The JBotSIM Library

Arnaud Casteigts
casteig@site.uottawa.ca

Version 1*
January 9, 2010

“A good software is one written twice..”
Martin Raspaud, CVT, 2005

Foreword

...or more. Better not mentioning how many times JBotSIM was rewritten, or you may find it pretentious (as per Martin’s phrase). JBotSIM is a small Java-based network simulation library whose purpose is to offer basic tools for studying distributed algorithms in dynamic networks. JBotSIM is not a competitor of NS [NS3] or OMnet [V+01] in the sense that it cannot simulate a IEEE 802.11 protocol stack or doppler effects while transmitting messages. It is not a competitor of the ViSiDiA platform [BM03, Der07] neither, this latter being incredibly more furnished than JBotSIM but dedicated to static networks (and closed source). JBotSIM enables a (very (very)) fast prototyping of distributed algorithms, provides basic primitives to manage dynamic topologies (either based on program or based on user interactions) while the algorithm is running. It also permits to prepare live demonstrations of your algorithms to explain them to students or colleagues.

The style of programming in JBotSIM is event-driven: algorithms are defined as subroutines to be executed when some particular events occur (e.g. ring of an alarm clock, appearance/disappearance of an adjacent link, arrival of a message, movement of the underlying node, etc.). Although the future purpose of this document is to offer a thorough presentation of JBotSIM, this first version will settle for a brief practical overview of JBotSIM’s key features. I hope this material will suffice to convince you of the relevance (or irrelevance) of JBotSIM to your activities, and give you a good programming start.

Getting JBotSIM and setting it up

JBotSIM can be obtained in two ways:

- by getting the sources:
svn co https://jbotsim.svn.sourceforge.net/svnroot/jbotsim jbotsim, and compiling them (make in the top directory) to generate the file jbotsim.jar.

- by downloading the latest pre-compiled version at http://jbotsim.sf.net/jbotsim.jar

*Please check for a possible new version of this document at http://jbotsim.sf.net/jbotsim-doc.pdf

1 Another unconfessed aim is that of pursuing simplicity above all criterions (which motivated most rewritings).
JBotSim can be linked to your programs by adding jbotsim.jar to the CLASSPATH environment variable, or using the -cp option while compiling (e.g. javac -cp jbotsim.jar MyClass.java) and executing (java -cp jbotsim.jar:. MyClass).

Besides the present document, a Javadoc is available at [http://jbotsim.sf.net/javadoc/](http://jbotsim.sf.net/javadoc/) and the example discussed below can be visualized at [http://jbotsim.sf.net/examples/](http://jbotsim.sf.net/examples/).

**Practical overview**

Let us consider a basic example: we want the nodes to be colored in green when they have at least one neighbor, in red otherwise, and the colors must be updated when the situation of the nodes changes. This section gives a quick overview of JBotSim’s Application Programming Interface (API) by means of showing several ways of implementing this Red-Green principle. The first three implementations are distributed (an algorithm is executed independently by each node), the fourth is centralized (a global entity watch the entire network and update all the states accordingly). A last example entangles the first implementation with another algorithm responsible for moving the nodes randomly (to illustrate some of JBotSim’s features for nodes movements).

The dominant paradigm in JBotSim is event-driven. This means that your algorithms mainly consist in providing the code to be executed in reaction to particular events. These events can be observed either from the node standpoint or the global standpoint, depending on whether the algorithm is distributed or centralized. The normal way of implementing a distributed algorithm in JBotSim is by extending the class Node and adding some event-driven code to it. In the first example (Listing 1), the source of event that we use is the time: each node periodically checks for the presence of neighbors and updates its color accordingly.

