A New Mobile Agent-based Middleware System Design for Wireless Sensor Network

Yuechun Wang, Ka Lok Man, Steven Guan, Danny Hughes

Abstract – **Wireless Sensor Network is playing a crucial role in daily life because of its distributed sensing ability in combination with wireless communication techniques and self-organising deployment approaches. To satisfy requirements of easily matching diversified dynamic sensing applications and highly heterogeneous sensor platforms from perspective of logistics, a low budget but high-efficient wireless sensor network middleware is in dire need. To maximise commercial benefits, a WSN middleware is designed in order to increase efficacy and return on investment. This paper presents a design of an inventive mobile agent-based middleware, which could optimally achieve the intensive requirements of a WSN middleware. The middleware proposed in this paper has considered resources limitation issues that are commonly addressed on normal sensor nodes, meanwhile it provides a possibility of platform independence as well as programming language independence. In addition, a mobile agent-based system can reduce network payload and dynamically adapt to environmental changes. Therefore, the middleware presented in this paper has ability of perceiving changes in operating environment and ability to automatically response to these changes.**

Index terms -- Mobile Agent, Code Mobility, Component Infrastructure, Middleware, Wireless Sensor Network

I. Introduction

As one of the large-scale Internet of Things (IoT) system's application, Wireless Sensor Networks (WSN) enable sensing systems moving out of laboratories into real world and having a rapid development with a growing range of application scenarios. With features of self-organisation, flexibility, as well as combination of environmental sensing ability and wireless communication techniques, WSN is widely used in tracking, monitoring, and creature-unfriendly space exploration [1]. However, the complexity of current sensor programming approaches and highly heterogeneous third party platforms are leading to a waste of resources with increasing number of sensing applications appeared. A dynamic WSN middleware [2], which should bridge the gap efficiently between two key stakeholders in the WSN value chain: sensing application developers and sensing platform providers, is therefore urgent demanded.

In this paper, an new WSN middleware is proposed to reduce development overhead for both platform providers and application developers. Being motivated from the perspective of logistics and transportation, we have noticed that contemporary logistics sensor platforms shape roadblocks to application developers in mainly two aspects: one is at hardware level, a wide range of sensing, computing and networking facilities are offered by platforms; another is at software level, sensor nodes run a variety of operating systems and middleware. Besides, current logistics sensor platforms are also highly distributed with third-party sensing infrastructure deployed in geographically distributed trucks, trailers and warehouses that should come together as required to enact a sensing application. Due to the application-specific nature of WSN, sensor nodes are required to have various capabilities to handle multiple applications. It is difficult to store all the potential applications in the local memories of sensor nodes economically. Therefore, an new and creative approach to dynamically deploying of new applications is in demand.

To overcome the barriers, a mobile agent-based middleware for WSN is presented in this paper. The core idea of this middleware system is to optimise the allocation of resources while taking the compatibility of heterogeneous platform into account. The middleware is duplex-friendly to both users and platforms suppliers. For users, the middleware provides an all-purpose network without preinstalled applications. Agents are injected to sensor nodes cooperatively in order to implement functions that are previously achieved by applications. Application-level inductions are provided to users for the development of agents as well as configuration of local components on sensor nodes. For platforms suppliers, this middleware provides heterogeneous platforms assistance as well as runtime components binding and test through a loosely-coupled component infrastructure. Based on this infrastructure, distributed concerns are cleanly separated from component implementation, also the application-level interoperability between heterogeneous WSN platforms is well supported.

The remainder of this paper is organised as follows. Section II introduces the background of our research, which consists of application scenarios of agent-based WSN system and requirement analysis. Section III presents the related work of the research that includes loosely-coupled component infrastructure and feasibility analysis of achieving the designed system. Detailed system design and implementation are proposed in Section IV. Concluding remarks of this paper and research future work are given in Section V.

