Research Article

INTERNATIONAL JOURNAL OF ADVANCED ROBOTIC SYSTEMS

International Journal of Advanced Robotic Systems
November-December 2016: 1–15

DOI: 10.1177/1729881416663662
arx.sagepub.com
SAGE

EmSBot: A modular framework supporting the development of swarm robotics applications

Long Peng[1,2], Fei Guan[1], Luc Perneel[1], and Martin Timmerman[1]

Abstract

Component-based approaches are prevalent in software development for robotic applications due to their reusability and productivity. In this article, we present an *E*mbedded *m*odular *S*oftware framework for a networked ro*BoT*ic system (EmSBoT) targeting resource-constrained devices such as microcontroller-based robots. EmSBoT is primarily built upon μCOS-III with real-time support. However, its operating system abstraction layer makes it available for various operating systems. It employs a unified port-based communication mechanism to achieve message passing while hiding the heterogeneous distributed environment from applications, which also endows the framework with fault-tolerant capabilities. We describe the design and core features of the EmSBoT framework in this article. The implementation and experimental evaluation show its availability with small footprint size, effectiveness, and OS independence.

Keywords

Robotic software framework, multi-robot system, real-time, EmSBoT

Date received: 28 January 2016; accepted: 25 May 2016

Topic: Mobile Robots and Multi-robot Systems
Topic Editor: Nak Young Chong

Introduction

Currently, advances in sensors, actuators, and computer technologies have led to the increasing complexity of robotic systems. It is a challenging and tedious task to develop new robotic applications for such platforms, especially when the robotic system is equipped with several heterogeneous embedded processors, considered as a networked system (e.g. the ATHLETE robot,[1] Triskar2,[2] and s-bot[3]). Engineers have to deal with diverse functional modules of the applications, from low-level device drivers to high-level functions such as planning and navigation. On the other hand, it has long been recognized that the use of a multi-robot system can outperform an individual versatile robot, such as space exploration, object transport, and distributed manipulation. Writing software for a multi-robot system further complicates the situation when considering coordination and communication problems. Overall, we can view an advanced robot with several processors and multi-robot system with communication capability as distributed computing systems connected via physical or wireless network, a special case of a networked robotic system (NRS).[4]

Due to the inherent difficulty in developing complex robotic applications from scratch, a number of robotic software frameworks (RSFs) or middlewares have been specifically designed and widely used to alleviate the process, such as Player,[5] ROS,[6] ASEBA,[7] RSCA,[8] MIRA,[9] Orocos,[10] Miro,[11] OpenRDK,[12] OPRoS,[13] and OpenRTM-aist.[14]

[1] Vrije Universiteit Brussel, Brussel, Belgium
[2] National University of Defense Technology, Changsha, Hunan Province, China

Corresponding author:
Long Peng, Vrije Universiteit Brussel (VUB), Pleinlaan 2, Brussel 1050, Belgium.
Email: longpeng@vub.ac.be

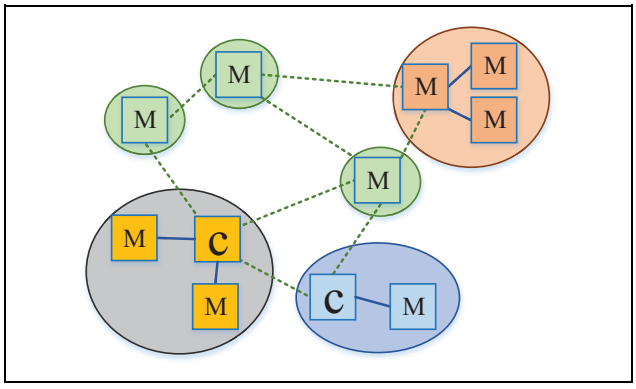

Figure 1. The extended networked robotic system. A robot could be composed of a single microcontroller, a central processor with several microcontrollers, or just several homogeneous microcontrollers. C: central processor; M: microcontroller; ○: robot; _____: linked via physical bus; . . . . . : wirelessly connected.

Nearly all these frameworks share the same paradigm—a component-based approach—an idea naturally originating from component-based software engineering (CBSE). By applying the principles of CBSE, the robotic application is divided into different software "agents", which are loosely coupled. Depending on the framework, an "agent" is also called a "component," "module," "node," or "resource." In the rest of the article, the term agent is used. These loosely coupled agents only export data or control interfaces to others, while the RSF is responsible for the message passing and event notification between agents. Besides the communication mechanism, some RSFs also specify the agent model for applications, such as OpenRTM-aist,[14] GenoM3,[15] and FINROC.[16] Using RSF in the development process of robotic applications introduces better software quality, code reusability, and collaborative development. However, without deliberately designing the interactions between agents, achievement of the quality-of-service requirements of some applications, such as the real-time constraints, is an intractable problem.

This article focuses on the problem of applying RSF in the domain of NRS. We extend the concept of NRS to a more general idea as illustrated in Figure 1, including not only a group of connected robots, such as swarm robots and modular robots with communication feature,[17,18] but also a complex self-contained robot with distributed processing units. We claim that every node in Figure 1 is equipped with a processing unit, some memory, and some peripheral interfaces that can be used to connect sensors or control actuators, as well as some communication channels. The processing unit could be a microprocessor, microcontroller, or digital signal processor. Using RSF to develop applications for such a heterogeneous system presents some challenges and requirements.

- *Support for heterogeneous hardware platforms and operating systems*. The framework should support diverse hardware platforms, the memory of which could be limited to dozens of kilobytes. Therefore, the framework must be configurable and scalable to support the diversity. On the other hand, the hardware varieties also result in different operating systems on top. Due to the resource constraints in embedded systems, real-time operating systems (RTOSs) are widely used. The programming interfaces and procedures are very different from those in desktop operating systems. The framework should abstract the underlying operating system in order to be ported to other operating systems without too much effort.
- *Communication network and transport protocol independence*. The framework should not depend on the communication network adopted. The communication between computing nodes on a complex robot may be based on a physical network bus, such as an Ethernet or CAN bus. In swarm and modular robotics, the communication may rely on a wireless network, such as ZigBee, Bluetooth, and WLAN. The framework should have the flexibility to switch to different communication networks.
- *Real-time support*. The real-time requirement is ubiquitous in robotic systems, especially in the automotive and avionic domains. Deploying the framework above or on top of the RTOS is the preliminary step in guaranteeing the real-time requirement. In addition, the framework should support the real-time requirement inherently or not jeopardize it. The distribution of the whole system further complicates the situation. It may need real-time communication, whereas we claim that the framework should be network transparent. Therefore, the framework should employ a mechanism to coordinate the real-time communication with the network layer.
- *Fault-tolerant support*. NRS provides reliability and fault tolerance inherently by adding redundant features and distributing task responsibilities to different nodes. An abnormal hardware node or software agent will not disable the system because of automatic reconfiguration.[19] The software framework should have such an advantage and should further simplify the reconfiguration process.
- *Simplicity and ease of use*. The framework should be simple and easy to learn. The programming model of the framework is intuitive and existing code can be adapted to the programming model with little or no change.

This article presents EmSBoT, an *E*mbedded *m*odular *S*oftware framework for a networked ro*BoT*ic system, which considers the aforementioned requirements. The rest of the article is organized as follows: Section "Related work" gives an overview of related work and highlights the distinguished features of EmSBoT. Section "The

EmSBoT framework'' describes the internals of EmSBoT design as well as its core features. Section ''Implementation and experimental evaluation'' presents its implementation details, some benchmarking results, and a multi-robot leader–follower case study to demonstrate its efficacy. Finally, we give some conclusion remarks and future work in section ''Conclusion and future work.''

Related work

Many RSFs with various characteristics are available. However, none of them is the de facto standard tool for building robotic applications in the educational and industrial communities, due to the diverse functional and non-functional requirements in different robotic projects. A comprehensive survey and comparison of existing frameworks are given in the literarure.[20–22] Here, we just analyze the attributes that EmSBoT is supposed to include.

After reviewing existing RSFs, we found that none are designed to support heterogeneous hardware platforms and operating systems. Most of them only support the desktop-level environment without taking into account resource-constrained platforms, such as microcontroller-based robots. On the other hand, there are also some RSFs that support embedded hardware, such as R2P,[2] ASEBA,[7] and μORB.[23] However, they do not consider the cross-platform problem. For programming a networked robotic system that includes heterogeneous nodes, an existing solution is to use an RSF in desktop-level or powerful embedded computers with or without an embedded RSF on other nodes. The communication between different RSFs is via a specific wrapper. For instance, in R2P, an μROSnode is developed for integration with the ROS system. In the study by Mellinger et al.,[24] a desktop with the ROS system is used to control multiple quadrotors without an RSF on board. The onboard processor is only used to read and transmit sensor data to the desktop, to receive control commands, and to set actuators. However, with advances in processor power and sensors, it is capable of deploying a (real-time) operating system with a lightweight RSF in the quadrotor, as it did in the study by Meier et al.[23] Instead of building a bridge between different RFSs, we propose the idea that a unified framework supporting various platforms can further alleviate the development of applications for NRSs.

Most of the existing frameworks build upon the TCP/IP or UDP/IP network, such as MIRA, ROS, OpenRDK, and Player; some frameworks are also based on general communication middlewares, such as OpenRTM-aist and MIRO adhering to CORBA. FINROC, R2P, and Orocos support communication over CAN bus. ASEBA supports both CAN bus[7] and Bluetooth networks.[25] A communication system based on a ZigBee network was evaluated for self-reconfiguring modular robots in the study by Fitch and Lal.[26] None of these frameworks are versatile enough to support all the networks, or to be network transparent. The capability to seamlessly switch to another network should be integrated into the framework. This feature is profitable for robots with multiprocessors, as well as multi-robot systems.

