Middleware for Multi-robot Systems

Yuvraj Sahni, Jiannong Cao and Shan Jiang

Abstract Recent advances in robotics technology have made it viable to assign complex tasks to large numbers of inexpensive robots. The robots as an ensemble form into a multi-robot system (MRS), which can be utilized for many applications where a single robot is not efficient or feasible. MRS can be used for a wide variety of application domains such as military, agriculture, smart home, disaster relief, etc. It offers higher scalability, reliability, and efficiency as compared to single-robot system. However, it is nontrivial to develop and deploy MRS applications due to many challenging issues such as distributed computation, collaboration, coordination, and real-time integration of robotic modules and services. To make the development of multi-robot applications easier, researchers have proposed various middleware architectures to provide programming abstractions that help in managing the complexity and heterogeneity of hardware and applications. With the help of middleware, an application developer can concentrate on the high-level logic of applications instead of worrying about low-level hardware and network details. In this chapter, we survey state of the art in both distributed MRS and middleware being used for developing their applications. We provide a taxonomy that can be used to classify the MRS middleware and analyze existing middleware functionalities and features. Our work will help researchers and developers in the systematic understanding of middleware for MRS and in selecting or developing the appropriate middleware based on the application requirements.

Y. Sahni · J. Cao (✉) · S. Jiang
Department of Computing, The Hong Kong Polytechnic University, Kowloon, Hong Kong
e-mail: csjcao@comp.polyu.edu.hk

Y. Sahni
e-mail: csysahni@comp.polyu.edu.hk

S. Jiang
e-mail: cssjiang@comp.polyu.edu.hk

© Springer International Publishing AG, part of Springer Nature 2019
H. M. Ammari (ed.), *Mission-Oriented Sensor Networks and Systems: Art and Science*, Studies in Systems, Decision and Control 164,
https://doi.org/10.1007/978-3-319-92384-0_18

1 Introduction

Recent advances in robotics and other related fields have made it feasible for developers to build inexpensive robots. The current trend in robotics community is to use a group of robots to accomplish task objectives instead of using single-robot systems. These group of robots working in collaboration with each other form an ensemble which is commonly referred as a multi-robot system (MRS). Use of MRS provides better scalability, reliability, flexibility, and versatility, and helps in performing any task in a faster and cheaper way as compared to single-robot system [6]. MRS system can be very useful in search and surveillance applications especially for areas which are difficult or impossible for humans to access. Another benefit of MRS is that it has better spatial distribution [97]. Many applications such as underwater and space exploration, disaster relief, rescue missions in hazardous environments, military operations, medical surgeries, agriculture, smart home, etc. can make use of distributed group of robots working in collaboration with each other [6, 50]. It would not only be difficult but may also lead to wastage of resources if such applications are developed using single-robot systems.

The benefits provided by MRS do not come at low cost. MRS is a dynamic and distributed system where different robots are connected to each other using wireless connection. Robots in MRS should collaborate with each other to perform complex tasks such as navigation, planning, distributed computation, etc. but it is not as easy as the systems are usually heterogeneous. Heterogeneity in MRS can arise due to the use of heterogeneous hardware, software, operating system, or communication protocol and standards. Besides, the large number of robots used in the system makes the system development even more complicated. It is extremely difficult for a robotic system developer to develop such complex systems that should be robust, reliable, scalable, and support the real-time integration of heterogeneous components. Developing a complete robotic application requires knowledge from multiple disciplines such as mechanical engineering, electrical engineering, computer science, etc.

These complexities can be reduced by the use of middleware layer. Middleware provides programming abstractions for a developer so that the developer can focus on application logic instead of low-level details [20]. Many middleware architectures have been proposed for MRS. There is a wide range of applications for MRS, and each application has some specific requirements. It is not trivial to develop a middleware for MRS due to peculiar characteristics of MRS and diverse application requirements. The complexity of middleware becomes higher as more features are incorporated. In fact, it is extremely hard to develop a common middleware for all robotic applications [86]. Therefore, it is important to study different types of middleware to help make a better decision while selecting the middleware for an application.

In this chapter, we first study the recent developments in building MRS. We describe the key applications and requirements of MRS. We then describe the design goals and provide a feature tree-based taxonomy of MRS middleware for systematic understanding of middleware. After giving the background of MRS and the motivation for using middleware, we survey state of the art of middleware for MRS.

Although survey of middleware for robotics can be found in literature in [28, 48, 65, 66], they do not focus specifically on middleware for MRS. Besides, many new middleware architectures have been developed and have not been discussed in previous survey papers.

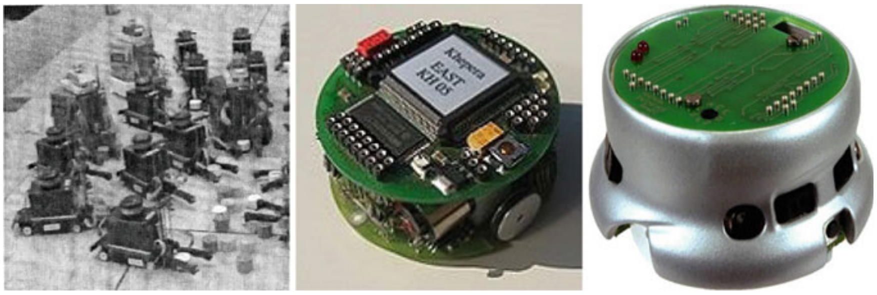
The contributions of this work are as follows:

- We describe the key requirements and applications of MRS. We also show the developments made by robotics community in building distributed MRS. This is useful for researchers and developers who are interested in developing a real testbed for MRS.
- We provide a feature tree-based taxonomy of MRS middleware features. We have considered features corresponding to both middleware and MRS. We utilize the structure of the phylogenetic tree to give a comprehensive framework that can be used by researchers for systematic understanding and comparison of different MRS middlewares. This is the first time such a taxonomy has been given specifically for MRS middleware.
- We have done a comprehensive review of existing middleware for MRS. 14 different middleware examples have been discussed in this work. We have also provided design goals for middleware and analyzed existing works. The review and analysis done in this chapter will be especially useful for beginners who are interested in developing their own multi-robot system. This work can also be used by developers and other researchers in selecting a suitable middleware based on their application requirements.

The remainder of this chapter is as follows. In Sect. 2, we discuss the recent developments in building MRS and provide a classification of robotic applications. In Sect. 3, we discuss the need of middleware for MRS and give some design goals for MRS middleware. In Sect. 4, we provide a feature tree-based taxonomy of MRS middleware. In Sect. 5, we do the comprehensive review of existing middleware for MRS. In Sect. 6, we provide an analysis of existing middleware for MRS.

2 Existing Multi-robot Systems and Applications

This section is divided into two subsections. In Sect. 2.1, we give some key requirements of MRS and then discuss the developments made by the robotic community in building distributed MRS. In Sect. 2.2, we give a classification of robotic applications. We answer two important questions in this section, which are: What is the current stage of development in MRS? and What are the different possible applications of MRS?



(a) Mataric R2 (b) Khepera (c) Khepera II

Fig. 1 Three kinds of robot for multi-robot system in early age

2.1 Existing Multi-robot Systems

Experimental evaluation and validation are important for research in MRS. It often happens that theoretical models and algorithms with perfect simulation results do not work under real-world conditions. In MRS, these divergences are even more amplified compared with single-robot system due to the large number of robots, interactions between robots, and the effects of asynchronous and distributed control, sensing, actuation, and communication. Therefore, it is crucial to build a testbed for MRS to conduct multi-robot research [64]. In this section, we list key requirements of an MRS and show how robotics community has progressed in building distributed MRS over the years.

One of the earliest multi-robot systems is the Mataric R2 robots built in the 1990s (shown in Fig. 1a). They use a group of four robots to demonstrate and verify the group behavior such as foraging, flocking, and cooperative learning [58]. For each Mataric R2 robot, it equips piezoelectric bump sensors for collision detection, two-pronged forklift for picking goods, six infrared sensors for object detection, and radio transceivers for broadcasting up to one byte of data per second. Nearly the same time, the K-Team from Switzerland developed Khepera robot team in 1996 and Khepera II robot team in 1999 [67] shown in Fig. 1b and Fig. 1c, respectively. The size of the robot is reduced from 36-cm long (Mataric R2 robot) to 8-cm long (Khepera and Khepera II). The Khepera II robot has stronger functionality than the Mataric R2 robot such as more powerful computation ability and more reliable wireless communication. Due to the development of electronic technology, the Khepera II robot also has a smaller size.

After the early age, more and more multi-robot systems are built in both laboratory and industry nowadays. Two representative multi-robot systems are Swarmbot [59, 60] developed by McLurkin and iRobot for research purpose in 2004 (shown in Fig. 2a) and Kiva [96] developed by Amazon for warehouse usage in 2007 (shown in Fig. 2b). Also, the research community has organized a lot of multi-robot competitions such as RoboCup for robotic soccer, MAGIC competition for military surveillance,

(a) Swarmbot (b) Kiva

Fig. 2 Two representative multi-robot systems in recent years: Swarmbot for research purpose and Kiva for industry usage

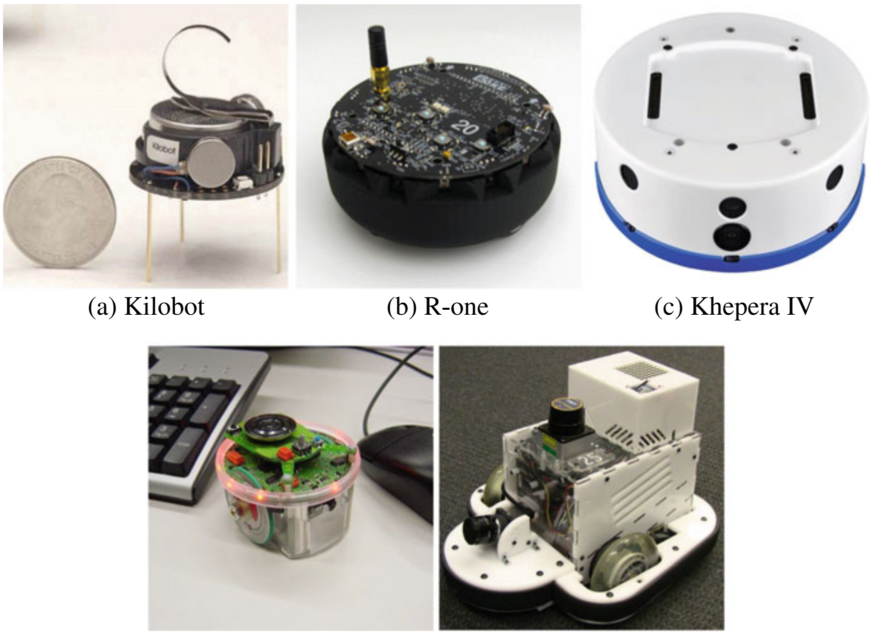
and MicroMouse for maze exploration. A lot of multi-robot systems result from these competitions such as AIBO dog [18], NAO humanoid [33], and Cmdragons [12].

We have observed several features of a multi-robot system:

- Cost: inexpensive for each single robot. A general purpose for MRS is to let quantities of agents, each of which owns limited ability, to achieve a complex system-level target. The system must be designed to be inexpensive to allow researchers to incrementally increase the size of the system. When a multi-robot system is scaled up, it will be hard to cover the fee if each individual robot is highly expensive.
- Size: small size for each single robot. Given limited space, robots with the large size may have problems of frequent collisions, communication blocking, and less flexibility. Also, robots in huge size go against the scalability of the whole system.
- Functionality: stable and strong sensibility for each single robot. If every robot has stable functionality, the whole system can be reliable enough. The stronger the sensibility is, the more the information it may acquire from itself, the environment, and other robots. Hence, the whole system may achieve more complex tasks.

As we know, stronger functionality may result in larger size and higher cost. Therefore, to build an MRS, it is crucial to find a balance between cost, size, and functionality.

Though there have been a lot of multi-robot systems, most of them are controlled in centralized way. In another word, there is a central controller to schedule the robots to perform cooperative tasks. Centralized multi-robot system can be hardly scaled up due to limited computation capability of the central controller. Hence, scholars transfer their research direction to distributed multi-robot system [30]. There are varieties of active research topics that explore efficient algorithms to control distributed multi-robot system, such as self-reconfiguration [7, 76] and exploration [14, 38]. Scholars generally envision their algorithms to be feasible for a distributed multi-robot system consisting of hundreds, thousands, and even more robots


(a) Kilobot (b) R-one (c) Khepera IV

(d) E-puck (e) Scarab

Fig. 3 Five representative robots suitable for multi-robot system nowadays

[7, 25, 90]. However, these algorithms are usually evaluated in simulator only [7, 76], or deployed on a small group of tens of robots or fewer [44, 45] due to cost, time, or complexity. As we previously mentioned, a simulator can hardly model robots' movement, communication, and sensibility in a precise way. Therefore, it would be significant if a large-scale distributed MRS can be built up for algorithm evaluation.