Yuechun Wang is with Xi'an Jiaotong-Liverpool University, China (email: yuechun.wang@xjtlu.edu.cn)
Ka Lok Man is with Xi'an Jiaotong-Liverpool University, China; and Swinburne University of Technology Sarawak, Malaysia (email: ka.man@xjtlu.edu.cn);
Steven Guan is with Xi'an Jiaotong-Liverpool University, China (email: steven.guan@xjtlu.edu.cn)
Danny Hughes is with IBBT-DistriNet, KU Leuven, Leuven, B-3001, Belgium (email: danny.hughes@cs.kuleuven.be)

II. BACKGROUND

In this section, research background is presented in two segments: (1) two basic application scenarios of agent-based WSN system and (2) requirement analysis of an agent based WSN middleware.

A. *Application scenarios of agent-based WSN system*

A Mobile Agent (MA) [3], which is described as a special software that includes executed codes, can migrate among sensor nodes in a monitoring area to collect target data. This definition indicates that MA is an application-specific software and has high mobility. Its ability of data processing decides that WSN system with MA can significant extend lifespan of terminal resources sensor nodes by reducing the energy consumption of data processing on local sensor nodes. An example of sensor network that involved agents is shown in Figure 1.

According to the periodicity of tasks, area monitoring and goods tracking are two basic application scenarios of agent-based WSN system [4]. In the case that agents are activated and migrated periodically, the agent system is applied to monitor. In case the system is applied to track goods, agents will be activated based on action of target tracking goods which is aperiodic.

When dealing with monitoring-based tasks, mobile agents cooperate with local processor in two approaches to optimise resource usage [5]. One approach is that mobile agents carry only execution codes related to specific required application so that processor deals with only part of raw datum instead of dealing the whole data, which will dramatically reduce amount of data to be transmitted. Once agents migrate among sensor nodes, which are geographically close by, it will be in high probability that data collected by these nodes are repeated. Thus, another approach is that the local data redundancy can be eliminated through data aggregation codes brought by mobile agents. When dealing with tracking-based tasks, such as fire track [6], agent activating and migrating are always triggered aperiodic. Agilla [3] and its latest version [7] [8] are one of the classic mobile agent-based middleware that are designed for tracking issues.

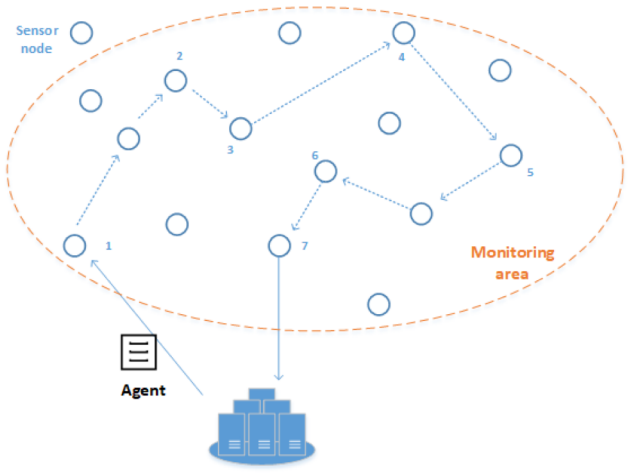

Figure 1. An example of WSN with agent system applied in. The little circles denote sensor nodes while the large ellipse in dash line denotes monitoring area. Numbered sensor nodes denote agent migration order and arrows denote one hop of agents among sensor nodes respectively.

B. *Requirements analysis*

In an agent-based system, applications are divided into functional separated pieces to facilitate injection and distribution by MA through the network. To clarify the requirements of designing an agent-based middleware, functionalities of principal agents and agent execution platform should be clearly analysed.

An agent is described as a special software that includes unique ID, states, execution codes, as well as datum. It owns executive resources such as stack, heap, register, and counter. Code execution will be supported by accessing virtual machine, meanwhile data can be stored by accessing memory on local nodes.

A backup platform in an agent system is the operation environment of agents. The platform provides a public memory for agents execution; at the same time, it can allocate and manage those memories. Besides, functions of the virtual machine are achieved on platforms, which include executing variety of instructions and managing executive resources at runtime. Cooperation among agents is supported by platform as well. On a single sensor node, multiple agents can be executed concurrently and exchange data through public memories that provided by the platform; Agents can be migrated to remote sensor nodes that have the same platform.

