A multithreaded discrete-event driven mobile sensor network simulator

Filippe C. Jabour [1,2], Eugênia Giancoli [1,2], Aloysio C. P. Pedroza [1]
[1] UFRJ - PEE/COPPE/GTA - DEL/POLI
[2] CEFET-MG - Campus Leopoldina
{jabour, eugenia, aloysio}@gta.ufrj.br

***Abstract*—In this paper, we present a multithreaded discrete-event driven mobile sensor network simulator. The simulator supports node mobility, application and MAC layer protocols, radio range variations and mobile agents execution. The multi-threaded approach can take advantage of the power of multi-core CPU efficiently. The schedule of the multi-threads increases the probability of the uncertainty in simulation processes, and makes the events which arrived at the same time interactive with each other sufficiently. The mapping of a mobility input file onto a memory data structure makes it easy to use any kind of mobility patterns generator.**

I. Introduction

In recent researches related to wireless sensor networks [1][2], we observed the lack of simulation tools that agregate mobility support and a mobile agents platform at the same time. Thus, we decided to build a Java application [3][4][5][6][7] to meet such requirements. The simulator developped can be found at http://www.gta.ufrj.br/~jabour/.

II. The simulator

The simulator reads an input data file in the pattern of *ns-2* [8] Tcl. This file contains the mobility instructions for every network node. All nodes are configured with the same radio range (in meters). Each node has a real position and a mesured position (generated by some location algorithm). The simulator is configured with a location algorithm error (percentage). All simulations have a clock precision. It is a value between zero and twelve decimal places for each simulation step. Each node, each mobile agent and the global simulation clock run as independent and concurrent threads. Just when all threads complete the current step, the clock advances to the next one (according to the clock precision). Some agents are injected into the mobile network with some parameters: initial host, initial execution instant, target zone coordinates, return region coordinates and type of algorithm to be used.

The last release (v6.2.0) was built with Java 1.6.0_04. The Integrated Development Environment used was Netbeans [9], version 6.0.1 (Build 200801291616).

As stated before, each node executes as an independent thread. Thus, each node runs a protocol stack, which may be different from one node to another. In this article, the simulator was used with the applications described in [1][2], the MAC layer implemented was 3M [10] and the physical layer was considered ideal (without losses or interference). New applications and protocols may be implemented and attached to simulator as new Java classes, without any change in the rest of the simulator.

A location algorithm is considered to be running and a new position is estimated for each node, in each simulation step. A parameter ϵ that represents the location algorithm error is given as input. Thus, the simulator keeps two sets of coordinates for each node, the real position and the measured position, which is affected by the error ϵ. The actual number of neighbors is provided by the simulator when demanded. The set of neighbors known is maintained by each node, as a result of implementation of the Protocol 3M. The simulator supports the execution of mobile agents and their migration between nodes.

A. The simulator core

Simulations are naturaly plenty of events running simultaneously. The simulator architecture considers this feature. This fact facilitated the design, documentation and understanding of the tool as a whole. Thus, by identifying a new entity (or class) in the system and to define the scope of their actions in time, one might simply create a new thread that represents that entity. This thread should accommodate the attributes ("what it is") and methods ("what it does") within the class and execute concurrently with the others.

As seen, the creation of a new class (thread) with a specific function in the simulation has minimal impact on the simulator as a whole. In the current version of the simulator, three distinct threads run: NodeThread, MobileAgent and Clock. There is an instance of NodeThread for each network node, an instance of MobileAgent for each mobile agent, and a single instance of the Clock for the entire simulation.

Below are listed the functions of the classes NodeThread and MobileAgent.

1) NodeThread:

- Read the input file with mobility instructions, generating a data structure (in RAM) with all the movements and stops (only once);
- Adjust the mobility pattern to the clock precision in use (each step);
- Move the node (each step);
- Execute 3M protocol (each step);
- Release the clock step (after each step);
- Write output files (for each step, with some final summaries).

2) MobileAgent:

- Run one of the mobile agent decision algorithms(each step) (any other developped application may be executed by the node) (no mobile agent may exists);
- Release the clock step (after each step);
- Write output files (for each step, with some final summaries).

Within one clock cycle, the events are carried out in a non-deterministic order, by the various threads running. Each thread may or may not have some work to be done within the current cycle. If so, it performs this task and sends a message to the global clock stating that finished their tasks for this cycle. If not, it releases the clock cycle immediately by the same method. The threads scheduling and concurrency within one clock step makes the simulations more random, which is desirable.

The release is done accessing a Clock class method:

clockInterface.nodeStepOk(nodeThreadID);
clockInterface.mobileAgentStepOk(mobileAgentID);

These methods are accessed with mutual exclusion (synchronized). This ensures that only one thread is running the method in a given time.

