



HAL
open science

JBotSim: a Tool for Fast Prototyping of Distributed Algorithms in Dynamic Networks

Arnaud Casteigts

► **To cite this version:**

Arnaud Casteigts. JBotSim: a Tool for Fast Prototyping of Distributed Algorithms in Dynamic Networks. Eighth EAI International Conference on Simulation Tools and Techniques (SIMUTOOLS), Aug 2015, Athènes, Greece. hal-01472926

HAL Id: hal-01472926

<https://hal.science/hal-01472926>

Submitted on 21 Feb 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

JBotSim: a Tool for Fast Prototyping of Distributed Algorithms in Dynamic Networks*

Arnaud Casteigts

LaBRI, University of Bordeaux

arnaud.casteigts@labri.fr

arXiv:1001.1435v6 [cs.MS] 18 May 2015

ABSTRACT

JBOTSIM is a java library that offers basic primitives for prototyping, running, and visualizing distributed algorithms in dynamic networks. With JBOTSIM, one can implement an idea in minutes and interact with it (*e.g.*, add, move, or delete nodes) while it is running. JBOTSIM is well suited to prepare live demonstrations of algorithms to colleagues or students; it can also be used to evaluate performance at the algorithmic level (number of messages, number of rounds, etc.). Unlike most simulation tools, JBOTSIM is not an integrated environment. It is a lightweight library to be used in java programs. In this paper, I present an overview of its distinctive features and architecture.

1. INTRODUCTION

JBotSim is an open source simulation library that is dedicated to distributed algorithms in dynamic networks. I developed it with the purpose in mind to make it possible to implement an algorithmic idea in minutes and interact with it while it is running (*e.g.*, add, move, or delete nodes). Besides interaction, JBOTSIM can also be used to prepare live demos of an algorithm and to show it to colleagues or students, as well as to assess the algorithm performance. JBOTSIM is not a competitor of mainstream simulators such as NS3 [5], OMNet [10], or The One [8], in the sense that it does not aim to implement real-world networking protocols. Quite the opposite, JBOTSIM aims to remain technology-insensitive and to be used at the algorithmic level, in a way closer in spirit to the ViSiDiA project (a general-purpose platform for distributed algorithms). Unlike ViSiDiA, however, JBOTSIM natively supports mobility and dynamic networks (as well as wireless communication). Another major difference with the above tools is that it is a *library* rather than a software: its purpose is to be used in other programs, whether these programs are simple scenarios of full-fledged software. Finally, JBOTSIM is distributed under the terms of the LGPL licence, which makes it easily extensible by the community.

Whether the algorithms are centralized or distributed, the natural way of programming in JBOTSIM is event-driven: algorithms are specified as subroutines to be executed when particular events occur (appearance or disappearance of a link, arrival of a message, clock pulse, *etc.*). Movements of the nodes can be controlled either by program or by means of live interaction with the mouse (adding, deleting, or moving nodes around with left-click, right-click, or drag and drop, respectively). These movements are typically performed while the algorithm is running, in order to visualize it or test its behavior in challenging configurations.

The present document offers a broad view of JBOTSIM's main features and design traits. I start with preliminary information in

Section 2 regarding installation and documentation. Section 3 reviews JBOTSIM's main components and specificities such as programming paradigms, clock scheduling, user interaction, or global architecture. Section 4 zooms on key features such as the exchange of messages between nodes, graph-level APIs, or the creation of online demos. Finally, I discuss in Section 5 some extensions of JBOTSIM, including TikZ exportation feature and edge-markovian dynamic graph generator.

Besides its features, the main asset of JBotSim is its simplicity of use – an aim that I pursued at the cost of re-writing it several times from scratch (the API is now stable).

2. PRACTICAL ASPECTS

In this short section, I show how to install JBOTSIM and run it with a first basic example (this step is not required to keep reading the present document). I also provide links to online documentation and examples, for readers who would like to explore JBOTSIM's features beyond this paper.