ROS, Player, and OpenRDK, from their original design goals, have not been dealing with a robotic system with the real-time requirement but are aimed at service robots and assistive robots. ASEBA employs the event-based programming paradigm to support the real-time requirement. RT-CORBA is used in RSCA to support real-time communication. Orocos, built on top of RTAI and RTLinux, supports real-time components. OpenRTM-aist supports real-time capability through its RT-component model. MIRA, by avoiding unnecessary copying of data when handling communication between agents, outperforms other frameworks in terms of communication latency and memory usage. ChibiOS/RT, an RTOS, is chosen in R2P to meet the real-time requirement. R2P also integrates a real-time CAN bus protocol to transmit messages between networked nodes.

Fault tolerance includes fault detection and recovery at runtime. As stated in the study by Cui et al.,[19] faults include two aspects: functional and nonfunctional. Detection of the functional and nonfunctional faults is rarely supported in existing frameworks except OPRoS. Fault recovery can be supported if the framework supports agent reconfiguration at runtime. This is strongly related to the programming mode and communication style of the framework. ROS, MIRA, OpenRDK, R2P, and μORB all adopt a similar topic-based communication style. They can achieve fault recovery by replacing the failure agent when finding fault artificially. Another favorable communication style is the port-based mechanism, which is adopted in OPRoS, OpenRTM-aist, FINROC, and COP frameworks.[27] This mechanism makes agents loosely coupled, and the communication between agents can be created and destroyed dynamically, which is highly suitable for failure recovery through agent reconfiguration. OPRoS is endowed with fault tolerance through a specific fault-manager module.[28,29]

What most distinguishes EmSBoT from existing frameworks is that it provides a lightweight unified programming and communication interface for heterogeneous distributed environments, ranging from resource-constrained embedded microcontrollers to desktop-level computers. In EmSboT, the communication networks are isolated as special agents, making the framework network transparent and adaptable to diverse communication protocols. This feature is highly beneficial for the devices that inhabit more than one network. Apart from being built on top of RTOSs intentionally as other RSFs do to support the real-time requirement, EmSBoT also employs a priority-based message passing approach to increase its soft real-time capability. Furthermore, the framework integrates an affordable distributed fault-tolerant mechanism that is based on the port-based communication style. All these features will be introduced in detail in the following section.

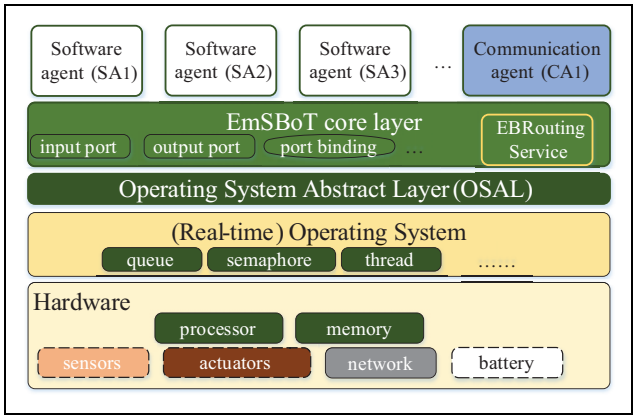

Figure 2. The architecture of the EmSBoT framework in one microcontroller-based robot. EmSBoT: *Em*bedded *m*odular *S*oftware framework for a networked ro*BoT*ic system.

The EmSBoT framework

The EmSBoT framework was initially proposed to address the problem of communication transparency between swarm robots in our laboratory. These robots are resource constrained, being equipped with ARM Cortex-M MCUs, but capable of embedding an RTOS inside. Also, a more powerful robot equipped with the Cortex-A processor could be used as the mother ship. The communication between them is achieved via a Digi XBee RF module considering the cost and power consumption. The framework must be able to support such a circumstance. If we abstract the underlying network layer, the framework is highly suitable for complex robots with multiprocessors. With this vision, the EmSBoT framework is developed to address the requirements and challenges mentioned earlier.

Overall architecture

The overall architecture of EmSBoT is deliberately divided into three layers (see Figure 2): the operating system abstraction layer, the core of the framework, and the application layer composed of software agents.

In order to achieve portability, we argue that EmSBoT should only utilize the generic system calls that operating systems provide, such as calls to thread (or task), semaphore, mutex, and message queue. This principle guides us to abstract these services to a separate layer, leading to the uniform wrapper structures and functions for the upper layer. Our previous experience in evaluating (real-time) operating systems using our benchmark suite proves its feasibility.[30,31]

Based on the OS abstraction layer, we implemented the core framework of EmSBoT with the aim of minimizing its memory footprint and dependencies on other libraries. This layer defines the programming model for applications, provides core services to the upper application layer, and routes messages between agents. The real-time requirement and fault-tolerant mechanism are also supported in this layer.

The modularity of the framework is implemented by dividing the application into software agents, which are managed by the EmSBoT core. These agents are functionally independent in principle and interact via message passing.

System and programming model

The construction of a distributed EmSBoT system stems from the notion of a computation "node," which is addressable. Formally, the system can be expressed as a tuple $< N, C >$, where N is the set of computation nodes and C is the set of communication channels. Normally, the system contains only one communication channel, whereas a more sophisticated system may have several communication channels connected via gateway nodes. For example, in the literature,[2] the CAN network and IP network are mixed by a special gateway node.

Node. Each node n_i in N is composed of a set of software agents and communication ports. We argue that all the elements of n_i share the same memory space. For example, a microcontroller with RTOS and a communication medium inside can be viewed as a node (see Figure 2); a process in the desktop OS such as Linux can also be a node, which could be addressed by a TCP or UDP port number. Every node in the system has a unique ID, NodeID, which can be set to its network address. Then, n_i can be generalized as a tuple $< NodeID, A, IP, OP, NC >$. A stands for the set of agents. IP and OP denote the set of input ports and the set of output ports, respectively. We will introduce these notions later. $NC \subseteq C$ is the communication channels owned by the node. Except for the gateway nodes, the cardinality of NC is equal to one in most cases, implying that this node is only connected to one communication network. In a few situations, in order to increase the robustness of the node, it could double the same communication link for redundancy, which is beyond the scope of this article. But we could regard them as two different links.

Agent. Each agent in the node also has a unique ID, *AgentID* so in the whole system an agent can be identified and addressed by $< NodeID, AgentID >$. An agent is associated with four hook functions: *Init*, *Exec*, *Destroy*, and *Faulthandling*, which denote the major phases of the agent's life cycle. Each agent also comprises a group of coupled threads: some optional worker threads and one mandatory main thread (MT) that controls the life cycle of the agent. In MT, the agent starts by executing the *Init* hook function to do some initialization work, such as allocating ports, binding ports, and creating worker threads; then, the *Exec* function is called periodically according to a specified period T. If an agent is designed with only the *MT* thread, the *Exec* can be used to carry out the main logic of the agent. When an agent has multiple worker threads, it is recommended to monitor and inspect the running of worker threads in *Exec*. If some faults are detected, the

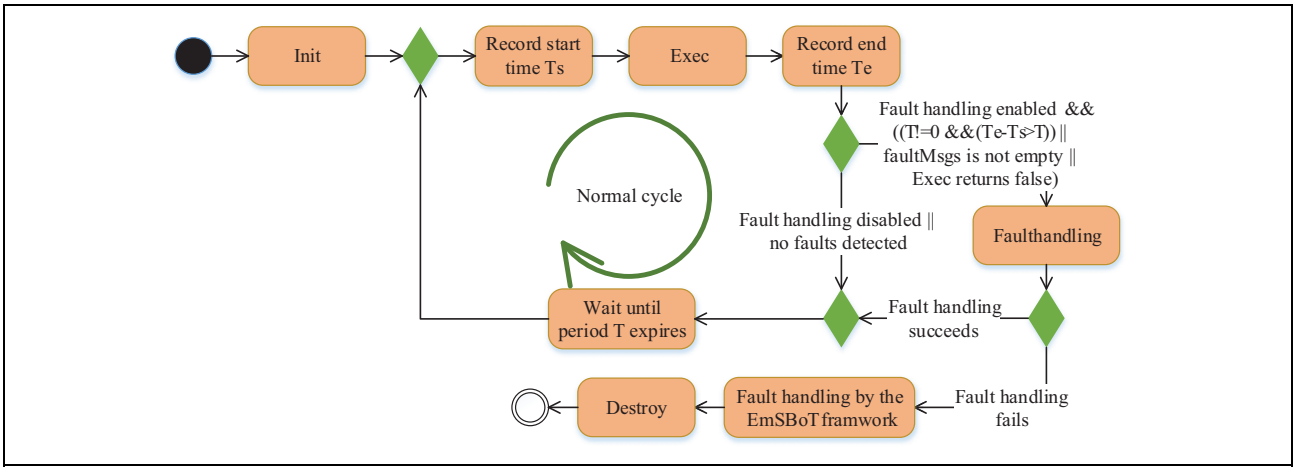

Figure 3. Activity diagram of the lifecycle of an agent, integrated with the fault-tolerant mechanism.

agent enters into the fault-handling procedure and execute the *Faulthandling* function. If the agent cannot recover from faults after *Faulthandling*, finally MT will invoke the *Destroy* function that can be used to release resources. Figure 3 shows the life cycle of an agent. The fault-tolerant mechanism will be presented in detail in section "Fault-tolerant support." An agent also has a priority window τ $([\tau_{\min}, \tau_{\max}], \tau_{\min} \leq \tau_{\max})$ to support the real-time requirement. The priority of every thread in the agent must lie in τ. Then, formally, an agent is a tuple $< AgentID, NodeID, Funcs, Thrs, \tau, T, IP^{'}, OP^{'} >$. *Funcs* is the set of four hook functions; *Thrs* is the set of threads; $IP^{'} \subseteq IP$ and $OP^{'} \subseteq OP$ are the ports that the agent holds. The framework always keeps the rule that for any two agents A_i and A_j, $IP^{'}[A_i] \cap IP^{'}[A_j] = \emptyset$ and $OP^{'}[A_i] \cap OP^{'}[A_i] = \emptyset$. This rule guarantees that a port is exclusively owned by one agent.