The management of agents is therefore a necessary function that should be achieved by an agent-based middleware, which should cover but not limit to agent clone, agent execute, and agent kill. Meanwhile, agent message packaging, transmitting, receiving, and debagging are implemented through calling the message interface on underlying OS.

Apart from the basic functions introduced above, a WSN middleware should also provide network management. For instance, topology management and neighbor list management. The agents should have ability to acquire network information as well.

For users of this system, the middleware should provide the instruction set for agent control and network management.

The dynamical deployment of agent applications can be implemented by adding an agent generating module and a migration message transceiver [9].

III. RELATED WORK

Refer to our previous work [10], this section presents an overview of footstone of our research: component infrastructure as well as the feasibility analysis of our approach.

A. Loosely-coupled component infrastructure

The Loosely-coupled Component infrastructure (LooCI) as presented in [11] is a state-of-the-art middleware system that bridges the gap between distributed WSN applications and diverse platforms. As described in its name, LooCI is composed of a reconfigurable component model, a hierarchical system, and a distributed event bus. These features denote that LooCI can support a separation of distributed applications from component implementation. Operating system that could support LooCI currently includes OSGi, Contiki, Squawk, and Android.

Several core definitions in LooCI are listed as follows.

Events. Events are the unit of networked communication and flow between components.

Components. Components are the unit of execution and run in the component runtime environment, which do not interest in where the source and destination of events are.

Codebases. Codebases describe functionality of components, which indicates that each codebase can spawn multiple components.

Event bus. Event bus is an entirely decentralized publish-subscribe communication medium, which can be seen as a distributed implementation of the mediator pattern.

Wires. Wires can be seen as connections between two components.

B. Feasibility analysis

As a component infrastructure and event driven middleware system, LooCI is a good option as the infrastructure to achieve MA. The analysis of LooCI operating principle connected with functional requirements of an agent-based middleware is as follows.

Three core concepts of LooCI are codebase and component, event, and wires. Codebase is an executable code file with a node unique ID that can be deployed to a node while component is a runtime unit which executes the functionality of codebase. In other words, codebases provide executable code and components are the instantiation of codebases, which means all the manager modules (will be presented in coming sections) can be achieved by declaring new codebases and components in LooCI. Components communicate by exchanging events through two interfaces – provided interface, which is for publishing; and required interface, which is for subscribing. These two interfaces can achieve the message exchange among four modules on supporting platform layer: control messages, status of agent, neighbour list, etc. An event is the basic unit for network communication. Events are delivered between sensor nodes, which are similar to the role of mobile agents. Every event contains an event ID and payload, which indicate the agent ID as well as storage space for both data and code on agents that could be achieved. Wires are the medium for component communication. Events that have been delivered among diverse components on local or remote sensor nodes which are transmitted through wires.

Table 1. LooCI-based agent middleware and typical agent-based middleware comparison

LooCI-based agent middleware	Typical agent middleware
Each codebase can spawn multiple components	Agents can be **spawned** on node
Event manager stores the information of event source and destination in the form of wirings	Have a **neighbor list** on node
Standard predefined events should work together regardless of the platform and location. Components which use these standard events should adhere to the format and meaning of the events	-
LooCI have local wires, outgoing wires and incoming wires. An event matches a wire, if all the source attributes the match.	The **agent** needs to provide a template that **matches** the **tuple** to extract a tuple from a tuple space.
	Tuple space provide some operation commands
Application software is realised in the form of 'compositions' of reusable components.	An Agilla application consists of **numerous autonomous agents**, possibly of different types, scattered throughout a network.
LooCI allows individual components to be developed on motes at runtime	Agilla allows **agents** to be injected on nodes at runtime.
Components are managed by a reconfiguration manager	Agents are managed by an **agent manager**
Components are uniquely identified by a <node address, component ID> tuple	A **tuple** is an ordered set of fields where each field has a type and value
Wildcards represent no-specific values for the arguments used during reconfiguration and inspection of a LooCI runtime. Their values will never refer to a single, specific instance of their respective argument type.	Templates are unique in that their fields may contains wildcards that match by type
Components, applications, and all the other participants interact through events.	**Agents** migrate carrying their code and state, but no tuple spaces.
In LooCI, using introspection one can discover which components are present on a node along with their interfaces, current state and bindings.	-

To clearly show the possibility of implementing an agent-based middleware based on the infrastructure we proposed, a comparison between LooCI-based agent middleware and typical agent middleware is illustrated in Table 1.

