02/01/2005

LSU SensorSimulator

User Manual
Version 1

Sensor Networks Research Group,
Department of Computer Science,
Louisiana State University, Baton Rouge, LA.

For details contact:

- Dr.S.S.Iyengar, **iyengar@bit.csc.lsu.edu**
- Vatsalya Kunchakarra, **vkunch1@bit.csc.lsu.edu**
- Ankur Suri, **asuri1@lsu.edu**

Table of Contents

1. Introduction
   1.1 Simulators in Sensor Networks
   1.2 What is SensorSimulator?
   1.3 SensorSimulator on OMNeT++

2. Overview of the Framework
   2.1 Concept
   2.2 Using SensorSimulator and Directory Structure

3. Building Simulations

4. Running Simulations in SensorSimulator

5. Implementation of Directed Diffusion and IEEE 802.11

1. Introduction:

1.1 Simulators in Sensor Networks:

The multitudes of design challenges imposed on Sensor Networks tend to be quite complex and usually defy the analytical methods that are quite effective for traditional networks. At current stage of technology very few Sensor Networks have come into existence. Although there are many unsolved research problems in this domain, actual deployment and study is infeasible. The only practical alternate to study Sensor Networks is through simulation, which can provide better insight to behavior and performance of various algorithms and protocols. The goal of SensorSimulator is to provide a framework to closely model and simulate various Sensor Networks scenarios.

The framework of SensorSimulator has been built on the OMNeT++ simulation environment.

1.2 What is SensorSimulator?

SensorSimulator is a framework developed on OMNeT++, mainly intended to support Sensor Network Simulations. The framework provides basic modules that can be derived in order to implement own modules. With this concept a programmer can easily develop own protocol implementations for the SensorSimulator Framework without having to deal with the necessary interface and interoperability stuff.

The source files have the structure of a basic Sensor Network. We are currently developing a library of standard protocols for the SensorSimulator (802.11, Directed Diffusion with GEAR). The framework is under development and we will be updating the code on a regular basis. Our goal is to have a rich library of such protocols to enable easy plug-and-play simulations of various kinds of widely used protocols.

1.3 SensorSimulator on OMNeT++:

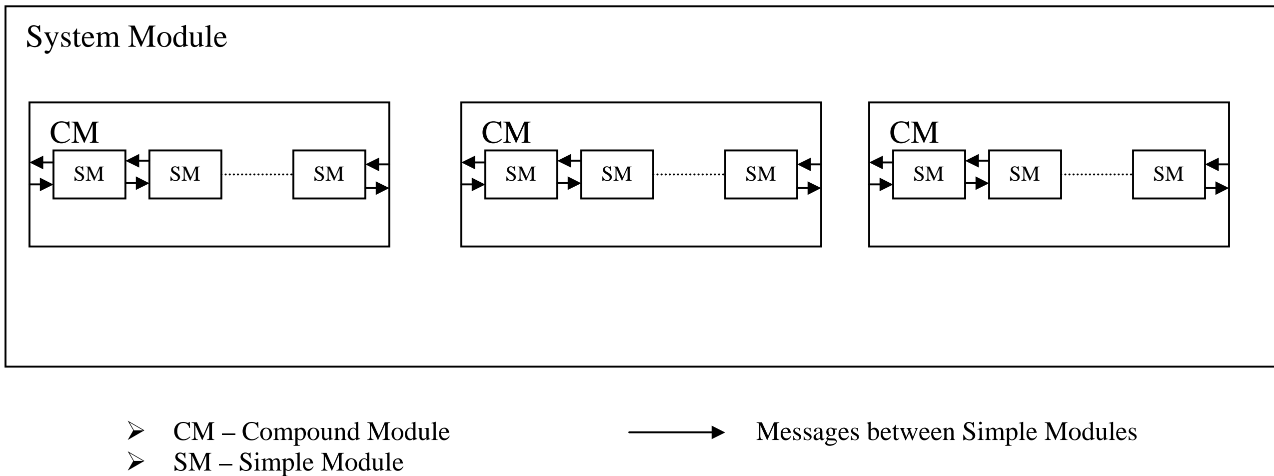

Figure 1: Simple and Compound modules in OMNeT++

Objective **M**odular **N**etwork Test-bed in C++ (OMNeT++) is a public-source, component-based, modular simulation framework. It is has been used to simulate communication networks and other distributed systems. The OMNeT++ model is a collection of hierarchically nested modules as shown in Figure 1. The top-level module is also called the System Module or Network. This module contains one or more sub-modules each of which could contain other sub-modules. The modules can be nested to any depth and hence it is possible to capture complex system models in OMNeT++. Modules are distinguished as being either simple or compound. A simple module is associated with a C++ file that supplies the desired behaviors that encapsulate algorithms. Simple modules form the lowest level of the module hierarchy. Users implement simple modules in C++ using the OMNeT++ simulation class library. Compound modules are aggregates of simple modules and are not directly associated with a C++ file that supplies behaviors. Modules communicate by exchanging messages. Each message may be a complex data structure. Messages may be exchanged directly between simple modules (based on their unique ID) or via a series of gates and connections. Messages represent frames or packets in a computer network. The local simulation time advances when a module receives messages from another module or from itself. Self-messages are used by a module to schedule events at a later time. The structure and interface of the modules are specified using a network description language. They implement the underlying