The simulator permits events occurring with frequencies that are multiples of the core clock. Thus, in intermediate steps, the holder of this event only frees the simulation clock. For example, if the class needs to perform the task every $0.1\,s$ and the clock is set to advance every $0.001\ s$, the class internal algorithm should release the clock 99 times, run its task and release the clock again. Then, the cycle repeats.

The simulator core is executed by the class Simulator. This class creates an instance of the Clock thread, as described above, and it controls the global simulation clock. The key point of this control is the release step messages coming from various running threads. The communication between different threads and Clock class happens through ClockInterface that is passed, as parameter, by Simulator class to each thread created (Figure 1). The thread can send data and access thread Clock's methods through such interface. An example is obtaining the present moment (currentInstant = clockInterface.getCurrentInstant()). This is done to realize if the clock has advanced to the next step, i.e., if all threads already released the current step. Thus, the thread in question can now decide to start executing the next step. The decision is made according to the Algorithm 1.

a

The class Simulator have an interface (SimulatorInterface) to provide the threads access to many core resources.

B. The simulator input interface

In the simulator, it was necessary to make a conversion of ns-2 setdest pattern of mobility to the granularity of time in use. As mentioned before, the simulator uses a time resolution of 0 to 12 decimal places. In every clock step, the node should move according to a fraction of the total distance between two setdest commands. The conversion was made as follows:

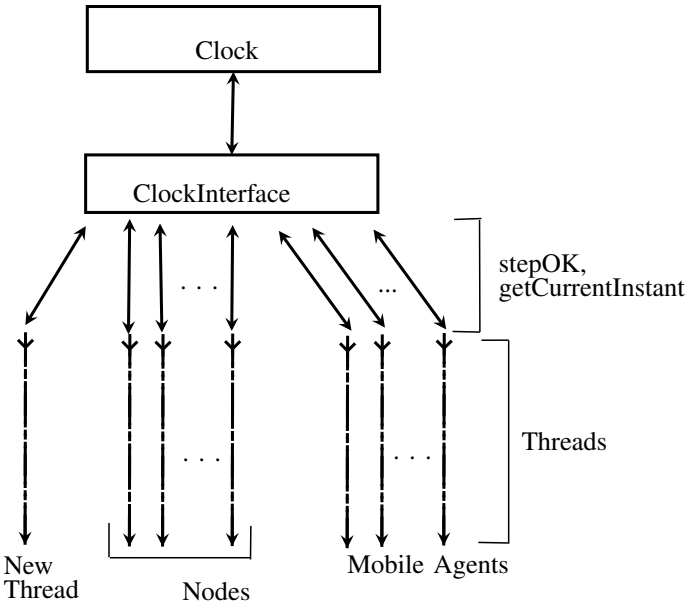

Fig. 1. Clock simulator operation

Algorithm: Local clock advance

```
currentInstant = clockInterface.getCurrentInstant() //current step
//maxInstant comes from mobility input file (input parameter)
lastInstant = 0 //previous Step
while currentInstant ≤ maxInstant do
    currentInstant = clockInterface.getCurrentInstant()
    if currentInstant ≠ lastInstant then
        lastInstant = currentInstant
        //Executes step code
        clockInterface.nodeStepOk(Thread identifier)
    else
        Frees the processor //Allows other threads to finalize
                            //the current step
```

Algorithm 1: Local clock advance

- It is computed the variable $time$, according the expression: $time = currentInstant - lastInstant$;
- The method **move(target, speed, time)** is executed. The parameters target and speed are read from the data structure in memory. This data structure contains the nodes movements instructions. These instructions are equivalent to two consecutive setdest commands related to same node, as shown in Figure 2;
- It is calculated the covered distance: $coveredDistance = speed \times time$;
- Then, it is calculated the total covered distance between two setdest commands. This calculation is based on current position and on target argument. $distanceFromRealPositionToDestiny =$
  $= \sqrt{(X_{final} - X_i)^2 + (Y_{final} - Y_i)^2}$;
- Finally, by using proportion (or similar triangles), the new

coordinates are calculated:

$$X_{i+1} = X_i + \frac{\big(coveredDistance \times (X_{final} - X_i)\big)}{distanceFromRealPositionToDestiny}$$