2.1 Fetching JBOTSIM

The straightest (and safest) way to obtain JBOTSIM is to fetch the latest release, as a *jar* package on JBOTSIM's website [1]. One can also get the latest development version from SourceForge's GIT repository and compile it as shown below, keeping in mind that online documentation refers to the official release rather than this version. Here is the command:

```
>git clone git://git.code.sf.net/p/jbotsim/git target
where target is the local directory in which to put JBOTSIM.
From that directory, one can produce the JAR package using a simple make.
```

After more than seven years of development, JBOTSIM is finally approaching version 1.0 (as of today, version 1.0-prealpha) and I think it is ready to be tested by the community.

2.2 First steps

As a first program, one can copy the code from Listing 1 into a file named `HelloWorld.java`.

Listing 1 HelloWorld with JBOTSIM

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

public class HelloWorld{
    public static void main(String[] args){
        new JViewer(new Topology());
    }
}
```

How to use the JAR file and run the HelloWorld example:

*A shorter version appeared in SIMUTOOLS 2015.

- If using *IntelliJ*, go to *Project structure* > *Modules (select the module)* > *Dependencies* > "+" and add `jbotssim.jar`.
- If using *Eclipse*, go to *Project* > *Properties* > *Java build path* > *Libraries* > *Add external jar* and add `jbotssim.jar`.
- From the terminal, the following commands can be used:


```
javac -cp jbotssim.jar HelloWorld.java
(compilation)
java -cp .:jbotssim.jar HelloWorld
(execution)
```

By running the program, one should see an empty gray surface in which nodes can be added, moved, or deleted using the mouse.

2.3 Sources of documentation

In this document, I provide a general overview of what JBOTSIM is and how it is designed. This is by no means a comprehensive programming manual. The reader who wants to explore further some features or develop complex programs with JBOTSIM is referred to the API documentation (see *javadoc* on the website).

Examples can also be found on JBOTSIM's website, together with comments and explanations. These examples offer a good starting point to learn specific components of the API from an *operational* standpoint – the present document essentially focuses on *concepts*. Most of the online examples feature embedded videos. Finally, most of the examples in this paper are also available on JBOTSIM's website. Feel free to check them when the code given here is incomplete (e.g. I often omit package imports and `main()` methods for conciseness).

3. FEATURES AND ARCHITECTURE

This section provides an overview of JBOTSIM's key features and discusses the reason why some design choices were made. I review topics as varied as programming paradigms, clock scheduling, user interaction, and global architecture.

3.1 Basic features of nodes and links

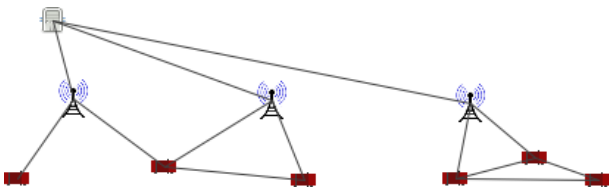


Figure 1: A highway scenario composed of vehicles, road-side units, and central servers. Part of the network is *ad hoc* (and wireless); the rest is *infrastructured* (and wired).

JBOTSIM consists of a small number of classes, the most central being `Node`, `Link`, and `Topology`. The contexts in which dynamic networks apply are varied. In order to accommodate a majority of cases, these classes offer a number of conceptual variations around the notions of nodes and links. Nodes may or may not possess wireless communication capabilities, sensing abilities, or self-mobility. They may differ in clock frequency, color, communication range, or any other user-defined property. Links between the nodes account for potential communication among them. The nature of links varies as well; a link can be directed or undirected, as well as it can be wired or wireless – in the latter case JBOTSIM's topology will update the set of links *automatically*, as a function of nodes distances and communication ranges.

Figures 1 and 2 illustrate two different contexts. Figure 1 depicts a highway scenario where three types of nodes are used: vehicles, road-side units (towers), and central servers. This scenario is semi-infrastructure: Servers share a dedicated link with each tower. These links are *wired* and thus exist irrespective of distance. On the other hand, towers and vehicles communicate through wireless links that are automatically updated.

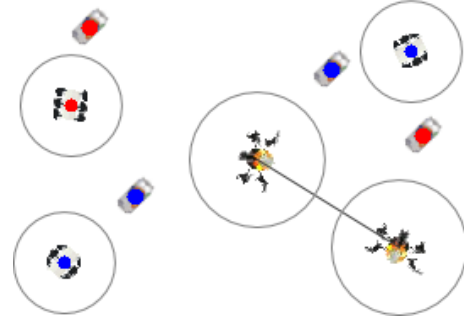


Figure 2: A swarming scenario, whereby mobile robots and UAVs collaborate in order to clean a public park.

Figure 2 illustrates a purely *ad hoc* scenario, whereby a heterogeneous swarm of UAVs and robots strives to clean a public park collectively. In this scenario, robots can detect and clean wastes of a certain type (red or blue) only if these are within their *sensing range* (depicted by a surrounding circle). However, they are pretty slow to move and cannot detect remote wastes. In the meantime, a set of UAVs is patrolling over the park at higher speed and with larger sensing range. Whenever they detect a waste of some type, they store its position and start searching for a capable robot. In addition to sensing capabilities, UAVs can exchange messages with each other to optimize the process.

Besides nodes and links, the concept of topology is central in JBOTSIM. Topologies can be thought of as *containers* for nodes and links, together with dedicated operations like updating wireless links. They also play a central role in JBOTSIM's event architecture, as explained later on.

3.2 Distributed vs. centralized algorithms

JBOTSIM supports the manipulation of centralized or distributed algorithms (possibly simultaneously). The natural way to implement a distributed algorithm is by extending the `Node` class, in which the desired behavior is implemented. Centralized algorithms are not constrained to a particular model, they can take the form of any standard java class.

Distributed algorithm.

JBOTSIM comes with a default type of node that is implemented in the `Node` class. This class provides the most general features a node could have, including primitives for moving, exchanging messages, or tuning basic parameters (e.g. communication range and sensing range). Distributed algorithms are naturally implemented through adding specific features to this class. Listing 2 provides a basic example in which the nodes are endowed with self-mobility. The class relies on a key mechanism in JBOTSIM: performing periodic operations that are triggered by the pulse of the system clock. This is done by overriding the `onClock()` method, which is called periodically by JBotSim's engine (by default, at every pulse of the clock). The rest of the code is responsible for moving the node, setting a random direction at construction time

Listing 2 Extending the `Node` class

```
import jbotsim.Node;

public class MovingNode extends Node{
    public MovingNode(){
        setDirection(Math.random() * 2*Math.PI);
    }
    public void onClock(){
        move(2);
    }
}
```

(in radian), then moving in this direction periodically. (More details about the movement API can be found online.)

Once this class is defined, new nodes of this type can be added to the topology in the same way as if they were of class `Node`, as illustrated in Listing 3. Here, a new topology is created first, then 10

Listing 3 Adding nodes manually

```
public static void main(String[] args){
    Topology tp = new Topology(400,300);
    for (int i=0; i<10; i++){
        tp.addNode(-1,-1, new MovingNode());
    }
    new JViewer(tp);
}
```

nodes of the desired type (here, `MovingNode`) are added through the `addNode()` method at random locations (using `-1` for x -coordinate or y -coordinate generates a random location for that coordinate; here both are random). In order to use `MovingNode` through the GUI, for instance when clicking with the mouse to add new nodes, this class must be registered as a *node model*. This is done by calling method `setDefaultNodeModel()` with the class itself as an argument, as shown on Listing 4. JBOTSIM will create new instances on the fly, using reflexivity. Several models can be registered simultaneously, using `setNodeModel()` with an additional argument that corresponds to the model name. If several models exist, JBOTSIM's GUI (the *viewer*) displays a selection list when a node is added by the user (in this example, only one model is set and it is used by default). In the scenario of Fig-

Listing 4 Using a defined node as default