In the comparison, several core definitions in a typical agent-based middleware are needed to be specified. Two core components are agent and tuple space, which have definition as follows.

Agent. In classic agent-based system, agents are actually belong to the application layer. 'An Agilla application consists of numerous autonomous agents, possibly of different types, scattered throughout a network.' [3] This denotes that one application creates a number of agents with interoperated functions to achieve the overall purposes.

Tuple space. The tuple space is local and shared by the agents residing on the node. Special instructions allow agents to remotely access another node's tuple space.

Left column in Table 1 lists the core features of LooCI that are related to MA middleware system. Right column lists the features of a classic agent-based middleware corresponding to LooCI's features on the left. The grids in Table 1 filled by a hyphen mark denote that there are no specific corresponding features.

The comparison shows not only the possibility of implementing MA on LooCI, but also the challenge to achieve transplantation in practical since LooCI is event-based which has gap to a fully functional middleware to support agent system, for instance the lack of specific tuple spaces.

In simple words, core modules of LooCI can satisfy the functional needs of the MA middleware. The challenge is how to declare the manager modules and set suitable properties for the components.

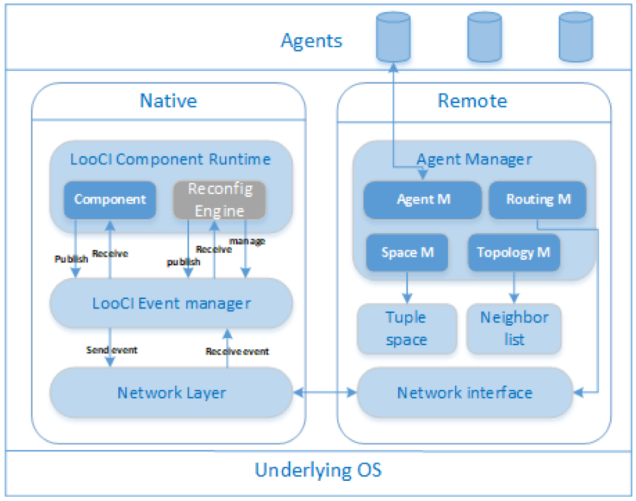

Figure 2. Overview architecture design of the middleware. Modules in Native block serve local component reconfiguration. Modules in Remote block serve dynamic deployment of applications.

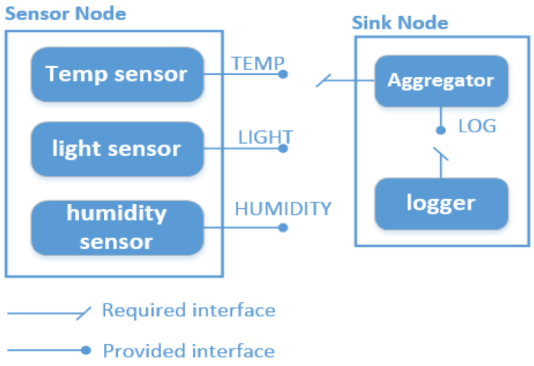

Figure 3. Wires of components before adding agent. Before agent system being added into the sensor nodes, each of the components transmit data to sink node directly. Suppose there are three types of sensors on local node, it demands three channels for sensing data transmission.

IV. System Design and Implementation

In this section, we present our approach in five sub-sections: we show the overview architecture of our design followed with detailed discussion of specific modules such as kernel managers, agent transceiver, and agent architecture. Example codes and implementation are presented at the end of this section.

A. Overview architecture

Figure 2 shows the basic elements in the designed middleware. Aiming to be duplex-friendly to both users and platforms suppliers, the middleware separate core elements into two purposes: one for operations that relates to agents, the other for operations that relates to local components.

