An Agent-Based Multi-Model Tool for Simulating Multiple Concurrent Applications in WSNs

Mo Haghighi

***Abstract*—Due to hardware resource constraints in WSN nodes, and lack of support for high-level development environments, application developers tend to avoid concurrent object-oriented models in designing WSN applications. In recent years due to new advances in microelectronics and embedded system design, there have been a number of attempts at manufacturing WSN node prototypes with resource-rich capabilities that enable running multiple applications on individual nodes or on groups of them collectively. However, generating realistic results for large-scale concurrent applications requires sophisticated simulation and debugging tools. In this paper we describe a novel simulation tool with multi-model execution environments that overcomes many complexities involved in simulating large-scale WSNs. SXCS, SensomaX Companion Simulator, is a standalone generic simulator particularly targeting agent-based architectures for densely distributed embedded systems. The proposed architecture, unlike many existing models, is not tied to any particular platform and can be fine-tuned through a set of powerful network APIs as well as being capable of hosting multiple highly dynamic virtual environments. It can also act as an emulator taking the entire middleware source code to replicate its core functionalities over a network of up to 2500 virtual nodes. An open-source release of SXCS is planned for later in 2013.**

***Index Terms*—Agent-based, simulator, SXCS, sensomax, multi-operational, multi-application, concurrency**

I. Introduction

There exist several WSN simulators aimed at dynamic environments, however none have capabilities for simulating multiple applications on a single node or a cluster. SXCS had initially been designed for emulating Sensomax [1], [2] middleware, which is an agent-based middleware with multiple concurrent application support for dynamic data gathering in large-scale WSNs. However, due to its many distinct features such as modularity, extensibility, reusability, object-orientation and dynamicity, it was optimized for specialized simulation tasks.

SXCS is a hybrid network simulator that like WISDOM in [3], has been written in Java and has no dependability on any other Java simulation packages such as common discrete event simulators like Jist [4] or J-Sim [5]. The key strengths of SXCS are: 1) Modelling concurrent WSN applications on node, cluster and network levels; 2) Modelling multiple exclusively-defined properties and allocating them to several sets of nodes and clusters; 3) Dynamic Injection of virtual and real sensory data into virtual environments via multiple data generators; 4) Dynamic runtime insertion and execution of applications data requirements, such as computations and aggregations, on node, cluster and network levels; 5) Emulation and simulation of up to 2500 virtual nodes in a multi-clustered fashion; 6) Keeping synchronized records of components' operational status, such as nodes' lifetime, activities' durations, number of transceived packets and etc.

The aforementioned features are intended to satisfy a number of objectives:

1) Providing common communication paradigms including broadcast, unicast and multicast on network, node and cluster levels.
2) Multi-agent intercommunications on all three levels.
3) Running four discrete simulation environments including event, time, query and data, under a unified emulation environment.
4) Splitting applications requirements into time, event, query and data portions and associating them with their relevant simulation environments.
5) Allocating exclusive set of virtual resources to individual/collective virtual environments.

Limitation of hardware resources in WSNs has imposed several major obstacles in design and development of powerful software solutions for such devices. Most of the existing software solutions for WSNs are in the form of middleware and operating systems, and generally lack numerous common functionalities available in conventional systems. Such limitations not only force developers to rely on low-level programing languages, but also require them to learn low-level details of the hardware resources. Also on the deployment-side, using low-level development environment contributes to many complexities in network entity interactions, which results in less autonomy and limits reusability of pre-deployed networks.

Reusability or shared utilization of pre-deployed networks requires developers to either rebuild the network with a new software solution from scratch or build on top of the existing software. In the latter case, this necessitates an adaptable, yet flexible environment where multiple applications can co-exist efficiently. Overcoming the aforementioned complexities have been the major motivations behind the development of Sensomax, in order to provide a high-level and object-oriented development environment. Constructing such an environment has only been made possible through utilization of core java APIs including sockets, threads and memory management, to deliver the features of conventional systems such as multi-tasking, multi-agent communications, serving multiple concurrent applications and dynamic application/network adaptability.