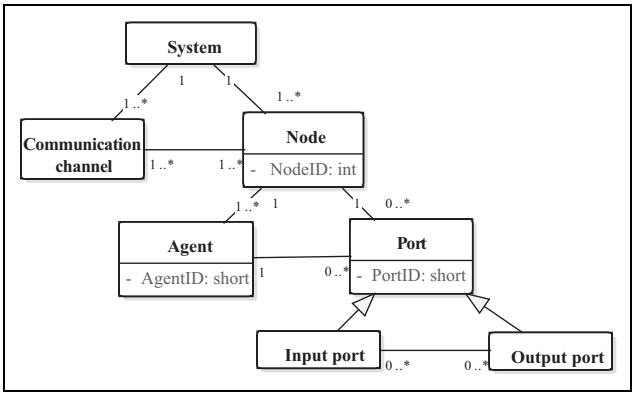

Figure 4. UML diagram of the EmSBoT system model. EmSBoT: *E*mbedded *m*odular *S*oftware framework for a networked ro*BoT*ic system.

Port. Similarly to other frameworks,[13,14,16,27] ports are used to pass messages between agents, giving the advantage of flexibility, low coupling, and ease of use. In our model, the input and output ports are explicitly distinguished. An agent can only receive data from input ports and send data to output ports. The data flow is established by binding input ports to output ports. In the OPRoS model, three types of ports are defined: service, data, and event ports.[13] However, in order to maintain simplicity and reduce the footprint size, only the type of data port is reserved in EmSBoT. We argue that the event port and service port can be easily implemented using a data port. Actually, an event usually includes the payload data,[7] so listening to an event can be converted to listening to the data port. One outstanding feature of EmSBoT is that only the pointer to messages is transmitted in the node domain, which guarantees real-time capability in inner-node communication. Every port in our model also has a unique ID, *PortID*, and a string, *Name* indicating its data type. The input (output) port can only bind to the output (input) port, which has the same *Name*. The model does not restrict the number of output ports that can bind to the same input port, and vice versa. Every port in EmSBoT inherits the same priority τ_{mt} from the MT thread. Then, either an input port or an output port can be expressed as a tuple $< PortID, AgentID, NodeID, Name, \tau_{mt}, PL >$. The *AgentID* denotes the agent that owns the port, and the *NodeID* indicates the node at which the port is located. *PL* is the set of ports that bind to it.

Figure 4 shows the UML diagram of these elements in the EmSBoT system. The framework is responsible for maintaining their relationships, which can be used to help analyze agent dependencies and fault recovery.

Communication mechanism

The communication between agents is achieved via the data flow from the output port of one agent to the input port of another agent. The communication process is explicitly divided into two steps: (1) port binding and (2) data sending and receiving.

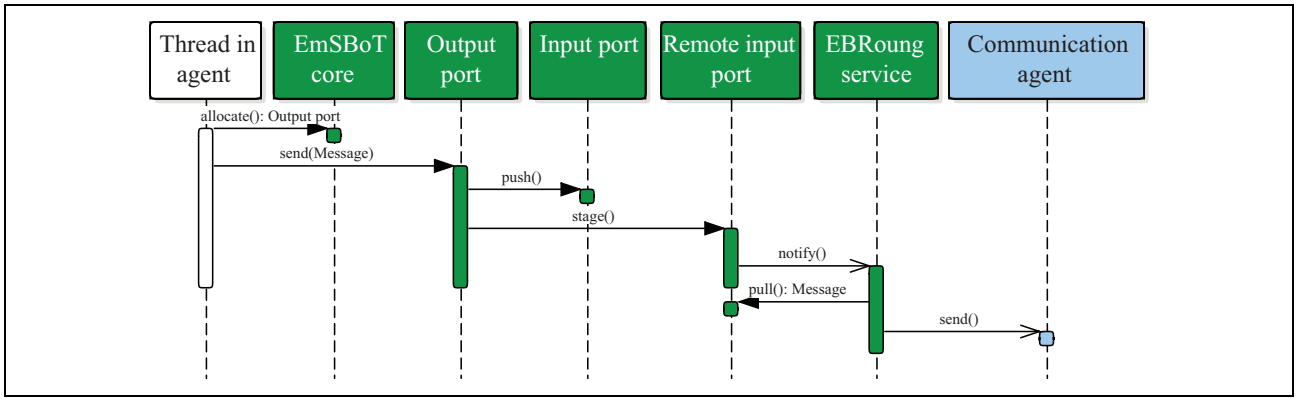

Figure 5. The sequence diagram of sending data.

Port binding. In the EmSBoT framework, an agent that needs data from an output port of another agent needs to request an input port with the same name from the framework and bind it to the output. Similarly, an agent that intends to send data to an input port of another agent must request an output port and bind it. In our model, every (input or output) port can be "active" or "passive" according to the action of the agent. For instance, an agent that provides other agents with normalized sensor information will use a "passive" output port to export the IR data because it has no perception of which agents need the IR data. On the other hand, if a navigation agent A_{nav} uses an input port IP_{tp} to receive the target position, then the input port is "passive" in A_{nav}. A high-level agent that relies on A_{nav} obviously needs to employ an output port OP_{tp} to send the position data to IP_{tp}, in which case OP_{tp} is "active." This kind of property can be declared explicitly when the agent requests a port. The EmSBoT framework provides two kinds of core binding methods: static binding and dynamic binding. Both sides are specifically assigned in static binding, whereas in dynamic binding, the "active" side is not aware of where the "passive" side is located. The framework will help solve the lookup or discovery problem. However, this process is completely transparent from the agent's side.

Sending and receiving messages. After port binding is finished, the agent with the output port can send messages to agents with the matching input port. If the input port is in the same node, EmSBoT directly pushes the message into the receiving buffer of the input port; if the input port is across nodes, the message is first staged in a dummy remote input port (RIP) and then a service named EBRouting, which is a dedicated thread employed to handle message routing asynchronously over node boundaries, is activated. Once EBRouting has been notified, it gets messages from RIPs and passes them to the corresponding communication agent. The whole sending process is illustrated in Figure 5. When data are available in the input port, instead of using callback to invoke the user-defined function, EmSBoT only provides the receiving function to retrieve the data from the input port with or without blocking, which is more flexible than the callback method.

In EmSBoT, the data items in the input port are also attached to the identity information, indicating the source of the data. This is very helpful when the input port is used to aggregate data from several agents. Additionally, output port multiplexing is supported in EmSBoT. As an example, take a multi-robot surveillance task in which three worker robots $(Node_{worker})$ are assigned to monitor different areas. The navigation agent (A_{nav}) in each $Node_{worker}$ is used to receive the target location from the input port (IP_{tp}) and then guides the robot to the desired location. A remote controller robot $(Node_{con})$ sends different target locations to the three worker robots via output ports. This idea of output port multiplexing is illustrated in Figure 6. Instead of creating one output port for each IP_{tp}, only one output port is created in A_{con} and is binded to all three IP_{tp}. The output port sends the data group (target locations) in multiplexing mode, and thereby every A_{nav} can receive a different target location. Such a mechanism is very helpful in a memory-limited robot, as it does create excessive output ports.

In order to increase its soft real-time capability, we implemented a priority-based message passing approach in EBRouting inspired by the optimization work of parallel intent broadcasts in real-time Android.[32] When multiple input ports pend on the same output port simultaneously, this approach guarantees that the agent with the higher priority will receive messages earlier than others. In the study by Kalkov et al.,[32] a priority message queue is employed to arrange the messages based on priorities. Instead of prioritizing the messages, our approach sorts the input ports according to the attribute τ_{mt} derived from agents. Analogously, in EBRouting, the remote input ports are sorted according to their priorities. It always pulls data from the RIP that has the highest priority. One problem with priority queue is that the time taken to insert one element varies with the priority of the element and the length of the queue, which makes it less deterministic. Our previous research shows that such a problem can be solved using a priority level bitmap table,[32] together with a highly efficient search algorithm (based on the count leading

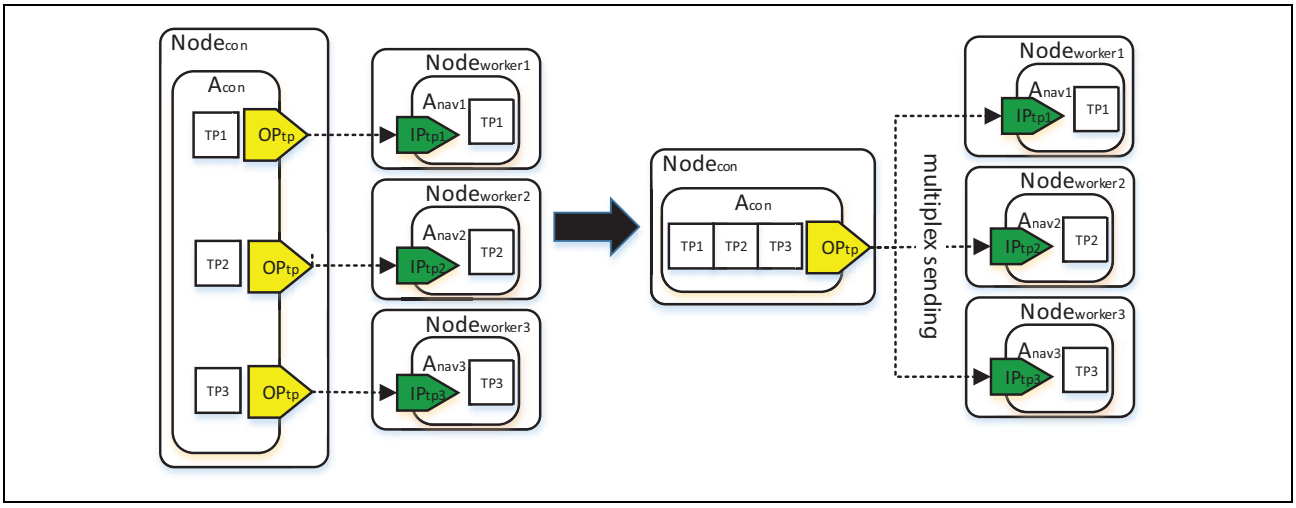

Figure 6. An illustration of output port multiplexing.

zeroes (CLZ) instruction or a bit pattern lookup table). We also use such a method in EBRouting, making the whole process quick and stable.