The proposed middleware incorporates and extends LooCI, which is briefly introduced in section III.A. LooCI is adapted extensively for the support of native module in our middleware. Individual local components are allowed to deploy on nodes at runtime. As shown in Figure 2, components are managed by

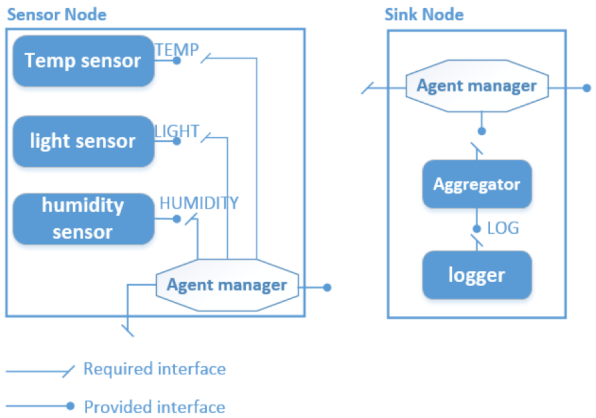

Figure 4. Wires of components after adding agent. After adding agent system into local node, agent manager will collect sensing data based on application requirement or even aggregate data directly on local nodes before data transmission.

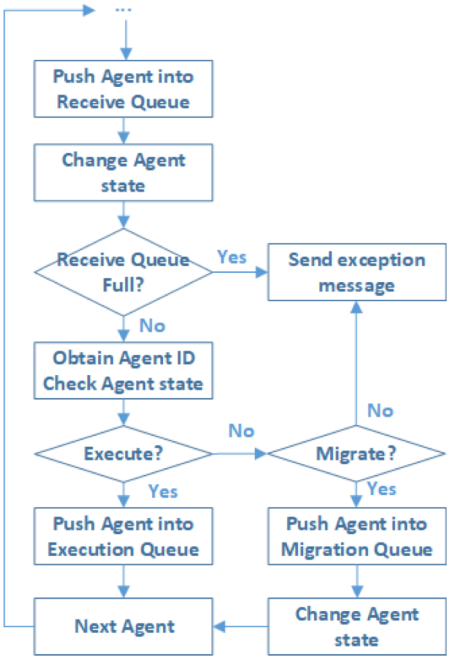

Figure 5. Flowchart of an agent's lifecycle in Agent manager

reconfiguration manager and exchange events through LooCI event manager. Related working mechanism can be found in [11]. Modules in Remote block extend our previous work in [10]. Apart from the management of agents and related components, tuple space and neighbour list are considered in the overall architecture. Wires of components before and after inserting agent system into the nodes are shown in Figure 3 and Figure 4 separately. It can be observed that after adding agent system into local node, agent manager will collect sensing data based on application requirement or even aggregate data directly on local nodes before data transmission. Bandwidth requirement will therefore be reduced.

Functionality of core modules will be presented in the following sections.

B. *Kernel managers*

Block diagrams of four managers in Figure 2 refer to our previous research outcomes. In this section, only core part of agent manager's block diagram is presented for discussion. For details of the other managers, we recommend reference [10].

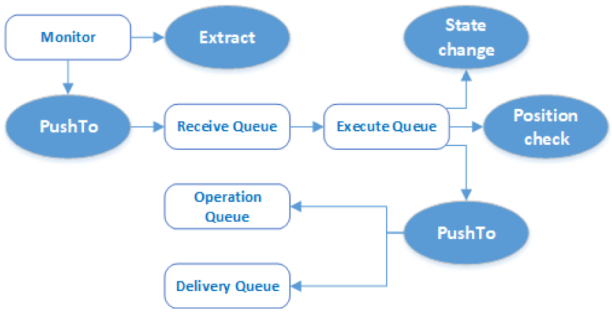

Figure 6. Basic components and core methods of the agent manager

In the lifecycle of an agent in agent manager as shown in Figure 5, three fundamental functions of agent manager are proposed: a) switch agent state; b) allocate resources (frame in queues and memory); and c) execute agent code.

Figure 6 illustrates the basic components and related methods of agent manager. According to the flowchart of agent manager, two basic components that are needed to be implemented, which include a monitor and three functional different queues.

