System-Level Approach to the Design of Collaborative Distributed Systems based on Wireless Sensor and Actuator Networks

Udayanto Dwi Atmojo, Zoran Salcic, and Kevin I-Kai Wang
Embedded Systems Research Group, Department of Electrical and Computer Engineering
University of Auckland
Auckland, New Zealand
udwi469@aucklanduni.ac.nz, {z.salcic, kevin.wang}@auckland.ac.nz

Abstract— **Wireless sensor and actuator networks (WSAN) have become pervasive and are found in many embedded and intelligent systems. However, the complexity of applications based on WSANs is limited due to the lack of programming tools for this type of networks. In this paper, we show how a concurrent programming language, SystemJ, is used to develop a middleware-free distributed system that consists of a combination of stationary and mobile WSAN nodes. A distributed Ambient Intelligence (AmI) monitoring and control scenario that consists of collaborating stationary and mobile WSAN nodes with high degree of reactivity is presented. The system is designed and implemented using SystemJ to demonstrate the proposed system-level design approach. SystemJ is designed specifically to handle reactive and concurrent behaviors while providing modular and scalable methodology for distributed system composition. In addition, SystemJ removes the need for a middleware and allows system designers to focus on implementing system functionality rather than to deal with the low level programming details.**

Keywords- Distributed AmI system; SystemJ; Wireless Sensor and Actuator Network; SunSPOT; Intelligent environments

I. Introduction

Wireless sensor and actuator networks (WSAN) are increasingly used in many mobile and embedded systems. Traditional Ambient Intelligence (AmI) systems [1] commonly utilize a specific WSAN for sensing and controlling physical environments, while sharing information among themselves. The growing trend of the Internet of Things (IoT) [2] gives two distinctive new capabilities to traditional AmI systems, (1) the individual nodes are IP-addressable and Internet accessible, and hence the AmI systems may comprise components distributed across remote geographical locations, and (2) the increasing use of nodes with mobility (nodes attached to mobile objects, e.g. vehicles and humans). Both capabilities extend the traditional AmI systems into distributed domains and therefore the design, modeling, and implementation of such systems become increasingly difficult without proper software tools.

A distributed system with both fixed and mobile nodes brings a number of challenges in the design and implementation of such systems, just to name a few. First, the distributed system may contain nodes operating asynchronously to each other, while performing collective synchronous behaviors in order to provide proper services to the end users. Second, such distributed system may contain heterogeneous nodes with different sensing, actuation and communication capabilities. Third, each individual node itself can be considered as a reactive and concurrent system that integrates multiple sensors, actuators and communication capabilities. Therefore, it is crucial to have a software design paradigm that allows the designers to easily design, model and implement software behaviors with a high level of abstraction on such diverse platform. The overall distributed system can be composed based on a collection of software behaviors, which provide different services to the end users. Furthermore, each software behavior should be easily verifiable for its functional correctness, or even timing correctness, which facilitates the reliability of the provided services.

In this paper, a motivating example of an intelligent warehouse control and monitoring, which illustrates a typical distributed AmI system, is used as a case study. Based on the motivating example, we demonstrate the system-level design approach of using a reactive and concurrent programming language, SystemJ [3], for designing distributed AmI systems based on a WSAN. The example shows how SystemJ allows designers to focus solely on the functional behavior level without worrying about the underlying physical devices, communication links and middleware. The example is designed and implemented on a WSAN that consists of SunSPOT [4] nodes based on ARM-9 processors and equipped with a small Java Virtual Machine called Squawk [5]. The system contains multiple fixed and mobile SunSPOT nodes, with different sensing and actuation capabilities. Each SunSPOT node can communicate with other nodes and networks using the 802.15.4-based 6LoWPAN protocol [6], which provides IP-addressability and Internet accessibility. Internet access or remote services can be provided through a fixed gateway (or edge router) SunSPOT node.

The rest of the paper is organized as follows. Section II provides related works on some existing AmI applications and available software design approaches. Section III illustrates the motivating example of distributed AmI system. Section IV explains the system-level design approach using the motivating example, with individual software modules described. Section V concludes the paper with descriptions of possible future works.