Besides, as `EBRouting` is implemented as a dedicated thread, its priority should be deliberately assigned in the platform with RTOS inside. The running of `EBRouting` should avoid destroying the real-time capability of the whole system. One solution could be assigning a static moderate priority. However, we argue that a high priority or a low priority is not appropriate. When it is given a low priority, the messages sent to agents with high priority could be delayed. On the other hand, if we give it a high priority, the real-time requirement of agents with high priority cannot be guaranteed if agents with lower priority receive a lot of messages, as `EBRouting` will have exclusive use of the CPU time until all the messages have been handled. Therefore, we take the strategy of tuning its priority dynamically according to the priorities of the RIPs. When the system starts up, `EBRouting` initializes its priority to the lowest one. We adjust its priority at two moments: (1) when `EBRouting` is notified that an RIP has staged messages and (2) between when `EBRouting` finishes handling the current RIP and when it pulls messages from the next one. In the first case, `EBRouting` inherits the priority from the RIP if its priority is lower than that of the RIP. This principle makes sure that the RIP with the highest priority receives messages as soon as possible. In the second case, `EBRouting` directly lowers its priority to that of the next RIP. This principle guarantees that `EBRouting` will not paralyze the real-time requirement of agents with high priority. Finally, if there are no messages to send, `EBRouting` goes back to the initial low priority state.

When managing the communication between heterogeneous platforms, dealing with the data format and type marshalling are inevitable. Frameworks such as R2P[2] and μORB[23] do not consider this problem because they are deployed in homogeneous nodes or in a single node only. As stated in the literature,[21] there are two main kinds of approaches: text-type format and binary data format. Text-type format can be XML, CSV, and JSON, which are more interpretable and language- and platform-independent. However, considering the performance, energy consumption, and communication network bandwidth limit in swarm and modular robots, it is better to use binary data format, which saves encoding time and network bandwidth. As EmSBoT primarily focuses on the resource-constrained platforms, the use of specific binary data format is adopted. It is preferred and advisable to specify the content of messages using standard C data structures. In its current version, EmSBoT directly transmits the content of memory that the message (data structure) occupies without data marshalling when the message is transferred between homogeneous nodes, which promotes the efficiency of the framework. Data marshalling happens only when the message is transferred between heterologous nodes. It is intended that future version EmSBoT will integrate the type specification and marshalling mechanisms of LCM.[33]

Some frameworks provide synchronous communication mechanisms, such as ROS, MIRA, and FINROC. They all implement the remote procedure call service, a typically synchronous communication mechanism. When a client agent sends a request to the server agent, it will be blocked until the response from the server agent is returned. EmSBoT does not support synchronous communication, as it mainly copes with microcontroller-based robots and diverse networks. Considering simplicity and efficiency as its design principles, EmSBoT only employs asynchronous communication based on data ports. In practice, if needed, the synchronous communication can be achieved by setting up two communication connections between two agents as shown in Figure 7. The server agent A_s provides service by receiving a request from the input port IP_s and sending the response back via the output port OP_s. The

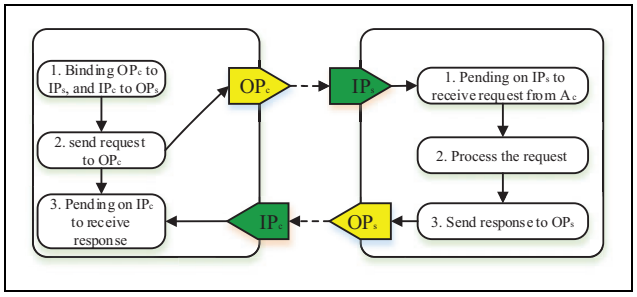

Figure 7. Implementation of synchronous communication using two-way port connections in EmSBoT. EmSBoT: *E*mbedded *m*odular *S*oftware framework for a networked ro*BoT*BoTic system.

client agent A_c consumes A_s's service just by setting up the binding with IP_s and OP_s, and synchronization is realized by pending on IP_c.

Communication agents

In our framework, to facilitate communication between different computing nodes, we creatively abstract the communication layer as individual agents, each of which is associated with one communication channel. A communication agent is in charge of sending (receiving) messages to (from) its corresponding communication channel. An agent is identified as a communication agent by providing three callbacks to the EmSBoT core via the system API

```
EB_err eb_comchannel_register(
    EB_comchannel **channel,
    Sendfunc sendfunc,
    Broadcastfunc broadcastfunc,
    Getnetaddrfunc netaddrfunc
),
```

where Sendfunc and `Broadcastfunc` perform unicasting and broadcasting, respectively; Getnetaddrfunc returns the communication channel's network address, which is actually regarded as *NodeID* because of its uniqueness in the whole system. Inside `eb_comchannel_register`, the framework also attaches a thread-safe hook function `eb_handle_data` to the communication agent. When one message is available in the communication agent, it is passed to the EmSBoT core by calling `eb_handle_data`.

To keep its flexibility, the framework does not strictly specify how the three callbacks are implemented in a communication agent. For instance, we have already implemented a UDP/IP communication agent, in which a public UDP port is shared among nodes and is used to broadcast messages, while each node holds an exclusive UDP port to send and receive unicasting messages. In reality, when performing port binding, the framework will first broadcast a port metadata discovery message to all other nodes and then the other nodes will reply with the matching results by unicasting. However, the port binding process can be completely different if we prefer to use a centralized hub that manages all the port metadata. In this scenario, we only need to implement the `Broadcastfunc` function just by sending messages to the hub server without enforcing real broadcasting.

Fault-tolerant support

The prevalence of NRS is due to its advantages, such as robustness and fault tolerance, over a single robot. A team of robots with different functionalities could cooperate to carry out the tasks assigned in parallel. In a high level, an abnormal robot could be replaced by another normal one. In a low level, when a functional software unit of a robot does not work correctly, the robot could benefit from utilizing the same software unit located in another robot. However, it is not straightforward to achieve these objectives autonomously at runtime, especially in a distributed environment. There are at least two major problems to consider: (1) how to detect the failure of an agent and (2) how to recover from the failure in a tractable way. EmSBoT promotes the separation of fault handling from the normal logic of an agent.

Fault detection. In EmSBoT, the faults for an agent either originate from the agent itself through self-checking or from other coupled agents through fault propagation. The framework now supports the detection of two types of faults: timing fault and functional fault. A timing fault is identified by recording the start and end time of *Exec* of the agent (see Figure 3). If the agent has timing specification $(T > 0)$ and $T_e - T_s > T$, it tells that the system cannot meet the agent's timing requirement. A functional fault is indicated by the return value of *Exec*. For the agent designer, *Exec* can return false if the agent does not work properly. For instance, consider a speed control agent, the *Exec* of which is to make the robot maintain a desired speed. After a period of time, the agent is unable to keep the speed due to low power. It is desirable that the *Exec* function returns a false value, making the agent enter into the fault mode. Meanwhile, an agent can receive fault messages from other agents that connect with it via port binding. A fault message means that the sender agent is out of order, and then the agent that receives it should reconfigure its port binding. As shown in Figure 3, each time *Exec* is finished, the framework will check the faults mentioned earlier. In this way, the fault detection is separated from the normal logic of an agent. If faults are detected and the agent itself enables fault handling, the agent enters into the fault recovery procedure.

Fault recovery. The fault recovery procedure is divided into two steps. First, if the *Faulthandling* function is provided by the agent designer, the fault is passed to it. The prototype of *Faulthandling* is defined as follows:

```
boolean Faulthandling(
uint32 nodeid,
uint16 agentid,
FAULT_TYPE type
),
```

where the first two parameters indicate the source of the fault, and *type* is a structure and denotes the fault type and value. In *Faulthandling*, the agent designer can implement different fault recovery strategies according to its context and different fault types. For instance, if the timing fault happens, the agent can go back to normal by requesting for more computation resource from the system (through increasing its priority) if this mechanism is supported by the operating system. Furthermore, when a fault message is received, the agent can identify the faulty agent by *nodeid* and *agentid* and then change the port binding with the faulty agent.