Method *extract()* is one of the core methods of monitor, which is used to extract the header, payload, purpose, etc. of the events or messages that are received.

pushTo() is another method of monitor, which is used to push agent body to a suitable queue. This method should be the same for component Monitor and component Queue.

statusChange() is used to change the status of agents based on their purpose.

positionCheck() is used to check whether there are still empty cell in queues.

C. *Agent transceiver*

The mechanism of migration and receiving of agents is referred as agent transceiver in our approach. A simple migration process is separated into three steps: a) transmitter judges whether destination of current agent is local node; b) if it is not, transmitter sends it to neighbours through network; c) neighbour nodes will deliver the agent through one-hop routing mechanism until it arrives target node.

During the migration, transmitter module on the local sensor node coordinates four sub-modules to achieve the migration function as shown in Figure 7. Once migration process is activated, coordinator will setup time-up control for delivered parameters and then wait for response from target node. Transmission of next parameter will be activated by a correctly receiving expected response within the allotted time, or coordinator will require transmitting failed parameter again. It

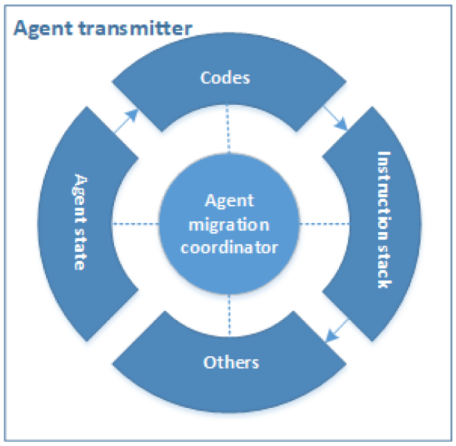

Figure 7. Agent transmitter. Migration modules for agent state, execution codes, instruction stack and others are coordinated together for agent migration. Agent receiver has similar structure to the transmitter.

is the same for receiver mechanism. Once parameter is received by target node, it will reply a specific message to source node and then setup time-out control. Memory is re-allocated by agent manager modules in case the time is overdue and no further parameters will arrive.

D. Other modules

As proposed in Section II.B, a feasible architecture of agents will influence efficiency of middleware apart from kernel modules in the supporting platform. Table 2 illustrates an agent packet, which consists of unique agent ID, initial code, target node, node list, next hop, state, register, instructions, execution code, as well as data. The upper five elements in Table 2 are proposed to identify the migration route of the agent, while bottom five elements are used for agent execution. Necessity of each element in Table 2 is presented as follows.

ID. the unique ID identifies which agent is under operation. For both multiple-agent tasks and single-agent tasks, a unique ID is useful for synchronisation and agent identification.

Initial node. the initial node identifies where the agent comes from. This actually makes it possible to withdraw agents that sent by sink node.

Target node. the nodes that will execute agent codes are set as target nodes. Once an agent arrives on the target node, it switches to next target node on node list.

Node list. all the target nodes on agent's route are recorded on the node list. It will be given by sink node for monitoring purpose.

Next hop. in case that target nodes are not close by (we refer the nodes who can be arrived by agents within one hop as 'close by'), next hop of current agent is necessary for continuous migration.

State. agent state is used to identify purpose of current agent: to execute on local node or is needed to be delivered to other nodes.

Register. register is used for addressing of next instruction and recording execution status.

Instructions. instructions are commands to control the activity of agents on the node.

Execution code. the function of applications is achieved through these execution code carried by mobile agents.

Data. after filtering and aggregation of data collected by local sensors, agent will carry selected datum back to sink node.

Adhere to workflow of agent transceiver and managers as proposed in Section IV.B and IV.C, the elements listed in Table 2 have already stored in this agent after inserting an agent into sink node by the developers. Each module that is accessed by agents will check and store agent ID. State of agent is checked and changed by agent manager. Followed the node list which is predetermined, agents on node will have two operations: (a) for the agents which have already arrived at target node and wait in queue for execution, virtual machine will be triggered to process instructions in the stack. (b) If current node is not the destination of the agent, manager will push this agent into migration queue waiting for transmission.