An MRS is said to be fully distributed [72] if each robot in the system supports:

- Distributed control: to process gathered information and to make the decision locally while achieving the system-level goal.
- Distributed sensing: to sense itself, the environment, and other robots locally.
- Distributed actuation: to navigate freely in the environment without collision with obstacles and other robots.
- Distributed communication: to receive and transmit data from other robots in a scalable robot network.

Knowing basic elements for a distributed MRS, we characterize some typical MRSs in detail and compare their functionality and cost. The criteria are to select open-source, still active, and relatively high-impact MRS. The summary of comparison can be seen in Table 1. In detail, five multi-robot systems are considered as follows: Kilobot [77, 79], r-one [61], Khepera IV [88] (evolved from Khepera III [73]), e-puck [68], and Scarab [64].

Table 1 A comparison of off-the-shelf multi-robot systems in terms of functionalities and hardware

	Kilobot	R-one	Khepera IV	E-puck	Scarab
Source	Harvard U	Rice U	K-Team	EPFL	Pennsylvania U
Locomotion	Vibration	Wheel encoders 3-axis gyro 3-axis accelerometer	Wheel encoders 3-axis gyro 3-axis accelerometer	Wheel encoders 3-axis gyro 3-axis accelerometer	Wheel encoders 3-axis gyro 3-axis accelerometer
Sensibility	1 IR range sensor	8 IR range sensors 8 bump sensors 4 light sensors a speaker	8 IR range sensors 8 light sensors 4 IR cliff sensors 5 ultrasonic range sensors 1 microphone 1 speaker 1 color camera	8 IR range sensors 8 light sensors 1 microphone 1 speaker 1 color camera	Laser range sensor high-res color camera
Communication	IR signal	IR signal radio	802.11 b/g Wi-Fi Bluetooth 2.0 EDR	Radio	Radio
Computation	8 MHz Atmega328 32 kB Memory	50 MHz ARM Cortex-M3 64 KB SRAM 256 KB Flash	800 MHz ARM Cortex-A8 512 MB RAM 512 MB flash 4 GB flash for data	Microchip dsPIC MCU 8 KB RAM 144 KB flash	/*
Battery life (h)	3–24	4	7	1–10	/*
Size (cm)	3.3	10	14	7.5	22.2
Cost ($)	14	220	2625	545	3000

*not specified

- The Kilobot[1] (shown in Fig. 3a) is designed by the K-Team and used in SSR lab of Harvard University. Kilobot is a low-cost robotic system especially suitable for research on swarm robotics. The functionality of each individual Kilobot is limited, i.e., only can sense the distance from its neighbor, sense the intensity of visible light, and receive/transfer message from/to its neighbors. However, a collective of Kilobot achieves relatively complicated behaviors such as generating different shapes [80] and transporting large objects [78]. This kind of robotic system in which every robot is with limited ability while can achieve complicated behavior

[1] http://www.eecs.harvard.edu/ssr/projects/progSA/kilobot.html.

together is called swarm robotics. It is inspired by biological swarm behaviors [71] such as bird flocking and ant manipulation. Another such kind of system is the I-Swarm [46] from the University of Stuttgart. However, the robot Jasmine in I-Swarm is far more expensive ($130) compared with Kilobot ($14) while the functionality is similar. Simple functionality makes low cost possible and, on the other hand, limits the feasible environment. For example, a message is transmitted using the reflection of infrared signals. Therefore, the floor where the Kilobots move must be smooth enough, or infrared signals may not reach individual's neighbors.

- The r-one[2] (shown in Fig. 3b) is designed and used in Rice University. The r-one is a relatively low-cost robot that enables large-scale multi-robot research and education. In terms of locomotion, each robot is equipped with two-wheel encoders, a 3-axis gyro, and a 3-axis accelerometer to move on a floor with awareness of odometer, speed, and acceleration. With respect to communication, there are two kinds of communication method. First one is to use infrared transmitter and receiver to achieve directional communication, and the second one is to use radio to achieve nondirectional communication with higher bandwidth. The sensing ability is provided by using 8 bump sensors for 360° detection. r-one provides ample functionalities at a low cost which has motivated its use for education area application [62]. Several courses are taught using r-one. r-one can also be used for multi-robot manipulation [63] and transportation [36] if each robot is equipped with a gripper.
- The Khepera IV[3] (shown in Fig. 3c) is designed and made by K-Team. It is a commercial robot with abundant and powerful functionality compared with non-commercial ones. A standard Khepera IV has the same equipment for locomotion as r-one. For the communication part, Khepera uses 802.11 b/g Wi-Fi and Bluetooth 2.0 EDR for wireless communication instead of infrared signals or radio. Khepera IV has strong sensibility due to the presence of multiple sensors. A Khepera IV is equipped with five ultrasonic transceivers and eight infrared sensors for obstacle detection, four extra infrared sensors for cliff detection, one microphone and one color camera for multimedia functions, and twelve light sensors and three programmable LED for human–robot interaction. Besides, Khepera IV is highly extensible. Developers may extend native functions using the generic USB, Bluetooth devices, and custom boards plugging into the KB-250 bus. Khepera IV wrap the remarkable abilities of sensing, communication, and locomotion in a small body of 14-centimeter diameter. However, the cost of each Khepera IV is over US$ 2600. The Khepera series robot is adopted by DISAL of EPFL and is used for various research topics such as multi-robot learning [26] and odor plume tracing [87].
- The e-puck[4] (shown in Fig. 3d) is designed and made by EPFL. E-puck designer Francesco Mondada started with the Khepera group and moved to make simpler

[2]http://mrsl.rice.edu/projects/r-one.

[3]http://www.k-team.com/khepera-iv.

[4]http://www.e-puck.org/.

education robots. An e-puck is equipped with two-wheel encoders, a VGA camera, three omnidirectional microphones, 3-axis accelerometer, eight infrared sensors, and eight ambient light sensors. Also, e-puck is only 7 cm long and easy to extend functionality. For instance, rotating scanner and turret with three linear cameras are two optional extensions. E-puck is specially designed and widely used for education purpose [21]. It is used in the teaching areas of signal processing, automatic control, behavior-based robotics, distributed intelligent systems, and position estimation and path finding of a mobile robot [68]. In addition, e-puck is also used in many research topics such as supervisory control theory [51] and distributed control strategy [83].

- The Scarab shown in Fig. 3e is designed and made at the University of Pennsylvania. Compared with other robots, the design of Scarab shifts from minimal multi-robots to a complex and robust system. Two of the major components in a Scarab are the Hokuyo URG laser range finder and the Point Grey Firefly IEEE 1394 camera. Using the laser and camera, Scarab is capable of the tasks requiring strong sensibility and high computation payload such as SLAM (simultaneous localization and mapping) [75] and vision processing. However, a Scarab is significantly large, heavy, and expensive with 23 cm diameter, 8 kg weight, and over \$3000 cost. Consequently, Scarab is not practical for large populations, i.e., more than ten Scarabs working together. But using less than five Scarabs for multi-robot SLAM is applicable [81].

2.2 *Multi-robot System Applications*

Robots contain both sensing and actuator components which make them useful for a wide range of applications. Applications which involve navigation, exploration, object transport, and manipulation benefit from the use of MRS. Researchers have been trying to develop biologically inspired robots that incorporate not only the structure of insects and animals but also their social characteristics to design multi-robot system. Researchers try to emulate the communication behavior in bees, birds, and other insects to design control and coordination system for MRS. We have classified the robotic applications into seven categories as shown in Fig. 4. A brief overview of the robotic application is also provided below. These applications are generic and not specifically related to MRS. However, the current research trend is that most applications are now being developed using MRS instead of single-robot system.

- *Healthcare Robots*: Robots have been used by healthcare and medical professionals for a long time. One of the most important uses of robots in health care has been for performing and assisting surgeries. Robots are used for performing precise and minimally invasive surgeries [9, 15]. The current research trend in this area is to use biologically inspired robots that can move in confined spaces and manipulate objects in complex environments [15]. Other areas where robots are being used in

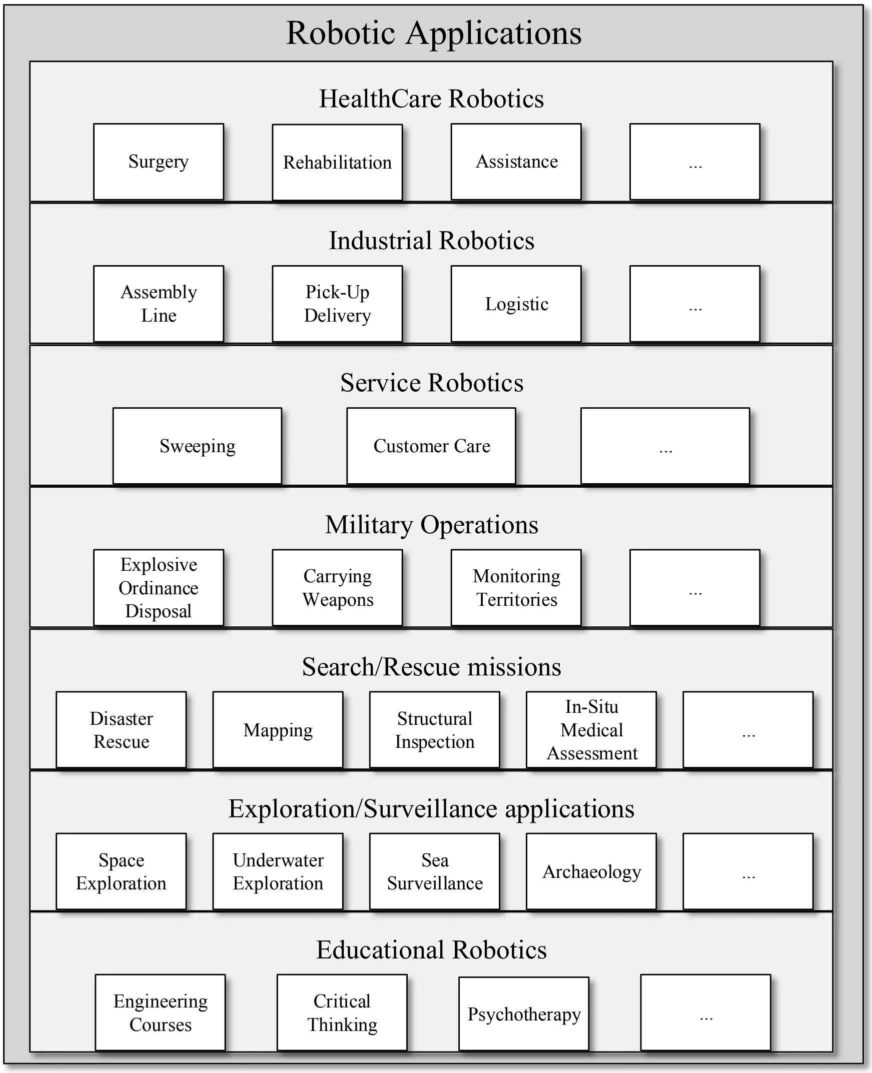


Fig. 4 Classification of robotic application

this application domain is rehabilitation and assistive robotics [34, 91]. Robots are used for recovery of patients with impaired motor and cognitive skills [34]. Robots are being used for assistance to elderly and other physically or mentally disabled individuals to help them live independently. There are even companion robots that help such individuals with special needs. However, due to lack of awareness and other reasons, patients and even healthcare professionals are reluctant to accept robots for medical purpose [11, 91].