```
public static void main(String[] args){
    Topology tp = new Topology(400, 300);
    tp.setDefaultNodeModel(MovingNode.class);
    new JViewer(tp);
}
```

ure 1, however, left-clicking on the surface would give the choice between *car*, *tower*, and *server*, the names of the three registered models for that scenario.

Centralized algorithms.

There are many reasons why a centralized algorithm can be preferred over a distributed one. The object of study might be centralized in itself (e.g. network optimization, scheduling, graph algorithms in general). It may also be simpler to simulate distributed things in a centralized way. Listing 5 implements such a version of the *random waypoint* mobility model, in which nodes repeatedly move toward a randomly selected destination, called *target*. Unlike a distributed implementation, the movements of nodes are here driven by a *global* loop at every pulse of the clock. For each node, a target is created if it does not exist yet or if it has just been reached; then the node's direction is set accordingly and the node

Listing 5 Centralized version of Random Waypoint

```
public class CentralizedRWP implements ClockListener{
    Topology tp;
    public CentralizedRWP (Topology tp){
        this.tp = tp;
        tp.addClockListener(this);
    }
    public void onClock(){
        for (Node n : tp.getNodes()){
            Point2D target=(Point2D)n.getProperty("target");
            if (target == null || n.distance(target)<1){
                target = new Point2D.Double(Math.random()*400,
                    Math.random()*300);
                n.setProperty("target", target);
                n.setDirection(target);
            }
            n.move();
        }
    }
    public static void main(String[] args){
        Topology tp = new Topology(400,300);
        new CentralizedRWP(tp);
        new JViewer(tp);
    }
}
```

is moved (by default, by 1 unit of distance). For convenience, the `main()` method is included in the same class.

Notice the use of `setProperty()` and `getProperty()` in this example. These methods allow to store any object directly into a node, using a key/value scheme (where key is a string). Both `Link` and `Topology` objects offer the same feature.

3.3 Architecture of the event system

So far, we have seen one type of event: clock pulses, to be listened to through the `ClockListener` interface. JBOTSIM offers a number of such events and interfaces, some of which become ubiquitous. The main ones are depicted on Figure 3 on the following page. This architecture allows one to specify dedicated operations in reaction to various events. For instance, one may ask to be notified whenever a link appears or disappears somewhere. Same for messages, which are typically listened to by the nodes themselves or can be watched at a global scale (e.g. to keep a log of all communications). In fact, every node is automatically notified for its own events; it just needs to override the corresponding methods from the parent class `Node` in order to specify event handlings (e.g. `onClock()`, `onMessage()`, `onLinkAdded()`, `onSensingIn()`, `onSelection()`, etc.). Explicit listeners, on the other hand, like the ones in Figure 3, are meant to be used by centralized programs which do not extend class `Node`.

Listing 6 gives one such example, consisting of a mobility trace recorder. This program listens to topological events of various kinds, including appearance or disappearance of nodes or links, and movements of the nodes. Upon each of these events, it outputs a string representation of the event using a dedicated human readable format called DGS [7]. Similar code could be written for Gephi [3].

Other events exist besides those represented in Figure 3, such as the `SelectionListener` interface, which makes it possible to be notified when a node is selected (middle-click) and make it initiate some tasks, for instance broadcast, distinguished role, etc.

3.4 Single threading: why and how?

It seems convenient at first, to assign every node a dedicated thread, however JBOTSIM was designed differently. JBOTSIM is single-threaded, and definitely so. This section explains the why and the how. Understanding these aspects are instrumental in developing well-organized and bug-free programs.

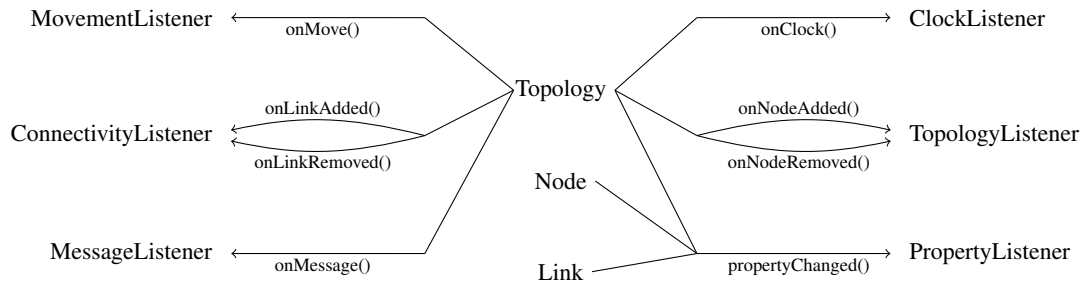


Figure 3: Main sources of events and corresponding interfaces in JBOTSIM.

Listing 6 Example of a mobility trace recorder