Manuscript received March 25, 2013; revised May 17, 2013.

Mo Haghighi is with the Department of Computer Science, University of Bristol, Bristol, BS8 1UB, UK and Large-Scale Complex IT Systems (LSCITS) (e-mail: Mo.Haghighi@bristol.ac.uk).

DOI: 10.7763/JACN.2013.V1.54

II. Existing Simulators

The primary objective of this paper is to present how concurrent applications can be simulated with regards to their operational paradigms in separated virtual environment. We also mentioned that the concept of concurrency is not very well-defined in WSNs due to lack of qualified hardware. Therefore in majority of research cases, support for multi-tasking and concurrency is seriously overlooked. It was also mentioned that SXCS was originally designed as an emulator for Sensomax, hence its architecture was under great impact of Sensomax architecture. However in order to convert SXCS emulative architecture into an independent simulator, a number of features in the existing simulation tools have been exploited to enhance its usability and competitiveness.

OMNeT++ [6] and ns-2 [7] are the most popular open source network simulating tools. They both offer a number of flexibilities that make them also suitable for WSNs. OMNeT++ in particular, is an interoperable simulation framework rather than an independent simulator; therefore it has become an outstanding foundation for new simulation architectures to be built on top of. Before designing SXCS, OMNeT++ had frequently been used to validate the functionalities and communication interactions in Sensomax architecture. Therefore OMNeT++'s functionalities had a major impact on the design of SXCS.

OMNeT++ or Objective Modular Network Test bed is an event-driven discrete time simulator, which consists of a wide range of components through a sophisticated library that lets users create the simulation environments of their choice in C++ or NED languages. The modular foundation of OMNeT++ is based on two major types of components: *Simple* and *Compound*. Simple components are the ones with fixed functionalities that cannot be extended any further. Compound components however can be consisted of as many simple and compound components as they users desire. Users' interactions can be implemented via two interfaces: a graphical interface (Tkenv) and a command line interface (Cmdenv). Tkenv provides several debugging facilities including *Module Output*: an interface for displaying the outputs from the components both as groups and individually; *Object Inspector*: an interface for displaying and editing objects properties; and *Animated Messages*: displaying messages flowing amongst components. OMNeT++ incorporates popular communication protocols including TCP/IP, FDDI and SCSI through a highly customizable topology creator as well as facilitating multiple operational environments including thread-based and FSM. In the process of designing SXCS, the aforementioned features were considered and combined in a uniform Java package that includes the best of what OMNeT++ offers, in addition to a number of novel capabilities offered by Sensomax, including:

1) Thread-based and FSM-based combined operation
2) Common communication protocols: TCP/IP, UDP
3) Command line and graphical user interfaces
4) Object-oriented deep modular nesting
5) Debugging tools for displaying components interactions
6) Interoperability for various platforms
7) Direct and indirect components interactions

ns-2 however, lacks many features of OMNeT++. Most notably, the process of composing a simulation scenario is very complex and requires thorough knowledge of the underlying classes that have been written in C++ and OTcl. In most cases, making new components requires replicating any underlying C++ classes in OTcl. Out of memory errors are also very common and making any modifications in particular, requires full compilation. Therefore, due to the aforementioned downsides, creating large objective scenarios including Sensomax, proved to be a very frustrating task.

III. Architecture

Sensomax has been described in detail in previous publications [1] and [2]. However since the architecture of SXCS shares a number of components with Sensomax, and in fact most of the functionalities exploited in SXCS are derived from emulating an instance of Sensomax, here the high-level operation of Sensomax as a WSN middleware is briefly described.

