The JBotSim Library

Arnaud Casteigts

The JBOTSIM Library

Arnaud Casteigts

LaBRI, University of Bordeaux

arnaud.casteigts@labri.fr

July 1, 2013

***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 your 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 tools, JBOTSIM is not an integrated environment. It is a lightweight library to be used in your program. In this paper, we present an overview of its distinctive features and architecture.**

I. INTRODUCTION

JBotSim is an open source (LGPL) simulation library dedicated to distributed algorithms in dynamic networks. With it, you can prepare live demos of your algorithm, interact with it while it is running, and evaluate its performance. JBOTSIM is not a competitor of mainstream simulators such as NS3, OMNet, or The One (see [3, 6, 9]), in that it does not implement real-world networking stacks. Quite the opposite, JBOTSIM aims to remain a technology-insensitive tool to be used mostly at the algorithmic level. It is closer, in spirit, to the ViSiDiA project – a general purpose visualization environment for distributed algorithms [4, 8].

Whether your algorithm is centralized or distributed, the natural way of programming in JBOTSIM is event-driven: algorithms are defined as subroutines to be executed when particular events occur (appearance or disappearance of a link, arrival of a message, timer 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. We start with some preliminaries regarding installation and documentation. Section III reviews JBOTSIM's main components and specificities such as programming paradigms, clock scheduling, user interaction, or global architecture. Section IV zooms on key features such as the exchange of messages between nodes, graph-level APIs, or the creation of online demos. Finally, we discuss in Section V some extensions of JBOTSIM, including a Ti*k*Z exportation feature and an edge-markovian dynamic graph generator.

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

II. PRACTICAL PRELIMINARIES

In this short section, we help you install JBOTSIM, write, and run a first program. Useful links to online documentation and examples are also given to explore the API further and practice it beyond this paper.

A. *Fetching* JBOTSIM

The straightest (and safest) way to obtain JBOTSIM is to fetch the latest official release, as a JAR package (`http://jbotsim.sf.net/jbotsim.jar`). If you are more daring, you can get the latest development version from SourceForge's repository and compile it yourself as shown below. Doing this, you likely get new features, but less conformity to online documentation. Here is the command:

```
> svn co svn://svn.code.sf.net/p/jbotsim/code/ target
```

where `target` is the place you want to put JBOTSIM's source code in. From within that directory, you can produce the JAR package by typing `make`.

JBOTSIM does not use version numbers. So far it proved more convenient. New features are released frequently and users get in touch with me to discuss any issue. This might change in the future.

B. *HelloWorld with* JBOTSIM

Whether you downloaded the JAR package or compiled it, you can check your setting with the following program. To do so, copy the code from Algorithm 1 into a file named `HelloWorld.java`.

Algorithm 1 HelloWorld with JBOTSIM

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

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

If you are working from the terminal, you may add `jbotsim.jar` to your `CLASSPATH` environment variable or use the equivalent options as follows:

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

If you are running *Eclipse* or a similar IDE, add `jbotsim.jar` to the build path of your project (*Project > Properties > Java build path > Librairies > Add external jar*). Now run your program. If you see an empty surface where you can add nodes, move them, or delete them with the mouse, then you are all set.

C. Sources of documentation

In this document, we provide a general overview of what JBOTSIM is and how it is designed. This is by no means a comprehensive programming handbook. The reader who wants to explore deeper some features or develop complex programs with JBOTSIM is referred to the API documentation, available at `http://jbotsim.sf.net/javadoc/`.

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 online examples are augmented with interactive visualization, which you can see if your browser supports java applets. Finally, most examples given in this paper are available on JBOTSIM's website. Feel free to check them when the code given here is incomplete (e.g. we often omit package imports and `main()` methods for conciseness).

III. JBOTSIM'S FEATURES AND ARCHITECTURE

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

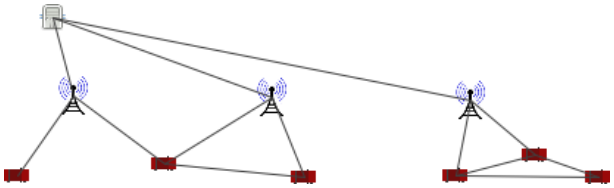


Figure 1. A highway scenario composed of vehicles, road-side units, and central servers. Part of the network is ad hoc (and wireless); the rest is infrastructured (and wired).

A. Basic features of nodes and links

JBOTSIM consists of a small number of classes, the most central of which are `Node`, `Link`, and `Topology`. The contexts in which dynamic networks manifest 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 this diversity of 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-infrastructured: Servers share a dedicated link with each tower. This link is *wired* and exists independently from the distance. On the other hand, towers and vehicles communicates through wireless links that are automatically updated.

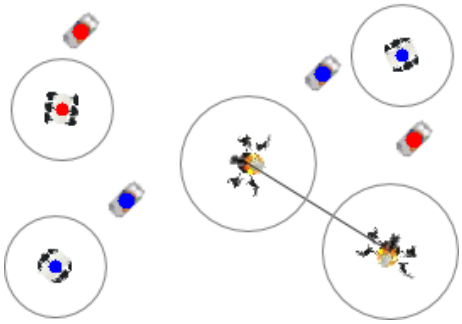


Figure 2. A swarming scenario, whereby mobiles robots and UAVs collaborate to clean a public park.

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 the position and start searching for a capable robot. In addition to sensing capabilities, UAVs can communicate wirelessly to share environmental information.

Besides nodes and links, the concept of topology is central in JBOTSIM. As far as we are concerned here, they 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 we will see later on.

