Multi-Agent Support for Multiple Concurrent Applications and 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—WSNs have gained increasing attention for monitoring various variables of interest for a wide variety of applications ranging from tracking environmental conditions to medical and structural monitoring. There are many WSN hardware platforms with a wide range of on-board resources. There also exist many software solutions for programming and re-programming WSNs. Most of the existing software solutions are either tightly coupled to their associated hardware, or very application-specific. Such diversity introduces many challenges for application developers. In this paper we propose a novel middleware solution, which runs on Java (SE and ME) programming platforms for easy task distribution and data gathering integrated in a modulated architecture that supports the serving of multiple concurrent applications, dynamic reprogramming, good scalability, and multiple operational paradigms.

Keywords—Sensomax; multi-agent; concurrency; dynamic; WSN; multi-paradigm; java;

I. INTRODUCTION

A wireless sensor network is a collection of small and embedded devices scattered across an environment for monitoring environmental parameters or observing phenomena of interest. WSNs have drawn a lot of attention over the past decade and their applications have increased dramatically surrounding every part of our lives ranging from structural monitoring, military solutions, health and medical observation, fire detection, habitat monitoring to monitoring volcanoes and space engineering.

WSNs are generally installed in remote areas where physical access is very difficult or in many cases impossible. Therefore they are normally required to run unattended for long durations. Also sensor nodes involved in WSNs are small-sized and battery-powered. This lack of sufficient power and enough space, limits their onboard resources and capabilities. Therefore WSNs are very resource constraint in terms of processing, memory and communications. MICA2 mote [1] is one of the most popular sensor platforms used in research and development of WSNs in both academia and industry. MICA2 mote is a very good example of resource scarcity with its specifications as follows: 4MHz processor, 512 KB of data and 128 KB of program memories.

Recent technologies in microelectronics and embedded systems however have brought a lot of benefits to WSNs and many new sensor devices are highly capable in terms of their hardware resources. Sun Spot [2] is one of these devices that was introduced a few years ago by Sun Microsystems and later continued and improved by Oracle. Sun Spot houses a 400MHz ARM processor, 1 MB of data and 8 MB of program memories. What makes Sun Spot an ideal candidate for embedded solution is its ability to run the full Java ME virtual machine (JVM) which enables developers to program the Sun Spots in Java.

Apart from the hardware perspective, WSNs need reliable software to manage all the communications, processing, storage, security, data aggregation, energy-management and etc. In conventional computer systems these objectives are accomplished by the operating systems. In both cases of resource-constraint and resource-rich WSN devices, due to limited resources and long operational periods, the concept of operating system is not very well-defined and not the same as conventional systems [3].

Middleware is a layer residing between the hardware resources and the application layer. Middleware should facilitate applications’ access to the underlying hardware resources in the most efficient and reliable manner with as little overhead as possible. There have been many attempts at developing middleware for WSNs and many of those proved to be quite effective. However, in many cases developers tend to build platform-specific middleware, which tightly integrates their software solutions (middleware) to their associated hardware, generally with low-level programming languages. Examples of such approaches include TinyOS[4], which is specifically built for MICA2 and uses a special C-like programming language called nesC[6], and Agilla[5] which has been written in low-level Assembly-like language and runs on TinyOS-based MICA2 and MICAZ motes. Many middleware also tend to be very application-specific such as Mate[7],Impala[8], Sina[9] and irisnet[10] which are the examples of data-driven models where data are accessed through SQL-like queries and nodes offer none or minimal autonomy. The above-mentioned obstacles bring a number of challenges to developers for either porting or modifying the existing middleware onto other hardware platforms. Hence they need to learn low-level hardware and software...
details of the existing systems as well as learning new programming languages like netS.C.

Java is an object-oriented and high-level programming language that offers cross-platform interoperability for conventional systems. Java ME is a light version of Java targeted at embedded systems ranging from TVs to mobile phones and WSNs. Recently a new version of Java (Java SE Embedded) has also been introduced by Oracle targeting midrange devices that fit somewhere between resource-constrained and resource-rich embedded systems such as the Raspberry Pi [14]. What matters is the ability of the core Java API that is extensive and includes features available in conventional PCs such as sockets, threads, security, graphics and other useful functions.

