Sensomax: An Agent-Based Middleware For Decentralized Dynamic Data-Gathering In Wireless Sensor Networks

Mo Haghighi and Dave Cliff
Department of Computer Science
University of Bristol
Bristol, UK
Mo.Haghighi@Bristol.ac.uk, dc@cs.bris.ac.uk

ABSTRACT—**In this paper we describe the design and implementation of *Sensomax*, a novel agent-based middleware for wireless sensor networks (WSNs), which is written in Java and runs on networks of various Java-enabled embedded systems ranging from resource-constrained *Sun Spot* nodes to resource-rich *Raspberry Pi* boards. Programming WSNs tends to be a complex task for developers, as it requires detailed knowledge of underlying hardware resources as well as their firmware or operating systems. Although many solutions have been proposed up to this date, only a few of those are capable of satisfying challenging demands such as serving multiple user applications and re-programming the network at run-time in popular high-level languages such as Java. Sensomax presents a novel combination of several best practices from existing solutions, facilitating fully distributed and decentralized bulk programming and/or updating of sensor nodes; serving multiple simultaneous applications deployed by single or multiple users; allowing dynamic run-time changes in the application requirements; and offering on-the-fly switching between time-driven, data-driven, and event-driven operational paradigms. Sensomax provides a sophisticated set of APIs, a feature-rich desktop application, a web application for cloud-based distributed networks, and a simulator. We demonstrate Sensomax in operation on a real network of 12 Sun Spots deployed as an environment-monitoring system, and 600 virtual Sun Spot nodes running continuously over periods of several weeks, using a novel statistically rigorous adaptive change-point detection algorithm to identify significant "anomalous" changes in the monitored data-streams.**

Keywords-Wireless Sensor Networks; WSN; Agents; Java; Sun Spot; Multi-tasking; Middleware

I. INTRODUCTION

Remote monitoring of key environmental variables as well as observing certain events of interest has become the focus of many scientific fields and is of increasing interest in engineering projects and consumer products. WSNs, due to their lack of fixed infrastructure and ease of installation are considered highly versatile technology platforms for a variety of application areas.

The high versatility of WSNs is due to many factors: WSN nodes are typically small in size, battery-powered, and low-cost. However such features come with several tradeoffs in the underlying hardware resources.

Due to lack of large energy capacity, crucial hardware resources such as processor, memory and wireless communication devices need to be very energy-efficient in their operations. Even if all such devices are maximally energy-efficient, battery-life can still be prolonged by deployment of appropriate resource-allocation methods and control mechanisms. Therefore, a set of coordinated policies among the resources and the applications are highly desirable and in many cases essential in order to meet the requirements of both the network and the applications [8]. From the application point of view, the adoption of such policies typically results in reduction of multi-tasking, distributed storage, serving multiple applications, data-aggregation and minimizing the total number of communications.

Applying such tradeoffs can enormously affect network performance, while serving multiple applications, and dynamic changes forced by applications could result in long delays or non-responsiveness. Several middleware solutions addressing these challenges have exploited a single operational paradigm such as time-driven, data-driven, or event-driven approaches, all to make a balanced tradeoff. (For survey reviews of WSN middleware, see e.g. [14], [15] and [16], and for a recent taxonomy see [17]). Applying a single operational paradigm seems like an

978-1-4673-6404-1/13/$31.00 ©2013 IEEE

effective solution when certain specific applications are concerned. However such application-specific behavior can be inflexible and brittle when dynamic changes are required. On the other hand, attempting to implement multi-tasking and serving concurrent multiple applications via a single operational paradigm can reveal severe limitations in that paradigm. Serving multiple applications can be very cost-effective and ultimately results in seamless re-purposing of the pre-deployed network. e.g. a wide-life monitoring network that simultaneously serves a fire-detection application.

The primary backbone of Sensomax is based on mobile agents. Nearly all operations and processing within the network and nodes are carried out by the agents. Mobile agents convey data, tasks, and different configuration policies throughout the network in order to inject or execute them in the appropriate nodes. In Sensomax, as with other WSNs, nodes are required to specify their required functionalities to the agents: this is one of the major differences between agent-based approach in conventional IP-based network and WSNs.

The mobile agent approach was originated from client-server communication paradigm developed in conventional computer networking, where a number of services are provided by the servers, and clients can use them [2]. Therefore Sensomax, as its first preference, partitions the network into several clusters to implement server-client paradigm. Another important motivation behind such approach, is decentralizing task-allocation and data-distribution.