behaviors of simple modules.. Simulation executions are easily configured via initialization files. It tracks the events generated and ensures that messages are delivered to the right modules at the right time.

To take the advantage of the above features of OMNeT++ we have chosen it as the framework for Sensor Network Simulations. Its salient features include:

- OMNeT++ allows the design of modular simulation models, which can be combined and reused flexibly.
- It is possible to compose models with any granular hierarchy.
- The object-oriented approach of OMNeT++ allows the flexible extension of the base classes provided in the simulation kernel.
- Model components are compiled and linked with the simulation library, and one of the user interface libraries to form an executable program. One user interface library is optimized for command line and batch-oriented execution, while the other employs a graphical user interface (GUI) that can be used to trace and debug the simulation.
- OMNeT++ offers an extensive simulation library that includes support for input/output, statistics, data collection, graphical presentation of simulation data, random number generators and data structures.
- OMNeT++ simulation kernel uses C++ which makes it possible to be embedded in larger applications
- OMNeT++ models are built with NED and omnetpp.ini and do not use scripts which makes it easier for various simulations to be configured

The topology of the Sensor Network field in our simulations is derived from the Simple and Compound Module concept of the OMNeT++ framework. As shown in Figure 1, layers of a node behave as Simple Modules and a Sensor Node behaves as a Compound Module and all these Sensor Nodes constitute the Sensor Network depicted as System Module.

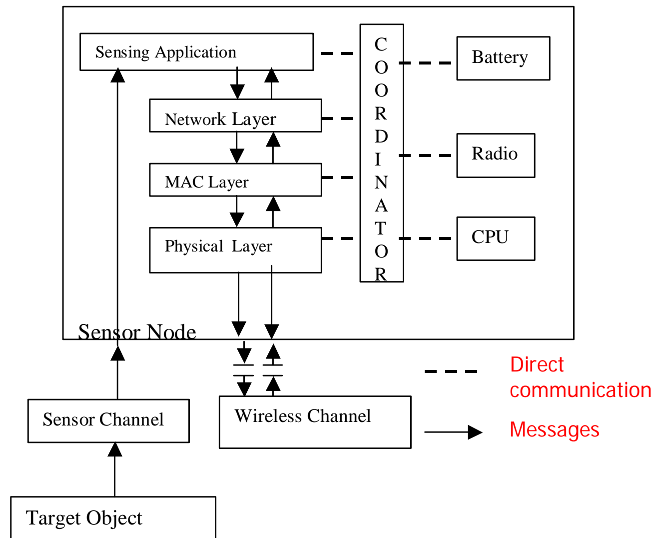

Figure 2: Basic Structure of the Sensor Node in our Simulator Structure

2. Overview of Framework:

The section will give the basic framework of SensorSimulator on OMNeT++. User needs to know about programming in OMNeT++. If not you should read the OMNeT++ manual (*http://www.omnetpp.org/*). The manual then explains how to install SensorSimulator on OMNeT++.

2.1 Installation:

You need a running OMNeT++ version 2.3 to use the LSU SensorSimulator with all of its functionality. This is available from omnetpp.org site. User needs to go through the User manual of OMNeT++ to run simulations on our Simulator. After downloading the most recent version of the SensorSimulator from the download area of LSU Web Site, copy the file to the desired directory. `cd` to this directory and

- tar xzf LSU-SensorSimulator.tgz
- cd LSU-SensorSimulator

- There are 3 directories – inc,samples,src with Makefiles already created.
- Set the path of the src directory to LD_LIBRARY_PATH
- Makefiles are created using opp_makemake of OMNeT++. Open each Makefile, copy the opp_makemake command which is shown commented at the starting of Makefile and execute the command again with the changed paths. A new Makefile with changed path will be created and then do "make".
- Repeat the procedure in the sub-directories of samples also.
- Then execute the simulation using "./Simulation" in samples directory.

Then user can implement his protocols and run the simulation with the executable ./Simulation.

