, L.E.: Current state of the art in distributed autonomous mobile robotics. In: Distributed Autonomous Robotic Systems, vol. 4, pp. 3–12. Springer, Heidelberg (2000)
32. Matson, E., DeLoach, S.: Enabling intra-robotic capabilities adaptation using an organization-based multiagent system. In: 2004 IEEE International Conference on Robotics and Automation. Proceedings. ICRA'04, vol. 3, pp. 2135–2140. doi:10.1109/ROBOT.2004.1307378 (2004)
33. Silva, D., Braga, R.A.M., Reis, L., Oliveira, E.: A generic model for a Robotic agent system using GAIA methodology: two distinct implementations. In: 2010 IEEE Conference on Robotics Automation and Mechatronics (RAM), pp. 280–285. doi:10.1109/RAMECH.2010.5513176 (2010)
34. Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia methodology for agent-oriented analysis and design. Auton. Agent. Multi-Agent Syst. **3**(3), 285–312 (2000)
35. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: the Gaia methodology. ACM Trans. Softw. Eng. Methodol. (TOSEM) **12**(3), 317–370 (2003)
36. COROS: http://www.iroboapp.org/index.php?title=COROS (2014)
37. Office courier delivery application. video demonstration, iroboapp project. http://www.iroboapp.org/index.php?title=Videos (2014)