Sensomax has been developed in Java(ME) programming language to be implemented on Sun Spot sensor nodes[14] and Java (SE Embedded) targeting resource-rich platforms such as the *Raspberry Pi* [18], where energy is not constrained. Sun Spot node is a sensor platform comprising a number of sensors and actuators such as temperature, light, 3D accelerometer, LEDs, IOs and etc. As illustrated in Figure 1, Sensomax runs atop Squawk which is a J2ME virtual machine on a bare ARM processor [3].

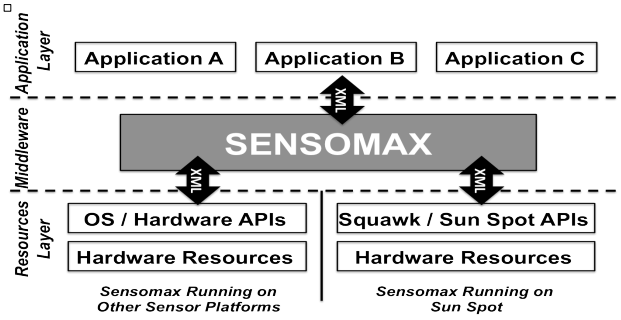

Figure 1. Sensomax middleware layer in SunSpot and other platforms

Sensomax middleware is very light and only takes 70KB of storage. There only exists a single version of Sensomax, integrated in every sensor node, regardless of their resources, which shows the heterogeneity of its architecture. On the applications side, there is a desktop application which contains the same version of Sensomax in Java (SE) combined with a GUI for easier user interaction.

Although Sensomax has currently been implemented and tested on Sun Spot nodes, there is nothing Sun Spot-specific in its design. We have successfully ported Sensomax to the *Raspberry Pi,* a popular small cheap single-board Linux platform [18] produced in the UK.

As Figure 1 depicts, all physical communications between desktop application, gateway and nodes are in XML formatting to facilitate interoperability with other applications and platforms. Due to length restriction of this paper we have omitted the details of design and implementation of Sensomax simulator and the Web application for cloud inter-networking capabilities, which have been submitted in two papers and are currently under review.

In the next section of this document we survey a number of agent-based Middleware approaches, comparing their pros and cons. In Section III we describe the Sensomax architecture and functionalities, and demonstrate its component interactions. In the final section we explain why Sensomax is an ideal candidate for multi-purpose WSNs.

II. EXISTING AGENT-BASED APPROACHES

A number of middleware systems based on agents have been developed and implemented for WSNs. Agents have been widely used for various applications: [10] used agents model to detect intrusion, [11] applied agents for monitoring dam safety, [9] provides a good example of agents-based observation in healthcare and [6] demonstrates highly adaptable behaviors of agents in fire detection. In this section we will provide a brief description of implemented middleware functionalities and how they relate to our proposed middleware in terms of using agents for multi-tasking, data collection, fusion and delivery.

Agilla [6], which was the first agent-based middleware to have implemented mobile agents in WSNs, integrates mobile agents into a tuple-space operational model. Application-specific agents are injected into the network from the base station and cloned onto other nodes in order to perform application tasks. Based on task requirements, Agilla can autonomously change behavior and adapt its operation to the environment. In Agilla, every node contains a single tuple space in which agents can reside or migrate to. Nodes' tuple spaces can collectively be viewed as a network shared memory space. Tuple space helps the agents to migrate freely without any prior synchronization

or memory allocation scheme. Agilla's focus on the local self-adaptive behavior of each agent, although promotes a great degree of decentralization, locally forces nodes into application-specific behavior which prevents executing multiple agents concurrently as well as imposing long latency for network-wide data-collection. Also in cases where multiple applications are involved, too many agents need to be deployed, cloned and moved around the network waiting for execution which causes a bottleneck.

SensorWare [4] is a dynamic middleware particularly developed for Wireless Ad-hoc Sensor Networks (WASN) targeting resource-rich devices (in WASN, sensors communicate in an ad-hoc fashion where the locations of sensors are not known to the applications). In SensorWare, based on the application requirements, nodes are configured as sets of services to be exploited by agents. Services are abstraction layers over the operating system and hardware resources which offers pre-defined operations in multiple single packages. Services can be re-configured at runtime based on new incoming requirements from the applications. Applications' requests are distributed through the network in form of agents that can execute their tasks using the available services in each node. Multiple agents can reside on a single node, waiting for execution, as state machines. SensorWare uses TCL scripting language for executing the agents and aggregating data, which requires rich processing resources. The dynamic mechanism for reprogramming nodes during runtime makes Sensorware an effective approach to achieve programming flexibility. However such approach is more appropriate for data-aggregation tasks, where aggregation mechanisms need to be changed based on new requirements, compared to a data-gathering approach where on-time delivery of data is of utmost importance.