$$Y_{i+1} = Y_i + \frac{\big(coveredDistance \times (Y_{final} - Y_i)\big)}{distanceFromRealPositionToDestiny}$$

```
#
# nodes: 50, speed type: 2, min speed: 1.00, max speed: 30.00
# avg speed: 2.3, pause type: 2, pause: 10.0, max x: 56.0, max y: 56.0
#
$node_(0) set X_ 48.87...
$node_(0) set Y_ 43.24...
$node_(0) set Z_ 0.00...
$node_(1) set X_ 38.50...
$node_(1) set Y_ 41.75...
$node_(1) set Z_ 0.00...
. . .
$node_(49) set X_ 0.15...
$node_(49) set Y_ 52.45...
$node_(49) set Z_ 0.00...
. . .
$ns_ at 2.06... "$node_(1) setdest 9.42... 34.29... 23.87..."
. . .
$ns_ at 3.32... "$node_(1) setdest 9.42... 34.29... 0.00..."
. . .
$ns_ at 13.86... "$node_(0) setdest 55.09... 10.54... 19.94..."
. . .
$ns_ at 15.53... "$node_(0) setdest 55.09... 10.54... 0.00..."
. . .
$ns_ at 17.65... "$node_(1) setdest 32.88... 49.58... 15.16..."
. . .
$ns_ at 87.14... "$node_(0) setdest 28.54... 38.08... 0.00..."
```

Fig. 2. Some lines of a mobility input data file

Thus, the node has moved only a fraction of the total displacement defined by the input file, equivalent to one clock step, whatever it is. Mobility occurs in the real node's coordinates.

Figure 3 shows the displacement of node 1 between the starting coordinates of Figure 2 and the first node's destination. The movement must occur between instants $2{,}06\,s$ and $3{,}32\,s$. The example assumes a time granularity of time $0,1\,s$, i.e., a step of the simulator every $0,1\,s$.

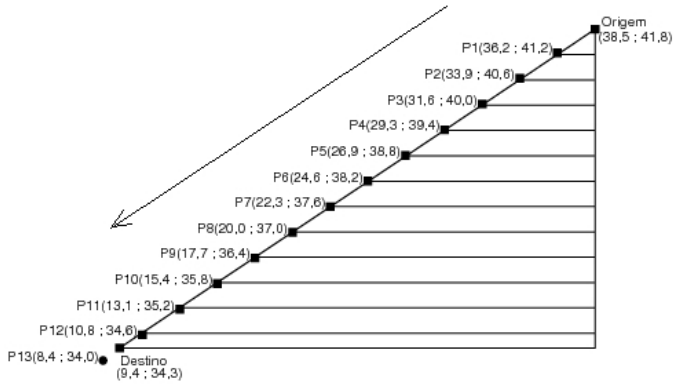

Fig. 3. The mapping of mobility file onto simulator pattern

Figure 3 shows the starting point (origin) and the end point (destination) of the predicted trajectory. Coordinates $P1, P2, P3, \cdots, P13$ show the positions that the node will take in every clock step.

It is observed that only after reaching the point $P13$, node 1 starts working with the next displacement. There is a minimal resolution that the method can achieve, according to the clock step used in the simulation. In many simulations with random mobility, these inaccuracies in the turning points of the trajectories are irrelevant. If greater accuracy is necessary to deal with displacements present in the input file, the steps of the clock should be reduced (more decimal places).

C. Simulator MAC and Physical layers

The simulator implements the 3M protocol on link layer, as described in Section II. However, any link layer protocol can be implemented as an independent class (thread). 3M assumes that each node thread sends "presence announce messages".

It was created the class Channel that models a communication channel. Each node can instantiate as many channels as needed. These channels are instances of class Channel and represent the channels in use by the node. Each node must instantiate at least the control channel.

Class Channel controls the medium state through the logical variable *busy*. When a node sends data to the medium, the same frequency channels for all nodes in its coverage area changes to busy state (**busy = true**). The channel is available again (*idle* or **busy = false**) when the node receives the "end frame". A node "listen" to the medium corresponds to read the state of the *busy* variable of its channel instance for that frequency.

The node detects the end of the frame through the information "frame size" (field FL – Frame Length), present in header of physical package. Upon receiving the "frame end", the physical layer sends the data to layer 2 by invocanting the method **sendToMACLayer(String phyBuffer)**.

During frame reception, the node's physical interface can receive other frame fragments. After reception, method **validate(phyBuffer)** is called. It checks whether or not collisions happened. If it did not happen, the frame is processed. Otherwise, it is discarded. In addition, records of successful frames and collisions are made, for further analysis.

The simulator calculates how many clock cycles are required to send the frame. During these cycles, the process described above is executed.

Due to difference between magnitude of mobility (seconds) and transmissions of data link layer (microseconds), topology changes during transmission of a frame (beacon) were not considered, i.e., variations in the "real neighbors set" were not treated. For greater frames and even to a more accurate consistency of this simulator module, these routines should be reviewed. The simulations performed did not show a negative behaviour related to this simulator feature.

III. Validation and sanity tests

In computer science, sanity tests represent a brief analysis of the functionality of a software, system, or calculation, in order to ensure that the system or methodology works as expected. It is often used before an exaustive round of tests [11].