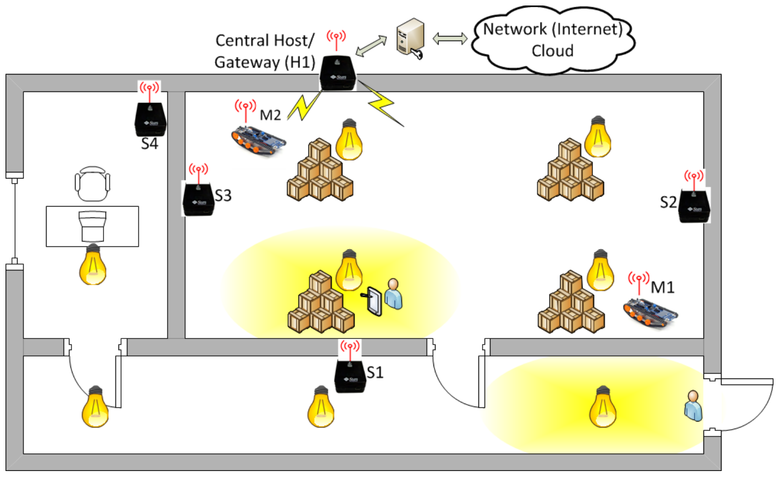

Figure 1. A motivating example on intelligent warehouse control and monitoring.

II. RELATED WORKS

There are a number of application scenarios, which have been developed or currently under development related to collaborations between fixed and mobile sensor and actuator nodes. Those applications includes services in smart homes [7] and surveillance [8]. Many existing software frameworks are designed to facilitate the aforementioned applications scenarios. The two major approaches are the robotics middleware and multi-agent system frameworks.

Robotics middleware such as OpenRTM-aist [9], Player /Stage [10], and Robot Operating System (ROS) [11], are designed mainly to facilitate the development process of mobile robots. While these middleware offer some advantages such as platform and language independency and mechanisms for inter-components communication, each of them possesses some drawbacks. Player has a centralized architecture (as it is based on a server-client communication), thus a system developed using this middleware may stop working entirely if the server fails to execute properly. OpenRTM-aist's platform and language independency, as well as its inter-components communication, is realized through CORBA, which has been outdated and has a complex implementation. CORBA itself is known to have an issue as it generally assumes that performing remote calls is always successful (reliable), whereas in distributed system context, it is unsafe to make such an assumption [12]. Similar to Player, ROS also possesses a centralized architecture, hence it faces the similar issue that Player has. In addition, ROS is known to have overheads in its messaging system. Unlike our approach, ROS users have to handle thread synchronization by themselves, therefore introducing more programming burden and more error-prone implementation.

Unlike the robotic middleware, agent frameworks provide a high level abstraction of software behaviors, where each software behavior (or agent) is loosely connected with other behaviors. However, in order to run a software agent on a specific hardware platform, extra layers of agent runtime environment and middleware are necessary. For example, software agents developed using JADE (Java Agent DEvelopment Framework) [13], AF-APL (Agent Factory Agent Programming Language) [14], and JAL (JACK Agent Language) [15] all require the accompanied Agent Runtime Environment (ARE). Additional middleware layer, such as SIXTH [14], UPnP [16], DPWS [17], or CORBA [18], underneath the ARE layer is also necessary for deploying agent systems on heterogeneous hardware platforms. In terms of performance, SystemJ has shown better efficiency compared to JADE, as elaborated in [19].

III. MOTIVATING SCENARIO

In this section, a motivating scenario of a distributed AmI system, which will later be used to demonstrate the proposed system-level design approach in SystemJ in the next section, is presented. In this scenario, an intelligent warehouse control and monitoring system is considered, where multiple stationary and mobile WSAN nodes are included in the system, as shown in Fig. 1. The warehouse building contains three areas, which are the corridor area in the bottom, the office area on the left and the warehouse area on the right. The nodes placed in different areas will have different sensing and actuation capabilities in order to provide different services.