Table 2. Elements in an agent

ID	Initial node	Target node	Node list	Next hop
State	Register	Instructions	Execution code	data

```
/**
 * LooCIComponent(<agentManager>,
 * <provided interfaces>,
 * <required interfaces>);
 */
    public agentManager() {
            super("agentManager",
            //provided interface
            new short[] {EventTypes.AGENT_EVENT };
            // required interface
            new short[] {EventTypes.AGENT_EVENT,
            EventTypes.STOP_COMPONENT_EV } ;
            };
```

Figure 8. Example code of codebase in LooCI

E. Example codes

Implementation of our proposed middleware is illustrated by two pieces of example codes and the execution result in LooCI management console.

Figure 8 shows part of codebase in LooCI-Java while Figure 9 shows part of component in LooCI-Java.

Codebase. besides the component name, provided interface *AGENT_EVENT* and required interface *AGENT_EVENT*, *STOP_COMPONENT_EV* are declared in codebase *agentManager()*. *STOP_COMPONENT_EV* is pre-defined in LooCI, which is used as control message in agent manager. *AGENT_EVENT* is self-defined to identify agents.

Component. as mobile agents (LooCI event) are deposited to the agent manager, method *receive()* is called to receive agents. In this method, *eventID* is used to identify control messages and mobile agents. If it is control message *STOP_COMPONENT_EV*, manager will deactivate component;

```
public void receive(short event_id, byte[] payload) {
/**check eventID to identify
*control messages and mobile agents
*/
  if(eventID == EventTypes. STOP_COMPONENT_EV){
     System.out.println("Monitored control messages");
     deactivate();
     } else if(eventID == EventTypes.AGENT_EVENT){
     System.out.println("Monitored mobile agent");
     publish(EventTypes.AGENT_EVENT,payload);
     } else {
     System.out.println("received invalid event");
     }
 }
```

Figure 9. Example code of component in LooCI

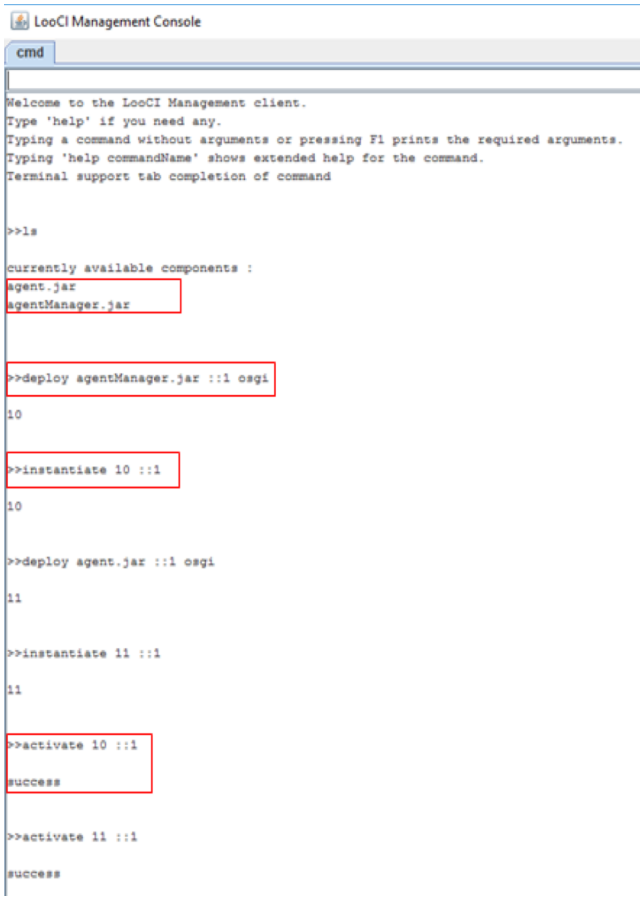

Figure 10. Operations of agent manager and agents on LooCI console

if it is an agent, manager will publish this agent; otherwise push out invalid event information.