- *Industrial robotics*: Robots are now a main component in manufacturing and logistics industries. Industries have been using robots for tasks which are impossible or difficult for humans, such as working in a room filled with hazardous substances, inside a furnace, etc. [34]. Several robotic application studies in manufacturing industries have been mentioned in [29] including die casting applications, forging applications, heat treatment applications, glass manufacturing applications, etc. All large-scale manufacturing industries especially automobile, component assembly, and many other industries involving tasks related to packaging, testing, and logistics rely on the use of robots for efficient task completion [85]. Besides automation, robots are also used for assisting humans in their activities in industries.
- *Service robotics*: Service robots are fully or semi-autonomous robots that perform tasks useful for the well-being of humans except in manufacturing related activities. Service robots are useful for performing tasks that are trivial, dangerous, or repetitive for humans. Home service robots are one such type of robots. They can be used for activities that range from cleaning floor, kitchen, bathroom, windows, swimming pool to lawn mowing, washing clothes, and many other activities [34, 85]. Besides home, service robots can also be used for other services such as object pickup and delivery, customer care, etc. [34].
- *Military operations*: Most of the military organizations around the world are using different types of robots for situations that are risky for humans [82]. Robots are also cheaper to maintain than having the human personnel. Military robots can be classified into three categories, which are ground robots, aerial robots, and maritime robots [82]. These military robots are very often used for battlefield surveillance from ground, air, and underwater level. Ground robots are also being utilized for explosive ordinance disposal. Besides carrying out surveillance operations in enemy territories, unmanned aerial vehicles (UAVs) are also used for carrying missiles to attack enemy sites.
- *Search and Rescue missions*: Rescue robots are used to provide real-time information about the situation to aid search and rescue missions. Rescue robots are used for performing tasks such as searching in unstructured and hazardous environments, reconnaissance and mapping, rubble removal, structural inspection, in-situ medical assessment and intervention, and providing logical support [85]. Rescue robots can be utilized for many situations including natural disasters, mining accidents, fire accidents, explosions, etc. [34]. Rescue robots are also useful for post-disaster experimentation [85]. A key aspect of this application is that rescue robots must be autonomous and they are supposed to work in an unstructured environment where any pre-existing communication network may not work properly.
- *Exploration/Surveillance application*: Robots are useful for collecting data in unstructured environments, unknown territories, and from areas which are difficult or impossible for humans to access. Space exploration, underwater exploration, and exploration in hazardous environments such as radiation prone areas, wilderness, mines, damaged buildings, etc. are some examples of this application [85]. Exploration or surveillance is an important part of other applications too such as military operations, and rescue missions. Navigation, coordination, and

collaboration are three important tasks performed by robots in surveillance applications. A lot of researchers are trying to develop biologically inspired robots that can navigate in confined spaces and perform complex tasks [47].

- *Educational robotics*: Robots are now being used in schools and universities for the educational purpose also. Students can learn about multiple disciplines such as computer science, electronics, mechatronics, etc. by developing robotic applications and learning from the experience [3]. However, there is a drawback with this approach as students only learn about robot-related fields. Several studies have been reviewed in [10], and it is observed that most studies only help in teaching concepts related to physics and mathematics such as Newton's Law of motion, kinematics, fractions, etc. Students who are interested in other fields such as music or arts do not get much benefit out of this. There are few instances where robots have been used for teaching students something different from mathematics or physics. In [95], Lego robots have been used to teach about evolution. Lego robots have also been utilized in [70] to improve social connection in individuals with autism and Asperger's syndrome. This shows that robots have huge potential for contribution toward education. Research efforts are required to find ways to use robots for the development of skills such as critical thinking, problem-solving, teamwork, etc.

3 Design Goals for MRS Middleware

The current trend in robotics is to use MRS for application development instead of a single-robot system. Multiple robots are connected using a wireless network and they work together as a group to accomplish application objectives. These robots are usually composed of heterogeneous hardware and software components that collaborate and coordinate with each other to perform complex tasks such as planning, navigation, distributed computation, object manipulation, etc. [66]. It is not trivial to design software architecture for MRS due to many challenges such as interoperability, dynamic configuration, real-time integration of heterogeneous components, etc. Middleware can resolve these issues by providing programming abstractions and help in reducing the development time and cost [66]. Middleware can also make the application development easier and flexible by providing reusable services. It is, however, challenging to develop a middleware as middleware needs to not only deal with complex issues related to MRS but also satisfy multiple application requirements. In this subsection, we have explained some design goals that should be considered while developing a middleware for MRS. An ideal middleware should be able to support all the features but it should be noted that the complexity of middleware becomes higher as more features are supported. Therefore, it is a trade-off between the number of features supported by a middleware and its complexity.

- *Hardware and software abstractions*: Developing a robotic application requires knowledge of multiple disciplines, which includes knowledge of hardware and software components being used, and corresponding application domain. Usually, robotic application developers have knowledge of their application domain but it is difficult for them to have expertise on low-level hardware and software issues. The primary purpose of using middleware is to make the application development easier and faster. Development of an application using MRS can be done easily if high-level abstractions are provided to a robotic application developer. Having hardware and software abstractions will enable developers to focus on high-level application requirements rather than low-level hardware and network issues. Besides making the application development easier, it will also help in enhancing the efficiency of the application.

- *Interoperability*: MRSs can have multiple sources of heterogeneity. Heterogeneity in MRS may arise due to the difference in either hardware or software of multiple robots. It is not uncommon to use robots from different hardware manufacturers within the same MRS. Even with the same hardware manufacturer, hardware heterogeneity can arise due to difference in the sensor and actuators being used for the robots. Different communication standards can be used within the same MRS which also leads to heterogeneity in the network. Even if the homogeneous hardware is used for MRS, there can be differences in the software architecture of multiple robots. Software modules developed by different programmers using different programming environments can also lead to heterogeneity. Middleware should provide abstractions for developers to enable interoperability between heterogeneous robots. Middleware should enable platform independence such that robots can be developed on different platforms. Middleware should allow robots developed using different platforms or containing heterogeneous hardware and software components to communicate with each other.
- *Real-time support for required services*: Time-critical robotic applications such as rescue operations, medical surgeries, military operations, etc. require real-time support for services. Most of the applications require real-time support that is required for many services that are responsible for collision detection and avoidance, collaboration between multiple robots in MRS, integration of multiple components in robots, etc. There are some tasks which can afford a delay in services but for most of the services used in MRS, real-time support is required.
- *Dynamic resource discovery and configuration*: MRS is a dynamic system where robots are mobile and since robots are usually used in unstructured environments, there is always change in connectivity. MRS is a scalable system where robots can be added, removed, or changed in configuration. There is always change in the configuration of the network. Middleware should enable dynamic discovery of resources which includes both robots and the software services being used. Middleware should enable autonomous detection and recovery from any fault in the network or software. Middleware should provide support for MRS to be self-adapting, self-configuring, and self-optimizing [66].
- *Flexibility and Software reuse*: Software reuse means using the same service even for a different application, hardware, or software environment. Middleware should enable flexibility in using software services such that services are defined by their functionalities and not based on the hardware, software, or the applications for

which they are used. This implies that a developer should not redevelop the service every time there is some change in hardware, operating system, or even application. Middleware should enable the developer to add new functionalities to a system without having to redevelop everything from scratch.

- *Collaboration among multiple robots*: In MRS, multiple robots collaborate with each other by sharing data. Distributed computation is necessary to enable collaboration between robots; however, due to heterogeneity in MRS, it becomes challenging to understand data belonging to the different types of robots. Another requirement for collaboration between robots is that it should be real-time which makes it even more complex for developers to support this functionality. Middleware should provide services that make it easier to do collaboration between robots. Robots should not only be able to transfer data between each other but also understand the meaning of shared data. Middleware should provide abstractions that can help achieve this objective.
- *Integration with other systems*: Nowadays, robotic applications are developed by integrating robots with other systems such as Internet of things (IoT) and cloud. Cloud robotics is a new paradigm where robots utilize computation and storage benefits of the cloud to perform tasks [39]. In near future, IoT and robotics will be combined to provide better services to humans. Issues and technological implication in implementing IoT-aided robotic applications have been studied in [34]. In coming future, more technologies will be integrated with robotics to provide improved services. Middleware should enable integration of MRS with other systems and technologies. Middleware should provide abstractions for a developer to integrate different technologies.
- *Management and monitoring tools*: A lot of components are involved in development, deployment, and functioning of MRS including multiple robots consisting sensors and actuators, software services, and many other resources. Due to the complexity of MRS, it is difficult for a developer to control everything unless there are some tools available that can help in management and monitoring of the overall system. Besides providing services to program MRS, middleware can also provide management tools to configure, debug, and view the overall MRS [20]. Middleware should also enable the developer to view whole system component-wise to provide a better understanding. This functionality will make it easier even for non-programmers to understand and contribute to the development of the robotic application.
- *Support for the addition of extra services*: MRS is usually deployed in an unstructured environment and every application requires some specific services. Middleware should be flexible to enable the addition of services at runtime. Middleware should support the addition of new services to address both network-specific and application-specific qualities of service (QoS) requirements. It should support the addition of services to address issues such as security, reliability, availability, energy optimization, collision detection and avoidance, etc.

4 A Taxonomy of MRS Middleware

There are tens of existing middleware for multi-robot systems focusing on various aspects and purposes. Among the off-the-shelf middleware, it is difficult for a beginner to choose an appropriate one suitable for a specific multi-robot system or multi-robot application. To address this issue, we propose a taxonomy of MRS middleware features to formally describe MRS middleware. In detail, we utilize the structure of the phylogenetic tree to provide a comprehensive, yet succinct framework that allows for a systematic comparison of MRS middleware. Developers could look up desired features in the phylogenetic tree for the purpose of finding a suitable MRS middleware. In biology, a phylogenetic tree or evolutionary tree is a branching diagram showing the inferred evolutionary relationships among various biological species [24]. In the field of computer science, phylogenetic tree has been used to visualize a taxonomy in many survey papers such as WSN programming abstractions [69], WSN middleware [94], and programming distributed Intelligent MEMS [49], but it has not been used for describing MRS middleware yet.

In Fig. 5, we decompose the MRS middleware features into ten leaf features. Between Fig. 6 and Fig. 13, we describe each leaf feature appeared in Fig. 5 in detail. In these figures, we utilize some notations to describe relationship among features. The relationship between a father feature and several child features can be either inclusive or alternative, notated by solid dot and hollow dot, respectively. Also, a child feature can be either necessary or optional, notated by solid square and hollow square, respectively.

As shown in Fig. 5, when we are investigating MRS middleware features, it can be divided into software features from middleware and hardware features from MRS. On the one hand, features from middleware can be divided into two parts. One is the services by the middleware and the other is the system architecture of the middleware. In terms of provided services, it includes functional services as well as nonfunctional services. With respect to features from the system architecture, three parts are included which are programming abstraction features, infrastructure features, and coordination method features, respectively. On the other hand, features from MRS come from both the infrastructure and concrete applications. The features from infrastructure can be node-level one and system-level one. The features from concrete applications are divided into subcategories based on environment, scope/area, and purpose/goal. In this way, MRS middleware features are divided level by level and result in ten leaf features. We explain and describe each leaf feature in detail in the following paragraphs.

There are a variety of functional features (shown in Fig. 6) for MRS middleware. Functional features of MRS middleware are the basic functions implemented by the middleware. Such functions include localization, mapping, collision avoidance, path planning, vision processing, and many others. With the off-the-shelf implementation, developers may use these basics but important functions conveniently. Nonfunctional features (shown in Fig. 7) are features provided by the middleware in terms of QoS, for example, security, fault tolerance, reliability, real-time support, etc.