SmartMessages [7] is another agent-based approach primarily targeting networks of embedded systems in which agents are called Smart Messages (SMs). This platform offers an operational model for programming distributed applications and adapting to network changes. It consists of two disciplinary components: a Tag Space and a Virtual Machine (VM). *Tag space provides a name-based memory and a unique interface to the local OS and I/O System [7]*. VM provides an abstraction layer in which unnecessary details of underlying hardware resources are hidden and SMs are executed. SMs use content-based routing in which it migrates through the network seeking the nodes of interest and only executed after matching their tag names. However SM contains some routing code which needs to be executed in every node it visits. VM needs to acquire some information about the next hop's available resources before deploying the SM. VM achieves this checking process using the SM declared minimum requirements. After executing the SMs, VM embeds their execution state in them before sending to the next hop. This causes some tight coupling between nodes where migrating large number of nodes could potentially take long. VM can accommodate multiple SMs, however only one can be executed at a time and others need to be queued for dispatching or execution until after dispatching the current one. Therefore multiple programming and updating of the network cannot be done concurrently. One of the other limitations of this approach is its heavy initialization where it requires rich resources for loading, linking and verifying class files for creating VM-level thread with a stack frame. Such operations need more powerful resources which WSNs are lacking.

MAPS [2] or Mobile Agents Platform for WSNs based on Java Sun Spot, provides an architecture where its components interact in an event-driven manner. *Mobile agents are modeled as multi-plane state-machine driven by ECA rules [2]*. Components can offer a number of services such as message transmission, agent creation, migration and cloning. Such services are provided based on the agent operational level: application, middleware and network levels. Therefore MAPS is an approach combining the agent, event and state based paradigms into a single framework.

MAPS has been developed in Java and implemented on Sun Spot nodes, hence in terms of using Sun Spot Java-based APIs for accessing the hardware resources such as communication protocols, sensing devices, flash memory and etc., it shares a number of similarities with our proposed middleware. MAPS relies heavily on event objects for its system, components and agents to interact with each other. Also components are represented as threads which can create other thread-based components. This thread instantiating process results in several tightly-coupled threads running in the main Isolate (main application space provided by squawk in which node initially boots up [3][13]). There arises a number of shortfalls in this approach. Since components are highly interactive through events, the failure of one thread can result in others failure. Running many threads have large data memory footprint as well as shortening the lifetime of the network by consuming constant energy for executing the threads. One of the most important limitations of such approach is implementing energy saving strategies. Sun Spot provides a major energy-saving strategy known as deep sleep where device energy consumption falls below 35μA [13]. In order to send the device into deep sleep mode, running threads need to be paused. In such approaches where a number of inter-dependent threads are running, achieving efficient energy strategies are highly unlikely. The other limitation of MAPS is MAMigrationMan component which handles the migration process of the agents. This component uses the Radiostream protocol [13], which establishes a synchronous communication with the destination node. Radiostream protocol requires a request-to-send packet to be transmitted to the destination as some sort of hand-

shaking which forces the transmitter into waiting for its acceptance. During data streaming, any other communications with other nodes cannot be established. therefore such approach interrupts the ongoing process in both the sender and the receiver nodes, as well as causing delays to other components operations which could result in rejecting other agents. The dynamic behavior of mobile agents are represented by a component called ECAA which combines ECA rules with State Machine mechanism, for handling state transitions of mobile agents according to certain events. In cases where multiple events are fired, such approach limits the concurrent checking of multiple events and transitioning multiple agents.

III. SENSOMAX ARCHITECTURE

This section describes the software architecture of Sensomax at user application side where Sensomax resides on the gateway node and also at the network side where it functions in sensor nodes. Before detailing the architecture, it is worth pointing out how Sensomax benefits from agent-based communication paradigm. As Agents can freely move throughout the network and promotes asynchronous functionalities, they offer benefits such as lowering the bandwidth and decentralization. Using agents as communication intermediary within Sensomax provides a number of key features in terms of programming, tasking, data-gathering and storage.

A. Key Features of Sensomax

1) Programming

- *a) Multi/single node initial programming*
- *b) Multi/single node re-programming/updating*
- *c) Multi/single node role assignment*
- *d) Regional Programming*

Deployed agents freely move around the network, carrying vital network properties, in order to program each node based on the application requirements.
Using their onboard data, agents can assign the properties required for node's operational paradigm such as its role in the network (Cluster-head, member or CH of CHs), its communication paradigm (ports, CH address, gateway address, cluster-name and clock synchronization) and assigning regional configuration to a group of CHs to represent a single abstract region.