In the corridor area, there is one stationary node (S1) installed for controlling individual lighting circuits and detecting human presence. Following the human movement along the corridor, S1 node will turn on the necessary lights according to the current user location and turn off all the other lights. The stationary node (S4) in the office area may have different capabilities and provides different services. For example, S4 may be equipped with a light intensity sensor, which senses ambient light intensity that changes dynamically by the external light source (i.e. window) and the installed lighting. The S4 node can adjust the lighting circuit intelligently to maintain the ambient light level. In addition, S4 can also be equipped with a human presence sensor, such as a PIR sensor, to turn off the lights when no human presence is detected.

Unlike the office and corridor areas, the warehouse area is bigger and more complicated. Within the warehouse area,

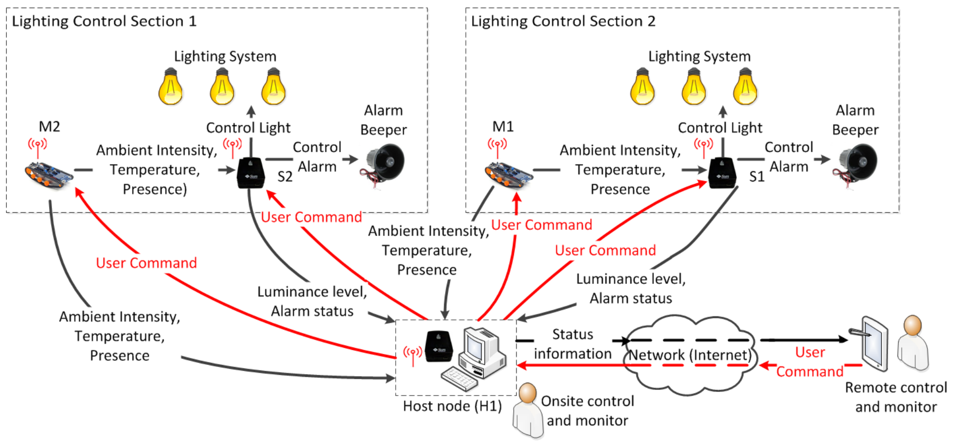
Lighting Control Section 1
Lighting System
M2
Control Light
Alarm Beeper
Ambient Intensity, Temperature, Presence)
Control Alarm
S2
User Command
User Command
Luminance level, Alarm status
Ambient Intensity, Temperature, Presence
Lighting Control Section 2
Lighting System
M1
Control Light
Alarm Beeper
Ambient Intensity, Temperature, Presence
Control Alarm
S1
Ambient Intensity, Temperature, Presence
User Command
User Command
Luminance level, Alarm status
Status information
Network (Internet)
User Command
Remote control and monitor
Host node (H1)
Onsite control and monitor

Figure 2. System-level design approach using a distributed collaborative WSAN.

there might be several sections for holding different inventories and each section may have its own lighting circuit. Due to the larger area coverage, multiple stationary nodes (S2 and S3) are installed and each controls several sections. The corresponding lighting circuit will be turned on when human presence is detected. In addition to the lighting control, these stationary nodes may also be equipped with extra temperature and humidity sensors and are able to actuate local ventilation systems, in order to maintain the warehouse at a prescribed condition. Besides stationary nodes, there can also be one or multiple mobile nodes (M1 and M2) patrolling within the warehouse area, performing security checks. The mobile node can communicate with the stationary node when unauthorized presence is detected and the stationary node can trigger the alarm system. Mobile nodes in this case give additional advantages by providing better sensing coverage compared to the stationary nodes, which have a limited view angle and can be blocked by obstacles.

Besides servicing mobile and stationary nodes, there is also a host gateway node, which, for instance, collects information from monitoring environment via stationary nodes or even inventory checking information from a mobile tablet node. The information is then made available for browsing via the Internet. In addition to the remote monitoring, an active control is also possible through the host node, which can accept commands from remote users to manually adjust stationary nodes settings or control mobile nodes movements.