**Listing 1 Red-Green algorithm (distributed version based on time).**

```java
import jbotsim.Node;
import jbotsim.Clock;
import jbotsim.event.ClockListener;

public class RedGreenNodeV1 extends Node implements ClockListener{
    public RedGreenNodeV1(){
        Clock.addClockListener(this, 10);
    }
    public void onClock(){
        if (super.getNeighbors().size()>0)
            super.setProperty("color", "green");
        else
            super.setProperty("color", "red");
    }
}
```

In practical terms, the new type of node implements the interface ClockListener and subscribes to a clock through Clock.addClockListener, which causes its method onClock() to be periodically called. The rate at which this notification occurs is specified while subscribing, by the second parameter, in clock’s time unit (here, 10). The questions of how long one time unit is, and why the clock is represented by a static class, will be answered in a later version of this document. A possible main function to start simulating this algorithm is given on Listing 2 on the next page.
In this `main()` function, an object of type `Topology` is created. The concept of topology is central in JBotSIM because it represents the environment in which nodes and links are contained. New nodes or links can be added to a topology by using one of the several `addNode()` and `addLink()` methods. These methods are overloaded to offer several ways of doing so. It is for example possible to add an already instantiated node, or to ask the topology to create the new node itself. The kind of nodes created can be tuned by means of `models` (special instances of nodes used to specify the default class and properties of new nodes). It is possible to define new models or override existing ones using `Node.getModel()` and `Node.setModel()`. In the example of Listing 2, we override the default model to be of type `RedGreenNodeV1`.

By default, all the nodes are created with wireless communication capabilities. This allows the topology to update the set of links automatically as the nodes move (according to the relative distance between nodes). JBotSIM provides a basic viewer (`JViewer`) to visualize a topology and interact with it during the execution. It allows for example to add, move, or remove nodes (left click, drag and drop, and right click, respectively) while an algorithm is running to test its behavior.

Rather than executing a piece of code periodically, one could prefer being notified only when the node’s connectivity changes. The interface `NodeListener` specifies such notifications (addition or deletion of a local link), among others (notification of a movement of the node or a property changed). In the example of Listing 3, we update the local color when adjacent links are added or removed.
Practically, each node initializes its color to "red" and registers to its own events (using the parent method `addNodeListener()`). Then, whenever an adjacent link is added or removed, the corresponding method is called. By reading the examples on Listings 1 and 3 you may wonder why the method `setProperty()` was used instead of using an independent variable for the color. The reason is that the viewer recognizes the property key "color" for a node and knows how to paint it accordingly. In fact, the viewer also listens to each node’s events using the same mechanisms, and updates the painting color of a node when notified by `propertyChanged()`.

We assumed so far that the existence of neighbors or adjacent links could be directly assessed by the nodes (e.g. `getNeighbor()`, `linkAdded()`, `linkRemoved()`). We consider now a more practical scenario, in which nodes do not have such information and discover their neighborhood by exchanging HELLO messages at regular interval (or beacons). (This kind of communication scheme is particularly popular in wireless networking.) The adaptation of the Red-Green algorithm is as follows: whenever a node receives a beacon, it sets its color to "green", and while sending its own beacon it checks if the last beacon was received since more than the beaconing period, in which case it sets its color to "red". The corresponding code is given on Listing 4.

**Listing 4** Red-Green algorithm (distributed version based on HELLO messages).

```java
import jbotsim.Node;
import jbotsim.Clock;
import jbotsim.Message;
import jbotsim.event.ClockListener;
import jbotsim.event.MessageListener;

public class RedGreenNodeV3 extends Node implements ClockListener, MessageListener{
    int lastReceptionDate = -30;
    public RedGreenNodeV3(){
        Clock.addClockListener(this, 30);
        super.addMessageListener(this);
    }
    public void onClock(){
        super.send(null, "HELLO");
        if (lastReceptionDate < Clock.currentTime() - 30)
            super.setProperty("color", "red");
    }
    public void onMessage(Message msg){
        super.setProperty("color", "green");
        lastReceptionDate=Clock.currentTime();
    }
}
```

The class `RedGreenNodeV3` uses two interfaces: `ClockListener` and `MessageListener`. `ClockListener` declares the method `onClock()`, which allows to execute a piece of code at regular interval (here, the emission of a beacon, and the inspection of the last reception date). We can see here the way messages are sent in JBotSim by calling the parent method `send()`. The first parameter of this method specifies the destination of the message (either a single node, or `null` to denote all neighbors), the second parameter is the message itself (being any kind of object whose reference can be directly passed to the destination(s)). The interface `MessageListener` also declares a single method: `onMessage()`, which is called whenever the node receives a message (don’t forget to subscribe via `addMessageListener()`). Finally, `Clock.currentTime()` returns the number of time units elapsed since the beginning of the execution.
The next example (Listing 5) presents a centralized version of the Red-Green algorithm. Here, rather than having each node taking care of itself, a global entity is in charge of updating all nodes colors as the network evolves. More precisely, whenever a link appears, the colors of the two corresponding nodes are set to "green", and when it disappears, the remaining number of neighbors is examined for each node and the color it sets to "red" if none are left.

