WIRELESS SENSOR NETWORK SIMULATOR

Problem description

1.1 Abstract

As descripted within the documents available, the software's first objective is to properly simulate the behaviour of a real sensor network. One sensor communicates in a distributed way with others via wireless connections. This configuration is more likely exposed to physical attacks: a node can be cloned and used to send fake data. The simulator should initially be able to run RED and LSM protocol in order to obtain an ambient to study and eventually improving the protocols.

1. Executables

As from the documentation, the software is composed by a server application and a client application. The client runs the simulations accordingly to the user's preferences, and sends statistics for each simulation run. The server reiceves the statistics and fills an output file.

2.1 Guide

This section covers a small user guide to setup the system and run the simulations.

2.1.1 Client Application

The client application starts up opening the control panel and the log panel. The control panel lets the user setup and control the flow of the simulations. The log panel shows the user the current status of the simulations. The simulations cannot be started unless the following are all satisfied:

1. All of the settings must be set either loading them from a configuration file or defining them manually.

2. The client must be connected to a remote and compatible RMI server

When the simulations start, the system will open the ambient panel which contains a graphical view of the nodes. The control panel can be used to control the ambient panel, showing or hiding additional information within the ambient panel.

Please use correctly the "fetch settings" and "lookup server" buttons to successfully start the simulations!

2.1.2 Server Application

The server application consists in a simple GUI which only shows the data received from the client application and writes the output file.

2.1.3 Compiling and executing (root folder is the extracted archive consigned)

There is no need to start the registry from the terminal. I have included 2 jar files that can be executed with JRE. They can be found in *root/jar*. To manually compile the source code just position to *root/server-sorce* and *root/client-source* and run:

user@computer[root/(server-source|client-source)]> *javac */*.java*

To start the compiled client:

user@computer*[root/client-source]> *java -cp . gui.UserInterface

To start the compiled server:

user@computer*[root/client-source]> *java -cp . ServerRMI.RmiServer

2.1.4 Reccomendations

While testing the application under hard settings (# of nodes > 2-300) please keep in mind that this software's GUI is slowing the overall speed of the simulation by a 10x multiplier (statistically). Simulations with more than 300 nodes should be run without mantaining the interface, otherwise it would take a lot of time running a good number of simulations.

The Control Panel offers various settings to study deeply the simulations:

1. "Paths" will draw the paths of the messages successfully sent between the nodes.

2. "Idles" will notify in real time which nodes are currently working or idle.

3. "IDs" will show the ids of the nodes.

4. "Ranges" will color up the nodes and show their transmission range according to the color of the node.

5. "Highlight" in couple with choosing an ID from the select box, will highlight the selected node.

Please take note that these are useless while the GUI is disabled.

6. Architecture Design

The application has been developed following the guidelines learned from the course itself. This section explains an high level overview of the system architecture.

3.1 Client packages

- *CommonInterface - Contains the shared interfaces between the server and the client application*
- *Enums - Contains the enums used through all the classes*
- *Exceptions - Contains the custom exeptions*
- *Gui - Contains the classes used to compose the user interface*
- *Logic - Contains the components for the business logic*
- *Utilities - Contains utility classes*

3.2 Server packages

- *CommonInterface - Contains the shared interfaces between the server and the client application*
- *Gui - Contains the classes used to compose the user interface*
- *Logic - Contains the components for the business logic*

3.3 Design patterns

3.3.1 Singleton

Singleton pattern is used to force a single instance for a class. It was used in a good number of classes to have static methods defined for external request.

3.3.2 Facade

Facade pattern is used to have a passtrough class which serves as interface for the communication between the Logic and the Gui packages

3.3.3 Abstract Factory

3.4 Problem modeling

The objective of the software is to write statistics concerning the simulations while studying the possible problems of the protocols. The modeling of the problem is really important to achieve good results. The following sections describe the details of the most important models for the simulations.

3.4.1 Design choices

1. *Object-orientation over perfomance:* Object oriented programming means expandibility, better modeling of the problem and quality of source code but produces slower software. The class structure and algorithms of this software are not optimized to speed up the simulations in order to produce clear and easily mantainable source code.

2. *Multi-windowed GUI:* Can cause some problems with older resolutions (800 x 600), but more logically connected to the classes from the source code.

3.4.2 Wireless communications

In real world situations the sensors are placed and communicate with each other through the ambient. The *Logic.Ambient* class acts as the "air" in the real world. All the messages pass through this class. Wireless communications are not as strong as wired cables, and can have problems that can be simulated through the class.

3.4.3 Sensors

Sensors are modeled through the *Node* hierarcy. The root class achieves the standard behaviour of a node for the most common actions. The subclasses specialize the processes of the particular protocol to simulate. This guarantees the future extendibility for possible new protocols to be implemented.

Sensors are implemented as threads.

3.4.4 Messages

Sensors communicate with each other sending messages through the ambient. The messages are modeled through the *Message* hierarcy. The hierarcy provides the flexibility for eventual new implementations. Some of the features included in these hierarcy were used in older versions of the software that have been removed.

3.4.5 Simulation hypervisor

The simulation hypervisor acts as an observer that react to specific events occurring in the simulation. The hypervisor is not related in any way with nodes and in a real world situation would not exists. It can be seen as some sort of witness taking care of the overall status of the nodes.

3.4.6 Server

As from the documentation, the server of this software only writes an output file. As the source code for this part of the system does not cover any of the concurrent topics of the course, I preferred to keep it straight simple. It offers only one method to remotely push strings to the output file.

5. Concurrency

The project covers some concurrent situations that are worth to mention in this document.

5.1 Simulation termination

The termination of a simulation depends on random factors and it was not simple to achieve. It was not enough to check for the nodes thread state because:

1. There were situations where all the nodes were detected as WAITING but some of their buffers were not empty.

2. The algorithm which checks the state is a thread itself and in my situation I was not able to lock all the nodes to effectively ensure the emptiness of all the buffers.

The way I found to correctly stop the simulation is to check both the state of the threads and the buffers of the nodes. To ensure the emptiness I wrote a recursive function that locks one by one all the buffer while checking if they are empty.

5.2 Simulation pause

The action of pausing a simulation is not simple as there are a lot of threads involved. The hypervisor itself is a thread. To get things simple I wrote a simple *Monitor* class. A monitor has only one boolean instance member that models if something is paused or not. Everything that should be paused, must implement methods to pause it. The object that needs to stop the simulation then will wait on his monitor. To unpause things, it is just needed to notify the monitor.

6. Code conventions

I have followed the source code conventions when it had sense. In particular for some classes I didnt comment everything and sometimes left to the reader the simpler things to understand.

7. Testing

I have tested successfully the application under:

1. OSX 10.7 (lion)
2. Windows 7
3. Ubuntu 10.04