This motivating scenario demonstrates a potentially large scale distributed AmI system with collaborating stationary and mobile nodes. Each node has different sensing and actuating capabilities and service requirements, which have different control software behaviors. This simplified example is implemented in SystemJ to demonstrate the proposed system-level design approach in the next section.

IV. System-Level Design Approach

Based on the motivating scenario described in Section III, the proposed system-level design approach is demonstrated in this section. The presented implementation captures only the warehouse area in Fig. 1, which contains sufficient details for the purpose of demonstration. Subsection A presents the top-down system composition for the implementation of the overall distributed system using multiple SystemJ programs, which execute on heterogeneous platforms. Subsection B, C and D explain how SystemJ captures the reactive and concurrent software behaviors of the mobile, stationary and host nodes. Subsection E briefly covers the SystemJ Runtime Support (RTS), which enables SystemJ programs to be deployed on heterogeneous platforms without using middleware.

A. *System-level design approach*

The design process of a distributed AmI system starts with a top-down decomposition of the overall system into multiple software behaviors, which can be deployed on either fixed or mobile nodes. For example, in our scenario, the system contains a WSAN with five nodes, which are two pairs of stationary (S-node) and mobile node (M-node) and a PC-connected host node (H-node) for the purpose of remote control and monitoring, as shown in Fig. 2. Each node within the system has its own capabilities and services, and can be considered as a subsystem within the overall distributed system. A SystemJ program that encloses a unique set of software behaviors represents a subsystem and provides necessary services. Within each subsystem, there exist asynchronous and synchronous software behaviors, which are captured by SystemJ programming entities, namely clock domains (CDs) and reactions, respectively. The use of a CD and a reaction are detailed in the following subsections.

A distributed system can be easily composed by a collection of SystemJ programs, running asynchronously on heterogeneous platforms. In order to facilitate system-level design, SystemJ provides a high level abstraction object, signal, to handle the communication between multiple SystemJ programs and physical devices. Referring to Fig. 2, for example, the interconnections between SystemJ programs on S1 and M1 or S1 and an alarm beeper are implemented as signals, regardless of what physical connection is used underneath. The signals and physical connections mapping and conversion are done by the SystemJ RTS, which is introduced in subsection E. Based on this demonstration, it is clear that SystemJ supports an easy approach to decompose a distributed system, while also provides a high level abstraction to help system designers to focus on implementing the system functionalities, rather than taking care of low level communication details. The strength of SystemJ in handling reactive and concurrent software behaviors within a subsystem is detailed in the following subsections, and more details on features of SystemJ can be found in [19].

B. *Mobile node implementation*

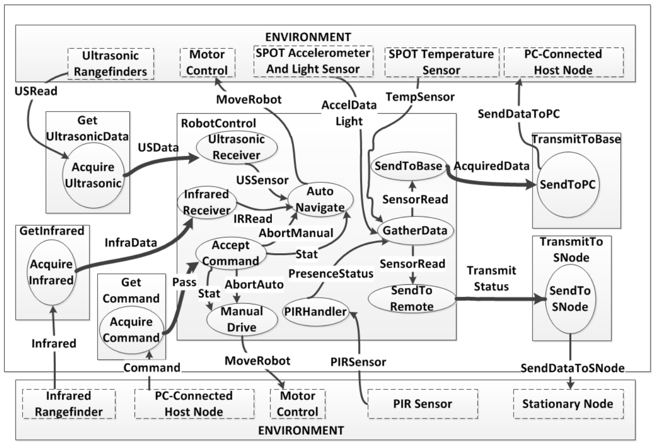

Figure 3. Graphical illustration of the SystemJ program on M-node

The graphical illustration of the top-level SystemJ implementation of the M-node is shown in Fig. 3. Reactions and CDs are depicted as ellipses and rectangles, respectively. As seen in Fig. 3, a single M-node contains a number of synchronous and asynchronous concurrent behaviors, each performing specific operations. For example, the **GetUltrasonicData** and **GetInfrared** CDs, each consists of one reaction that awaits for the ranging results from the ultrasonic and infrared (IR) rangefinders used by the mobile node for navigation data acquisitions through signals **USRead** and **Infrared**, and send them to the **RobotControl** CD via channels **USData** and **InfraData**, respectively. The example shows how SystemJ can be used easily to describe interactions between heterogeneous components (e.g. sensors) in a node.