```

public class MyRecorder implements TopologyListener,
    ConnectivityListener,
    MovementListener{
    public MyRecorder(Topology tp){
        tp.addTopologyListener(this);
        tp.addConnectivityListener(this);
        tp.addMovementListener(this);
    }

    // TopologyListener
    public void onNodeAdded(Node node) {
        println("an_" + node.hashCode() +
            "_x:" + node.getX() + "_y:" + node.getY());
    }
    public void onNodeRemoved(Node node) {
        println("dn_" + node.hashCode());
    }

    // ConnectivityListener
    public void onLinkAdded(Link link) {
        println("ae_" + link.hashCode() + "_" +
            link.endpoint(0) + "_" + link.endpoint(1));
    }
    public void onLinkRemoved(Link link) {
        println("de_" + link.hashCode());
    }

    // MovementListener
    public void onNodeMoved(Node node) {
        println("cn_" + node.hashCode() +
            "_x:" + node.getX() + "_y:" + node.getY());
    }
}

```

In JBOTSIM, all the nodes, and in fact all of JBOTSIM's life (GUI excepted) is articulated around a single thread, which is driven by the central *clock*. The clock pulses at regular interval (whose period can be tuned) and notifies its listeners in a specific order. JBOTSIM's internal engines, such as the message engine, are served first. Then come those nodes whose wait period has expired (remind that nodes can choose to register to the clock with different periods). These nodes are notified in a random order. Hence, if all nodes listen to the clock at a rate of 1 (the default value), they will all be notified in a random order in each round, which makes JBOTSIM's scheduler a non-deterministic 2-bounded fair scheduler. (Other policies will be available eventually.) A simplified version of the current scheduling process is depicted on Figure 4.

One consequence of single-threading is that all computations (GUI excepted) take place in a sequential order that makes it possible to use unsynchronized data structures and simpler code. This also improves the scalability of JBOTSIM when the number of nodes grows large. One can rely on other user-defined threads in the program, however one should be careful that these thread do not interfere with JBOTSIM's. The canonical example is when a sce-

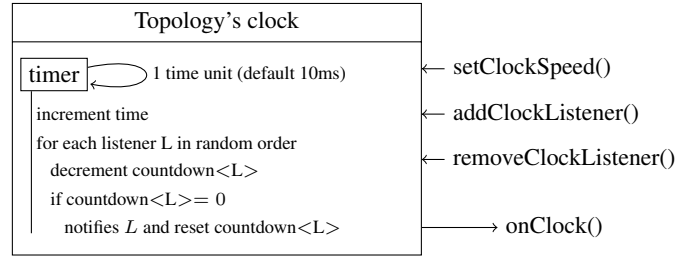


Figure 4: Simplified version of the internal scheduler.

nario is set up by program from within the thread of the `main()` method. If the initialization makes extensive use of JBOTSIM's API from within that thread and the clock starts triggering events at the same time, then problems might occur (and a `ConcurrentModificationException` be raised). The easy way around is to pause the clock before executing these instructions and to resume it after (using the `pause()` and the `resume()` methods on the topology, respectively).

3.5 Interactivity

I designed JBOTSIM with a clear separation in mind between GUI and internal features. In particular, it can be run without GUI (i.e. without creating the `JViewer` object), and things will work exactly the same, though invisibly. As such, JBOTSIM can be used to perform batch simulations (e.g. sequences of unattended runs that log the effects of some varying parameter). This also enables to withstand heavier simulations in terms of the number of nodes and links.