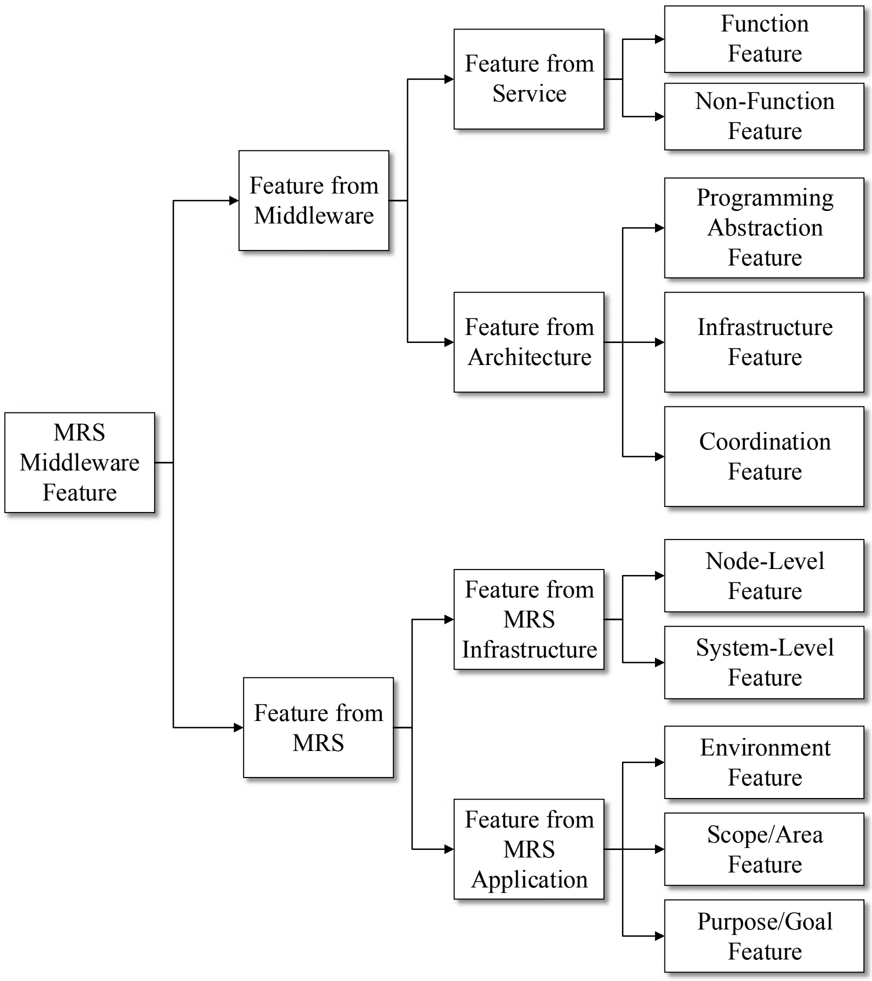


Fig. 5 Overview of feature tree of MRS middleware

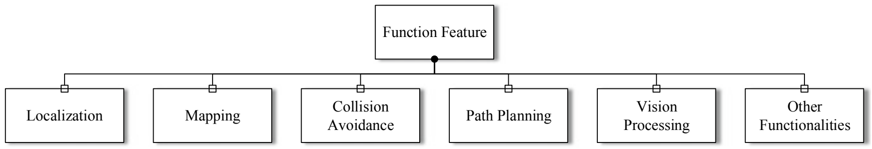


Fig. 6 Function feature

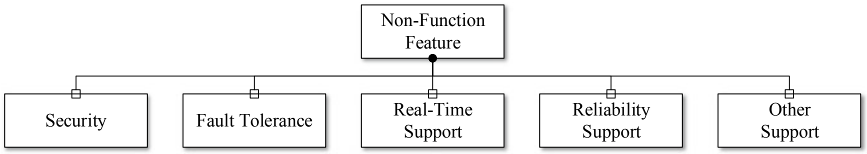


Fig. 7 Nonfunction feature

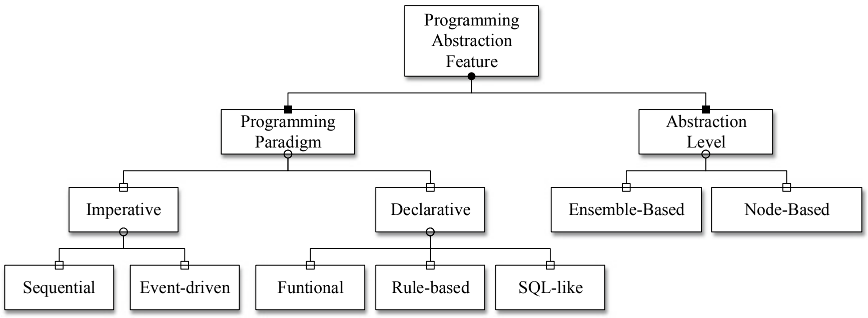


Fig. 8 Programming abstraction feature

Holonomic MRS middleware provides as many nonfunctional features as possible for developers so that they can choose a set of the features depending on specific applications. You cannot have your cake and eat it too, it is very suitable for some of the nonfunction features. For example, if the developer desires the privacy features, it may consume more time and consequently affects the real-time feature. Similar issue may be observed when fault tolerance and real-time support are provided by the middleware. There is always some form of trade-off between different nonfunction features. Such conflicts of nonfunction features are ubiquitous for middleware in other fields too such as wireless sensor network [19] and cloud computing [16].

Modern MRS middleware always provides a programming model or programming abstraction to facilitate development. A programming model masks the complexity of the system. Programming paradigm and abstraction level serve as the two fundamental elements of a programming model (shown in Fig. 8). The programming paradigm refers to the abstractions used to represent individual elements of a program. The individual elements of a program include constants, variables, clauses (iterations, conditions, etc.), and functions. Programming paradigm of a programming model can be imperative or declarative. While programming with imperative approach, the state of the program is explicitly expressed through statements. Relevant subcategories of imperative approaches include sequential and event-driven. On the other hand, while using a declarative programming model, the application goal is described without specifying how it is accomplished. Declarative approaches can be further classified into functional, rule-based, SQL-like, and special-purpose. The abstraction level refers to how developers view the multi-robot system and can be either node-based or ensemble-based. Node-based abstraction is used in traditional

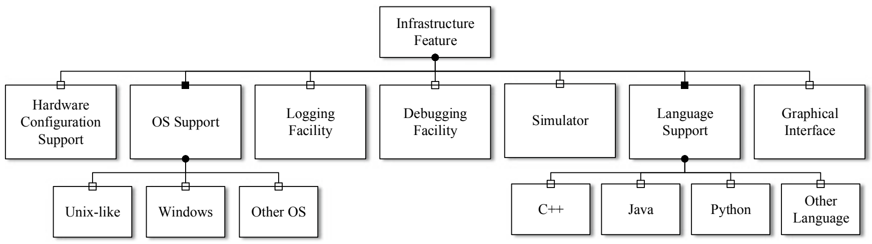


Fig. 9 Infrastructure feature

programming model where each robot is programmed, respectively. When a group of robots is assigned a task, it is natural to think about what the robot ensemble should do. This leads us to consider the ensemble-level abstraction. The entire MRS can be viewed as a single and monolithic unit for the programmer. Ensemble-level abstraction is referred to as macro-programming in wireless sensor networks [35].

There is a wide range of infrastructures (shown in Fig. 9) for MRS middleware. Infrastructures of MRS middleware include hardware configuration support, operating system support, logging facility, debugging facility, simulator, language support, and graphical interface.

- Since MRS middleware may be applied to all kinds of robotic system, hardware configuration support is required to configure the hardware of a specific kind of robot.
- MRS middleware must support a specific operating system or be cross-platform. Traditional operating systems can be UNIX-like OS, Microsoft Windows, Java virtual machine, and others.
- Logging facility and debugging facility are essential and useful for application development, algorithm evaluation. Looking up the log and debugging information, developers can have direction for development and improvement, which save significant amount of time.
- Simulator is useful when deployment is costly, hardware is unavailable, or developers want to testify algorithms before deployment.
- Language support is another necessary feature for MRS middleware. It can support one or several languages, for example, C++, Java, Python, etc.
- Graphical interface can be used to visualize the MRS and for human–robot interaction purpose.

Coordination (shown in Fig. 10) is a general issue in multi-agent system as well as in MRS where each realistic robot is regarded as an agent. A robot is a computational device capable of sensing, computing, and locomoting. The sequence of sensing, computing, and locomoting form a computation cycle of a robot. The coordination method is classified based on the relationship among computation cycles of the robots. In the asynchronized setting, the robots in the MRS do not have a common notion of the time. That is to say, there is no assumption on the relationship among the cycles of the same robot or different robots. The only assumption is that all

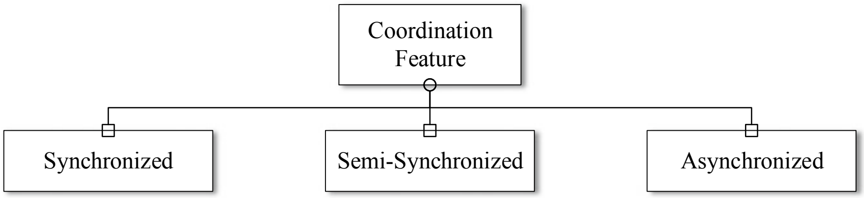


Fig. 10 Coordination feature

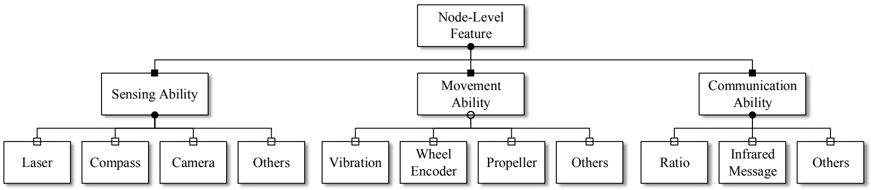


Fig. 11 Node-level feature

cycles finish in finite time. The robots are said to run in a semi-synchronized setting if all robots share a global clock and their actions are atomic. The robots can be either active or inactive at each clock tick and only robots in active state perform their cycles. To make sure every clock tick and every robot to be meaningful, it is restricted that at least one robot is active at every clock tick and every robot becomes active for infinite time instants. In a special case, every robot is active at every time instant. In this case, the robots are said to be fully synchronized. In this setting, all the robots are in the same state at each clock tick.

With respect to node-level features (shown in Fig. 11) of a single robot, it refers to the hardware features relating to sensing ability, locomotion ability, computation ability, and communication ability. For the sensing part, an individual robot may contain laser, compass, camera, microphone, etc. For the locomotion part, each robot may use vibration, wheel encoders, or propellers to navigate the environment. For the computation part, the CPU frequency, the memory size, and the data storage size vary a lot. For the communication part, ratio, infrared signals, Wi-Fi, and Bluetooth can be utilized to achieve it.

In terms of system-level features (shown in Fig. 12) of the whole MRS, it can be categorized by coordination method, embedded network protocol, and communication model. For the coordination method in an MRS, it can be centralized where there is a central controller, decentralized where the MRS is divided into groups, or fully distributed where all robots are equal. For communication, robot network may utilize TCP, UDP, ZigBee, or other network protocol. Communication is a general issue in network systems as well as in MRSs which can be regarded as robot networks. Communication features can be classified in the light of awareness, scope, and addressing. Awareness feature within communication can be further classified into explicit or implicit. If the communication is explicitly exposed to developers,

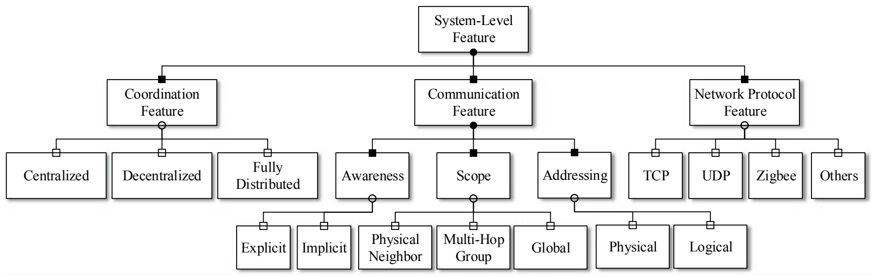


Fig. 12 System-level feature

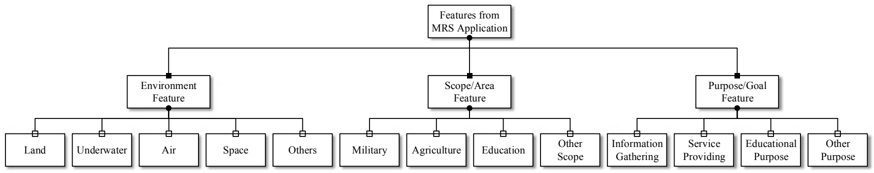


Fig. 13 Features from MRS application

it is termed as explicit. On the other hand, if the communication is hidden behind some higher level construct, it is said to be implicit. The scope of communication refers to the set of robots that exchange data to accomplish a given task. Physical neighborhood, multi-hop group, and system-wide serve as the three approaches for the scope of communication. The scope is physical neighborhood if programmers are only provided with method for exchanging data among robots within direct radio range. The scope is said to be multi-hop group if data exchange can be achieved with using multiple-hop transmission. The scope is system-wide if all the robots in the MRS are possible to be involved in data exchange. With respect to addressing in MRS middleware, it utilizes physical addressing if the target robots are identified using statically assigned identifiers. Otherwise, the target robots are identified through properties provided by the applications. This kind of addressing approach is logical addressing, which is generally called attribute-based addressing in wireless sensor networks [2].

In terms of application features, we sort it using environment, scope/area, and purpose/goal, respectively (shown in Fig. 13). An MRS can be deployed in one or more kinds of environments including land, underwater, air, and even space environment. The area of MRS application can be military, agriculture, education, household, manufacturing, etc. The purpose of an MRS application can be information gathering, service providing, educational usage, and others. Full discussion of the MRS applications can be found in Sect. 2.2.

5 Representative Middleware for MRS

In this section, we give a comprehensive overview of some popular middleware for MRS. This list is not exhaustive and there are many more middleware such as OROCOS [13], SmartSoft [84], CLARAty [93], etc. that have not been discussed in this work. We have focused on more recent and popular middleware for robotics. For each middleware that has been included below, we have discussed the architecture, objective, development tools and utilities provided, and platforms and programming languages supported by the middleware. An overview of various MRS middlewares, discussed in this section, has been shown in Table 2. This overview is done on the basis of middleware-specific features illustrated in Fig. 5, which shows the overview of feature tree of MRS middleware.

Player/Stage: Player[5] is device server that provides clients with programming interfaces to control robots comprising of sensors and actuators [31]. Player is implemented in C++ as a multi-threaded TCP socket server for transparent robot control. Socket-based robot server provides many benefits such as platform independence, language independence, and location neutrality which means a client can access and control robotic devices anywhere on the network. Player has been designed to support heterogeneous devices and clients simultaneously at different timescales [31]. One-to-many client/server architecture has been followed which implies that one server can serve multiple clients. Each client is connected to Player by a TCP socket connection, while a device can be connected to Player by any appropriate method. Client can be implemented in any language providing socket mechanism such as C, C++, Tcl, Java, Python, etc.