2) Tasking

- *a) Multi/single node task distribution and allocation*
- *b) multi/single node task execution*
- *c) Regional task execution*
- *d) mixed-paradigm task creation/execution*

The most important goal of sensor networks is assigning the right task to applicable nodes and ensuring of their orderly and on-time execution. Task agents migrate around the network, carrying task from one node to another, cloning the task in those involved in the application requirements. They also make sure the node is capable of executing the task by checking its resources as well as checking the task scheduler queue. Task agents can also be applied to a group of nodes, CHs or mixture of both.

3) Data Gathering

- *a) Multi/single node data-gathering*
- *b) Regional data-gathering*
- *c) Aggregated data-gathering*
- *d) mixed paradigm data-gathering*

Gathering, aggregating and forwarding data are considered the most crucial duties of agents in Sensomax. Data-gathering agents are called QFeeds which carry sensory data among nodes. Based on the application demands, some analysis and computations can be applied to sensory data before forwarding data to the application. Such data-gathering process could be done based on meeting one or more conditions, in which captured data is matched against a pattern consisting of various parameters specifically defined by the applications. Therefore the data-gathering model could switch between time-driven, event-driven and data-driven paradigms based on the type of condition.

4) Storage

- *a) Global storage*
- *b) Local storage*
- *c) Gateway storage*

Agents can convoy data for storage in other parts of the network. Depending on the application and the available storage capacity, data can be moved around the cluster-members, cluster-heads or among cluster-heads and the gateway. Distributed storage is useful when low-storage capacity is detected in a node or a cluster.

B. Application-Side Programming

Application-side programming refers to the Sensomax software, running on the desktop machine which can be accessed by the end-users for programming the network through either a GUI, command line or a web browser that connects to the host socket. For a quick demonstration, Figure 6 shows the GUI running on a Mac OS X PC. Due to the limited scope of this document, the detailed features of Sensomax APIs and software are omitted.

As Figure 1 illustrated, all communications between the system, application layer, gateway and all the nodes in the network are in XML format. This helps applications from

various sources to submit their requirements in a generic and widely-used formatting which promotes inter-operability. XML offers a number of advantages over other formatting such as its wide application in web semantic, human understandability, available libraries and etc., However the main motivation behind XML adoption is its application in Sensor Web Enablement, a set of standards defined by OGC (open geospatial consortium) [12], which could ultimately make Sensomax more compliant to web standards.

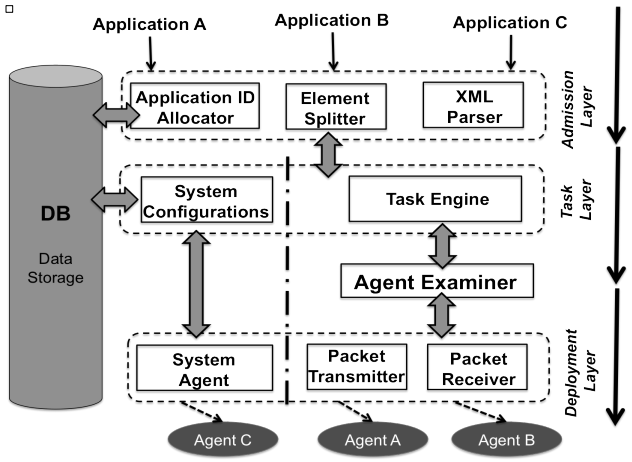

Figure 2. Sensomax End-user Application Framework

Figure 2 shows Sensomax's software architecture embedded in the gateway node. As depicted in the same figure, at the forefront of the system, end-user applications physically interact with the network through a gateway. In the gateway, there exists a layer known as the Assessment Layer with a principle component called the Agent Examiner (AE). This layer provides a simple XML parsing tool in order to split the application requirements and label them with their relevant application sources.

In the next step, those requirements submitted by different applications are processed and broken down into a single or multiple agents based on their demands and complexity. This sort of top-layer processing is the result of two components' coordination: Task Engine and System configuration.

Task Engine (TE) is primarily involved in identifying the spatial and temporal elements of the submitted requirement and forming a correlation of both aspects via its sub-components. The result of task engine analysis is a unified package called the QTASK which is classified as an independent agent.