This being said, one of the most distinctive features of JBOTSIM remains *interactivity*, e.g., the ability to challenge the algorithm in difficult configurations through adding, removing, or moving nodes during the execution. This approach proves useful to think of a problem visually and intuitively. It also makes it possible to explain someone an algorithm through showing its behavior.

The architecture of JBOTSIM's viewer is depicted on Figure 5. As one can see, the viewer relies heavily on events related to nodes, links, and topology. The influence also goes the other way, with mouse actions being translated into topological operations. These features are realized by a class called `JTopology`. This class can often be ignored by the developer, which creates and manipulates the viewer through the higher `JViewer` class. The latter adds external features such as tuning slide bars, popup menus, or self-containment in a system window.

While natural to JBOTSIM's users, the viewer remains, in all technical aspects, an independent piece of software. Alternative viewers could very well be designed with specific uses in mind.

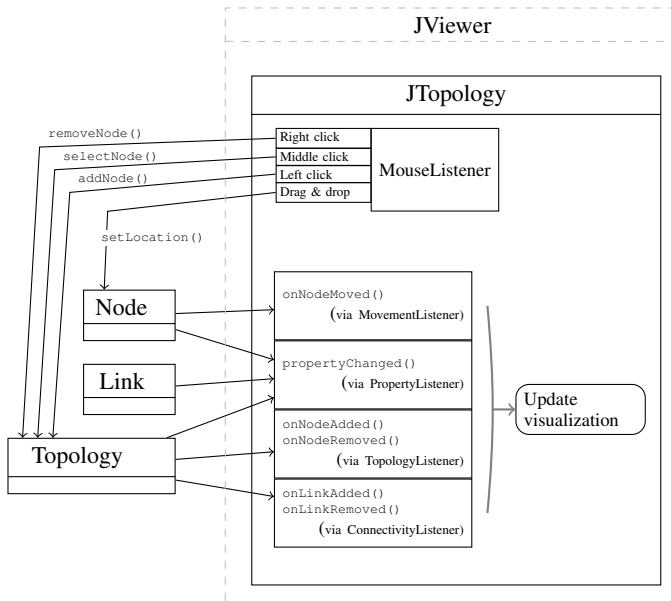


Figure 5: Internals of JBotSim's GUI

4. A ZOOM ON SELECTED FEATURES

This section provides more details on a handful of selected features. It covers, namely, the topics of message passing, graph-level algorithms, and java applets for online web demos.

4.1 Exchanging messages

The way messages are used in JBOTSIM is independent from the communication technology considered. Indeed, the API is quite simple, messages are sent directly by the sender, through calling the `send()` method on its own instance (inherited from `Node`). Messages are typically received through overriding the `onMessage()` method (also from class `Node`). Another way to receive messages, which is not event-based, is for a node to check its mailbox manually through the `getMailbox()` method, for instance when it is executing the `onClock()` method (this is the natural way to implement round-based communication models).

Listing 7 shows a message-based implementation of the flooding principle. Initially, none of the nodes are informed. Then, if a node is selected (either through middle-click or through direct call to the `selectNode()` method), then this node is notified in the `onSelection()` method and initiates a basic broadcast scheme. Here, the algorithm consists in retransmitting the received message upon first reception to all the local neighbors (`sendAll()`). In this example, the message is empty, however in general any object can be inserted in the message (no copy is made and the very same object is to be delivered, unless a copy is made at sending time). By default, a message takes one time unit to be transmitted (i.e. it is delivered at the next clock pulse). The delivery happens only for those links which exist by the time of reception. Since any object can be used as message content, it often has to be cast upon reception. All these aspects correspond to JBOTSIM's default message engine. Other message engines can be written, and indeed some exist in the `jbotsimx` package. For instance, the `DelayMessageEngine` makes it possible to add deterministic or random delays to the delivery of messages, possibly even in a non FIFO manner.

4.2 Working at the graph level

Listing 7 Example of message passing algorithm