Player is modular; therefore, devices can be added dynamically. UNIX model of treating devices as files has been chosen to provide an abstraction for a variety of devices. To receive sensor readings, client opens the device with read access while for controlling an actuator, client must open the device with write access. Each device has an associated command and data buffer that provides an asynchronous communication channel between device threads and client reader and writer threads. Clients and devices are decoupled from each other. Player also supports request–reply mechanism, similar to `ioctl()`, for configuration requests that can be used to access specific hardware features. There is no device locking mechanism implemented in Player; therefore, clients can overwrite commands of the other clients.

Stage is a simulator that is used for simulating population of mobile robots, sensors, and environmental objects. This enables development and testing of clients without accessing real hardware and environment. Stage simulator is also useful for experimentation of novel devices that have not been developed yet [31]. Sensors and actuator models in Stage are available through normal Player interface. Usually, clients cannot differentiate between real and simulated stage equivalents. Stage also supports non-locking, platform independence, and language independence characteristic of interfaces in Player.

[5]http://playerstage.sourceforge.net/.

Table 2 Overview of existing MRS middleware

Middleware	Function features	Nonfunction features	Programming abstraction	Infrastructure features	Coordination feature
Player	Localization, mapping, path planning, collision avoidance, vision processing	Security, robustness	Imperative, node-based	Unix-like/Windows, logging/debugging facilities, Simulator—Stage/Gazebo, C, C++, Java, Python, etc.	Asynchronized
Orca	Localization, mapping, etc. reusable from other projects	None	Imperative, node-based	Unix-like/Windows/Mac OS X, logging/debugging facilities, graphical interface: GOrca, C++, Java, Python, PHP, C#, and Visual Basic	Asynchronized
Miro	Mapping, localization, path planning, collision avoidance, speech recognition, vision processing	Robustness	Imperative, node-based	Unix-like/Windows, logging/debugging facilities, graphical interface: Qt GUI, C++	Asynchronized (Event-driven control)
MIRA	Localization, mapping, path planning, collision avoidance, vision processing	Security, robustness, reliability, fault tolerance, real-time support	Imperative, node-based	Linux/Windows, logging/debugging facilities, graphical interface: Qt GUI, C++, Python, JavaScript, etc.	Asynchronized
OpenRDK	Mapping, localization, path planning, navigation, collision avoidance, vision processing	Robustness	Imperative, node-based	Unix-like, logging/debugging facilities, simulator: USAR-Sim/Stage/Gazebo, C++	Asynchronized

Table 2 (continued)

Middleware	Function features	Nonfunction features	Programming abstraction	Infrastructure features	Coordination feature
MARIE	Mapping, localization, path planning, collision avoidance, vision processing	Robustness	Imperative, node-based	Unix-like, logging/debugging facilities, simulator: Stage/Gazebo, graphical interface: logviewer, C++	Asynchronized
Urbi	Vision processing	None	Imperative, node-based	Unix-like/Windows/Mac OS X, logging/debugging facilities, simulator: Webots, graphical interface: UrbiLab, C++, Java, MATLAB, Python, etc.	Both sychronized and asychronized
MRDS	Speech recognition, vision processing	Security, fault tolerance, robustness	Declarative, ensemble-based	Windows, logging/debugging facilities, simulator: Visual Simulation Environment, graphical interface: Visual Programming Language, C#, Visual Basic, and Iron Python	Asynchronized

Table 2 (continued)

Middleware	Function features	Nonfunction features	Programming abstraction	Infrastructure features	Coordination feature
RoboComp	Mapping, localization, etc. reusable from other projects such as Player, Orca, and ROS	None	Imperative, node-based	Unix-like/Windows/Mac OS X, logging/debugging facilities, simulator: Stage/Gazebo, graphical interfaces: managerComp, monitorComp, etc., C++, Java, Python, Ruby, C#, PHP, and Objective C	Asychronized
ROS	Mapping, localization, etc. reused from other projects such as Player, OpenRAVE, etc.	Real-time support, robustness	Imperative, node-based	Unix-like/Windows (partial)/Mac OS X, logging/debugging facilities, simulator: Stage/Gazebo, graphical interfaces: rxplot, rxgraph, C++, Python, Octave, and LISP	Asynchronized
WURDE	Localization, collision avoidance, vision processing	Robustness	Declarative, node-based	Unix-like, logging/debugging facilities, simulator: Stage, graphical interface: RIDE, C++	Asynchronized

Table 2 (continued)

Middleware	Function features	Nonfunction features	Programming abstraction	Infrastructure features	Coordination feature
OPRoS	Mapping, localization, path planning, collision avoidance, vision processing	Fault tolerance, robustness, real-time support (under development)	Declarative, node-based	Linux/Windows, logging/debugging facilities, simulator: OPRoS Simulator, Robot Builder, graphical interface: Component Composer, C++, Java	Asynchronized (event-driven)
RT-Middleware	Mapping, localization, path planning, collision avoidance, vision processing	Real-time support, robustness	Imperative, node-based	Unix-like/Windows, logging/debugging facilities, simulator: Stage, graphical interface: RTSystemEditor, C++, Java, and Python	Both synchronized, and asynchronized
ASEBA	Mapping, localization, path planning, collision avoidance, vision processing	Real-time support, robustness	Imperative, node-based	Unix-like/Windows, logging/debugging facilities, simulator: Enki Simulator, graphical interface: ASEBA IDE based on Qt4, ASEL (ASEBA Event Scripting Language)	Asynchronized (Event-based)

Since Player is freely available as open-source, many improvements have been done since the original version [32]. Some major improvements related to simplicity and flexibility have been done in [22]. Player has been divided into two parts, the core and transport layer. Separation of Player core from transport layer provides more flexibility. Original version [32] was a TCP-based device server, but now it can support many other configurations. Other transport layers, or no transport layer, can also be used. Player is supported on most of the UNIX flavors and on Windows using Cygwin.

Orca: Orca[6] is a framework that can be used for the development of component-based robotic systems. Complex robotic systems can be developed by piecing together the components provided by Orca. The main objective of Orca is to promote software reuse. Orca does not impose any constraint on the component granularity (size of modules used to make up the complete system), system architecture (any architecture such as centralized, blackboard, strictly-layered, strictly-decentralized, or mixed can be implemented), interfaces, and component architecture [56].

Orca uses Internet Communication Engine (Ice) for communication between interfaces [56]. Slice, a specification language for Ice, is used for defining interfaces. There are many Ice services such as IceGrid Registry, IceGrid Node, IceBox, and IceStorm which are extensively used in Orca. IceGrid Registry is a centralized registry for naming service. IceGrid Node is a software activation service. IceBox is an application server that is responsible for starting and stopping of application components. Application components are deployed as a dynamic library which makes them easy to deploy and administer, and also optimizes the communication between components within the same application server. IceStorm is an event service which forward the messages received from a server to multiple clients without marshaling or demarshalling them. IceStorm can also weaken client dependencies by configuring multiple threads.

Orca also provides a library called libOrcaIce which provided simplified API that can be used for development of robotic applications [56]. This lowers the barrier for developers as the majority of functionalities used for robotic applications are provided by Orca library. To allow the use of Orca on wider platforms, CMake is used to build system. Orca can be used on different operating systems including Linux, several flavors of Windows, and Mac OS X. Programming languages that are supported are C++, Java, Python, PHP, C#, and Visual Basic. Also, Ice Client and server are language independent so they can be implemented in any programming language.

Miro: Miro[7] is a three-layered middleware for mobile robot applications which is designed and implemented using object-oriented approach [92]. The three layers from bottom to top are MIRO device layer, MIRO service layer, and MIRO class framework layer. The higher layers access lower layers using interfaces. MIRO device layer is a platform dependent layer that provides classes to interface and abstracts the low-level sensors and actuators within a robot. The classes also allow access to

[6] http://orca-robotics.sourceforge.net/index.html.

[7] https://sourceforge.net/projects/miro-middleware.berlios/.

low-level hardware resources using ordinary method calls. MIRO service layer provides service abstraction for sensors and actuators with event-based communication by using CORBA interface definition layer (IDL). The services are implemented as network transparent CORBA objects which enable language and platform independence. Sensors and actuators are presented in a platform-independent manner by the use of classes in this layer. MIRO class framework layer provides functional modules such as mapping, localization, behavior generation, logging and visualization facilities, etc. which are extensively used for mobile robotic application development. Besides providing common functionalities for application development, MIRO class framework also provides functionality for experimental evaluation.

All MIRO functionalities have been implemented in C++. The communication mechanism is developed using TAO package which is an implementation of CORBA-based on adaptive communication engine (ACE). Client–server model has been used for communication between objects. MIRO implementation includes three types of clients (sample client, test client, and monitoring client) for testing and evaluation of service functionalities. Apart from event-driven communication between services, synchronous and asynchronous communications are also used.

MIRA: MIRA[8] is a decentralized middleware that supports the development of fully distributed robotic applications. The objective of MIRA is to support the development of real-world applications; therefore, mechanisms have been used to address issues such as memory consumption, latency, fault tolerance, and robustness. Each application is composed of different processes that can be located on different machines. Each process further consists of multiple software modules called units which implement algorithms to solve any task. In case multiple units are present in a single process, then each of the units runs in its own thread.

MIRA is written in C++ but it can be interoperable with other programming languages such as Java, Python, etc. Reflection and serialization are two concepts that have been widely used in implementing different mechanisms in MIRA. MIRA uses a reflect method to reflect and serialize any arbitrary class since reflection and serialization are not supported natively by C++. This mechanism enables the complete use of object-oriented programming paradigm. This allows transport of not only simple data but also complex objects including robot models, GUI components, etc. to the remote side. Use of serialization makes MIRA interoperable with other programming languages and middleware. Serialization formats such as XML, JSON, and binary are currently supported by MIRA.

Two communication mechanisms are supported by MIRA, message passing, and remote procedure calls (RPC). There is no central server used in MIRA for name look or other management tasks. MIRA supports robustness and reliability by using peer-to-peer architecture for communication between different processes. Communication between units and message exchange is done using named channels. Channels allow one-to-one, one-to-many, and many-to-many communication [27]. MIRA supports autonomous handling of multi-threading and data synchronization. Slot-based communication avoids unnecessary copying and blocking of data when there

[8]http://www.mira-project.org/joomla-mira/.

is simultaneous read and write access. Slot-based communication also helps in reducing memory usage. MIRA reduces latency of RPC by using futures, which act as proxy for the result of asynchronous calls. MIRA is currently supported for Linux and Windows operating system.

OpenRDK: OpenRDK[9] is a modular software framework designed to develop distributed and mobile robotic applications. The objective of OpenRDK is to support modularity and code reusability of software to enable easier and faster development of robotic application. The main entity of the software framework is a software process called agent. Single thread inside an agent process is called module, which can be loaded and started dynamically once agent is running [17]. An agent configuration is a list of modules that are to be loaded and executed, their interconnection layout, and value of their parameters. All modules publish the data they want to share in a repository. Variables published by modules, i.e., input, output, and parameter, are called properties. Each property is assigned to a globally unique URL address. These URL addresses enable modules to transparently access modules within the same agent or remotely. There are some special queue objects that are also addressed using global URL just like other local properties.

OpenRDK uses multiple processes with multiple threads. Since information sharing between modules is done with the help of URL, it introduces some coupling between modules which adversely affects the modularity of the whole system. To resolve this issue, property links are specified in configuration file that allows modules to refer to different names for the properties, thus avoiding any coupling between modules. While sharing of information within the same agent can be done using repository, inter-agent information sharing can be done by either property sharing or message sending. OpenRDK also contains RConsole which is a graphical tool used for remote inspection and management of modules. Other modules for connecting to simulators or for logging are also provided by OpenRDK. OpenRDK is written in C++. It can run on a UNIX-like operating system. OpenRDK does not focus on platform independence of the software framework.

MARIE: Mobile and Autonomous Robotics Integration Environment (MARIE)[10] is a distributed component-based middleware framework designed to develop robotic applications by enabling integration of new and existing systems [23]. The objective of MARIE is to enable software reuse, support multiple sets of concepts, and support a wide range of communication protocols, mechanism, and robotic standards. MARIE supports multiple levels of abstraction by utilizing layered software architecture consisting of three layers, which are core layer, component layer, and application layer. Core layer is the lower level layer that provides tools for low-level functionalities such as communication, distributed computation, data handling, and many low-level operating system functions. Component layer implements a framework to add components and support domain-specific concepts. Application layer consists of tools required to build robotic applications from available components.

[9]http://openrdk.sourceforge.net/index.php?n=Main.HomePage.

[10]http://marie.sourceforge.net/wiki/index.php/Main_Page.