2.2 Directory Structure:

Our directory structure is divided as follows:

LSU-SensorSimulator/

inc/	This has the header files of base classes
src/	This has the base classes for all the layers

CoOrdinatorBase

LayerBase

PhyLayerBase

MacLayerBase

NetLayerBase

AppLayerBase

RadioBase

BatteryBase

CPUBase

WirelessChannelBase

TargetNodeBase

cConsumer

samples/	This has the simple classes derived from base classes. Also includes the implementation of Directed Diffusion with GEAR and MAC-802.11.

phy_layer/
mac_layer/
netw_layer/
app_layer/
hwmodels_layer/
wireless_ch/
common/

Each sub-directory has implementations of that layer.

- Hardware models include Battery, CPU and Radio modules. These include simple hardware models.
- Common directory has CoOrdinator, packets structures for Network and Mac layer and other constants and attributes used for simulation. This directory is derived by all other directories of the sample folder.
- The Wireless Channel sub-directory has a Simple Wireless channel which introduces a

  Delay = Distance between the two communicating nodes/ Speed of Light

The class name has to be specified in the omnetpp.ini file if user wants to try the other implementations specified above.

2.3 To try a new implementation:

1. Every layer has three basic files: .cc,.h and .ned files. User has to have all these three files in a layer for his implementation. We are providing these three files for every layer with minimum functionality. User can add his code and just do "make" and run the simulation by changing the necessary configurable parameters in the omnetpp.ini file in the samples directory.
2. **Procedure**: Move into samples directory. This directory has sub-directories each for a layer. Each directory has examples of various implementations at each layer. Also, you can see the files New* Layer.cc, New*Layer.h and New*LayerModuleDefn.ned for a new user to start using the simulator. You can

add the various parameters of a module which you will be using in your implementation in the .ned file.

After adding the code or making changes for the existing implementation, do make and then go back to samples directory.

The class name of this new implementation has to be specified in omnetpp.ini so that the simulation considers your implementation for execution.

Eg: sensorNetwork.Nodes[*].strMACLayerType="MAC_802_11"

With this functionality, user can just add his protocol at that particular layer with all other protocol layers being same.

3. Building Simulations:

This section explains the basic concepts behind the SensorSimulator. The class hierarchy is explained and all relevant functions of the base modules are introduced. Detailed description is also available in the API reference.

Base Layer Concept:

The functionality of any layer of a node is defined in the LayerBase file. This itself gets the properties from the SimpleModule of OMNeT++. All the base classes of the node are derived from LayerBase. This is defined in the src directory.

Base Modules:

We provide base modules for each layer, which gets the properties from LayerBase. It is primarily implemented to clearly define the interface that can be understood easily and which can be extended if necessary. It implies that these source code files take care of the basic functionalities of that layer and the user need not go through all this for his implementation. He just has to derive his class from these base modules. These modules have header files in the "inc" directory. The ned files and the source code files are available in the "src" directory.

Simple Modules:

A simple functionality of each layer is defined here. These are derived from Base Modules. User has to implement his protocols at this level. These are available in the "samples" directory. The Simple source code files for each layer are available in each

sub-directory of samples directory with respect to that layer. The .ned files for these simple source files are available in the “src” directory. The .ned files for new implementations are available in the sub-directories itself.

4. Running Simulation:

We can run the simulation after all the layers are defined by configuring the parameters in the omnetpp.ini file. After writing your code at various layers, do make in that corresponding sub-directory. After the make is done successfully, get back to the samples directory and change the parameters in the omentpp.ini file. The topology, simulation time etc can be varied in this file. These sections are clearly explained in the OMNeT++ manual. Change the class name of the module you created (as discussed in section 3) and run the simulation. You can redirect the output to another text file.

5. Implementation of Directed Diffusion with MAC-802.11

We have implemented Directed Diffusion along with Geographic Routing. The AppLayerSimple generates interests that specify the region, the kind of data required and rate of delivery of data. The structure of a query is shown below. Nodes that initiate the interest are called Subscribers. On receiving the interest message, the broadcasts beacon messages in the network.

Query -	attribute
Query -	rate of data
Query -	duration

The immediate neighbors of the node on receiving beacon messages reply back with beacon-reply type of message that contains their geographic location and the energy left in them. On receiving the beacon-reply messages, the neighbor table of the node that sent the beacon is updated. The node waits a fixed duration of time to receive the beacon-reply from all the neighbors. The interest message is then forwarded to the node that has a lower estimated cost to the region as calculated by the GEAR protocol. The next node follows the same procedure and forwards the message towards the region by Geographic Routing. If a node in the path does not have any neighbors or all its

neighbors are away from the region, then it sends a message to its parent node that it is a dead-end. The parent node on updating the cost of the unreachable node, forwards the query in an alternate route towards the region. In the target region, the interest is disseminated by using recursive flooding. The interest cache is maintained at each of the nodes in the path with its gradient of interest to each of the neighbors. The nodes in the region that have the specified properties of the interest send out data. Nodes that send data out are referred to as Publishers.