Using Java covers all those low-level details in underlying hardware and provides an interoperable solution in a highly popular programming language. Also from the developer’s point of view, Java and its IDEs such as Netbeans and Eclips are freely available for different operating systems such as Mac OS, Windows and Linux.

There have been many attempts at developing java-based solutions for WSNs including: MagnetOS[11], MAPS[12], MASPOT[15] and ATaG[16]. Out of those, ATag, MAPS and MASPOT have been developed for Sun Spots.

Our Proposed middleware is called Sensomax, which is a component-based, multi-threading and dynamic software solution that runs atop various Java-enabled embedded devices. Sensomax provides an end-to-end solution for various applications to program and update WSNs based on their requirements as well as offering a reliable mechanism for capturing data from the network and fetching it to the applications. In summary, Sensomax provides the following features: (1) Macro-programming; (2) Multi-tasking; (3) Multiple concurrent applications support; (4) Dynamic task modification/ re-programming at runtime; (5) Supporting various scalability mechanisms through clustering the network.

II. ARCHITECTURE

In this section, the architecture of Sensomax is described in 7 different stages: Agents, Isolated Processing, Multi-tasking, Resource Abstraction, Communication domain, Profiling and Operational Paradigms.

As Figure 1 shows, Sensomax divides the network into multiple groups. Each group, composed of multiple nodes, is dedicated to a single or multiple application(s). Applications can potentially use multiple overlapping clusters by running application-specific agents on each cluster. This works in a way that network entities (cluster-heads and cluster-members) execute application agents differently based on the entity’s role and properties for that application in the network. i.e. Cluster-head of an application run that application’s agent with cluster-head’s functionalities while the same node acting as cluster-member of another application plays a member for another cluster-head and executes its agents with cluster-member functionalities.

![Figure 1. Clustering in Sensomax](image)

Abstract Regions[20] is a middleware approach that fragments a WSN into multiple groups known as regions. It allows the application programmers to treat each group of sensors as a single entity. Sensomax utilizes the same clustering mechanism for (1) Running concurrent applications in the network, (2) Maximizing scalability and (3) Decentralising application control to the cluster-heads rather than the base station.

A. Agents

In Sensomax, agents are the primary means of communications amongst modules by acting as couriers moving around the network. They carry various pieces of data such as instructions, tasks, updates, network status and etc. Based on their integrated data, they can be divided into three major categories; Global: basic agents used for communication amongst various network entities; Local: agents involved in inter-cluster communication and only exchanged between cluster-head and its members. Local agents are used by cluster-heads to address their requirements to their members and vice versa; System: These agents are one-way agents only sent from the base station to the cluster-heads and from cluster-heads to their members for instructing systematic configurations. There are five other types of agents considered as sub-categories of major types: QTask, QFeed, QQuery, QResponse and QCommand. In Sensomax, the base station creates a cluster for every application and interprets their requirements into multiple agents and transmits them to the assigned cluster-head.

B. Isolated Processing

Unlike Agilla[5] our approach doesn’t use a single shared space known as tuple-space for executing agents. Such approach prevents maintaining concurrent execution of multiple agents belonging to distinctive applications on individual nodes. However, it seems highly efficient for coordinating multiple agents of the same application on a single or multiple node(s).
Within each application process, several sub-tasks can be initialized. Each application task needs to be associated with an operational paradigm, which will be explained in more details. Most existing middleware support one or two operational paradigms such as Mate[7] and Impala[8] that follow event-driven paradigm, SINA[9] and COUGAR[13] follow a highly-coupled combination of data-driven and Query-driven paradigms. Sensomax provides a mechanism at the base station where task requirements are split into four major categories (1) Data, (2) Event, (3) Timing and (4) Queries. These sub-requirements define the overall demand of an application that need to be exposed to the underlying hardware.

Many applications tend to be dynamic or run in dynamic environments, in such cases WSNs need to support dynamic modification/updating of the applications requirements. i.e. [17] Investigates the effects of reprogramming and code modification of sensor nodes at runtime, [18] investigates how changing cluster-size can affect energy consumption and lifetime of WSNs and [19] looks into dynamic partitioning of network for oil slick monitoring for efficient data-collection. Such modifications should be implemented seamlessly to the relevant applications only, without affecting running applications and interrupting network.