MARIE uses mediator interoperability layer (MIL) to act as a mediator for interaction with each component independently. MIL is implemented as virtual space where components can interact with each other using a common language. This design leads to the decoupling between components, increases reusability, interoperability, and reduces the complexity of managing a large number of centralized components. MIL is composed of four types of components, which are application adapter (AA), communication adapter (CA), application manager (AM), and communication manager (CM). AA is responsible for interfacing applications with MIL. CA is responsible for communication between components by adapting different communication mechanisms. AM is responsible for management of all components in the system, and CM is responsible for management of communication between AAs.

MARIE has been written in C++. MARIE does not focus on any specific communication mechanism; rather it uses communication abstraction framework, called port, for provided communication protocols and component interconnection. It uses adaptive communication environment (ACE) library to implement for transport layer and low-level operating system function implementation.

Urbi: Urbi[11] is a software platform for developing robotic applications. Urbi is based on an event-driven scripting language, URBISCRIPT, and distributed component architecture [8]. URBISCRIPT is designed not only to create robotic applications and controlling robots but also it is the foundation of the communication protocol based on which client/server architecture for Urbi software platform is built. Multiple clients can interact concurrently with a server by means of URBISCRIPT. Urbi server which lies above the operating system is responsible for abstracting low-level hardware details. Urbi platform interacts with underlying operating system using engines which are also responsible for running the Urbi server. Urbi kernel is another part of the Urbi server that provides primitive services including urbi virtual machine (UVM). URBISCRIPT running on top of urbi virtual machine (UVM) is responsible for providing CPU independence.

Diversity in robots is addressed by the use of UObject architecture. UObject enables communication between low-level and high-level components, and their interaction with URBISCRIPT. Complex data flowing between multiple components can also be handled by the use of UObject API. UObject can be either plugged into the server or also used as standalone remote process. Besides low-level abstraction provided by UObject, high-level abstraction is also provided by using Urbi naming standard. These abstractions enable development of portable applications.

There are many graphical applications such as UrbiLive and UrbiLab provided by Urbi platform to enable easy interaction with robots. UrbiLive is a graphical editor useful for composing and chaining actions based on external events. UrbiLab is a graphical tool used as Urbi server inspector and effector. UrbiLab can also be used for remote control of robots. Urbi is open to programming environments such as Java, MATLAB, and Python. Although Urbi platform is based on C++ and URBISCRIPT, it is not necessary to know these languages to program robots and components.

[11] http://www.gostai.com/products/urbi/.

Microsoft Robotics Developer Studio (MRDS): Microsoft Robotics Developer Studio (MRDS)[12] is a service-driven robotic studio that follows representational state transfer (REST) pattern [40]. Decoupled software services are used for interaction with robots. Use of decoupled services enables modularity and code reuse. Services are used for both robot interaction and implementation of functionalities such as web-based error reporting, wireless communication, etc. The interaction between services is done by the use of XML-based configuration manifest file. Manifest file enables start-up of services by MRDS by defining partnership between services. The partnership of services also enables registration between services, message passing, and fault notification. Partnership and distributed messaging are enabled by the use of software library called Decentralized Software Services Protocol (DSSP). MRDS also uses another software called Coordination and Concurrency Runtime (CCR) for handling state updates and message processing. CCR also enables abstraction of complex functionalities such as memory locking and communication between various operating systems. There are two other main components in MRDS, which are visual programming language (VPL) for graphical interface and visual simulation environment (VSE) for running simulation.

There are some utility services provided by MRDS. A control panel enables the user to view all currently running services and links between them. There is a message logging service that runs built-in filtering to provide a debugging view of the system. Resource diagnostic service is also provided to enable the developer to obtain additional debugging and performance evaluation information. A 3D simulator, based on Microsoft DirectX technology, is also included in MRDS. The simulator is used for both graphics and physics simulations. MRDS is implemented in .NET. Services in MRDS can be written in any .NET compatible language. Service implementations have been done in C#, Visual Basic, and Iron Python. Simple object access protocol (SOAP) interface can also be used to interface services with other programming interfaces. MRDS is a popular proprietary middleware that only supports Windows platform.

RoboComp: RoboComp[13] is a component-oriented robotic framework that focuses on ease of use and rapid development of robotic applications [57]. Robocomp is based on Ice which is extended further by the use of different classes and tools. Components used in Robocomp consist of three main elements, which are server interface, worker class, and proxies that are used for communicating with other components. Worker class implements the core functionality of components. Server interface and worker class run in different threads to avoid delays. There is another optional common interface called CommonBehavior that is used for accessing the parameters and status of components.

Different tools provided by RoboComp are used for providing functionalities such as monitoring, management, debugging, simulation, etc. These tools are as follows:

[12]https://www.microsoft.com/en-us/download/details.aspx?id=29081.

[13]http://robocomp.github.io/website/.

(a) componentGenerator: This tool makes the task of the programmer easier by automatically generating the skeleton of the new component and even the code pieces for the programmer.
(b) managerComp: managerComp is a graphical tool that can be used for building and running the system. Both local and remote components are managed can be managed by use managerComp. This tool also makes use of CommonBehavior interface to visually access the parameters of the components.
(c) monitorComp: This tool is used for connection and monitoring of components. monitorComp provides a graphical interface for testing the components in an easier way. Testing is done either by the use of custom monitoring code or template available to test HAL components.
(d) replayComp: This tool records the output of components to replay them. This is also a graphical tool that is useful for debugging purposes.
(e) Simulation Support: RoboComp makes use of two widely used open-source simulators, Stage and Gazebo, for simulation purpose.
(f) loggerComp: This tool is used for analyzing the execution and interaction of components. This tool also provides a graphical interface to display different types of information.

RoboComp can be deployed on any computer system supporting Ice. Platforms supported by RoboComp are Linux, Windows, Mac OS X, Android, and iPhone. Any programming language that supports Ice can be used for RoboComp, which includes C++, Java, Python, C#, Ruby, PHP, and Objective-C.

ROS: ROS[14] is a modular framework for developing robotic systems [74]. ROS provides a structured communication layer above operating system of the host. Multiple processes running in a system are connected using peer-to-peer topology instead of using a central server. ROS is language-neutral. A simple and language-neutral interface definition language (IDL) is used to provide support for cross-language development. All the functionalities in ROS are developed using small modules. Use of modular architecture reduces complexity and enhances stability. All the driver and algorithm development is done in standalone libraries which are independent of ROS. Small executables are created inside the source code which exposes library functionality to ROS. This mechanism makes code reuse easier and helps in unit testing.

There are some fundamental concepts that have been defined for ROS implementation, which are nodes, messages, topics, and services. A system is composed of multiple nodes which are processes that perform computation [74]. Communication between nodes is using messages. A message can consist of other messages, or array or messages. Topic-based publish-subscribe communication paradigm has been used. To enable request/response communication, the concept of services has been defined. There can be multiple simultaneous publishers and/or subscribers for a single topic, and a single node can publish and/or subscribe to multiple topics.

ROS follows tool-based design, and thus there are many tools provided with ROS for different scenarios. ROS uses rconsole library to enable logging and monitoring

[14]http://www.ros.org/.

of distributed system. Packaging functionality is enabled by the use of roslaunch tool. Collaborative development is enabled by the use of utilities such as rospack and rosbash. ROS uses a utility named rostopic for filtering messages. There are tools such as rxplot and rsgraph that are used for generating plots and graphs. ROS has been designed to be language-neutral. ROS supports four different types of programming languages, which are C++, Python, Octave, and LISP.

WURDE: WURDE (Washington University Robotics Development Environment) is a modular middleware for developing robotic systems [37]. The objective of WURDE is to develop middleware that is easy to use even for beginners. WURDE provides set of abstraction and utilities to achieve this objective. Four layers of abstraction are provided by WURDE, which are communication, interface, application, and system. WURDE does not use specific communication protocol instead the communication layer defines types and methods for moving data to communication adapter [28]. Interface layer describes the data required by each type of robot. Interfaces are described using XML. Application layer provides API for controlling different aspects of applications. System layer is topmost abstraction layer which is used for connecting different applications.

WURDE does not use any specific software architecture; instead, the robotic system is developed as system of small interconnected modules. WURDE uses asynchronous communication for communication between modules. One of the modules provided by WURDE is a tasking and control interface called Robot Interaction Display Environment (RIDE). RIDE enables single user to control and task multiple robots at same time. Implementation of WURDE is not yet complete as there are many software packages that are still needed to be developed.

OPRoS: Open platform for robotic services (OPRoS)[15] is a distributed component-based platform for developing robotic systems [41]. The objective of OPRoS is to enable full development of robot software by providing required developmental tools, middleware services, component execution engine, simulation environment, and other utilities. Robotic service in OPRoS is composed of loosely coupled components. There are two types of components, atomic component and composite component. Components can have different granularities. Communication between components is done by the use of ports on each specific component. Each component can have multiple ports that are used for transmission of different types of information such as method invocation, data, and events. Components in OPRoS can support either of the three different types of execution modes, which are periodic, nonperiodic, and passive.

All the information related to components such as port types, execution semantics, properties, and other relevant information is stored in an XML file called component profile [41]. There are other XML profiles such as service profile, data profile, and application profile. Component and application profile is used by communication execution engine for execution and management of components. Component execution engine provides abstractions to developers by hiding low-level details such as thread management, resource allocation, and other functions offered by the oper-

[15]http://www.opros.or.kr/display/opros/OPRoS+Wiki.

ating system. Component execution engine also provides a self-configurable fault tolerance module for detecting faults and anomalies.

OPRoS also provides development tools for authoring atomic components and composing components. Component authoring tool, as the name suggests, is used for authoring atomic components. This tool also supports debugging, execution control, and monitoring of atomic components. Component composer is used for composing components to develop the robotic application. Component composer can also be used for remote control and monitoring of multiple component execution engines concurrently. All these tools can be supported on any operating system where eclipse is installed. OPRoS currently supports both Windows and Linux operating systems.

RT-Middleware: RT (Robot Technology)-Middleware[16] serves as a distributed component-based middleware for developing robotic systems [4]. It studies modularization of robotic elements and proposed RT-Components as the basic software unit based on Common Object Request Broker Architecture (CORBA). RT-middleware supports the construction of various networked robotic systems by the integration of various RT-Components. An open-source implementation called OpenRTM-aist [5] was developed for feedback from the robotic research community.

An RT-Component is composed of a component object as the main body, activity as the main process unit, and input ports (InPorts) and output ports (OutPorts) as data stream ports. The activity serves as a controller of the device and processes the designed tasks continuously. The activity has eleven states, and each state is possible to have three methods called entry, do, and exit. These three methods will be called automatically on entry to, being in, and on exit from the state, respectively. The transition of the states is uniform for all RT-Components. Hence, developers may implement the methods for each state to build a new RT-Component.

The InPorts and OutPorts take advantages of publisher/subscriber model. On the one hand, an InPort serves as a subscriber and may subscribe several OutPorts. It also provides a common method called `InPort::put` to allow data to be written. On the other hand, an OutPort serves as a publisher and write data to those InPorts who have subscribed it by using the method of `InPort::put`. It also provides several subscription types, e.g., Once, Periodic, and Triggered.

The RT-Middleware provides several methods for integrating RT-Components. These methods include assembly GUI tool, script language, XML file, other RT-Components, and other application programs. By integration of RT-Components, applications can be built from bottom to up. Such applications include network distributed monitoring system [42] and intelligent home service robotic system [43].

ASEBA: ASEBA[17] is an event-based middleware supporting distributed control and efficient resources exploitation among multiple microcontrollers in a robot [52, 55]. The ASEBA is specially designed for robots with more than one microcontroller sharing a bus for communication.

The ASEBA abandons traditional architecture where the main microcontroller controls all the other microcontrollers and manages all data transfers. As an event-

[16]http://openrtm.org.

[17]http://mobots.epfl.ch/aseba.php.

based middleware, it utilizes multi-master bus in which all microcontrollers can initiate data transfers. The multi-master bus enables asynchronous messages, called events, transferred between microcontrollers. Without control of a centralized main processor, load on bus is significantly reduced. Also, the main process can get released from processing messages and be dedicated for CPU-intensive tasks such as path planning and image processing.

A scripting language called AESL (ASEBA Event Scripting Language) is provided to describe even emission and reception policy in ASEBA. A piece of AESL program can be compiled into bytecode using the designed compiler and ran on the implemented virtual machine. The compiler, together with a script editor and a distributed debugger, forms into the integrated development environment for ASEBA. The virtual machine is lightweight with less than 1000 lines of C program including debugging logic.

The ASEBA has been successfully deployed in the handbot for the task of climbing a shelf and in the marXbot to improve the performance of behaviors in terms of a polling-based approach. The ASEBA has been utilized for the purpose of education [53] and managing a collection of single-microcontroller robots [54].