The sub-components of the task engine are: QTIMING, QREGION and QEVENT plus two logical analyzer. As their names suggest, they are in charge of handling the timing, regional and event-based parameters of the task. QLogic and Task Executor are two more components which handle the logical analysis of a task and its execution policies. These components are tightly coupled with system resources in order to perform the computational analysis of the task. Hence, the task engine receives raw task agents and outputs bundled packages in the form of QTasks.

System Configuration (SC) is an independent component which constantly deploys free agents through the network, querying the cluster-heads for local information such as their cluster-members, running tasks, available resources and clock synchronization. System Configuration agents, known as System Agents (SA), migrates from one cluster-head to another, gathering data without any interruption to the run-time processes of the node. SA processing is in total isolation and there is a separate component allocated in GLS for handling SAs.

The SA deployment and collection in System Configuration happens in the network background and there is no timing or any other requirement assigned to that. To Summarize, System Configuration creates a virtual map of network resources and their availabilities.

Task Engine can make use of this virtual map to decide on how best to utilize the resources, what adaptability policies to take on and what reconfigurations to carry out in order to execute a task. This initial bi-component collaboration can ultimately result in minimizing the extent and number of reconfigurations during the life-time of the network.

The next phase deals with transmitting agents onto the network. Agents are broadcasted and received by all cluster-heads and only processed by those involved in the execution of the task. Agent Deployment layer is in charge of this process by using PacketTransmitter component. PacketTransmitter can operate in different modes based on the types and number of recipients. It can unicast, broadcast or multicast cluster-heads or nodes based on the transparency and physical range of deployment.

Agents can actively look for their targets in other cluster-heads while processed in a different one. They keep on migrating until after reaching their targets.

In cluster-heads, agents are received by Deployment Layer through PacketReceiver component. PacketReceiver along with Agent Examiner component, survey the contents of the agent to define the type of agent. Agents can be classified into several categories: Local, Global, System, Feed and Task.

- Global Agent (GA) deals with the acts of individual nodes and does not necessarily engage in clustering configuration. Its data aggregation, collection, and injection are executed in the recipient nodes regardless of its cluster membership. Global agents can be made and deployed by the gateway for the cluster-heads or by the cluster-heads for the cluster-members.

- Local Agent (LA) deals with internal functions, aggregated data and command dissemination within a cluster. Therefore, LAs are only executed by the cluster-heads. LAs automatically create relevant Global sub-agents to be deployed onto cluster-members. Global sub-agents are like GAs, only made by the Cluster-heads, and will return to their cluster-heads after execution.

- System Agent (SA), which was explained earlier, is an agent executing in isolation with the rest of the system and its job is to gather systematic information about its recipient.

- Task Agent (TA), which was also explained before, is composed of single/multiple GAs or LAs, with timing, event and regional conditions. TAs can reside on a node and only executed when they are scheduled for execution by the applications. They can also look for events of interest and only act when those events are detected.

- Feed Agent (FA) is an agent returning to the gateway or the cluster-head with data content. FAs only contain sensory data collected from cluster-heads or cluster-members in raw or aggregated form.

C. *Node-Side Programming and Operation*

This section describes the operation of Sensomax on sensor nodes. Sensomax provides similar functionalities on the node-side although by involving more components where required. As Figure 3 shows, agents are received through the Packet Receiver component in the deployment layer. If Agents are non-SA types, they will be examined and forwarded to the Agent Examiner (AE) component. For each agent accommodated in the deployment layer, there will be an exclusive thread of Agent Examiner allocated. This thread allocating process allows handling multiple agents concurrently. Each AE thread is destroyed after fetching the agent to the task engine.

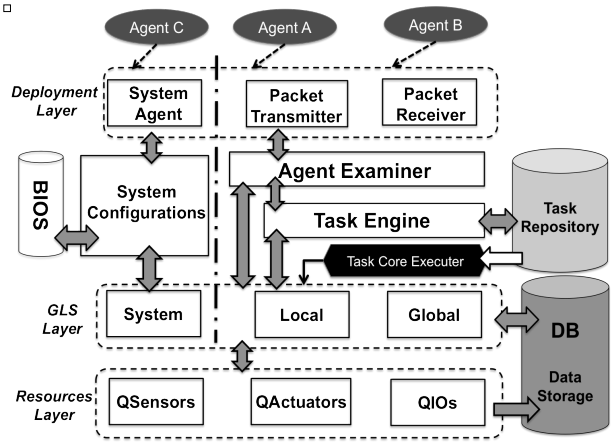

Figure 3. Sensomax Node-side Framework

If the agent is an SA, it will be forwarded to the System Configuration for further processing. In cases where the node is acting as a CH, received agent could potentially be an FA received from the cluster members. In this case, agent is processed by the EA, and extracted sensory data bypasses the Task Engine to the GLS for further storage.