```
public class FloodingNode extends Node{

    boolean informed = false;

    @Override
    public void onSelection() {
        informed = true;
        sendAll(new Message());
    }

    @Override
    public void onMessage(Message message) {
        if (!informed){
            informed = true;
            sendAll(message);
        }
    }
}
```

Implementing an algorithm in the message passing model is sometimes difficult. JBOTSIM makes it possible to sketch an idea, play with it, and share it with others, in minutes, thanks to working at a (more abstract) graph-level. This level of abstraction also is relevant in its own right, when the object of study is itself at the graph level, such as classical *graph algorithms* or distributed coarse-grain models like *graph relabeling systems* [9] or *population protocols* [2].

To illustrate the simplicity of the graph level, let us consider a scenario where a type of node called `SocialNode`, dislikes being isolated. Such a node is happy (green) if it has at least one neighbor, unhappy (red) otherwise. In the message passing paradigm, this principle would require to send periodic messages (beacons) and track the reception of these messages, as well as using a timer to decide when a node becomes isolated. Listing 8 shows a possible implementation at the graph level, which is pretty concise and self-explanatory. Topological events are directly detected by the

Listing 8 Example of graph-based algorithm

```
public class SocialNode extends Node{
    public SocialNode(){
        setColor(Color.red);
    }
    public void onLinkAdded(Link l){
        setColor(Color.green);
    }
    public void onLinkRemoved(Link l){
        if (!hasNeighbors())
            setColor(Color.red);
    }
}
```

nodes, which can update their status. (These events could also be listened to globally through the `ConnectivityListener()` interface. This is the way one might want to implement centralized dynamic graph algorithms.) Of course, from a message passing perspective, this implementation is cheating. Thus, methods like `onLinkAdded()`, `hasNeighbors()`, or `getNeighbors()` should not be used in a message-passing setting.

4.3 Embedding JBOTSIM in a java applet

One of the features of JBOTSIM's viewer is to create a windowed frame automatically for the topology. This is the default behavior of `JViewer`'s constructor when a single parameter of type `Topology` or `JTopology` is used. Other behaviors can be obtained by using different versions of the constructor. In particular, one can specify that *no* windowed frame should be created, and the `JTopology` object be plugged manually into a different

container. This feature enables the customization of JBOTSIM’s UI at will, as well as the creation of java applets.

Listing 9 shows an example java applet corresponding to the HelloWorld program from Listing 1. Here, a JTopology object

Listing 9 Embedding a JBOTSIM demo into a java applet

```
import javax.swing.JApplet;

import jbotsim.Clock;
import jbotsim.Topology;
import jbotsim.ui.JTopology;
import jbotsim.ui.JViewer;