Sensomax implements on-the-fly updating and reprogramming of nodes by utilizing System agents that are in charge of carrying modifications to network entities. System agents inject themselves to the relevant process needing an update. In Sensomax, a copy of each active task agent is maintained within each process, residing in two memory spaces known as: (1) Task Repository and (2) sub-tasks Repository. Task Repository simply contains a copy of the entire application task for that node whereas the Sub-Tasks Repository contains sequential blocks of sub-tasks, with their activity flags representing their running state, extracted from the main task. In case of minor updates, only sub-tasks concerning with the modifications are replaced, whereas major updates replaces the entire task repository with a new task.

C. Multi-tasking

In order to simplify the root operational mechanism of Sensomax, its overall structure can be best described in the context of conventional operating systems. If we omit the existence of JVM, Sensomax takes a hybrid-kernel approach in which every user application acts as a Microkernel running atop a single Monolithic Kernel. The Execution Engine’s structure, containing all user applications as processes, resembles a Monolithic Kernel, where every process (Microkernel) has access to all underlying hardware resources. Therefore Sensomax, which acts as a Monolithic, abstracts all the available resources around each user application on a higher level. This mechanism allows multiple applications to operate concurrently within the main process whilst accessing all resource. It also creates a virtual operational space in which components can interact with resources through the agents.

Our proposed middleware follows a components-based design where each component represents a software module in charge of a set of homogenous sub-tasks aimed at fulfilling a higher-level task. Such modulated design helps the application developers to envision the entire system as a collection of interactions amongst various modules and in turn facilitates the development process by interpreting the application requirements into a set of coordinated interactions amongst modules.

D. Resource Abstraction

Resource abstraction follows the same mechanism used for concurrency and was explained in the multi-tasking part. Sensomax, which acts as a micro-kernel, divides resources into multiple groups by assigning boundaries for privileged and non-privileged access. Applications (monolithic), on the other hand, have access to all resources whilst being managed by the main process. The aforementioned boundaries split resources into; Global: all sensors, actuators and aggregation processes that are shared amongst all network entities; Local: collective abstraction of all resources within a cluster and only shared amongst members of the same cluster; System: resources tied to system properties where crucial resources states are registered. In summary splitting resources into these three categories defines the access domain of user applications and their associated tasks, while isolating their exclusive requirements from those shared with other applications. Such isolated resource division eases the process of importing any optional resource allocation protocols. By default our middleware split resources fairly amongst all user applications. However, Sensomax exploits several automated computational mechanisms such as game theory and probability distributions for allocating/distributing data storage and processing amongst network entities. We have also considered various market-based algorithms to administer resource-allocation in WSNs.
perspective at a higher-level of abstraction, where the core of the system can be pictured as a number of user-application layers. Each agent, associated with an individual user application, is directed into its own layer where it is stored for processing. Agents are then ported into the execution space where they queue to be executed individually or together inside the core executor. Core Executor is the space where agents have access to all system resources based on their requirements. This is also where our proposed optional resource allocation algorithm is imported and implemented. Execution space also operates as a filter where agents are refined and their execution domain is assigned based on node’s status. Our multi-operational paradigm mechanism is implemented in this module.

E. Communication domain

Sensomax offers a number of clustering benefits: (1) Any node can belong to more than one cluster, (2) Every Cluster-member can act as a cluster-head of another group, (3) Every cluster head can also act as a cluster-member of another group, (4) Cluster-member of one group can also act as a member of other groups. Sensomax communication paradigm assumes four types of communications:

- **Exclusive Public Communications:** Group interaction only visible to the group members and the cluster-head (invisible to other clusters)(Multicast)
- **Exclusive Private Communications:** one-to-one communication only visible to both ends. (Unicast)
- **Inclusive Private Communications:** one-to-one communication visible to the group members and the cluster-head. (Insecure Unicast or Unicast + Multicast)
- **Inclusive Public Communications:** Group interaction visible to all groups and their members. (Broadcast)

The aforementioned communication scheme together with profile assignment, which will be explained in the next section, facilitates multiple overlapping abstraction regions where multiple applications can utilize the same set of sensors simultaneously. Nodes can switch between different applications based on the agent they receive or their stored tasks. Such application-switching paradigm helps nodes to act as members of multiple groups as well as potentially being cluster-heads of others. Based on Figure 2 (A), every application has its own space where all associated tasks are isolated from other application space.