6 Future Directions and Challenges

In this section, we have given some observations regarding future directions and challenges for MRS middleware. These observations have been made after reviewing existing middleware for MRS. We discuss which design goals have been commonly addressed by existing middleware and which ones need more research effort. Since design of middleware is dependent on hardware, it is important to develop multi-robot systems, which are cheaper, smaller, and have better sensing, processing, and communication technologies. Due to the limitation in technology, each robot in MRS has very limited processing power, storage, communication capability, and battery capacity. These resource limitations make it difficult to develop sophisticated middleware for MRS. One of the major research directions is to develop lightweight middleware that can be used to enable different design goals. Another challenging issue arises due to the heterogeneity of robots. MRS work by collaborating and coordinating with each other, however, heterogeneity of robots and communication protocols makes it a challenging task. Collaboration involves partitioning and distribution of complicated tasks among multiple robots; therefore, middleware should provide lightweight algorithms for task partitioning and management. Scalability will be another major challenge in coming future. Middleware should be designed to support scalability and dynamic configuration of system.

Middleware evaluation is a major part in analyzing the middleware. Middleware is usually evaluated by quantifying the system parameters based on some application example; however, there are some issues with this approach. Application examples that are used for middleware evaluation only focus on some specific system requirements which means evaluation depends highly on the application example being used [69]. Besides, different middleware architectures are developed for different applications, and thus it may not be the best criteria to evaluate a middleware based on some specific parameters. Another issue with middleware evaluation is that it is difficult to quantify the usability of middleware [69]. Several mechanisms such as the use of a questionnaire, number of lines of code, or time to develop the system have been used to determine whether middleware is easy to use or not but, this is not very efficient. Middleware evaluation is currently a challenging issue for researchers and developers which requires more research efforts.

Robot software architecture can usually be classified into three types, which are object-oriented robotics, component-based robotics, and service-driven robotics [1]. Most middleware examples that have been discussed in Sect. 5 use small modules or components for developing a robotic system. Very few middleware, in general, follow monolithic approach for developing robotic software platform. It is preferred to use small modules for system development as it reduces the complexity of the whole system and enables reusability. Component-based approach is a common choice for building middleware for MRS. Orca, MARIE, Urbi, Robocomp, OPRoS, and RT-Middleware follow component-based architecture for developing middleware. Miro and Mira follow object-oriented paradigm, and MRDS is a service-driven middleware that follows REST pattern. All the other remaining middlewares such as ROS, WURDE, OpenRDK, Player, etc. use modular architecture. Out of all these middleware architectures, ROS and Player are most popular among robotic system developers. Although component- and modular-based approaches offer many benefits, a challenging issue is to integrate different components which results in issues related to communication, interoperability, and configuration [65]. Currently, a new trend is emerging in robotics community to use model-driven engineering for robotics software [1]. SmartMDSD Toolchain [89] is a toolchain based on model-driven software development approach that provides an integrated modeling environment to create an overall workflow for robotics software development.

We described some design goals for middleware in Sect. 3; however, we observed that each middleware is usually designed focusing on some specific goals. It is very difficult to achieve all design goals simultaneously [86]. Future work in MRS middleware should consider satisfying multiple design goals to enable multidimensional benefits. Out of all the design goals, flexibility and software reusability is the most common objective for existing middleware. Software reusability is enabled by the use of modular or component-based middleware architectures. Second observation is that most of the middleware are freely available as open source. This is usually done to enable debugging of the software and make it more popular. Since heterogeneity is a major issue in robotic system development, most middleware tries to provide some programming abstractions and make their system platform and language independent. Linux is the first choice of operating system that is supported by most middleware, Windows being the second, and there are some middlewares such as Orca and RoboComp that support Mac OS too. Most middlewares provide some graphical interface that enables management and monitoring of the robotic system. Orca, OpenRDK, MARIE, Urbi, MRDS, RoboComp, ROS, and OPRoS are some middleware examples that provide specific management and monitoring tools. Apart from graphical interfaces, other mechanisms can also be used for management and monitoring purposes.

Although the features currently supported by existing middleware help in decreasing the complexity and provide some other useful features, there are many design goals which have not been fully addressed. One such design goal is collaboration among multiple robots which is currently achieved by existing middleware as they are usually modular and support distributed control. However, existing middleware does not focus much on providing some specific abstractions or tools to facilitate collaboration. A similar case is observed for dynamic resource discovery and configuration where specific tools and abstractions are not provided to enable dynamic configuration of system. Most middlewares do not provide explicit facilities to make the system more robust. Very few middlewares provide explicit fault tolerance capabilities. Out of all the middleware discussed, OPRoS, MIRA, and MRDS explicitly consider fault tolerance. Real-time support is another design goal which has not been addressed by many middleware. RT-Middleware, MIRA, ROS, and ASEBA are among the few middlewares that provide some form of real-time support. RT-Middleware provides some support for real-time processing, MIRA provides a mechanism to minimize latency, ROS enables real-time inspection and monitoring of any variable, and ASEBA also provides real-time support. Real-time support is essential for many robotic applications and thus, more research efforts are required to address this issue. Most middlewares currently do not focus much on supporting Quality of Service (QoS) requirements such as security, reliability, etc. Security and privacy are important concerns as MRSs are used for critical applications such as battlefield surveillance, exploration missions, etc. Very few middlewares provide security mechanism. Since MRSs are distributed in nature, it is challenging to develop distributed security algorithm. Although existing middlewares are developed to integrate sensors and actuators within the robots, integration with other technologies such as cloud computing and IoT has not been taken into much consideration.

Acknowledgements This work was supported by the ANR/RGC Joint Research Scheme [grant number A-PolyU505/12], the NSFC Key Grant [grant number 61332004], and the NSFC/RGC Joint Research Scheme [grant number N-PolyU519/12].

References

1. Ahmad, A., Babar, M.A.: Software architectures for robotic systems: a systematic mapping study. J. Syst. Softw. **122**, 16–39 (2016)
2. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. Comput. Netw. **38**(4), 393–422 (2002)
3. Alimisis, D.: Educational robotics: open questions and new challenges. Themes Sci. Technol. Educ. **6**(1), 63–71 (2013)
4. Ando, N., Suehiro, T., Kitagaki, K., Kotoku, T., Yoon, W.K.: Rt-middleware: distributed component middleware for rt (robot technology). In: 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 3933–3938. IEEE (2005)

5. Ando, N., Suehiro, T., Kotoku, T.: A software platform for component based rt-system development: Openrtm-aist. In: International Conference on Simulation, Modeling, and Programming for Autonomous Robots, pp. 87–98. Springer (2008)
6. Arai, T., Pagello, E., Parker, L.E.: Editorial: advances in multi-robot systems. IEEE Trans. Rob. Autom. **18**(5), 655–661 (2002)
7. Arbuckle, D., Requicha, A.A.: Self-assembly and self-repair of arbitrary shapes by a swarm of reactive robots: algorithms and simulations. Auton. Robots **28**(2), 197–211 (2010)
8. Baillie, J.C., Demaille, A., Hocquet, Q., Nottale, M., Tardieu, S.: The URBI universal platform for robotics. In: First International Workshop on Standards and Common Platform for Robotics (2008)
9. Beasley, R.A.: Medical robots: current systems and research directions. J. Robot. 2012 (2012)
10. Benitti, F.B.V.: Exploring the educational potential of robotics in schools: a systematic review. Comput. Educ. **58**(3), 978–988 (2012)
11. Broadbent, E., Stafford, R., MacDonald, B.: Acceptance of healthcare robots for the older population: review and future directions. Int. J. Soc. Robot. **1**(4), 319–330 (2009)
12. Bruce, J., Zickler, S., Licitra, M., Veloso, M.: Cmdragons: dynamic passing and strategy on a champion robot soccer team. In: IEEE International Conference on Robotics and Automation, 2008. ICRA 2008, pp. 4074–4079. IEEE (2008)
13. Bruyninckx, H.: Open robot control software: the orocos project. In: IEEE International Conference on Robotics and Automation, 2001. Proceedings 2001 ICRA, vol. 3, pp. 2523–2528. IEEE (2001)
14. Burgard, W., Moors, M., Fox, D., Simmons, R., Thrun, S.: Collaborative multi-robot exploration. In: IEEE International Conference on Robotics and Automation, 2000. Proceedings. ICRA'00, vol. 1, pp. 476–481. IEEE (2000)
15. Burgner-Kahrs, J., Rucker, D.C., Choset, H.: Continuum robots for medical applications: a survey. IEEE Trans. Robot. **31**(6), 1261–1280 (2015)
16. Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A., Buyya, R.: Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw. Pract. Exp. **41**(1), 23–50 (2011)
17. Calisi, D., Censi, A., Iocchi, L., Nardi, D.: Openrdk: a modular framework for robotic software development. In: 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 1872–1877. IEEE (2008)
18. Chalup, S.K., Murch, C.L., Quinlan, M.J.: Machine learning with AIBO robots in the four-legged league of robocup. IEEE Trans. Syst. Man Cybern. Part C (Appl. Rev.) **37**(3), 297–310 (2007)
19. Chen, D., Varshney, P.K.: Qos support in wireless sensor networks: a survey. In: International Conference on Wireless Networks, vol. 233, pp. 1–7 (2004)
20. Chitic, S.G., Ponge, J., Simonin, O.: Are middlewares ready for multi-robots systems? In: International Conference on Simulation, Modeling, and Programming for Autonomous Robots, pp. 279–290. Springer (2014)
21. Cianci, C.M., Raemy, X., Pugh, J., Martinoli, A.: Communication in a swarm of miniature robots: the e-puck as an educational tool for swarm robotics. In: International Workshop on Swarm Robotics, pp. 103–115. Springer (2006)
22. Collett, T.H., MacDonald, B.A., Gerkey, B.P.: Player 2.0: toward a practical robot programming framework. In: Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005), p. 145 (2005)
23. Cote, C., Brosseau, Y., Letourneau, D., Raïevsky, C., Michaud, F.: Robotic software integration using marie. Int. J. Adv. Robot. Syst. **3**(1), 55–60 (2006)
24. Darwin, C., Beer, G.: The origin of species. Dent (1951)
25. De Rosa, M., Goldstein, S., Lee, P., Campbell, J., Pillai, P.: Scalable shape sculpting via hole motion: motion planning in lattice-constrained modular robots. In: Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006, pp. 1462–1468. IEEE (2006)

26. Di Mario, E., Martinoli, A.: Distributed particle swarm optimization for limited-time adaptation with real robots. Robotica **32**(02), 193–208 (2014)
27. Einhorn, E., Langner, T., Stricker, R., Martin, C., Gross, H.M.: Mira-middleware for robotic applications. In: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 2591–2598. IEEE (2012)
28. Elkady, A., Sobh, T.: Robotics middleware: a comprehensive literature survey and attribute-based bibliography. J. Robot. 2012 (2012)
29. Engelberger, J.F.: Robotics in Practice: Management and Applications of Industrial Robots. Springer Science & Business Media (2012)
30. Farinelli, A., Iocchi, L., Nardi, D.: Multirobot systems: a classification focused on coordination. IEEE Trans. Syst. Man Cybern. Part B (Cybernetics) **34**(5), 2015–2028 (2004)
31. Gerkey, B., Vaughan, R.T., Howard, A.: The player/stage project: tools for multi-robot and distributed sensor systems. In: Proceedings of the 11th International Conference on Advanced Robotics, vol. 1, pp. 317–323 (2003)
32. Gerkey, B.P., Vaughan, R.T., Stoy, K., Howard, A., Sukhatme, G.S., Mataric, M.J.: Most valuable player: a robot device server for distributed control. In: 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2001. Proceedings, vol. 3, pp. 1226–1231. IEEE (2001)
33. Gouaillier, D., Hugel, V., Blazevic, P., Kilner, C., Monceaux, J., Lafourcade, P., Marnier, B., Serre, J., Maisonnier, B.: The nao humanoid: a combination of performance and affordability. CoRR abs/08073223 (2008)
34. Grieco, L.A., Rizzo, A., Colucci, S., Sicari, S., Piro, G., Di Paola, D., Boggia, G.: Iot-aided robotics applications: technological implications, target domains and open issues. Comput. Commun. **54**, 32–47 (2014)
35. Gummadi, R., Gnawali, O., Govindan, R.: Macro-programming wireless sensor networks using kairos. In: International Conference on Distributed Computing in Sensor Systems, pp. 126–140. Springer (2005)
36. Habibi, G., Xie, W., Jellins, M., McLurkin, J.: Distributed path planning for collective transport using homogeneous multi-robot systems. In: Distributed Autonomous Robotic Systems, pp. 151–164. Springer (2016)
37. Heckel, F., Blakely, T., Dixon, M., Wilson, C., Smart, W.D.: The wurde robotics middleware and ride multirobot tele-operation interface. In: Proceedings of the 21st National Conference on Artificial Intelligence (AAAI06) (2006)
38. Howard, A., Parker, L.E., Sukhatme, G.S.: Experiments with a large heterogeneous mobile robot team: exploration, mapping, deployment and detection. Int. J. Robot. Res. **25**(5–6), 431–447 (2006)
39. Hu, G., Tay, W.P., Wen, Y.: Cloud robotics: architecture, challenges and applications. IEEE Netw. **26**(3), 21–28 (2012)
40. Jackson, J.: Microsoft robotics studio: a technical introduction. IEEE Robot. Autom. Mag. **14**(4), 82–87 (2007)
41. Jang, C., Lee, S.I., Jung, S.W., Song, B., Kim, R., Kim, S., Lee, C.H.: Opros: a new component-based robot software platform. ETRI J. **32**(5), 646–656 (2010)
42. Jia, S., Takase, K.: Network distributed monitoring system based on robot technology middleware. Int. J. Adv. Robot. Syst. **4**(1), 69–72 (2007)
43. Jia, S., Hada, Y., Gakuhari, H., Takase, K., Ohnishi, T., Nakamoto, H.: Intelligent home service robotic system based on robot technology middleware. In: 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 4478–4483. IEEE (2006)
44. Jiang, S., Cao, J., Liu, Y., Chen, J., Liu, X.: Programming large-scale multi-robot system with timing constraints. In: 2016 25th International Conference on Computer Communication and Networks (ICCCN), pp. 1–9. IEEE (2016a)
45. Jiang, S., Liang, J., Cao, J., Liu, R.: An ensemble-level programming model with real-time support for multi-robot systems. In: 2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops), pp. 1–3. IEEE (2016)