The **TransmitToBase** and **TransmitToSNode** CDs, each also contains one reaction, which obtains information from the **RobotControl** CD, and transmits them to the H-node and S-node, respectively. The **GetCommand** CD, contains only one reaction that accepts any commands to the M-node and passes it to the **RobotControl** CD. The use of CDs shows how SystemJ can model interactions between distributed nodes, which are typically asynchronous each to the other.

Apart from receiving readings from the sensors, the **RobotControl** CD processes incoming information (e.g performs sensor fusions, and data processing) and performs physical actuations (e.g the M-node navigation) based on the received information. Listing 1 displays a fraction of the two reactions **AcquireInfrared** (in **GetInfrared** CD) and **AutoNavigate** (in **RobotControl** CD) to show how such functionalities are described in SystemJ. We have abstracted out most of the source code due to lack of space.

```
1    reaction AcquireInfrared(: input Vector signal Infrared, output
     Vector channel InfraData)
2    {
3       while (true)
4       {
5         await (Infrared);
6         Vector infra=(Vector)#Infrared;
7         send InfraData(infra);
8         pause;
9       }
10   }
11   reaction AutoNavigate (:input String signal IRRead, input String
     signal USSensor, input String signal Stat, input signal AutoAbort,
     output String signal MoveRobot)
12   {
13     while (true)
14     {
15       await (Stat);
16       String mode=(String)#Stat;
17       if (mode.equals("Automatic"))
18       {
19         abort (AutoAbort)
20         {
21           while (true)
22           {
23             {await (USSensor);}||{await (IRRead);}
24             speed= ComputeSpeed ((String)#USSensor,(String)#IRRead);
25             emit MoveRobot(speed);
26             pause;
27           }
28         }
29        mode="Manual";
30       }
31       pause;
32     }
33   }
34   //Further reaction declaration...
```

Listing 1. Reaction code example

In Listing 1, line 1-10 describe reaction **AcquireInfrared**. First, this reaction acquires distance measurement result from the IR rangefinders via signal **Infrared** (line 5). After obtaining the result, it sends the reading to **RobotControl** CD through channel **InfraData** (line 7). The **AutoNavigate** reaction is used to control the mobile node movement by utilizing IR and ultrasonic rangefinders (line 11-33). This reaction initiates by awaiting for **Stat** signal that carries a state variable (line 15). This signal is immediately present in the first tick and then saved in a variable named mode (line 16) which carries the value of "Automatic", thus the behavior will proceed. In the next tick, the reaction is concurrently waiting for rangefinder measurements from ultrasonic and IR rangefinders through signal **USSensor** and **IRRead**, respectively (line 23).

Once received, both results are obtained and processed using a Java method to calculate the M-node movement (speed and direction) (line 24). Line 23-24 show a simple example on how to perform sensor fusions in SystemJ. Finally, the reaction emits **MoveRobot** signal which will actuate the M-node to navigate its immediate environment (line 25). The user may manually control the M-node at any time. When such command arrives, the **AutoAbort** signal is present which will preempt the behavior execution inside the body (line 20-28). Once the mode variable is changed (line 29), the control functionality is then given to the **ManualDrive** reaction that enables the user to control the mobile node manually. This example shows how SystemJ can easily describe a software behavior with multiple states. The overall example also shows how SystemJ can easily describe a node that performs both control or data dominated operations.