Finally, if *Faulthandling* is incapable of handling the fault, the fault will be further processed by the framework. At this step, we can conclude that the agent is impossible to recovery from the fault, so this agent is marked as faulty, and the framework will propagate a specific fault message to other agents that are connected with the faulty agent. Through fault propagation and dynamic port binding, EmSBoT supports distributed fault tolerance in a novel and tractable way.

Implementation and experimental evaluation

EmSBoT is initially proposed to achieve transparent communication between devices with constrained resources, so we implement it primarily on top of the μC/OS-III RTOS on both the ARM Cortex-A8 and Cortex-M4 processors. As the goal of EmSBoT is to support heterogeneous hardware platforms and operating systems, it does not use any standard libraries that depend on specific operating systems. EmSBoT only employs a minimal set of OS features including threads (tasks), semaphores, mutexes, and timing, all of which are available in RTOSs and general-purpose operating systems. By introducing the OSAL, EmSBoT also supports QNX, Windows, and Linux platform. Our practical experience shows that porting EmSBoT to other platforms can be done effortlessly just by rewriting around 200 lines of code for the OS abstraction layer. Considering performance and programming of embedded devices, the framework is written entirely in C language and compiled using the GNU toolchain. MinGW (Minimalist GNU for Windows) is used in a Windows environment, while for ARM Cortex processors, the gcc-arm-embedded toolchain is used.

Memory footprint

One of the merits of the EmSBoT framework is its small memory footprint, which makes it feasible and suitable for deployment on MCUs. Table 1 shows the memory usage when running the EmSBoT framework on the STM32F4DISCOVERY board, which has 1 MB of flash memory for code and 192 KB of SRAM for data. The source code contains the μC/OS-III RTOS, necessary BSP files, the framework, and agents. Each agent has one input port and one output port. All the data in the table are statically allocated through the memory pool. When the program is executing, it will never acquire memory from the heap, which avoids deterioration of determinism using dynamic memory allocation. However, the framework also supports dynamic memory heap through configuration when compiling. It is favorable to use dynamic memory when the size of the program is hard to estimate or when the behavior of the program is unpredictable. The firmware is compiled by the GNU toolchain for ARM processors with Optimization Flag 'O3' (Optimize most) and with 'gc-sections' enabled (remove unused sections). From the table, EmSBoT will cost about 13 KB of flash memory and 5 KB of data memory when there are two agents in the system. When one agent is added, it will additionally cost about 1 KB of data memory.

Table 1. Memory usage of the firmware when deploying the EmSBoT framework on the STM32F4DISCOVERY board with μC/OS-III RTOS.

Section	Without EmSBoT (bytes)	With EmSBoT and two agents (bytes)	With EmSBoT and three agents (bytes)
.text	11,732	24,932	24,932
.data	16	16	16
.bss	5308	10,316	11,268

EmSBoT: *Em*bedded *m*odular *S*oftware framework for a networked ro*BoT*ic system

In conclusion, it has been shown that EmSBoT is very suitable for most microcontroller-based robots. When provided with a well-designed communication agent, EmSBoT is a desirable candidate for swarm robotic applications.

Performance evaluation

We have carried out three primary tests to check the framework's performance and real-time support. The first one measures the message delivery latency between agents in node domain. In this test, one agent (AG_S) with the output port is assigned to send 4 bytes of data at 500 Hz to the other agents ($AG\_R_{0\ldots n}$), all of which pend on the corresponding input port binding with the output port. In order to verify the framework's real-time support, these agents are classified into three categories with different priorities: AG_S, AG_R_0, and $AG\_R_{1\ldots n}$. We set the priority of AG_R_0 to be always greater than that of $AG\_R_{1\ldots n}$. In the test, 4096 messages are transmitted every round and then is repeated with different priorities of AG_S and increasing

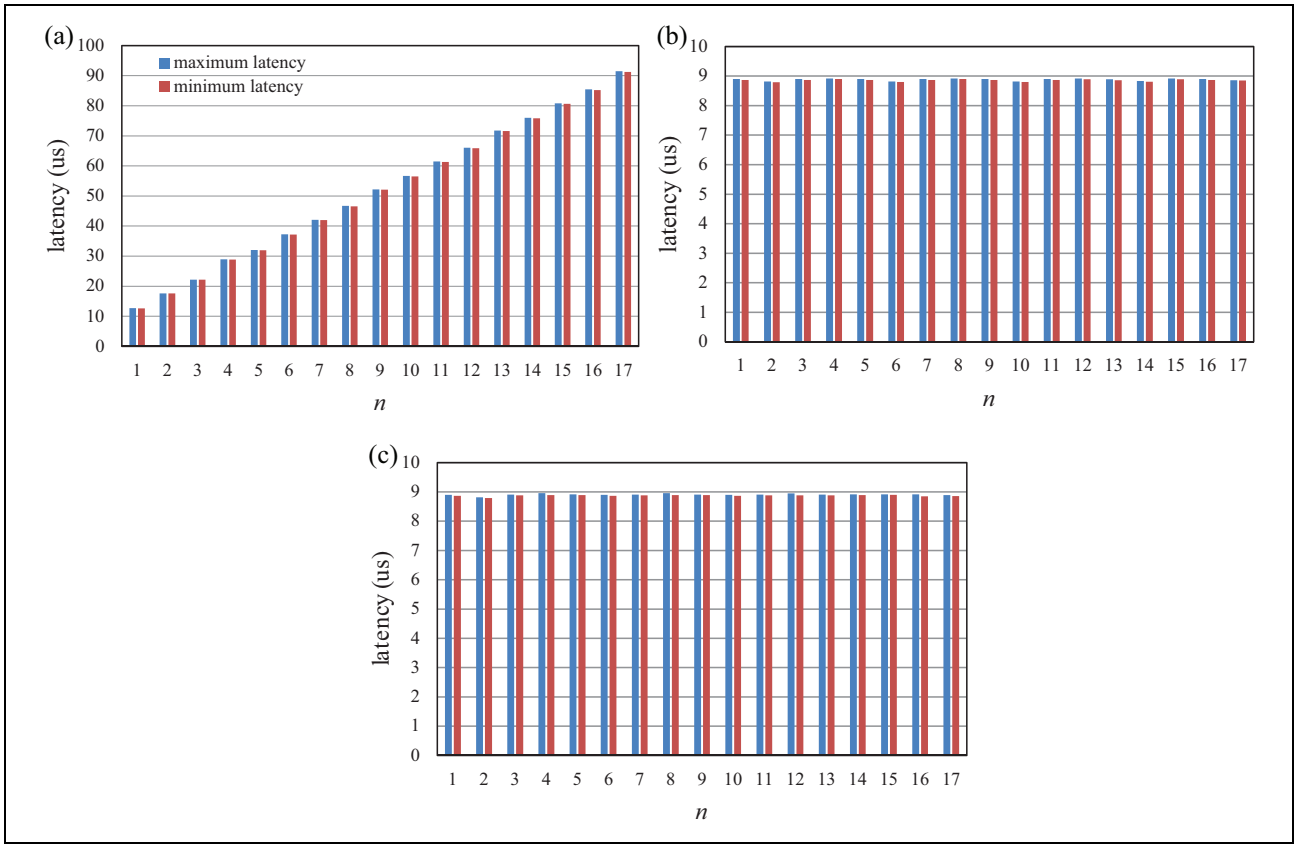

Figure 8. Data delivery latency between *AG_S* and *AG_R*$_0$ with different priorities of *AG_S* and increasing *n*. (a) *AG_S*: high priority; (b) *AG_S*: middle priority; and (c) *AG_S*: low priority.

n (from 0 to 16). In such a way, we can identify whether low priority agents have influence on the high priority agent. The time is measured from the moment *AG_S* sends the message to the moment *AG_R*$_0$ consumes the message received. The hardware and OS used are the same as in the previous section. The CPU is running at 144 MHz. The embedded SRAM has zero wait state, and the flash used for storing code and read-only data has five wait states (six CPU cycles). Figure 8 reports the results that we obtained.

Figure 8(a) shows that the latency increases linearly with *n* if *AG_S* has higher priority than all *AG_R*$_{0...n}$. This is in conformity with reality, because *AG_R*$_0$ can only consume the message after *AG_S* sends the message to all other agents. The time added for every *AG_R*$_i$ is nearly 5 µs, including the time consumed by the framework, and the time taken to signal semaphore and activate the agent. When *AG_S* has lower priority than *AG_R*$_0$, the latency is almost constant (8.9 µs; see Figure 8(b) and (c)), with jitter of less than 0.1 µs. EmSBoT distributes the data to the input ports according to their priorities, so *AG_R*$_0$ will always receive the data first and will then be activated to execute. From the test results, we can conclude that EmSBoT has real-time capability. If the priorities of agents are properly assigned, the message delivery latency between agents will be deterministic. It should also be noted that in this test we did not observe any effect of tick interrupts during the message delivery (only the tick interrupt was enabled). Because *AG_S* always transmits data immediately after sleeping for 2 ms (500 Hz), it will never encounter the tick interrupts during message delivery due to its short latency. However, in reality, the effect of interrupts needs to be thoroughly verified for hard real-time applications.