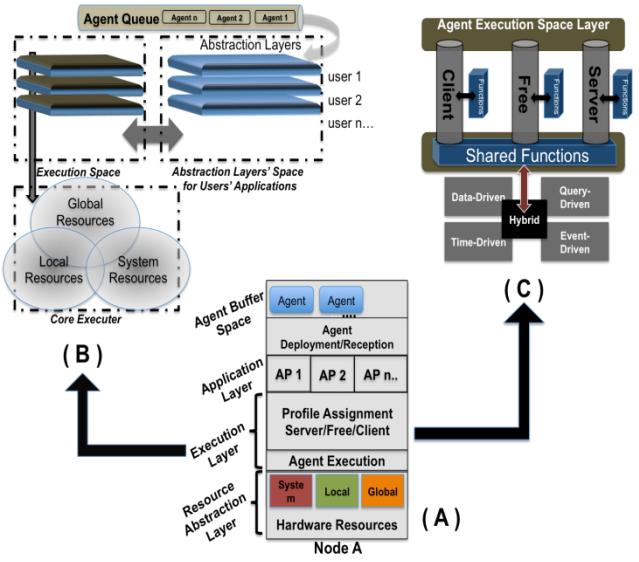

Fig. 1. Sensomax middleware architecture

Fig. 1 (A) shows the overall hierarchical architecture of Sensomax as software layers, where (B) illustrates the execution engine's architecture within the execution layer and (C) depicts the profile assignment layer.

Sensomax creates a collaborative execution environment where multiple applications demands are passed around the network, in the form of agents, attempting to exploit the available resources based on their requirements. It uses a clustering scheme in which the network is divided into multiple clusters, each composed of a collection of resources required for meeting a particular end-user application. Every node allocates an exclusive execution space to each new application agent. This mechanism not only isolates the execution of multiple agents from each other but also helps the node to maintain exclusive roles for each application at network level. Basically, a node can act as a cluster-head for other slave nodes, to manage the execution of a task, and yet the cluster-head can simultaneously play the role of a slave member of a different cluster, helping a different cluster-head execute a different task. This is to ensure the decentralized execution of tasks on network level as well as their centralized execution on cluster level.

Sensomax also abstracts both agents and available resources into three major categories of *global*, *local*, and *system*. This process safeguards exclusive interactions of agents and resources, where each type of agent is only privileged to access the resources of its own type. This mechanism benefited Sensomax enormously, resulting in more rapid runtime updates and better scalability. Therefore, multiple applications run simultaneously on both network and node levels whilst injecting their updates at runtime, without affecting others' processing, and potentially use/reuse the same set of resources. On the application-side, Sensomax refines applications demands into four major categories of *Event*, *Time*, *Query* and *Data* requirements, whilst switching operational paradigm on the node-side (Event-driven, Time-driven, Query-driven and Data-driven) and executing requirements in their relevant execution paradigm. It is worth noting that agents are still executed in their own application spaces within each operational paradigm. The definitions of time and event-driven paradigms are self-explanatory. Query-driven however, refers to instant queries that require executing immediately upon receipt, whereas Data-driven refers to instant queries that consider the node as a database of multiple variables and instruct some data extraction commands to fetch a specific portion of data with or without conditional perquisites.

Theoretically, a node equipped with Sensomax middleware, can serve as many as applications as its memory and processor allow. However in [2] we conducted an extensive research investigating this matter in which, SunSpot devices [8] as prototypes, achieved seamless running of 40 concurrent soft applications with reasonable latencies.

As was pointed out, our proposed simulator, SXCS, was initially developed as an emulator for Sensomax. Its basic original architecture resembled a container with multiple instances of Sensomax source code running in the form of multiple threads that communicate through a number of virtual sockets, and their sensing resources were abstracted as randomized data generators. The role of original SXCS was crucial in debugging and fine-tuning Sensomax for large-scale networks. However addition of more features like dynamic virtual environments, virtual radio channel, real data replicator and more importantly, implementing the mechanism of multiple operational paradigms, transformed it into a standalone simulator that can achieve most of its goals through running an instance of Sensomax.

SXCS allows multiple end-user applications to create their own WSNs with highly customizable environments. It enables end-users to build essential simulation components through a set of provided APIs and package them into virtual nodes, clusters, networks and environments. Those components include virtual hardware resources such as sensors, actuators and I/O ports; network resources such as virtual radio channels and routing protocols and finally computational resources such as market-based algorithms and energy-aware MAC protocols.