C. *Stationary node implementation*

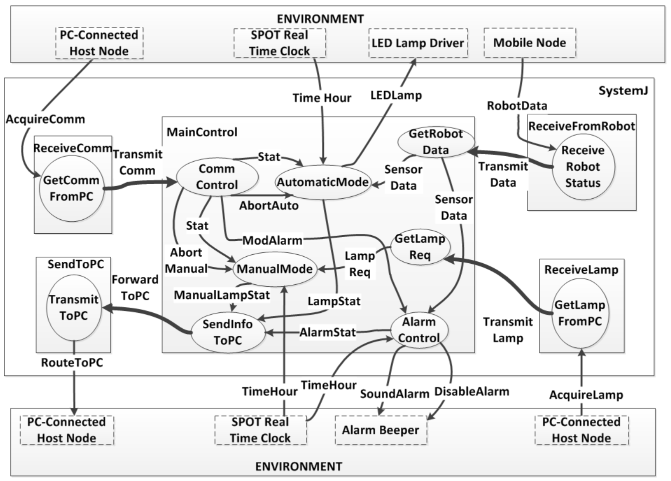

Figure 4. Graphical illustration of the SystemJ program on S-node

The graphical illustration of the top-level SystemJ implementation of the S-node is shown in Fig.4. Referring to Fig. 4, the implementation has five asynchronous CDs. The **ReceiveComm** CD has one reaction (**GetCommFromPC**) that awaits for any user commands (to switch between lighting control modes and to control the alarm beeper) via signal **AcquireComm** and sends it to the **MainControl** CD via channel **TransmitComm**. The **ReceiveLamp** CD, also consists of one reaction that receives any user commands to manually switch on or off lamps in the lighting system. The **ReceiveFromRobot** CD has one reaction, which accepts human presence information from the M-node and transfers it to the **MainControl** CD. The **SendToPC** CD obtains all information to be sent and sends them to the H-node. Similar to the example in Subsection B, these four asynchronous CDs describe the communication between the distributed nodes.

The **MainControl** CD, having seven synchronous reactions, performs the actual lighting and alarm control of the S-node. The lighting system is governed by two reactions (**AutomaticMode** and **ManualMode**) controlling the lighting system interchangeably. Such behavior is possible by making use of SystemJ abort statement, similar to the implementation described in Listing 1.

The alarm beeper is controlled by the **AlarmControl** reaction in the **MainControl** CD. This reaction awaits for signal **SensorData** (containing human presence information sent by the M-node) and **TimeHour** signal (carrying information about the current time). If a human presence is detected during a restricted-access time, the **SoundAlarm** signal is emitted to trigger the alarm beeper. Once enabled, the alarm beeper can only be disabled by an authorized user. Such command is received by the reaction via **ModAlarm** signal, in which then the **DisableAlarm** signal is emitted when the the **ModAlarm** signal is present, turning off the alarm beeper.