The data is marked as Exploratory to reinforce the path that was taken by the interest. On receiving the data marked as Exploratory by the subscriber, a positive reinforcement message is sent out by the Subscriber node. Each node on the path forwards this message - thus reinforcing the path to the region. When a node reinforces a path, its cost to the region is known and this cost is sent back to its source node, which updates the cost information of that node to the particular region of interest. Thus the path with the lowest cost is always maintained, reinforcing the route.

The data from the region follows the path established by the reinforced messages. The nodes in the region send out data at the rate that is specified in the query. Data caching is implemented in intermediate nodes and so the data requested by different subscribers from the same region can be satisfied by the common node in the path thus reducing the traffic and redundant messages. The data marked as exploratory are sent to identify better paths and reinforce at regular intervals. Also the neighbor- updating procedure is carried out, i.e. at regular intervals the beacon messages are broadcast and beacon-reply messages are sent by neighbors thus maintaining latest neighbor information.

The MAC layer with 802.11 places the Network packet on the Wireless Channel. The NetworkPacket maybe a broadcast or unicast packet to a specific node (sink node).

Any network layer packet received by the MAC-802-11 module is encapsulated into MAC frame with the MAC header added to it. The Network layer packets have the information whether the packet has to be broadcast or unicast. Broadcast packet is encapsulated into Broadcast MAC frame with appropriate MAC Header and is put in the Messages-queue of the MAC Layer. If the Network packet is for a particular destination, RTS frame is created and is inserted in the Messages-queue of MAC layer. If the Network packet length is more than the MAC frame, it is fragmented and the fragments for that Network Packet are created with MAC headers and are inserted into the Fragments Queue.

The MAC layer then waits for the channel to be idle to send its frame from the Messages-queue. MAC layer has a NAV Timer, which specifies the busy/idle state of the medium. NAV Timer set for a node implies that the channel is busy. When the NAV Timer expires the MAC layer waits for the channel to be free for DIFS time and if the channel is still idle after DIFS timer gets expired, it then goes into Exponential BackOff. It then waits for a random time set by the BackOff Timer. The BackOff Timer decrements its value during the idle period of channel. The node whose BackOff Timer expires earlier will get the chance to transmit its next frame. All the intermediate nodes receive this frame, set their NAVTimer to the value obtained from the Header field of the received frame. Then the BackOff Timer of the intermediate nodes is stopped from decrementing. Once the channel becomes idle (when the NAVTimer expires) all the nodes start decrementing their BackOff Timer. The node whose Back Off Timer expired earlier and got the channel will send the first message from the Messages Queue. If it is a broadcast message, then all the nodes in its region receive it and the MAC layer of those nodes decapsulate the Network packet and send it to the Network Layer. If it is a RTS frame, the Destination node checks whether its NAV timer is set or not (its transmission

region is busy or not) and then responds to it by sending CTS. All the other intermediate nodes receiving this RTS update their NAV Timer to the CTS+DATA+ACK duration which implies that the channel is busy for that duration and hence refrain from transmitting during this interval. If the Destination node receives more than two RTS requests within a time interval then collision occurs and the Destination node does not respond (send CTS) to any of these RTS requests. The Source node which is sending RTS have an RTSExpired Timer set for RTS frames, when they are sent to the Destination node. This timer is scheduled to expire after RTS+CTS duration. If the Source node does not receive CTS within this duration, RTSExpired Timer gets expired and retry counter of that RTS frame is incremented. If the retry counter is less than ShortRetyLimit (as per the specification), then the Contention Window is doubled and the random time set by the BackOff Timer is chosen between 1 and the Contention Window size. If the retry counter reaches ShortRetryLimit then the message (RTS and corresponding Fragment) is dropped by the MAC.

If the Destination node responds to RTS by sending back the CTS, the intermediate nodes for CTS will update their NAVTimer obtained from the Header field of CTS frame (Data+Ack duration) and hence refrain from transmitting during this interval. Once the Source node gets the CTS, it will send the corresponding fragment of the Network Packet to the Destination and waits for an Acknowledgement. The Destination node upon receiving the Data frame extracts the Network packet, sends it to the Network layer and sends back the Acknowledgement to the Source node. Once the Source node gets the Acknowledgement it checks and sends if there are any other fragments to be sent to this node without any additional RTS frames.

We conducted various experiments to verify the data received by the Subscribers from Publishers. This Delivery Ratio is 100% for smaller networks. The delivery ratio decreases for larger networks according to Directed Diffusion.

User can run Directed Diffusion with 802.11 by including the class file name in the omnetpp.ini file. The topology, number of queries, number of nodes in region, the region size and the simulation time are configurable in the omnetpp.ini. User can also run various simulations using different seeds as described in the OMNeT++ manual.