In a basic form, every component represents an object in Java, which can be extended further so long as maintaining its parent properties, which are defined by SXCS. This is different from the OMNeT++ in a way that we do not distinguish between simple and compound modules in the way OMNeT++ does. In contrast, SXCS allows all modules to extend the main functionalities so as long as conforming to the design rules.

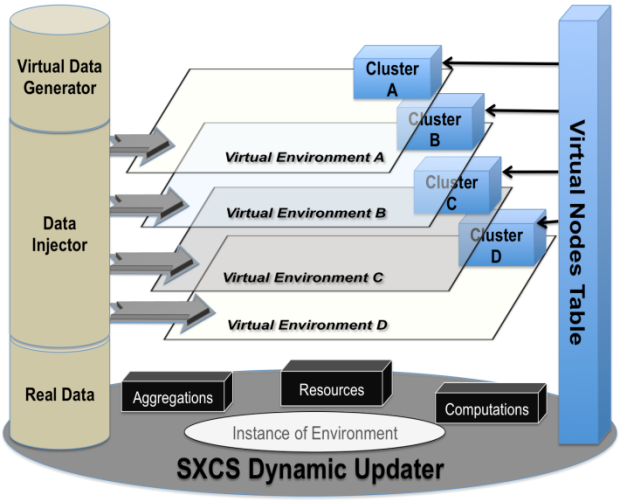

Fig. 2. SXCS high-level architecture

Fig. 2 depicts a high-level architecture of how SXCS operates. Users may create as many nodes as they require (the current version supports up to 2500) with their required properties integrated in them. The same applies to creating virtual environments with required properties. Users are then instructed to associate the generated nodes, individually or collectively, to the created environments. SXCS forms a logical layer, containing all nodes associated to the same environment, representing an independent cluster.

In the next phase, the *data-generator* component comes into play, through which users can disseminate sensory data to virtual environments. The data-generator can switch between real and randomly generated data, with customizable user-defined distributions. Data generator disseminates data throughout the environment in order to be received by the associated clusters. The same process applies to injecting computation, data aggregation and resource management.

All the aforementioned injection tasks are managed by SXCS Dynamic Updater module. All the interactions amongst these modules are visible through a terminal interface and results of each computation process can be viewed for each node individually or in groups collectively.

To systemically clarify component-based interactions in the aforementioned operations, every component represents a Java object, including ***QNode***.*java*, ***QCluster***.*java*, ***QEnvironment***.*java*, ***QAggregate.***.*java*, ***QComputation***.*java* and etc., each containing variable amount of codes, which are customizable by the users. These modules technically either extend or interface a parent class in Java, which is defined by SXCS and not customizable by the users. Parent classes basically define their communication domain and how they interact with other system modules.

In this section, SXCS operation goes lower by one level to illustrate how components interactions are implemented and to describe how the simulator engine operates.

WSN applications generally differ in terms of their operational paradigms. Existing WSN middleware support one or two operational paradigms such as Mate [9] and Impala [10], that follow an event-driven paradigm, or SINA [11] and COUGAR [12], which represent highly-coupled

combinations of data-driven and Query-driven paradigms. The same applies to WSN simulation tools where most are discrete-event-based simulators. Therefore to simulate different middleware, SXCS needs to offer an adaptable environment in which various behaviour models can be simulated. This is one of the challenges faced by many simulators where running multiple simulation models concurrently affects systems performance and realistic near-real-time results are hard to obtain. Many existing simulation tools attempt to solve this problem by customizing the middleware behaviour in a way that fits in Discrete event model.

As we mentioned, Sensomax is the first approach in WSN middleware design in which all four major operational paradigms are implemented under one hood in order to meet various application types concurrently. Since SXCS fundamentally extends Sensomax architecture, by nature it incorporates all four simulation models: *Discrete Event Simulation*, *Discrete Time Simulation*, *Discrete Data Simulation* and *Discrete Query Simulation*.