D. *Host node implementation*

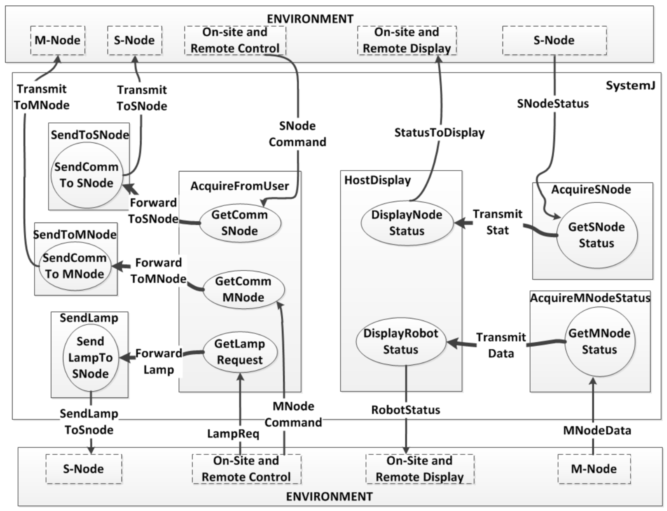

Figure 5. Graphical illustration of the SystemJ program on H-node

The SystemJ program on the H-node mainly performs data fusions of the received information from the distributed nodes and provides services for the user (e.g displaying information, and accepting incoming requests and serving them). Fig. 5 shows the graphical illustration of the SystemJ program on the H-node. The implementation consists of five asynchronous CDs (i.e **SendToSNode, SendToMNode**, **SendLamp**, **AcquireSNode**, and **AcquireMNodeStatus**), which handle information exchanges between H-node and the other distributed nodes, and two asynchronous CDs (i.e **AcquireFromUser** and **HostDisplay**) that model interactions between the nodes and other involved software systems (i.e user interface (UI) software). For example, a UI software receives a user command to manually control a lighting system or an M-node movement, in which the command is converted by the SystemJ RTS into a signal (i.e signal **SNodeCommand** or **MNodeCommand**), and then received by the SystemJ program. Two reactions (**GetCommSNode** and **GetCommMNode**) capture it, process, and send the command to the sending CDs to be transmitted to the destination nodes. To provide information to the user, the **AcquireMNodeStatus** and **AcquireSNode** CDs receive information from the M-nodes and S-nodes and transfer it to the **HostDisplay** CD, in which the information will be made available to the user by emitting signal **StatusToDisplay** and **RobotStatus**.

E. *SystemJ Runtime Support (RTS)*

During the design process of a GALS [20] system or a SystemJ program, system designers should consider interactions between reactions, CDs and external environment using only signals and channels. These two objects provide high-level abstractions for the system designers to focus on the design of system functionalities, rather than to handle low-level details of sensor/actuator events and physical communication links. All the channel and signal communications are handled by the SystemJ RTS, which is implemented in Java. SystemJ RTS makes use of the APIs provided by various JVM, including J2EE, J2SE, J2ME, Squawk VM, Dalvik and LeJOS, to convert physical signals from sensors, actuators and communication links of a specific hardware platform into SystemJ signals and channels, or vice versa, as shown in Fig. 6. The compiled SystemJ program executes on top of the RTS. The primary output of SystemJ program compilation is Java code, which is then compiled by a standard Java compiler and runs on various JVM-enabled execution platforms. Java code generated by the SystemJ compiler is compliant with CLDC 1.1 specification, which is supported by most Java execution environments, including all the aforementioned JVMs and other embedded Java solutions. It is important to note that SystemJ program entities (i.e. reactions and CDs) communicate using abstracted signals and channels, and the RTS that converts signals/channels into physical events are completely transparent to the system designers.

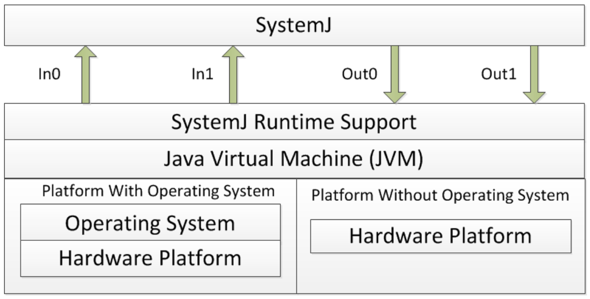

Figure 6. Software layers for executing a SystemJ program.

V. CONCLUSION

In this paper, a system-level design approach based on the use of the concurrent programming language, SystemJ is presented. A typical distributed AmI system that implements an intelligent warehouse control and monitoring system is used as a motivating example. Based on this example, the proposed approach and SystemJ demonstrated the strength of system decomposition using powerful synchronous and asynchronous software behaviors, and high level abstractions that lead to platform independent deployment and middleware-free implementation. The GALS formal MoC followed by SystemJ provides a nature fit of distributed, reactive and concurrent systems such as modern distributed AmI systems. With the support of formal MoC, the proposed approach not only provides an easier way for designing the system, but also makes behavior verification possible during design time. A full implementation of the motivating example is currently under development. A complete software framework that supports dynamic system composition is also planned.

REFERENCES

[1] Ramos, C., Augusto, J. C., and Shapiro, D., "Ambient intelligence—The next step for artificial intelligence," Intelligent Systems, IEEE, vol. 23, pp. 15-18, 2008.

[2] Atzori, L., Iera, A., and Morabito, G., "The Internet of Things: A survey," Computer Networks, vol. 54, pp. 2787-2805, 2010.