F. Profiling

Figure 2 (B) represents the execution space in figure 2 (A), where profiling and operational paradigm assignment are implemented. According to figure 2 (B) sensor nodes can operate in one of these three profiles; (1) **Free:** nodes run in neutral states where they act independently. Free Nodes can freely join other networks, clusters or switch roles into a cluster-head; (2) **Client:** Same features and functionalities of free mode while being exclusive to a cluster. Node can only communicate with its cluster-head and other cluster-members of the same cluster; (3) **Server:** Cluster-Heads operate in Server profile in which outgoing and incoming communications only comes from and goes to their cluster-members and the base station.

Each profile (operational state) owns a number of exclusive functionalities where each set of functions represents a tiny virtual machine. Agents associated with each profile can utilize those functionalities, along with a number of extra functionalities that are shared amongst all three profiles, such as Copy Agent, Extract agent properties, Register new properties and etc.

Functionalties differ based on the type of profile they operate in: e.g. the available agent deployment and reception functionalities of each profile vary as follows:

<table>
<thead>
<tr>
<th>Profile \ Functions</th>
<th>Agent Deployment</th>
<th>Agent Reception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free</td>
<td>Inclusive-Public</td>
<td>Inclusive-Public</td>
</tr>
<tr>
<td>Server</td>
<td>Exclusive-Public</td>
<td>Inclusive-Public</td>
</tr>
<tr>
<td>Client</td>
<td>Inclusive-Private</td>
<td>Exclusive-Public</td>
</tr>
</tbody>
</table>

![Figure 3. Functions variation in different profiles](image)

G. Operational Paradigms

Every agent needs to be associated with an operational paradigm in order to be executed. Our proposed system facilitates multi-operational paradigm mechanism where agents can execute in different paradigms based on their requirements.

**Data-Driven:** Data-driven paradigm is the most fundamental paradigm that conventional sensor networks and many current WSNs offer. In this paradigm a WSN is envisioned as a network of multiple data sources where each chunk of data represents the data captured by the sensors. Applications extract their required data by sending SQL-like queries to the relevant node. SINA[9] and COUGAR[13] are example of such paradigm. **Event-driven:** in this paradigm, applications view the network as multiple sources of events of interest. They subscribe to specific types of event and get notified of their occurrences. This operation, which is known as publish and subscribe, is a very well known paradigm and applicable to many fields of WSNs. Agent-based WSN middleware based on Event-driven paradigm are Impala[8] and Mate[7]. **Time-driven:** In this paradigm applications define specific single/multiple time intervals to be notified of sensed-data. This operation involves a number of timers to trigger the operations. **Query-driven:** In this paradigm applications instant-query the network for various requirements. Queries are generally used in conjunction with the data-driven paradigms in order express the specific data abstraction demands. **Hybrid:** this is a combination of multiple paradigms to provide more flexibility in a unified form.
Support for scalability is implemented through one-hop re-broadcasting of (received) multicast-type agents. Normally, recipients relay every agent that has been flooded throughout the network in a multicast fashion. Once an agent is re-sent, it will never be accommodated in that node again as its ID is saved in that node permanently until re-programming is required.

This mechanism ensures every network entity re-broadcast to its nearest node. On the downside, this mechanism also increases network traffic and causes congestion, but it proved to be an effective solution in our tests both in prototype and simulation as we prove it in the next section.

III. EVALUATION

We have evaluated the multi-tasking and multi-user capabilities of Sensomax in both prototype and simulation. Our prototype experiment was implemented on 12 Sun Spot nodes and for simulation we used Sensomax Simulator or SXCS. SXCS is a java-based emulator which has been fine-tuned for desktops running MAC OS X and Windows. Our Linux version is currently under-development. Operational details of SXCS is out of the scope of this paper, however a brief description of its architecture is given below.