It is essential that the simulator has the expected behavior, in a deterministic way, in each of their roles and routines. For this reason, the simulator has been thoroughly tested and validated at every step of its evolution. Despite the complexity of the simulator reached, the validation of each one of the most important was feasible and effective.

An example is related to precision of mathematical operations. Consistent with ns-2 simulator, which was used as the default file mobility, the simulator manipulates coordinates with 12 decimal places. This gave rise to minor errors in last decimal places. This fact caused the use of BigDecimal class. This class is appropriated to handle large real numbers and handle with great precision numbers after the decimal point. Thus, the simulator worked without precision error until the twelfth decimal place, for all operations used.

A. Simulator clock validation

It was developped a special thread that do not release the clock at a given instant. It was observed that all others threads stopped at the same step.

IV. Performance

Simulations with up to 1000 nodes and 1000 mobile agents were performed. The simulator reached a total of more than 2000 threads in concurrent execution. This demanding scenario was run on a system with an Intel Pentium 4, with $384MB$ of RAM and Linux operating system. In this scenario, the performance of the system to other running processes is compromised, but not impossible.

Java implements threads in user space and not in kernel space [12][13][14]. However, it was observed that operating systems and Java Virtual Machine (JVM) explore the full potential of parallelism of current processor architectures. On a platform running the operating system Microsoft Windows XP Professional – Version 2002 – Service Pack 2, processor Intel Core2 Quad CPU – Q9400 – 2.66GHz, was observed that the simulator occupies all four processor cores. This fact is highly advantageous for performance, reducing the execution time due to CPU cycles to about one quarter. Simulator TOSSIM [15], running in same system, uses just one processor core. The same platform running Linux operating system, Fedora distribution version 11, has the same behavior, using 4 CPU cores to a single simulator instance.

A simulation with 700 nodes and 10 mobile agents uses from 50 to 82 MB of RAM. It can be considered a small footprint.

Multiple instances of the JVM (up to eight) with one simulation running in each JVM instance were placed on concurrent execution in the Pentium 4 system described above, without fail.

V. Conclusions

This paper presented the simulator developed for testing, development and validation of sensor networks protocols, especially networks with mobile nodes.

The operation of the tool, its interfaces and core were described. It has been shown the safe and correct behaviour of simulator through execution analysis and sanity tests.

Due its architecture and project decisions, it was observed that the simulator can be extended to other applications and can easily support the implementation of other protocols.

References

[1] F. C. Jabour, E. G. Jabour, and A. C. P. Pedroza, "Mobility support for wireless sensor networks," in *Proceedings of IEEE International Conference on Computer and Electrical Engineering – ICCEE*, Phuket Island, Thailand, Dec. 2008.

[2] ——, "Redes de sensores móveis: análise da velocidade, comunicação e esforço computacional," in *7th International Information and Telecommunication Technologies Symposium - I2TS*, Foz do Iguaçu, Brasil, Dec. 2008.

[3] "Java," http://java.sun.com, acesso em Abril de 2009.

[4] "Java platform, standard edition 6 api specification," http://java.sun.com/javase/6/docs/api/, acesso em Maio de 2009.

[5] H. M. Deitel and P. J. Deitel, *Java - Como programar*. PEARSON, 2005, 6. ed.

[6] C. S. Horstmann and G. Cornell, *Core Java 2 - Fundamentos*. Alta Books, 2005, 7. ed.

[7] ——, *Core Java - Advanced Features*. Prentice-Hall, 8. ed.

[8] "ns-2 network simulator," http://www.isi.edu/nsnam/ns, 1998, acesso em Maio de 2009.

[9] "Netbeans," http://www.netbeans.org/, acesso em Maio de 2009.

[10] F. C. Jabour, E. G. Jabour, and A. C. P. Pedroza, "3m: Um protocolo de enlace multicanal para redes de sensores com alto grau de mobilidade," in *Conferência Latino-Americana de Informática - CLEI 2009*, Pelotas, Brasil, May 2009.

[11] R. Pressman, *Engenharia de Software*. McGraw-Hill Interamericana do Brasil, 2005.

[12] H. Blaar, M. Legeler, and T. Rauber, "Efficiency of thread-parallel java programs from scientific computing," *Parallel and Distributed Processing Symposium, International*, vol. 2, p. 0115b, 2002.

[13] M. Philippsen and M. Zenger, "JavaParty: Transparent remote objects in Java," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1225–1242, November 1997.

[14] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance java dialect," in *In ACM*, 1998, pp. 10–11, workshop on Java for High-Performance Network Computing, Stanford, 1998.

[15] P. Levis, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications," proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003), November 2003.