The second test measures the message delivery latency in distributed domain. As it is intractable to measure absolute latencies between two systems, in this test, the round-trip message delivery time is measured between two distributed heterogeneous systems. EmSBoT is first deployed in the Beaglebone Black board with 1 GHz ARM Cortex-A8 processor, 512 MB RAM, and QNX Neutrino 6.5 running on top. On the other side, the processor is Intel Core i5 750@2.67 GHz with four cores, 8 GB of memory, and Ubuntu 14.04 LTS 64 bit running on top. The two systems are connected via a GigE switch, and the UDP/IP communication agent is employed in EmSBoT. On the QNX side, agent *AG_S*$_1$ is assigned to send a 4 byte message to *AG_R*$_1$ on the Ubuntu side. Once *AG_R*$_1$ receives the message, it immediately sends an echo message (4 bytes) to *AG_S*$_1$. The time from the moment *AG_S*$_1$ send a message to the moment it receives the echo is called the round-trip latency. We also call *AG_S*$_1$ and *AG_R*$_1$ as a round-trip pair. In order to check the performance under the competitive environment, we measure the round-trip

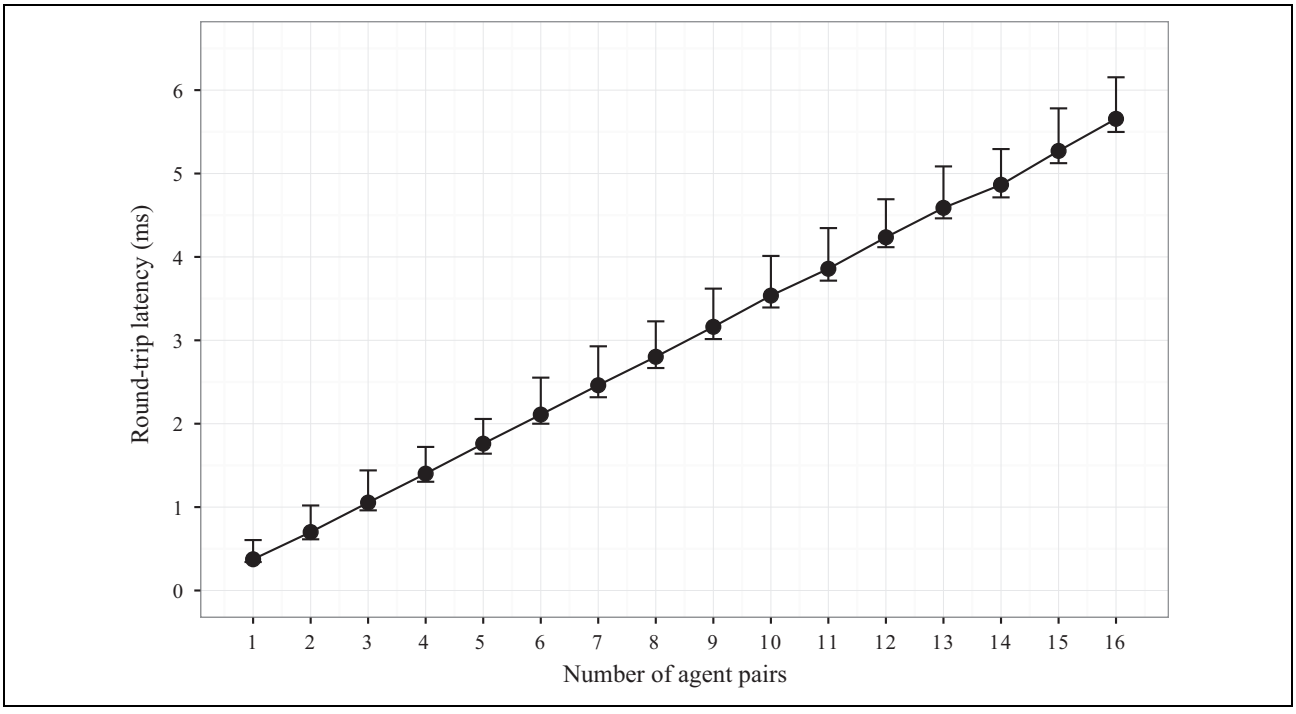

Figure 9. Round-trip delivery latency between AG_S_1 and AG_R_1 with increasing number of round-trip pairs.

latency with increasing number of round-trip pairs. All the round-trips pairs send message simultaneously contending for CPU time and network bandwidth. Figure 9 shows the result. We can see that the round-trip latency between AG_S_1 and AG_R_1 increases linearly as the increasing number of round-trip pairs. This is reasonable because agents on both sides share the same network but send messages simultaneously. When there is only one round-trip pair in the system, EmSBoT can send messages in the 1000 Hz range, while 125 Hz range can be achieved when 16 round-trip pairs are needed. Besides, the result also shows that EmSBoT is deterministic with stable jitters as the increasing number of round-trip pairs. In summary, this test demonstrates that EmSBoT's implementation is highly efficient and deterministic in distributed heterogeneous environment.

The third test is used to check the framework's maximum throughput in node domain under different circumstances. It measures the maximum number of transmitted messages with different n and different priorities of AG_S as the first test does. However, in this test, AG_S only sends messages to $AG\_R_{1...n}$ continuously without sleep. $AG\_R_{1...n}$ have the same priority. Figure 10 reports the benchmark result with the priority of AG_S higher than, equal to, and less than that of $AG\_R_{1...n}$. It is obvious that the throughput is affected by the priority assignment. When AG_S has the highest priority, it sends messages without preemption, so that other agents have no time slot for consuming the messages, and then the system has very high throughput (16,3861 messages transmitted per second when $n = 1$). With this configuration, the system can reach maximum throughput, although it is impractical in real applications. Practically, AG_S should yield to let other agents run to avoid starvation, which makes the throughput much lower than the values given here. The maximum "worst-case" throughput occurs when the priority of AG_S is set much lower than that of the others. In this scenario, AG_S can send the next message only after all other agents have consumed the message. However, we still observed a maximum throughput of 60,316 messages per second when $n = 1$. When all the agents have the same priority, the throughput depends on the scheduling policy of the OS. In our test, all agents with the same priority are scheduled in a round robin fashion with time quanta 10 ms, so the result is slightly different from the scenario in which AG_S has the highest priority.

In conclusion, the performance evaluation shows that EmSBoT preserves and supports the real-time capability of the system and meets the latency and throughput requirements for most real applications.

A case study

We used the EmSBoT framework in a simple leader–follower robotic application, which proved the effectiveness of EmSBoT. Two differential drive wheeled robots (Figure 11, modified QuickBoT) are used in the experiment. Both feature two wheel encoders and are controlled by the Beaglebone Black board. The state information (position and velocity) of the robot is approximately estimated by measuring the distances travelled by each wheel at 1000 Hz using the wheel encoders. The communication between the two robots is achieved via the Digi XBee

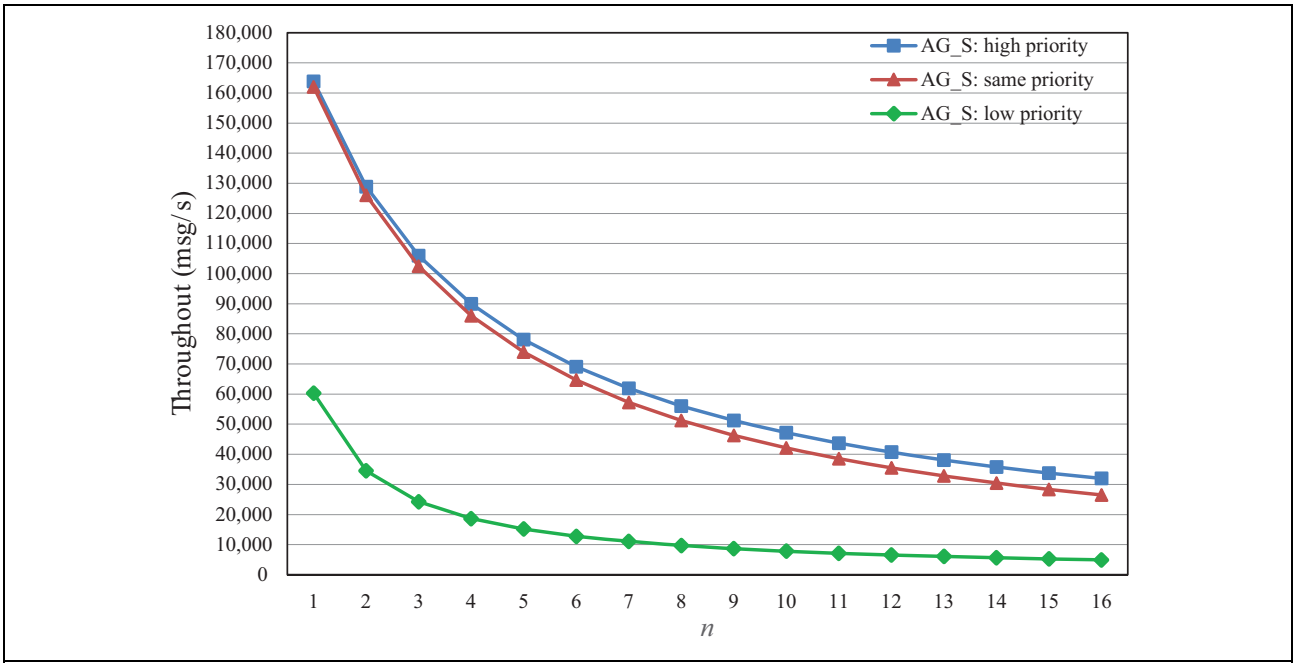

Figure 10. Maximum throughput with respect to different priorities of *AG_S* and increasing *n*.