**Listing 5 Red-Green algorithm (centralized version based on connectivity events).**

```java
import jbotsim.Link;
import jbotsim.Node;
import jbotsim.Topology;
import jbotsim.event.TopologyListener;

public class RedGreenCentralized implements TopologyListener {
    public void nodeAdded(Node n) {
        n.setProperty("color", n.getNeighbors().size() == 0 ? "red" : "green");
    }
    public void linkAdded(Link l) {
        for (Node n : l.endpoints())
            n.setProperty("color", "green");
    }
    public void linkRemoved(Link l) {
        for (Node n : l.endpoints())
            if (n.getNeighbors().size() == 0)
                n.setProperty("color", "red");
    }
    public void nodeRemoved(Node n) {}
    public void propertyChanged(Topology t, String key) {}
}
```

The class RedGreenCentralized does not extend Node. In fact, the type of node used in this version is JBotSim’s default class Node. RedGreenCentralized is an independent class which subscribes to the events of the topology by means of the TopologyListener interface. The methods declared in this interface allows to be notified whenever the topology is modified (addition or deletion of a link or a node) or when a property of the topology is changed. In our case, we are only interested in the appearance or disappearance of a link (to color both endpoints in green, or to check the remaining neighborhood, respectively), and the addition of a node (to initialize its color). In order to instantiate RedGreenCentralized and register it as a listener of the topology, the main() function can be as shown on Listing 6.

**Listing 6 Main function (for the centralized version of the Red-Green algorithm).**

```java
import jbotsim.Topology;
import jbotsim.ui.JViewer;
...

public static void main(String[] args) {
    Topology tp = new Topology();
    tp.addTopologyListener(new RedGreenCentralized());
    new JViewer(tp);
}
```
The last example presents some capabilities for nodes movements in JBotSim. It consists in implementing a simple version of the well-known RandomWayPoint mobility model [BRS03], where each node repeatedly choose a destination at random and move towards it. We define two static methods: init() and move(), which initializes and moves the specified node, respectively (Listing 7).

Listing 7 A (stateless) implementation of the RandomWayPoint mobility model.

```java
public class RandomWayPoint {
    private static Random r = new Random();

    public static void init(Node node) {
        node.setProperty("target", new Point(r.nextInt(400), r.nextInt(300)));
    }

    public static void move(Node node) {
        Point target = (Point) node.getProperty("target");
        node.setDirection(target);
        node.move(5);
        if (node.distance(target) < 5) {
            node.setProperty("target", new Point(r.nextInt(400), r.nextInt(300)));
        }
    }
}
```

In this implementation the movements are performed step by step (each step corresponding to a call of the method move()). In every step, the next movement is defined as two complementary elements: a direction and a distance. The direction is set by means of a reference point (here, the target), although this point is not stored and only serves to determine the angle in which the node will move (this angle can also be specified directly, by using another version of setDirection()). When the node arrives closer to the target than the step distance, a new target is selected. Using inner properties of the node makes it possible to store the data within the node itself (and thus make RandomWayPoint’s method stateless). The code in Listing 8 shows an example of how RandomWayPoint can be used by a node (here, the time-based version of the Red-Green algorithm).

Listing 8 Example of use of the class RandomWayPoint from a node’s definition.

```java
import jbotsim.Clock;
import jbotsim.Node;
import jbotsim.event.ClockListener;

public class RedGreenNodeV4 extends Node implements ClockListener {
    public RedGreenNodeV4() {
        RandomWayPoint.init(this);
        Clock.addClockListener(this, 10);
    }

    public void onClock() {
        RandomWayPoint.move(this);
        if (super.getNeighbors().size() > 0)
            super.setProperty("color", "green");
        else
            super.setProperty("color", "red");
    }
}
```
By running this version using a `main()` similar to the one of Listing 2, you should see the nodes moving in random directions and updating their colors simultaneously.