Such multi-model simulation offers a number of advantages to the applications, environments, nodes and the network:

1) Allowing more realistic interactions between multi-operational applications.
2) Allocating tailored execution model to each application based on its architecture, in which, true behavioural patterns can be analysed.
3) Enabling virtual environments to be simulated individually/collectively based on their own exclusive operational paradigms.
4) Displaying more realistic and faster interactions between the environments and the applications with different/same multi-operational behaviours.
5) Allowing concurrent evaluation of multiple nodes with different/same operational paradigms.
6) Faster delivery of applications requirements/updates to both nodes and environments.
7) Observable impacts of paradigm-shifting on multiple levels.

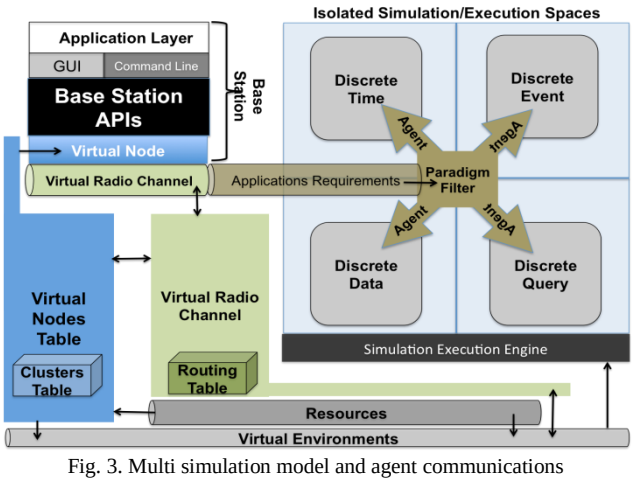

Fig. 3. Multi simulation model and agent communications

We evaluate these advantages both in emulation and in simulation, as discussed in the evaluation section.

Fig. 3 illustrates the execution of multiple simulation models in SXCS as well as multi-agent communications among components. The **Virtual Nodes Table** (VNT) is the most fundamental component, which generates and holds all virtual nodes and their clustering properties.

There always exists a node known as the **Base Station**, which handles all users' interactions with the simulator. The VNT provides a virtual node instance to the base station from which the base station can act as an independent virtual node with no associated environment or characteristics. There is also a set of APIs, provided for the users, aiming to ease the process of programming simulation scenarios.

Like Sensomax, all components' communications in SXCS are implemented through extensive exchange of multi-agents on all levels. The **Simulation Execution Engine** component runs all four models concurrently and provides a filtering component in which application requirements are filtered out to the relevant execution model. Fig. 3 only shows this filtering process for the application requirements, however the same operation applies to all the agents delivered to both environments and resources as well. Simulation Execution Engine is a multi-threaded process with an instance of Sensomax, in a fine-grained form, integrated in each thread. Each thread represents a Finite State Machine that takes in a type of agent that matches its operational paradigm.

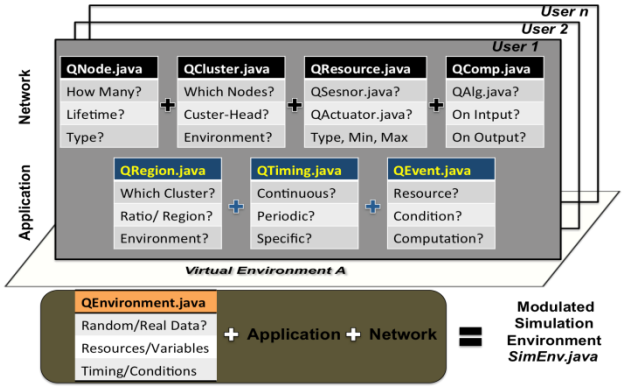

Fig. 4. Composing simulation environments through combining modules

Inter-components communications are all implemented in an indirectly through the **Virtual Radio Channel (VRC).** Virtual Radio Channel acts an intermediary between the virtual nodes and the rest of the architecture. It assigns a logical port and a 16bit address to every node and forms a routing table by which every logical port is associated with an address. Virtual Radio Channel also provides an extensive set of common communication protocols including *broadcast*, *unicast* and *multicast*. Application developers can access and customize all network entities as well as the VRC, through a set of provided APIs. Two major components that considerably simplify the process of designing communication protocols for the application developers are ***PacketTransmitter***.*java* and ***PacketReceiver***.*java*. Developers can utilize these modules in combination with any other components to manage inter-components interactions for their applications.