[3] Malik, A., Salcic, Z., Roop, P. S., and Girault, A., "SystemJ: A GALS language for system level design," Computer Languages, Systems, & Structures, vol. 36, pp. 317-344, 2010.

[4] Smith, R. B., "SPOTWorld and the Sun SPOT," presented at the Proceedings of the 6th international conference on Information processing in sensor networks, Cambridge, Massachusetts, USA, 2007.

[5] Simon, D., Cifuentes, C., Cleal, D., Daniels, J., and White, D., "Java on the bare metal of wireless sensor devices: the squawk Java virtual machine," presented at the Proceedings of the 2nd international conference on Virtual execution environments, Ottawa, Ontario, Canada, 2006.

[6] Mulligan, G., "The 6LoWPAN architecture," presented at the Proceedings of the 4th workshop on Embedded networked sensors, Cork, Ireland, 2007.

[7] Lu, F., Tian, G., Zhou, F., Xue, Y., and Song, B., "Building an Intelligent Home Space for Service Robot Based on Multi-Pattern Information Model and Wireless Sensor Networks," Intelligent Control and Automation, vol. 3, pp. 90-97, 2012.

[8] Lambrou, T. P. and Panayiotou, C. G., "Collaborative area monitoring using wireless sensor networks with stationary and mobile nodes," EURASIP Journal on Advances in Signal Processing, vol. 2009, p. 7, 2009.

[9] Ando, N., Suehiro, T., and Kotoku, T., "A software platform for component based rt-system development: Openrtm-aist," Simulation, Modeling, and Programming for Autonomous Robots, pp. 87-98, 2008.

[10] Gerkey, B., Vaughan, R. T., and Howard, A., "The player/stage project: Tools for multi-robot and distributed sensor systems," in Proceedings of the 11th international conference on advanced robotics, 2003, pp. 317-323.

[11] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A., "ROS: an open-source Robot Operating System," in ICRA workshop on open source software, 2009.

[12] Goff, M., Network distributed computing: fitscapes and fallacies: Prentice Hall Professional Technical Reference, 2003.

[13] Bellifemine, F., Bergenti, F., Caire, G., and Poggi, A., "JADE—a java agent development framework," Multi-Agent Programming, pp. 125-147, 2005.

[14] O'Hare, G. M. P., Collier, R., Dragone, M., O'Grady, M. J., Muldoon, C., and Montoya, D. J., "Embedding Agents within Ambient Intelligent Applications," Bosse, T.(ed.). Agents and Ambient Intelligence: Achievements and Challenges in the Intersection of Agent Technology and Ambient Intelligence, 2012.

[15] Bordini, R. H., Braubach, L., Dastani, M., El FSeghrouchni, A., Gomez-Sanz, J. J., Leite, J., O Hare, G., Pokahr, A., and Ricci, A., "A survey of programming languages and platforms for multi-agent systems," INFORMATICA-LJUBLJANA-, vol. 30, p. 33, 2006.

[16] Wang, K. I. K., Abdulla, W. H., and Salcic, Z., "Ambient intelligence platform using multi-agent system and mobile ubiquitous hardware," Pervasive and Mobile Computing, vol. 5, pp. 558-573, 2009.

[17] Familiar, M. S., Martínez, J. F., and López, L., "Pervasive Smart Spaces and Environments: A Service-Oriented Middleware Architecture for Wireless Ad Hoc and Sensor Networks," International Journal of Distributed Sensor Networks, vol. 2012, 2012.

[18] Augusto, J. C. and Nugent, C. D., Designing smart homes: the role of artificial intelligence vol. 4008: Springer, 2006.

[19] Malik, A., Salcic, Z., Chong, C., and Javed, S., "System-level approach to the design of a smart distributed surveillance system using SystemJ," ACM Transactions on Embedded Computing Systems (TECS), vol. 11, p. 77, 2012.

[20] Chapiro, D. M., "Globally-asynchronous locally-synchronous systems," PhD, Department of Computer Science, Stanford University, California, 1984.