46. Kernbach, S., Thenius, R., Kernbach, O., Schmickl, T.: Re-embodiment of honeybee aggregation behavior in an artificial micro-robotic system. Adapt. Behav. **17**(3), 237–259 (2009)
47. Kim, S., Laschi, C., Trimmer, B.: Soft robotics: a bioinspired evolution in robotics. Trends Biotechnol. **31**(5), 287–294 (2013)
48. Kramer, J., Scheutz, M.: Development environments for autonomous mobile robots: a survey. Auton. Robots **22**(2), 101–132 (2007)
49. Liang, J., Cao, J., Liu, R., Li, T.: Distributed intelligent mems: a survey and a real-time programming framework. ACM Comput. Surv. (CSUR) **49**(1), 20 (2016)
50. Lima, P.U., Custodio, L.M.: Multi-robot systems. In: Innovations in Robot Mobility and Control, pp. 1–64. Springer (2005)
51. Lopes, Y.K., Leal, A.B., Dodd, T.J., Groß, R.: Application of supervisory control theory to swarms of e-puck and kilobot robots. In: International Conference on Swarm Intelligence, pp. 62–73. Springer (2014)
52. Magnenat, S., Longchamp, V., Mondada, F.: Aseba, an event-based middleware for distributed robot control. In: Workshops and Tutorials CD IEEE/RSJ 2007 International Conference on Intelligent Robots and Systems, LSRO-CONF-2007-016. IEEE Press (2007)
53. Magnenat, S., Noris, B., Mondada, F.: Aseba-challenge: an open-source multiplayer introduction to mobile robots programming. In: Fun and Games, pp. 65–74. Springer (2008a)
54. Magnenat, S., Rétornaz, P., Noris, B., Mondada, F.: Scripting the swarm: event-based control of microcontroller-based robots. In: SIMPAR 2008 Workshop Proceedings, LSRO-CONF-2008-057 (2008b)
55. Magnenat, S., Rétornaz, P., Bonani, M., Longchamp, V., Mondada, F.: Aseba: a modular architecture for event-based control of complex robots. IEEE/ASME Trans. Mechatron. **16**(2), 321–329 (2011)
56. Makarenko, A., Brooks, A., Kaupp, T.: Orca: components for robotics. In: International Conference on Intelligent Robots and Systems (IROS), pp. 163–168 (2006)
57. Manso, L., Bachiller, P., Bustos, P., Núñez, P., Cintas, R., Calderita, L.: Robocomp: a tool-based robotics framework. In: International Conference on Simulation, Modeling, and Programming for Autonomous Robots, pp. 251–262. Springer (2010)
58. Mataric, M.J.: Interaction and intelligent behavior. Technical report, DTIC Document (1994)
59. McLurkin, J., Smith, J.: Distributed algorithms for dispersion in indoor environments using a swarm of autonomous mobile robots. In: In 7th International Symposium on Distributed Autonomous Robotic Systems (DARS). Citeseer (2004)
60. McLurkin, J., Smith, J., Frankel, J., Sotkowitz, D., Blau, D., Schmidt, B.: Speaking swarmish: Human-robot interface design for large swarms of autonomous mobile robots. In: AAAI Spring Symposium: To Boldly Go Where No Human-Robot Team Has Gone Before, pp. 72–75 (2006)
61. McLurkin, J., Lynch, A.J., Rixner, S., Barr, T.W., Chou, A., Foster, K., Bilstein, S.: A low-cost multi-robot system for research, teaching, and outreach. In: Distributed Autonomous Robotic Systems, pp. 597–609. Springer (2013)
62. McLurkin, J., Rykowski, J., John, M., Kaseman, Q., Lynch, A.J.: Using multi-robot systems for engineering education: teaching and outreach with large numbers of an advanced, low-cost robot. IEEE Trans. Educ. **56**(1), 24–33 (2013)
63. McLurkin, J., McMullen, A., Robbins, N., Habibi, G., Becker, A., Chou, A., Li, H., John, M., Okeke, N., Rykowski, J., et al.: A robot system design for low-cost multi-robot manipulation. In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 912–918. IEEE (2014)
64. Michael, N., Fink, J., Kumar, V.: Experimental testbed for large multirobot teams. IEEE Robot. Autom. Mag. **15**(1), 53–61 (2008)
65. Mohamed, N., Al-Jaroodi, J., Jawhar, I.: Middleware for robotics: a survey. In: 2008 IEEE Conference on Robotics Automation and Mechatronics, pp. 736–742. IEEE (2008)
66. Mohamed, N., Al-Jaroodi, J., Jawhar, I.: A review of middleware for networked robots. Int. J. Comput. Sci. Netw. Secur. **9**(5), 139–148 (2009)
67. Mondada, F., Franzi, E., Guignard, A.: The development of khepera. In: Experiments with the Mini-Robot Khepera, Proceedings of the First International Khepera Workshop, LSRO-CONF-2006-060, pp. 7–14 (1999)

68. Mondada, F., Bonani, M., Raemy, X., Pugh, J., Cianci, C., Klaptocz, A., Magnenat, S., Zufferey, J.C., Floreano, D., Martinoli, A.: The e-puck, a robot designed for education in engineering. In: Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions, IPCB: Instituto Politécnico de Castelo Branco, vol. 1, pp. 59–65 (2009)
69. Mottola, L., Picco, G.P.: Programming wireless sensor networks: fundamental concepts and state of the art. ACM Comput. Surv. (CSUR) **43**(3), 19 (2011)
70. Owens, G., Granader, Y., Humphrey, A., Baron-Cohen, S.: Lego® therapy and the social use of language programme: An evaluation of two social skills interventions for children with high functioning autism and asperger syndrome. J. Autism Dev. Disord. **38**(10), 1944–1957 (2008)
71. Parker, L.E.: Current state of the art in distributed autonomous mobile robotics. In: Distributed Autonomous Robotic Systems 4. Springer, pp. 3–12 (2000)
72. Prencipe, G., Santoro, N.: Distributed algorithms for autonomous mobile robots. In: Fourth IFIP International Conference on Theoretical Computer Science-TCS 2006, pp. 47–62. Springer (2006)
73. Pugh, J., Raemy, X., Favre, C., Falconi, R., Martinoli, A.: A fast onboard relative positioning module for multirobot systems. IEEE/ASME Trans. Mechatron. **14**(2), 151–162 (2009)
74. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA Workshop on Open Source Software, Kobe, Japan, vol. 3, p. 5 (2009)
75. Rogers III, J.G., Trevor, A.J., Nieto-Granda, C., Cunningham, A., Paluri, M., Michael, N., Dellaert, F., Christensen, H.I., Kumar, V.: Effects of sensory precision on mobile robot localization and mapping. In: Experimental Robotics, pp. 433–446. Springer (2014)
76. Rubenstein, M., Shen, W.M.: Automatic scalable size selection for the shape of a distributed robotic collective. In: 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 508–513. IEEE (2010)
77. Rubenstein, M., Ahler, C., Nagpal, R.: Kilobot: a low cost scalable robot system for collective behaviors. In: 2012 IEEE International Conference on Robotics and Automation (ICRA), pp. 3293–3298. IEEE (2012)
78. Rubenstein, M., Cabrera, A., Werfel, J., Habibi, G., McLurkin, J., Nagpal, R.: Collective transport of complex objects by simple robots: theory and experiments. In: Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems, International Foundation for Autonomous Agents and Multiagent Systems, pp. 47–54 (2013)
79. Rubenstein, M., Ahler, C., Hoff, N., Cabrera, A., Nagpal, R.: Kilobot: a low cost robot with scalable operations designed for collective behaviors. Robot. Auton. Syst. **62**(7), 966–975 (2014)
80. Rubenstein, M., Cornejo, A., Nagpal, R.: Programmable self-assembly in a thousand-robot swarm. Science **345**(6198), 795–799 (2014)
81. Saeedi, S., Trentini, M., Seto, M., Li, H.: Multiple-robot simultaneous localization and mapping: a review. J. Field Robot. **33**(1), 3–46 (2016)
82. Sapaty, P.: Military robotics: latest trends and spatial grasp solutions. Int. J. Adv. Res. Artif. Intell. **4**(4), 9–18 (2015)
83. Sartoretti, G., Hongler, M.O., de Oliveira, M.E., Mondada, F.: Decentralized self-selection of swarm trajectories: from dynamical systems theory to robotic implementation. Swarm Intell. **8**(4), 329–351 (2014)
84. Schlegel, C., Worz, R.: Interfacing different layers of a multilayer architecture for sensorimotor systems using the object-oriented framework smartsoft. In: 1999 Third European Workshop on Advanced Mobile Robots, 1999 (Eurobot'99), pp. 195–202. IEEE (1999)
85. Siciliano, B., Khatib, O.: Springer Handbook of Robotics. Springer Science & Business Media (2008)
86. Smart, W.D.: Is a common middleware for robotics possible? In: Proceedings of the IROS 2007 Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware. Citeseer, vol. 1 (2007)
87. Soares, J.M., Aguiar, A.P., Pascoal, A.M., Martinoli, A.: A graph-based formation algorithm for odor plume tracing. In: Distributed Autonomous Robotic Systems, pp. 255–269. Springer (2016)

88. Soares, J.M., Navarro, I., Martinoli, A.: The khepera iv mobile robot: performance evaluation, sensory data and software toolbox. In: Robot 2015: Second Iberian Robotics Conference, pp. 767–781. Springer (2016)
89. Stampfer, D., Lotz, A., Lutz, M., Schlegel, C.: The smartmdsd toolchain: an integrated mdsd workflow and integrated development environment (ide) for robotics software. J. Softw. Eng. Robot. **7**(1), 3–19 (2016)
90. Stoy, K., Nagpal, R.: Self-repair through scale independent self-reconfiguration. In: 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2004 (IROS 2004). Proceedings, vol. 2, pp. 2062–2067. IEEE (2004)
91. Tsui, K.M., Yanco, H.A.: Assistive, rehabilitation, and surgical robots from the perspective of medical and healthcare professionals. In: AAAI 2007 Workshop on Human Implications of Human-Robot Interaction, Technical Report WS-07-07 Papers from the AAAI 2007 Workshop on Human Implications of HRI (2007)
92. Utz, H., Sablatnog, S., Enderle, S., Kraetzschmar, G.: Miro-middleware for mobile robot applications. IEEE Trans. Robot. Autom. **18**(4), 493–497 (2002)
93. Volpe, R., Nesnas, I., Estlin, T., Mutz, D., Petras, R., Das, H.: The CLARAty architecture for robotic autonomy. In: Aerospace Conference, 2001, IEEE Proceedings, vol. 1, pp. 1–121. IEEE (2001)
94. Wang, M.M., Cao, J.N., Li, J., Dasi, S.K.: Middleware for wireless sensor networks: a survey. J. Comput. Sci. Technol. **23**(3), 305–326 (2008)
95. Whittier, L.E., Robinson, M.: Teaching evolution to non-english proficient students by using lego robotics. Am. Second. Educ. 19–28 (2007)
96. Wurman, P.R., D'Andrea, R., Mountz, M.: Coordinating hundreds of cooperative, autonomous vehicles in warehouses. AI Mag. **29**(1), 9 (2008)
97. Yan, Z., Jouandeau, N., Cherif, A.A.: A survey and analysis of multi-robot coordination. Int. J. Adv. Robot. Syst. **10** (2013)