Based on the application requirements, gathered data can be stored in the DB which handles storing all sensory data in the node. Storage in DB is done for further computation or transmission or for data to be directly sent out to the applications as FAs. GAs and LAs simply query the available sensing resources for sensory data or inject code to actuators. Resources layer is an abstract layer to shield the underlying hardware and its operational details for easier access.

The Task Engine is one of the core components of the system in which thread-scheduling, timer-allocation, garbage-collection, script analysing and regionalizing the network take place.

The Task Engine keeps track of all running tasks and the ones which have already completed their executions. It reclaims the allocated memory and resources of completed tasks by removing their objects from the thread-scheduler.

Figure 4 illustrates the Task Engine modulated components. They are exactly the same components as in the Gateway application, however their inputs (Agents) come from the AE and their outputs (QTasks) go to the Task Repository module and GLS.

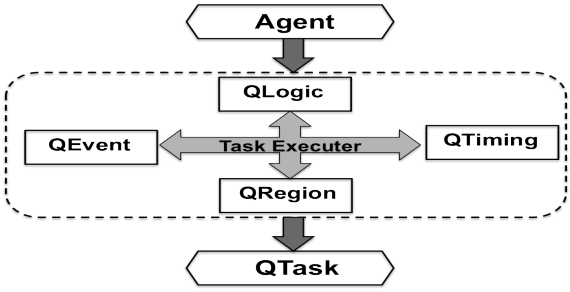

Figure 4. Task Engine Components

QTasks are either queued in the task repository which stores the tasks that need to be scheduled for later execution or send them directly to the GLS layer. GLS processes the QTask, and relevant LAs or GAs are either forwarded directly to the resources layer or back to the Task Engine for cluster members.

IV. CASE STUDY

For our prototype, we ran a data-gathering experiment in one of the computer labs at the University of Bristol. Twelve Sun Spots running Sensomax were used to monitor temperature and light levels for 36 hours with a resolution of 10 seconds. Sensomax desktop application was used to

program the nodes into three clusters with an abstract regional layer to cover five sensors in the lab and two sensors outside the windows. In total, 3 different tasks were injected to the network initially: 1. Sensing temperature in every node and storing data locally in the node. 2. Sensing light level and storing data locally in the node. 3. Aggregating both sensory data (temperature and light) and store them in the regional layer with 10 seconds interval. To check the dynamism of the system, using the internet-connected gateway, we imposed some operational changes at runtime such as cancelling and resuming tasks, changing the sensing intervals and changing the regional layers. As depicted in Figure 5, the gateway application provides an Internet socket using Sensomax APIs to remotely monitor the nodes and their status.

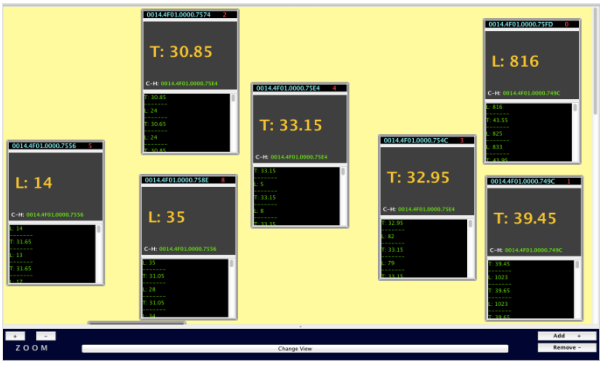

Figure 5. Remote Access to Sensory Data via the Web

We also had 40 different virtual user applications accessing the network. The virtual applications queried the network constantly at random intervals and all their requirements were met successfully.

After 36 hours, 350000 records were stored across the network. Stored data were optionally generated either as 7 separate XML files for each sensor, 3 XML files for 3 clusters or 2 regional XML files for observing data inside and outside the lab and 40 folders containing different data-sets associated with tasks deployed by 40 virtual user applications. Our experiment proved Sensomax's easiness and versatility in programming and applying changes at runtime.

Figure 6 shows the user interface for programming the sensors. Sensors can be seamlessly discovered and clustered, based on their locations, without getting into the burden of knowing every operational details of the system. In essence, the desktop application represents the flexibility, power and dynamism of Sensomax APIs, hidden below a user-friendly interface, in order to simplify the aforementioned complexities. Figure 6 (from right to left) shows the discovery and clustering steps for the aforementioned experiment. Nodes are formed into three clusters and deployed inside and outside the lab to measure the temperature and light levels. Nodes' properties such as their battery level, running times and available resources are also easily accessible through the user interface.