**Extending JBOTSIM**

In my view, JBOTSIM is a *kernel*, in the sense that it encapsulates a small number of generic features whose purpose is to be used by upper layer programs. As for today, the plan is to keep it this way and maintain the number of features to a minimum. This does not mean, of course, that JBOTSIM should not be extended by external components. Figure 1 illustrates an example of basic extension, the class `Tikz`, which allows to export topologies as Tikz pictures (Tikz is a TeX package for creating sophisticated graphics using high-level drawing primitives). Note that this particular extension is released with JBOTSIM, as part of the non-documented extension package `jbotsimx` (in `jbotsim.jar`), but it could have been as well released as an independent extension by some other people (which I strongly encourage!). By the way, if you develop an extension for JBOTSIM (e.g. an Evolving Graph [Fer04] or GraphStream [DGOP07] encoder/decoder, or a set of high-level primitives for bio-inspired approaches [Gui07] or (dynamic)-graph-relabeling approaches [CCF09], or even (why not?) a micromachine-like game involving races of nodes with shooting messages...), just let me know and I will be glad toadvertize your extension from the website.

![Diagram](image)

Figure 1: The class `jbotsimx.Tikz`, as an example of feature built on top of JBOTSIM. Here, the topological part of each picture was generated by JBOTSIM (and other elements added afterward).

**References**

[BM03] M. Bauderon and M. Mosbah. A unified framework for designing, implementing and visualizing distributed algorithms. *Electronic Notes in Theoretical Computer Science*, 72(3):13–24, 2003.

[BRS03] C. Bettstetter, G. Resta, and P. Santi. The node distribution of the random waypoint mobility model for wireless ad hoc networks. *IEEE Transactions on Mobile Computing*, 2(3):257–269, 2003.

[CCF09] A. Casteigts, S. Chaumette, and A. Ferreira. Characterizing topological assumptions of distributed algorithms in dynamic networks. In *Proc. of 16th Intl. Conference on Structural Information and Communication Complexity (SIROCCO’09)*, volume 5869 of *Lecture Notes in Computer Science*, pages 126–140, Piran, Slovenia, May 2009. Springer-Verlag.
[CNS10a] A. Casteigts, A. Nayak, and I. Stojmenovic. *Multicasting, Geocasting and Anycasting in Sensor and Actuator Networks*, chapter 5 of Wireless Sensor and Actuator Networks - Algorithms and Protocols for Scalable Coordination and Data Communication, Nayak, A. and Stojmenovic, I. (eds). Wiley, 2010.

[CNS10b] A. Casteigts, A. Nayak, and I. Stojmenovic. *Topology Control in Sensor, Actuator and Mobile Robot Networks*, chapter 7 of Wireless Sensor and Actuator Networks - Algorithms and Protocols for Scalable Coordination and Data Communication, Nayak, A. and Stojmenovic, I. (eds). Wiley, 2010.

[Der07] B. Derbel. A Brief Introduction to ViSiDiA, 2007. http://www2.lifl.fr/~derbel/resources_visidia/visidia_tutorial.pdf

[DGOP07] A. Dutot, F. Guinand, D. Olivier, and Y. Pigné. GraphStream: A Tool for bridging the gap between Complex Systems and Dynamic Graphs. *EPNACS: Emergent Properties in Natural and Artificial Complex Systems*, 2007.

[Fer04] A. Ferreira. Building a reference combinatorial model for MANETs. *IEEE Network*, 18(5):24–29, 2004. A preliminary version appeared as On models and algorithms for dynamic communication networks: The case for evolving graphs, *Algotel’02*, Meze, FR.

[Gui07] F. Guinand. Nature-Inspired Approaches for Complex Biological Systems. In *Proc. of the 22nd European Conference on Operational Research (EURO’07)*, Prague, Czech Republic, July 2007.

[NS3] NS3. The ns-3 network simulator. http://www.nsnam.org/ 2009.

[V+01] A. Varga et al. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM’01)*, pages 319–324, 2001.