@SuppressWarnings("serial")
public class HelloWorld_Applet extends JApplet{
    Topology tp;
    public void init(){
        tp = new Topology();
        JTopology jtp = new JTopology(tp);
        new JViewer(jtp, false);
        this.add(jtp);
    }
    public void destroy(){
        tp.pause();
    }
}
```

(which is a JPanel) is created manually from the topology. Its reference is then used to augment it with standard viewer features, and finally added to the applet container. Another important step is to pause the clock in the destroy() method (otherwise JBotSim’s engine would keep running in background).

5. CONCLUDING REMARKS

In my view, JBOTSIM is a *kernel*, in the sense that it encapsulates a number of generic features whose purpose is to be used by higher programs. As of today, the plan is to keep it this way and try containing the growth of the number of features, to the profit of quality and simplicity. This does not mean, of course, that JBOTSIM should not be extended *externally*.

In particular, JBOTSIM’s distribution already incorporates an extension package called `jbotsimx`, in which more specific features could be found. For instance, it incorporates a static class called `Tikz` that makes it possible to export the current topology as a TikZ picture – a powerful format for drawing pictures in L^AT_EX documents. Figure 6 illustrates this with two pictures that I

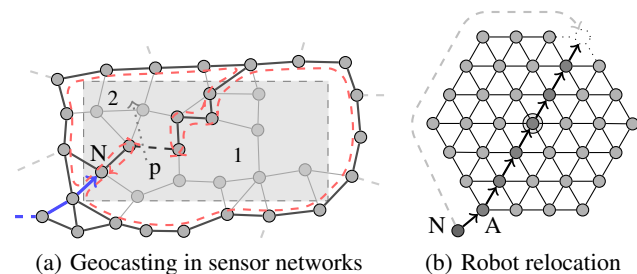


Figure 6: Two examples of pictures whose underlying topology was generated using JBOTSIM Tikz extension. The topology on the left was created by adding and moving nodes using the mouse; the topology on the right was generated by program. Once exported as a Tikz picture, they were tweaked manually to add shading, color, etc.

have generated. The Tikz class is composed of a single method,

`exportAsTikz()`, which takes a mandatory `Topology` argument, and an optional scaling argument.

Other extensions include basic topology algorithms for testing, e.g., if a given topology is connected or 2-connected, if a given node is critical (its removal would disconnect the graph) or compute the diameter. A set of extensions dedicated to dynamic graph are currently being developed (by myself and others). For instance, the `EMEGPlayer` takes as input a birth rate, death rate, and an underlying graph (given as a `Topology`), and generates an edgemarkovian dynamic graph based on these parameters, the dynamics of which can be listened to through the `ConnectivityListener` interface.

Contributions are most welcome, as well as suggestions of improvement or feature requests.

References

- [1] JBOTSIM’s website: <http://jbotsim.sf.net/>.
- [2] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta, “Computation in networks of passively mobile finite-state sensors,” *Distributed Computing*, vol. 18, no. 4, pp. 235–253, 2006.
- [3] M. Bastian, S. Heymann, and M. Jacomy, “Gephi: An open source software for exploring and manipulating networks,” in *International AAAI conference on weblogs and social media*, vol. 2. AAAI Press Menlo Park, CA, 2009.
- [4] M. Bauderon and M. Mosbah, “A unified framework for designing, implementing and visualizing distributed algorithms,” *Electr. Notes Theor. Comput. Sci.*, vol. 72, no. 3, pp. 13–24, 2003.
- [5] Collective Authors, “The NS-3 network simulator,” <http://www.nsnam.org/>, 2009.
- [6] B. Derbel, “A Brief Introduction to ViSiDiA,” USTL, Tech. Rep., 2007.
- [7] 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.
- [8] A. Keränen, J. Ott, and T. Kärkkäinen, “The one simulator for dtn protocol evaluation,” in *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, p. 55.
- [9] I. Litovsky, Y. Métivier, and É. Sopena, “Graph relabelling systems and distributed algorithms,” *Handbook of graph grammars and computing by graph transformation*, vol. 3, pp. 1–56, 1999.
- [10] A. Varga *et al.*, “The OMNeT++ discrete event simulation system,” in *Proceedings of the European Simulation Multiconference (ESM’01)*, 2001, pp. 319–324.