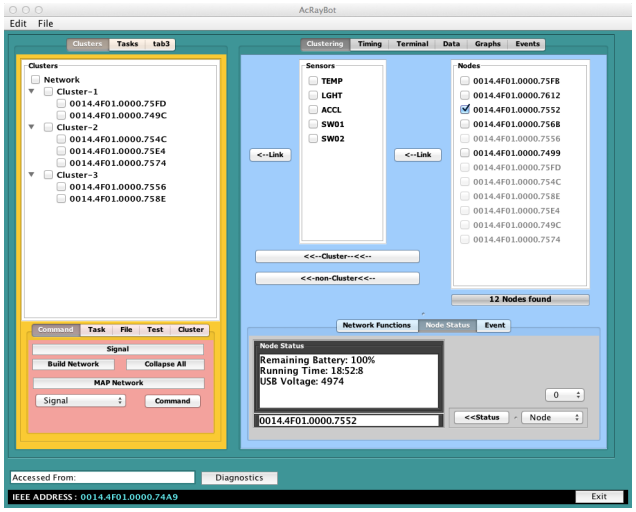

Figure 6. User Interface for Discovering and Clustering Nodes

The desktop application includes a graph-generating tool, which enables visualising sensors' captured data individually or collectively based on their regions. Figure 7 and 8 show the variation of light and temperature levels respectively, over the 36 hours period. Green and yellow lines denote the sensors located outside of the lab, the blue line representing the one inside but adjacent to the window and rest are the ones inside but away from the window.

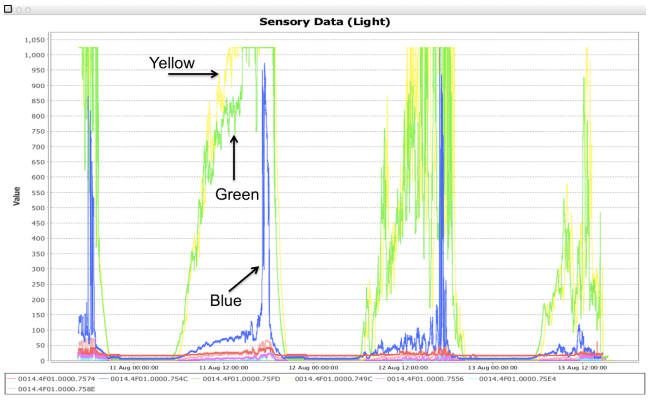

Figure 7. Light level Variation Graph

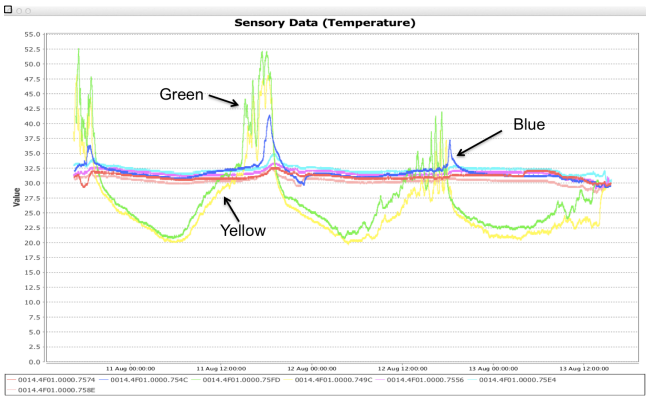

Figure 8. Temperature Variation Graph

One of the most important purposes of our experiment was checking out the reliability of our task engine. Based on our captured dataset, the task engine continually executed all the tasks and stored data locally inside each node as well as storing them into the cluster-heads. In a different experiment, we used Sensomax to capture and feed data to an algorithm written in python programing language to detect anomalous behaviour of environmental variables such as light and temperature as well as monitoring acceleration and infrared beacons for movement detection in security surveillance applications.

The high-level description of end-to-end operations of our proposed middleware was detailed in this document. To summarize, the three key features of Sensomax are as follows:

Multi-Agent Processing. As depicted in Figure 3, AEs' thread nature, can handle multiple agents and all non-system communications with the network which need to pass through this object layer. This mechanism enables the AE to be accessible concurrently at all levels amongst various components. AE objects are not persistent and will be destroyed after each agent processing.