There exists another important module in charge of resource allocation. ***QResource***.java represents a repository for all existing resources including sensors and actuators. Users may define a resource of their choice, or by default select one of template. A typical resource definition involves

setting the type of sensing variable as well as a minimum and a maximum value.

We have described the major components in SXCS and how the overall interaction works. To further clarify how users can compose a simulation environment and design an application using the modules, Fig. 4 illustrates how interconnecting modules are bundled together. Based on this figure, the composition of a simulation environment involves three major stages: Defining the network; building the application; and setting up the virtual environment. The same process applies to building the application in which users can assign the nodes that the application is going to interact with, as well as their conditional and timing requirements.

Due to the length restriction of this paper, a number of details on the operation of SXCS have been omitted.

IV. Evaluation

In this section a number of experiments have been conducted to evaluate SXCS characteristics against OMNeT++ in terms of energy, memory and performance profiling. We will also validate the effectiveness of applying multiple operational paradigms to the simulating environments.

As we mentioned before, it is not feasible to simulate multiple concurrent applications on a single node in OMNeT++. However, since concurrency is the primary focus of SXCS, it was imperative to compare SXCS's multitasking performance against OMNeT++. Hence we managed to construct a very weak resemblance of SXCS's concurrency model, by combining numerous compound modules under OMNeT++. That approach however, resulted in poor performance; therefore we reconstructed it differently, by combining agents in a collaborative multi-level model.

For the first experiment, a virtual environment, containing 20 variables was constructed in SXCS and OMNeT++, and between 10-1000 nodes were associated with the aforementioned environment, in an incremental step by step joining process. For the first phase, the multi-operational paradigm of SXCS was disabled in order to keep it compatible with OMNeT++. In the next phase we evaluated the same scenario under SXCS only, this time with operational paradigm selection enabled.

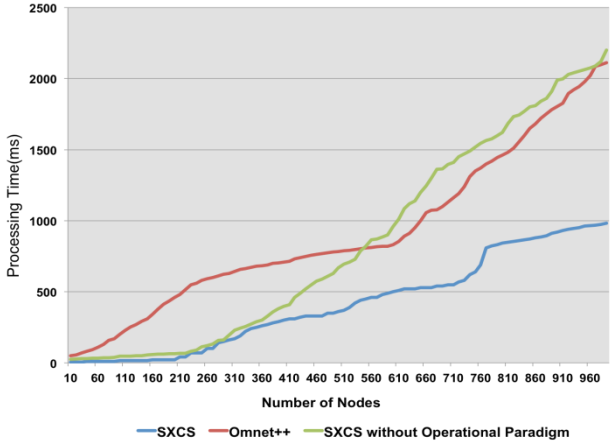

Fig. 5. Agents processing time vs. number of nodes

Fig. 5, shows the average agent processing time of 20 agents between 10-1000 nodes. Green line shows the processing time of agents in SXCS, whereas the red line presents the same processing time in OMNeT++. The blue line, on the other hand, shows the same process in SXCS, with the operational paradigm selection enabled. As the red line denotes, the agent processing time for up to 570 nodes is lower under SXCS, and only slightly higher, after exceeding 5870. Also the blue line shows, the latency will be much lower with the operational paradigm selection.

In order to validate the energy profiling of SXCS against our prototypes and OMNeT++, we conducted the second experiment using 14 Sun Spot nodes in which they were left running for 80 hours with a heavyweight application that required frequent sensing and transmission both globally and locally within the network. We constructed the same environment in OMNeT++ and SXCS in order to check the energy deprecation against the real-time application on Sun Spots.