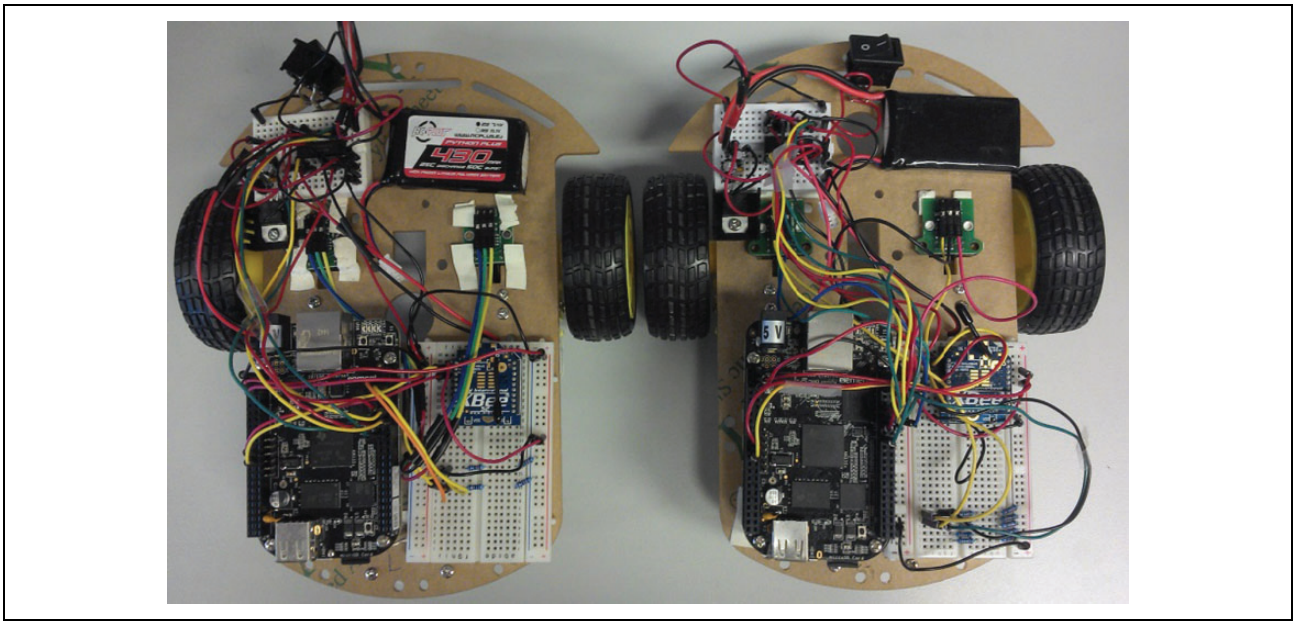

Figure 11. The robots used in the experiment.

ZigBee RF module. The μC/OS-III RTOS is ported to the board.

Each robot (the control board) is viewed as a single node and features three agents (Figure 12). The main functions of these agents are as follows:

- *AG_QBCtr*: Agent QuickBot Controller. This agent controls the velocity of the two wheels using a closed-loop PI controller at 50 Hz. It receives the reference velocity from the input port *IP_rbvel* and provides other agents with the state information through the output port *OP_rbstate*.
- *AG_Leader*: The leader agent keeps a list of goal points. In order to visit these points, it needs to calculate the desired linear and angular velocities according to its current state received from *IP_rbstate*. This process is also implemented via a PID controller at 50 Hz. The velocity command is sent to *AG_QBCtr* through *OP_rbvel*.

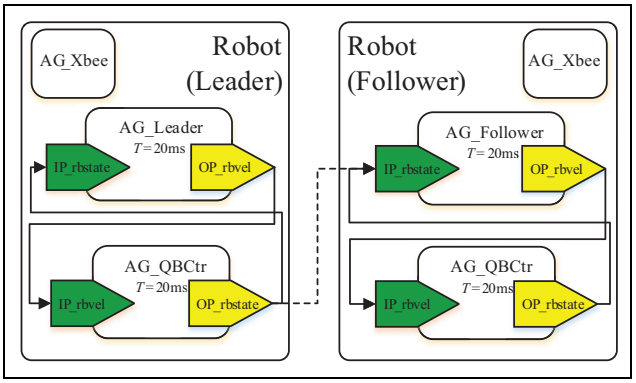

Figure 12. The architecture of the control software for the experiment. Solid lines denote the binding between input and output ports inside nodes, while the dashed line indicates that the binding is across nodes.

- *AG_Follower*: The follower agent takes charge of keeping a desired distance from the leader robot, so it calculates the follower robot's velocity constantly at 50 Hz according to state information from itself and the leader robot. All the binding information between ports is shown in Figure 12.
- *AG_Xbee*: This is the communication agent that employs Xbee to transmit state information between agents across nodes.

Figure 13 reports the experimental results that we obtained. The data are collected from the follower robot, which has the state information of both robots. The leader robot remains at a fixed linear velocity while visiting all the points. When it arrives at one point, the goal point is switched to the next one accordingly. Figure 13(b) shows

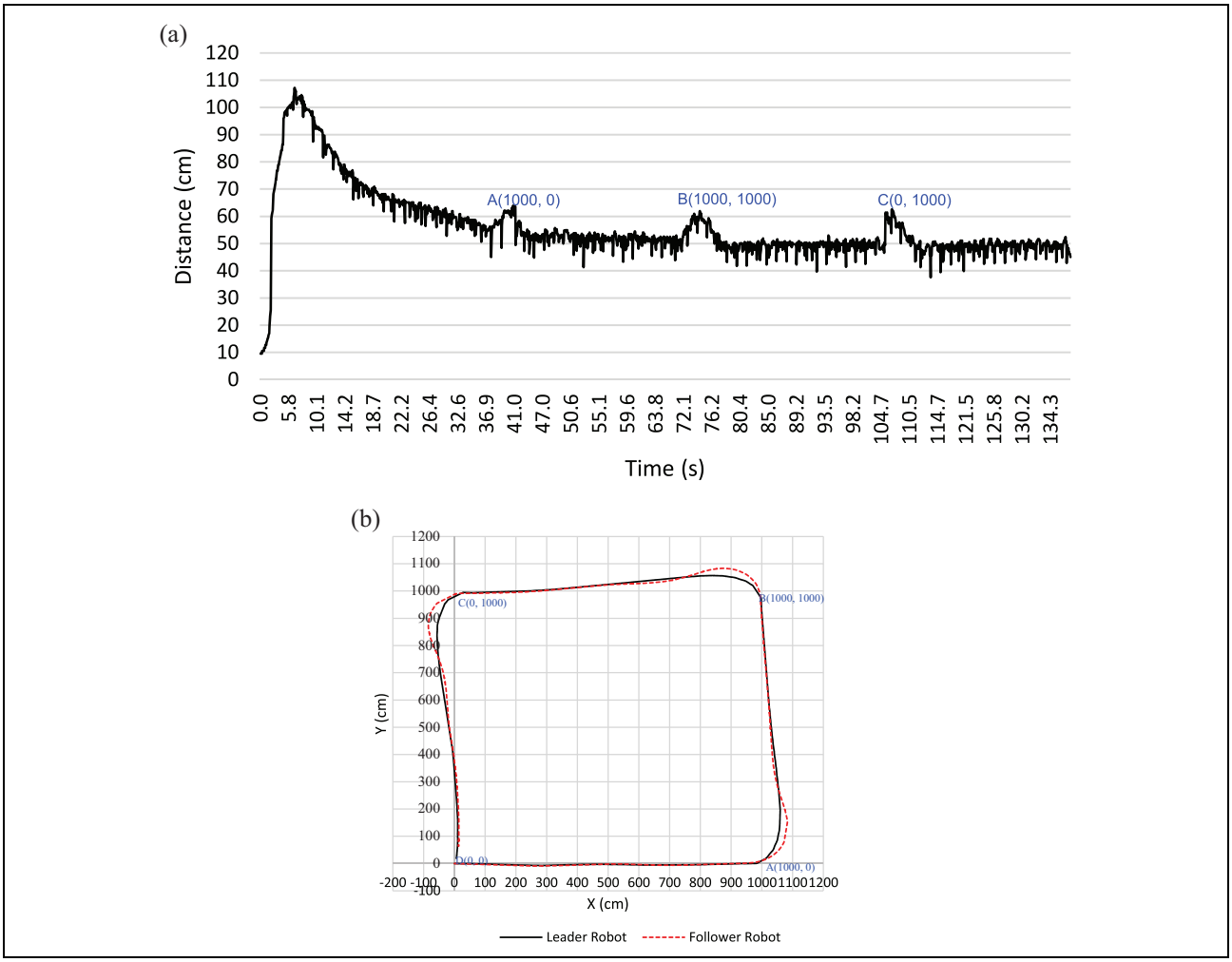

Figure 13. Distance data and trajectory data of the experiment. The leader robot starts from (0, 0), and is assigned to visit four points in order: A(1000, 0), B(1000, 1000), C(0, 1000), and D(0, 0). The follower is required to keep 50 cm distance and 0° relative bearing angle with respect to the leader robot, and thus the two robots remain in (50 cm, 0°) formation.[34]

the trajectories of both robots. The follower robot can follow the path of the leader perfectly when they remain in (50 cm, 0°)-formation,[34] except near goal points. This is acceptable because *AG_Follower* only utilizes the leader's position information, without other states such as its angular and angular velocity. Figure 13(a) reports the distance between the two robots. In the initial stage, the distance increases until the follower adjusts its speed to a high value in order to catch up with the leader. Afterward, the distance gradually becomes close to 50 cm, and, ideally, remains the same.

One major problem in the experiment is the indeterminism of message latency of the Xbee communication channel, which meant that *AG_Follower* could receive obsolete state data of the leader robot. Sometimes, we observed a delay of one second. We use two strategies to solve this problem. First, every message in EmSBoT includes a timestamp to indicate when the message is sent from the output port. The time stamp can be used to judge whether the message is received in time. If the message is received within a reasonable time window, it will be kept and used by *AG_Follower*; otherwise, it will be discarded. Second, if the state information received is obsolete and discarded, or if no state information is received in this round, the previous state information of the leader robot is employed to predict its current state. Using such two strategies, we can calculate the velocity of the follower robot as accurately as possible.