Multi-tasking. Based on Figure 3, the multi-tasking capability of our proposed middleware arises from the centralized access of the Task Engine which takes care of handling, storing and executing the task agents and at the same time, providing a secondary execution engine known as Task Core Executer which handles executing stored tasks in the task repository. Task Core Executer (TCE) operates an independent entity within in a thread that runs during the entire lifetime of a node. The independent behaviour of Task Core Executer provides a robust failure-recovery function that in case of failure in all other components, TCE reads and restarts all the stored tasks from the repository and re-injects them into the system.

Dynamic Behaviour. According to Figure 3, applying system changes at runtime is done in a separate space where SAs are handled by system configuration. System configurations are applied without affecting the task processing and other operational components.

V. CONCLUSION

The proposed Sensomax architecture has been demonstrated to be an effective framework for programming WSNs, applying dynamic changes at runtime, observing multiple regions and serving multiple applications concurrently. Sensomax achieves those key features by exploiting the agent-based communication paradigm in conjunction with data-driven, event-driven and time-driven operational paradigms, which are imposed collectively or separately onto the system operational paradigm based on the applications requirements. Sensomax detaches system-level operations from processing and executing the agents, in order to provide prompt response to dynamic requirements of the applications, without interrupting the on-going network operations. Our future works will include the integration of automated mechanisms such as market-based algorithms and game theory for clustering the network and distributing captured data.

REFERENCES

[1] M. Chen, S. Gonzalez, & V. Leung, "Applications and design issues for mobile agents in wireless sensor networks," *IEEE Wireless Comms*, 14(6): 20–26, 2007.

[2] Aiello, G. Fortino, A. Guerreri, & R Gravina, "Maps: a mobile agent platform for WSNs based on java sun spots" *3rd Int. Workshop on Agent Technology for Sensor Networks*, 2009.

[3] D. Simon, C. Cifuentes, D. Cleal, J. Daniels & D. White, "Java on the bare metal of wireless sensor devices - the squawk java virtual machine," *2nd Int. Conf. Virtual Execution Environments,* ACM Press, June 2006.

[4] A Boulis , C Han, & M Srivastava, "Design and implementation of a framework for efficient and programmable sensor networks," *Proc. 1st Int Conf. Mobile Systems, Applications & Services*, pages 187-200, San Francisco, USA, May 2003.

[5] S. Hadin, & N. Mohamed, "Middleware: middleware challenges and approaches for wireless sensor networks," I*EEE Computer Society*, vol. 7, no. 3, March 2006.

[6] C. Fok, G. Roman, & C. Lu, "Mobile agent middleware for sensor networks: an application case study," *Proc. 4th Int. Symp Information Processing in Sensor Networks*, pp.382-387, 2005.

[7] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer & L. Iftode, "Spatial programming using smart messages: design and implementation," *24th Int. Conf. Dist. Comput. Sys.* (ICDCS), 2004.

[8] K. Romer, O. Kasten, & F. Mattern, "Middleware challenges for wireless sensor networks," *ACM SIGMOBILE Mobile Communication & Communications Review*, 6(2), 2002.

[9] Z. Dai, B. Sun, & C. Gui, "Hybrid and multi-agent approaches on healthcare sensor information fusing," *Fourth Int. Conf. Networked Computing and Advanced Information Management.* 2008.

[10] M. Ketel, "Applying the mobile agent paradigm to distributed intrusion detection in wireless sensor networks", *40th Southeastern Symp. System Theory*, March 2008.

[11] Y. Zhu & C. Chai, "Sensor networks based dam safety monitoring system" *Int. Conf. on Computer & Comm. Tech. in Agriculture Eng.,* 2010.

[12] M. Botts, G. Percivall, C. Reed, & J Davidson, "OGC Sensor web enablement: overview and high level architecture", White Paper by open Geospatial Consortium Inc, 2007.

[13] Oracle, "Sun Spot Programmer's manual", Release v6.0, Sun Labs, Oracle, 2010.

[14] S. Hadin & N. Mohamed "Middleware for Wireless Sensor Networks: A Survey" *Proc. COMSWARE 2006*, IEEE.

[15] M. Wang, J. Cao, J. Li, & S. Dasi "Middleware for Wireless Sensor Networks: A Survey" *J. Comp Sci. & Tech.* 23(3):305-26, 2008.

[16] M. Molla & S. Ahamed "A Survey of Middleware For Sensor Networks and Challenges" *Proc. 2006 Int. Conf. on Parallel Processing Workshops (ICPPW'06*), pp.223-228, 2006.

[17] L. Motolla & G. Picco 2011. "Programming Wireless Sensor Networks" *ACM Computing Surveys* 43(3):1-51.

[18] E. Upton & G. Halfacree. *Raspberry Pi. User Guide*, John Wiley, 2012.