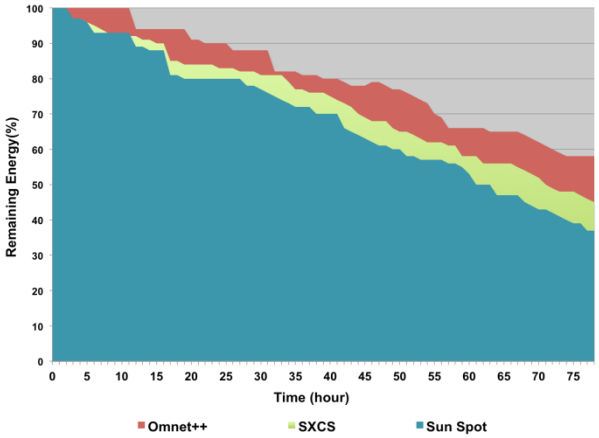

Fig. 6. Remaining energy profiling

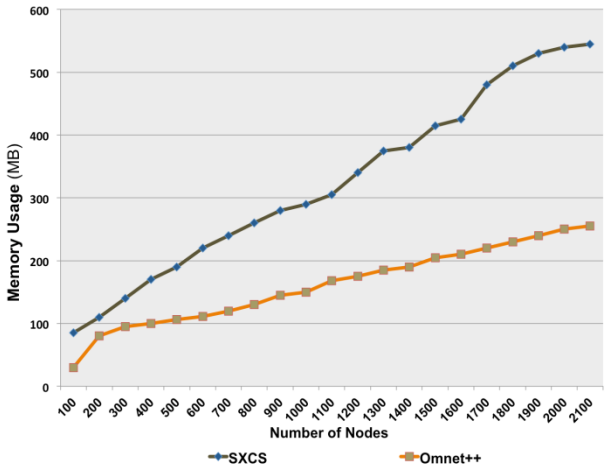

Fig. 7. Memory usage vs. number of nodes

As Fig. 6 shows, the turquoise histogram represents the remaining battery level of Sun Spot nodes from 100% in the beginning of the experiment until after 80 hours of operation with 40% battery remaining. Green and red histograms show the energy profiling of the same experiment under SXCS and OMNeT++ respectively. As this figure shows, SXCS results are closer to the Sun Spot results and therefore SXCS did a better energy estimation.

The amount of memory used by different simulators is one of the important factors in the simulators' performance. As we mentioned in section 2, ns-2 suffers a number of inefficiencies in terms of memory consumption and users

often encounter with out of memory errors.

In SXCS, the complex task of memory management is systemically assigned to the JVM. However for the third experiment we made an analysis on the memory consumption of OMNeT++ and SXCS.

In Fig. 7, orange and brown lines represent the memory usage, with respect to the virtual network density, for OMNeT++ and SXCS respectively. As can be seen, OMNeT++ takes noticeably less memory, with 2100 nodes merely taking over 250 MB, whereas SXCS occupies over 500 MB. This considerable difference is due to the higher memory footprint of Java.

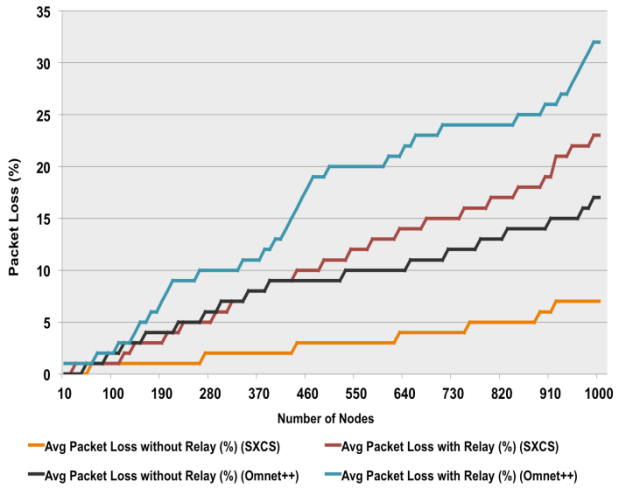

Fig. 8. Packet Loss vs. number of nodes

Good scalability is one of the features of Sensomax, which has been achieved by relaying packets to immediate one-hop neighbours. There are a number of mechanisms around it that have been thoroughly explained in [2]. For our fourth experiment, we evaluated the packet loss of OMNeT++ and SXCS with and without Sensomax's relaying mechanism. In Fig. 8, red and orange lines show the SXCS packet loss ratio in percentage with and without relaying respectively. Blue and brown lines on the other hand, represent the packet loss ratio with and without relaying in OMNeT++ respectively. Based on Fig. 8, both SXCS and OMNeT++ demonstrate lower packet loss without relaying. Relaying packets creates large backlogs of waiting packets in both OMNeT++ and SXCS, which results in more packet loss. In SXCS this bottleneck is created in the VRC, whereas in OMNeT++ that is imposed on the gates.