Conclusion and future work

The inherent complexity of developing (distributed) robotic applications promotes the popularity of RSFs, most of which adopt the component-based approach with different communication patterns. However, few RSFs consider embedded devices with constrained resources. EmSBoT, presented in this article, is an embedded modular component-based robotic software framework targeting heterogeneous platforms. It is deliberately built upon lightweight RTOSs, making it suitable for resource-constrained devices such as microcontroller-based robots. Furthermore, the OS abstraction layer extends it to other operating systems without too much effort. It uses the port-based communication mechanism as the only way to exchange messages between agents, which makes the agents loosely coupled, and endows the system with fault-tolerant capability by binding and rebinding ports dynamically at runtime. By isolating the communication channels as separate agents, we provide uniform message-passing APIs for agents, making the communication transparent over node boundaries. It also employs a priority-based message passing approach, providing the application with real-time capability. We also introduce a distributed fault-handling mechanism in EmSBoT. The agent can either have its own fault-handling procedure or propagate a fault message to other agents that bind to it.

The EmSBoT framework is fully implemented in C language. So far, it has already been ported to μC/OS-III, QNX, Windows, and Linux operating systems. The footprint (together with μC/OS-III) measured in the STM32F4Discovery board shows that EmSBoT is very suitable for microcontroller-based robots. We also conducted an exhaustive performance evaluation of EmSBoT inside one node, proving that EmSBoT is capable of maintaining real-time performance and providing high throughput. A leader–follower distributed application, which employs the XBee ZigBee mesh network to pass messages, was conducted to verify its effectiveness. In the future, we intend to develop other communication agents for the framework, such as agents over CAN and TCP/IP. Further experiments need to be conducted to evaluate its distributed performance, to compare it with other frameworks, and to demonstrate its fault-tolerant capability. At present, EmSBoT is in its initial version, so we do not consider the problem of message marshalling, which is a necessity when messages are exchanged between heterogeneous platforms. We will add this feature in the next version.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work is partially supported by the scholarship from China Scholarship Council (CSC) under grant no. 201206110039.

References

1. Wilcox BH, Litwin T, Biesiadecki J, et al. Athlete: a cargo handling and manipulation robot for the moon. *J Field Robot* 2007; 24(5): 421–434.
2. Bonarini A, Matteucci M, Migliavacca M, et al. R2P: an open source hardware and software modular approach to robot prototyping. *Robot Auton Syst* 2014; 62(7): 1073–1084.
3. Mondada F, Pettinaro GC, Guignard A, et al. Swarm-bot: a new distributed robotic concept. *Auton Robots* 2004; 17(2–3): 193–221.
4. Sanfeliu A, Hagita N and Saffiotti A. Network robot systems. *Robot Auton Syst* 2008; 56(10): 793–797.
5. Collett TH, MacDonald BA and Gerkey BP. Player 2.0: toward a practical robot programming framework. In: *Proc. Australasian Conf. Robotics and Automation (ACRA 05)*, Sydney, Australia, 5–7 December 2005, pp. 12–19. ARAA.
6. Quigley M, Conley K, Gerkey B, et al. ROS: an open-source robot operating system. In: *Proc. Open-Source Software Workshop Int. Conf. Robotics and Automation*, Kobe, Japan, 12–17 May 2009, pp. 1–6.
7. Magnenat S, Rétornaz P, Bonani M, et al. ASEBA: a modular architecture for event-based control of complex robots. *IEEE/ASME Trans Mechatron* 2011; 16(2): 321–329.

8. Hong S, Lee J, Eom H, et al. The robot software communications architecture (RSCA): embedded middleware for networked service robots. In: *Proceedings of the 20th international conference on parallel and distributed processing*, Rhodes Island, Greece, 25–29 April 2006, pp. 168–176. IEEE.
9. Einhorn E, Langner T, Stricker R, et al. Mira-middleware for robotic applications. In: *2012 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, Vilamoura, Portugal, 7–12 October 2012, pp. 2591–2598. IEEE.
10. Bruyninckx H. Open robot control software: the OROCOS project. In: *Proceedings of the 2001 IEEE international conference on robotics and automation (ICRA)*, Seoul, Korea, 21–26 May 2011, Vol. 3, pp. 2523–2528. IEEE.
11. Utz H, Sablatnog S, Enderle S, et al. Miro-middleware for mobile robot applications. *IEEE Trans Robot Automat* 2002; 18(4): 493–497.
12. Calisi D, Censi A, Iocchi L, et al. OpenRDK: a modular framework for robotic software development. In: *2008 IEEE/RSJ international conference on intelligent robots and systems (IROS 2008)*, Nice, France, 22–26 September 2008, pp. 1872–1877. IEEE.
13. Jang C, Lee SI, Jung SW, et al. OPRoS: a new component based robot software platform. *ETRI J* 2010; 32(5): 646–656.
14. Ando N, Suehiro T and Kotoku T. A software platform for component based RT-system development: OpenRTM-Aist. In: *Proceedings of the 1st International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2008)*, Venice, Italy, 3–6 November 2008, pp. 87–98. Springer.
15. Mallet A, Pasteur C, Herrb M, et al. GenoM3: building middleware-independent robotic components. In: *2010 IEEE international conference on robotics and automation (ICRA)*, AK, USA, 3–8 May 2010, pp. 4627–4632. IEEE.
16. Reichardt M, Fohst T and Berns K. On software quality motivated design of a real-time framework for complex robot control systems. In: *Proceedings of the 7th international workshop on software quality and maintainability (SQM)*, Genova, Italy, 5–8 March 2013, pp. 1–9.
17. Dorigo M, Floreano D, Gambardella LM, et al. Swarmanoid: a novel concept for the study of heterogeneous robotic swarms. *IEEE Robot Automat Magaz* 2013; 20(4): 60–71.
18. Yu CH and Nagpal R. A self-adaptive framework for modular robots in a dynamic environment: theory and applications. *Int J Robot Res* 2011; 30(8): 1015–1036.
19. Cui Y, Voyles RM, Lane JT, et al. ReFrESH: a self-adaptation framework to support fault tolerance in field mobile robots. In: *2014 IEEE/RSJ international conference on intelligent robots and systems (IROS 2014)*, pp. 1576–1582. IEEE.
20. Kramer J and Scheutz M. Development environments for autonomous mobile robots: a survey. *Auton Robots* 2007; 22(2): 101–132.
21. Iñigo-Blasco P, Diaz-del Rio F, Romero-Ternero MC, et al. Robotics software frameworks for multi-agent robotic systems development. *Robot Auton Syst* 2012; 60(6): 803–821.
22. Elkady A and Sobh T. Robotics middleware: a comprehensive literature survey and attribute-based bibliography. *Journal of Robotics* 2012; 2012: 15.
23. Meier L, Honegger D and Pollefeys M. PX4: a node-based multithreaded open source robotics framework for deeply embedded platforms. In: *2015 IEEE international conference on robotics and automation (ICRA 2015)*, Seattle, Washington, 26–30 May 2015, pp. 6235–6240. IEEE.
24. Mellinger D, Shomin M, Michael N, et al. Cooperative grasping and transport using multiple quadrotors. In: *Distributed autonomous robotic systems*, 2013, pp. 545–558. Springer.
25. Magnenat S, Rétornaz P, Noris B, et al. Scripting the swarm: event-based control of microcontroller-based robots. In: *Proceedings of the 2008 SIMPAR workshop*, Venice, Italy, 3–6 November 2008, pp. 1–14.
26. Fitch R and Lal R. Experiments with a ZigBee wireless communication system for self-reconfiguring modular robots. In: *2009 IEEE international conference on robotics and automation (ICRA'09)*, Kobe, Japan, 12–17 May 2009, pp. 1947–1952. IEEE.
27. Mamat R, Jawawi A, Dayang N, et al. A component oriented programming for embedded mobile robot software. *Int J Adv Robot Syst* 2007; 4(3): 371–380.
28. Kim J, Yoon H, Kim S, et al. Fault management of robot software components based on OPRoS. In: *14th IEEE international symposium on object/component/service-oriented real-time distributed computing(ISORC 2011)*, CA, USA, 28–31 Mar 2011, pp. 253–260. IEEE.
29. Ahn H, Loh WK and Yeo WY. A framework-based approach for fault-tolerant service robots. *Int J Adv Robot Syst* 2012; 9(200): 1–10.
30. Fayyad-Kazan H, Perneel L and Timmerman M. Linux PREEMPT-RT v2. 6.33 versus v3. 6.6: better or worse for real-time applications? *ACM SIGBED Rev* 2014; 11(1): 26–31.
31. Peng L, Guan F, Perneel L, et al. Behaviour and performance comparison between FreeRTOS and μC/OS-III. *Int J Embedd Syst* 2016; 8(4): 300–312.
32. Kalkov I, Gurghian A and Kowalewski S. Predictable broadcasting of parallel intents in real-time android. In: *Proceedings of the 12th international workshop on Java technologies for real-time and embedded systems, (JTRES 2014)*, NY, USA, 13–14 October 2014, pp. 57–66. ACM.
33. Huang AS, Olson E and Moore DC. LCM: lightweight communications and marshalling. In: *2010 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, Taipei, Taiwan, 18–22 October 2010, pp. 4057–4062. IEEE.
34. Consolini L, Morbidi F, Prattichizzo D, et al. Leader–follower formation control of nonholonomic mobile robots with input constraints. *Automatica* 2008; 44(5): 1343–1349.