SXCS is a discreet event simulator/emulator in which the entire Sensomax middleware source code is loaded onto the simulator and up to 2000 nodes can be emulated as virtual nodes. Sensors, actuators and the communication medium for each node are provided by SXCS. SXCS provides a shared communication channel in which all common network communication mechanisms such as broadcast, unicast and multicast, both in private and public domains, are provided for each virtual node and the base station. As we explained, Sensomax capabilities are mainly focused on the high-level data distribution/gathering, multi-tasking and multi-application support and dynamic reprogramming and multi-agent execution. Therefore the objectives of our simulation experiments are not concerned with lower-level protocols such as routing, energy and etc.

In our first experiment, we evaluated Sensomax’s multi-tasking, multi-application support and dynamic updating, using 12 Sun Spots nodes with a variable number of applications querying the network at randomly selected times.

The objectives of this experiment are: (1) Total processing time of each agent based on the number of concurrent tasks (2) Maximum number of cluster membership each node can support with a reasonable agent processing latency (3) Total processing time of each dynamic update and (4) Total percentage of packet loss in the network. We conducted our experiment with variable number of applications between 1-30 and registered the total processing time of each agent. It is worth noting that all these processing times include an average of 35ms propagation delay that occurs in the Sun Spot routing protocol.

As illustrated in figure 4, total processing time of each agent increases with the number of concurrent applications inside the node. For the purpose of our experiment 200ms is a reasonable processing time for 30 concurrent applications running inside the node. It is also worth mentioning that applications used for this experiment were mostly lightweight, containing hybrid combination of event-driven, time-driven and data-driven requirements.

![Figure 4. Processing Time vs. Number of Applications](image)

An example of such lightweight application is one that demands reporting of temperature and light levels in two different clusters every 30 seconds to the base station, saves both parameters every 1 minute inside the nodes and decreases the reporting interval if temperature exceeds e.g. 20°C. 29 other applications follows the same patterns asking for these variables with various conditional and timing requirements.

We also evaluated the dynamic updating of the applications by changing their requirements at different occasions during the experiment. According to figure 4, processing times of updating agents are much faster than other types of agents.

![Figure 5. Number of Applications vs. Clusters](image)

Figure 5 (blue line) shows the average number of clusters made based on the total number of concurrent applications. Figure 5 (red line) shows the average number of cluster membership per node that increases with the number of applications in a less aggressive manner. Finally figure 5 (green line) shows the average number of clusters serving an application. Each experiment was conducted 10 times to average out the anomalies for higher accuracy. The same experiment was repeated in the simulation environment using 1000 virtual nodes.
Figure 6 shows the agent processing time (blue line) and dynamic updates processing time (red graph) based on number of virtual nodes running on Sensomax. Comparing Figure 4 with Figure 6 highlights a much faster processing and a lack of 35ms latency in the latter.

Finally we focused on the total number of packet loss in the simulation environment as the number of nodes increases. Reliable agent delivery was one of the main concerns of Sensomax architecture. One-hop relaying mechanism decreased packet loss by almost 20%. Figure 7 shows the total percentage of packet loss with (blue graph) and without (red graph) the one-hop relaying mechanism.

IV. CONCLUSION

In this paper we have proposed a middleware architecture for WSNs by exploiting agents-based communication over a multi-layer execution environment, which results in easing programming WSNs, offering support for multiple concurrent applications, applying runtime dynamic changes and increasing scalability. Sensomax takes advantage of object-orientated nature of Java to enable the co-existence multi-agents and multi-operational paradigms in hybrid kernel. Sensomax provides concurrency for multiple applications in resource constraint WSNs, as well as providing multiple overlapping clusters to promote reusability of available resources. Sensomax proved its effectiveness through a number of experiments in simulation environments and using prototypes such as Sun Spots and resource-rich hardware platforms like the Raspberry Pi[14]. Our future works will include embedding market-based algorithms, Game Theory and probability distributions for autonomous resource-allocation, distributing storage and processing in order to conserve energy and improve network longevity in WSNs.

REFERENCES