V. Conclusion

In this paper we have described the architecture of SXCS, which is an agent-based multi-operational simulator for simulating multiple concurrent applications in WSNs. SXCS features a user-friendly interface as well as offering an extensive library of predefined components, through which, up to 2500 virtual nodes over multiple virtual environments can be constructed in Java. In order to obtain more realistic results, SXCS incorporates four isolated execution environments where simulation scenarios are broken down into time, event, query and data requirements for separate executions. The effectiveness of this approach has been validated against OMNeT++, which is one of the most popular simulation frameworks.

References

[1] M. Haghighi and D. Cliff, "Sensomax: An Agent-Based Middleware For Decentralized Dynamic Data-Gathering in Wireless Sensor Networks," in *Proc. The 2013 International Conference on Collaboration Technologies and Systems*, CTS 2013, May 2013.

[2] M. Haghighi and D. Cliff, "Multi-Agent Support for Multiple Concurrent Applications and Dynamic Data-Gathering in Wireless Sensor Networks," in *Proc. Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, IMIS-2013, July 2013.

[3] H. B. Lim, B. Wang, C. Fu, A. Phull, and D. Ma, "WISDOM: Simulation Framework for Middleware Services in Wireless Sensor Networks," in *Proc. 5th IEEE Consumer Communications and Networking Conference*, pp. 1269-1270, January 2008.

[4] I. Mahgoub, A. Badi, and M. Ilyas, "Design and implementation of parallel JiST to support distributed wireless network simulation," *High-Capacity Optical Networks and Enabling Technologies (HONET)*, December 2010, pp. 154-160.

[5] J-Sim. [Online]. Available: http://www.j-sim.org/.

[6] A. Varga. *OMNET++ Discrete Event Simulation System Version 3.2 User Manual*. [Online]. Available: Available: http://www.omnetpp.org.

[7] NS-2. [Online]. Available: http://www.isi.edu/nsnam/ns/

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

[9] P. Levis and D. Culler, "Mate: a tiny virtual machine for sensor networks," in *Proc. the 10th International Conference on Architectural support for programming languages and operating systems ASPLOS-X*, 2002, ACM Press, pp. 85-95.

[10] T. Liu and M. Martonosi, "Impala: A middleware system for managing auto-nomic, parallel sensor systems," in *Proc. ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 2003.

[11] C. Shen, C. Srisathapornphat, and C. Jaikaeo, "Information Networking Architecture and Applications," *IEEE Personal Communications*, 2001, pp. 52-59.

[12] P. Bonnet, J. Gehrke, and P. Seshadri, "Towards sensor database systems," in *Proc. 2nd International Conference on Mobile Data Management (MDM)*, LNCS vol. 1987, 2001.

Mo Haghighi is a doctoral researcher at the University of Bristol, UK. He is currently pursuing his research in the area of "Decentralized Agent-based Adaptive Dynamic Data Gathering in Large-scale Wireless Sensor Networks". He has obtained a BEng in Electronic and Telecommunications engineering followed by an MSc in Wireless Sensor Networks. He began his research in a joint collaboration between the University of Bristol, the BAE Systems and Large-Scale Complex IT Systems (LSCITS). Prior to his PhD, he had worked for Sun Microsystems/Oracle for over two years, primarily involved in academic projects. As a member of LSCITS, his research has broadened to include complexity science, Cloud computing, multi-agent systems and quantitative data analysis for large-scale complex systems. Mo has extensive programming experience in Java, C++ and Assembly. He also specializes in designing embedded systems (ARM, Freescale, Microchip and Intel), large-scale distributed systems, network security, machine learning, LoWPANs and Microwave communications.