To test the implemented components, we use command system on LooCI-OSGi. Operations of all components on LooCI console are as shown in Figure 10. After 'ant' in directory of agent and agent Manager, It can be checked by command 'ls' whether all components are available. Since two components are listed as expected, instruction 'deploy' is used for deployment of agentManager.jar and it returns codebase ID 10 by default. After deploying successfully, instruction 'instantiate' is used for instantiating codebase and it returns component number 10. The deployment and instantiation of agent are the same as agent manager except for the returned codebase ID and component number; both are 11 for the agent. After deploying and instantiating, command 'activate' can check whether it is possible to activate all components. As shown in Figure 8, the feedback 'SUCCESS' appears after 'activate' command.

V. CONCLUSION AND FUTURE WORK

A mobile agent-based middleware design for WSN is proposed in this paper. Core idea of our middleware is to optimise the allocation of resources while taking the compatibility of heterogeneous platform into account. Based on our previous research outcome, the middleware is designed on an event-based component infrastructure. Feasibility analysis and requirement analysis are presented as a proof-of-concept. The overview architecture of our design followed with detailed discussion of specific modules such as kernel managers, agent transceiver, and agent architecture is shown. Examples of codes and implementation are presented by means of a simple test.

Based on the implementation and test result we currently have, performance evaluation of our middleware system as well as case studies are necessary to show the applicability of our middleware in the coming future. Finally, we consider mobile agent technique is one of the state-of-the-art techniques in WSN but still have plenty of barriers for a commonly usage.

ACKNOWLEDGEMENT

The research presented in this paper is supported by the Research Development Fund (#RDF14-03-12) of the Xi'an Jiaotong-Liverpool University, Suzhou, China.

REFERENCES

[1] L. M. Borges，F. J. Velez，A. S.Lebres, "Survey on the Characterization and Classification of Wireless Sensor Network Applications," *IEEE Communications Surveys & Tutorials,* vol. 16, no. 4, pp. 1860-1890, 2014.

[2] M.A.Clarke,M. Razzaque, A. Milojevic-Jevric，S. Palade, "Middleware for Internet of Things: A Survey," *IEEE Internet of Things Journal,* vol. 3, no. 1, pp. 70-95, 2016.

[3] C. Fok,G. C. Roman,C. Lu, "Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications," in *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, 2005.

[4] M. Chen, S. Gonzalez,Victor, C. M. Leung, "Applications and design issues for mobile agents in wireless sensor networks," *IEEE Wireless Communications,* vol. 14, no. 6, pp. 20-26, 2007.

[5] N. Tziritas,T. Loukopoulos,S. Lalis,S. U. Khan,C. Z. Xu, "Coordination Strategies for Agent Migrations in Wireless Sensor Networks," in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, 2015.

[6] C.L. Fok, G.C. Roman, C. Lu , "Mobile agent middleware for sensor networks: an application case study," in *International Symposium on Information Processing in Sensor Networks, IEEE*, 2005.

[7] K. Lingaraj,Rajashree V. Biradar, V. C.Patil,, "Eagilla: An Enhanced Mobile Agent Middleware for Wireless Sensor Networks," *Alexandria Engineering Journal.*

[8] L. Corradetti, D. Gregori,S. Marchesani,L. Pomante,M. Santic,W. Tiberti, "A renovated mobile agents middleware for WSN porting of Agilla to the TinyOS 2.x platform," in *2016 IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*, 2016.

[9] A. Amel,D. Laurent,Deprez, J.Christophe, "Mobility Management for Wireless Sensor Networks A State-of-the-Art," *Procedia Computer Science,* vol. 52, pp. 1101-1107, 2015.

[10] Y. Wang, K. Man, S. Guan, D. Hughes, "Feasibility analysis of achieving mobile agents for wireless sensor network based on LooCI," *Lecture Notes in Engineering and Computer Science: Proceedings of The International MultiConference of Engineers and Computer Scientists*, IMECS 2017, 15-17 March, Hong Kong, 2017, pp. 714-718.

[11] D. Hughes, K. Thoelen,J. Maerien,N. Matthys,W. Horré,J. Del Cid,C. Huygens,S. Michiels,W. Joosen, "LooCI: The Loosely-coupled Component Infrastructure," in *2012 IEEE 11th International Symposium on Network Computing and Applications*, 2012.