B. Distributed vs. centralized algorithms

JBOTSIM supports the manipulation of centralized or distributed algorithms (sometimes both simultaneously). The natural way to implement a distributed algorithm is by defining a class that inherits from the `Node` class. Centralized algorithms are not constrained to a particular model, they can take the form of any standard java class.

a) 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. Algorithm 2 provides a basic example in which the nodes are endowed with self-mobility. The class relies on a key mechanism

Algorithm 2 Extending the `Node` class

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

public class MovingNode extends Node
                        implements ClockListener{
    public MovingNode(){
        Clock.addClockListener(this, 5);
        setDirection(Math.random() * 2*Math.PI);
    }
    public void onClock(){
        move(2);
    }
}
```

in JBOTSIM: performing periodical operations that are triggered by the pulse of the system clock. This is done by registering your object as a listener to the static instance `Clock`, typically during construction, using `addClockListener()`, then filling in your code into the `onClock()` method. This code will be executed by JBOTSIM at the specified frequency – here, every five pulses of the clock (whose rate can be set independently). The rest of the code is responsible for moving the node, setting a random direction at construction time (in radian), then moving in this direction periodically. (More details about the movement API can be found online.)

Once your class is defined, there are two ways of using it, depending on whether the scenario is set up by program or interactively. In the first case, simply call JBOTSIM's methods with instances of your class instead of the original node instances. In Algorithm 3, we create a new topology, then add ten randomly positioned nodes of the new type using the `addNode()` method. In order to use moving nodes in an interactive scenario, you can register them as as a node model. This is done by calling the static node method `setModel()` with a prototypal instance of this class as argument, as shown on Algorithm 4. JBOTSIM's GUI (also called the *viewer*) will consult the list of model whenever a new node is added with the mouse and display a selection menu if several models exist (in this case, only one model is set and will be used automatically). In the scenario of Figure 1, left-clicking on the surface would give the choice between *car*, *tower*, and *server*, the names of the three registered models.

Algorithm 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);
}
```

Algorithm 4 Using a defined node as default

```
public static void main(String[] args){
    Node.setModel("default", new MovingNode());
    new JViewer(new Topology(400,300));
}
```

b) 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. Algorithm 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 every four pulses of the clock. For each node, a target is created if it does

Algorithm 5 Centralized version of Random Waypoint

```
public class CentralizedRWP implements ClockListener{
  Topology tp;
  public CentralizedRWP (Topology tp){
    this.tp = tp;
    Clock.addClockListener(this, 4);
  }
  public void onClock(){
    for (Node n : tp.getNodes()){
      Point2D target=(Point2D)n.getProperty("target");
      if (target == null ||
          n.getLocation().distance(target)<2){
        target = new Point2D.Double(Math.random()*400,
                                    Math.random()*300);
        n.setProperty("target", target);
        n.setDirection(target);
      }
      n.move(2);
    }
  }
  public static void main(String[] args){
    Topology tp = new Topology(400,300);
    new CentralizedRWP(tp);
    new JViewer(tp);
  }
}
```

not exists yet or has just been reached; then the node's direction is set accordingly and the node is moved (by 2 units). For convenience, we include the `main()` method in the same class.

Notice the use of `setProperty()` and `getProperty()` in this example. These methods allow to store any object directly into the node, then retrieve it with a string identifier (alike a map object). Both links and topologies also have the same feature.

C. Architecture of the event system

Up to now, we have met one type of events – clock pulses, to be listened to through the `ClockListener` interface. JBOTSIM is architectured around a number of such events and interfaces, some of which become ubiquitous in any reasonably large program. The most important of them are depicted on Figure 3 on the following page. This architecture allows one to monitor the network state and associate reactive operations to various events. For instance, you may ask to be notified whenever a link appears or disappears somewhere. This can be done relative to a single node (local changes) or relative to the whole topology. 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).

Algorithm 6 gives yet another example, consisting of a mobility trace recorder. This program listens to topological events at large, 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 [5]. Similar code could be written for Gephi [2].

Algorithm 6 Example of a mobility trace recorder

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

  // TopologyListener
  public void nodeAdded(Node node) {
    println("an␣" + node + "␣x:" + node.getX() +
            "␣y:" + node.getY());
  }
  public void nodeRemoved(Node node) {
    println("dn␣" + node);
  }

  // ConnectivityListener
  public void linkAdded(Link link) {
    println("ae␣" + link + "␣" + link.source +
            "␣" + link.destination);
  }
  public void linkRemoved(Link link) {
    println("de␣" + link);
  }

  // MovementListener
  public void nodeMoved(Node node) {
    println("cn␣" + node + "␣x:" + node.getX() +
            "␣y:" + node.getY());
  }
}
```

Note that other events exist beyond those represented in Figure 3. In particular, the `SelectionListener` interface allows us to be notified when a node is selected. This feature is particularly helpful to trigger computation during interaction, e.g., to have some node initiate a broadcast.

D. Mono threading: why and how?

It seems natural to assign every node a dedicated thread, so that each node executes its algorithm in parallel – just as in the real world. Yet, it was decided that JBOTSIM be mono threaded, and definitely so. This section explains why and how this was done. Understanding these points can be instrumental in developing well-organized and bug-free programs.

In JBOTSIM, all the nodes, and in fact all of JBOTSIM's life at large (GUI excepted) is articulated around a single thread, which is driven by the central *clock*. The clock wakes up at regular interval (default rate is 10ms) and notifies its listeners in a specific order. JBOTSIM's internal engines, e.g. the message delivery engine, are served first. Then comes the turn of those nodes whose

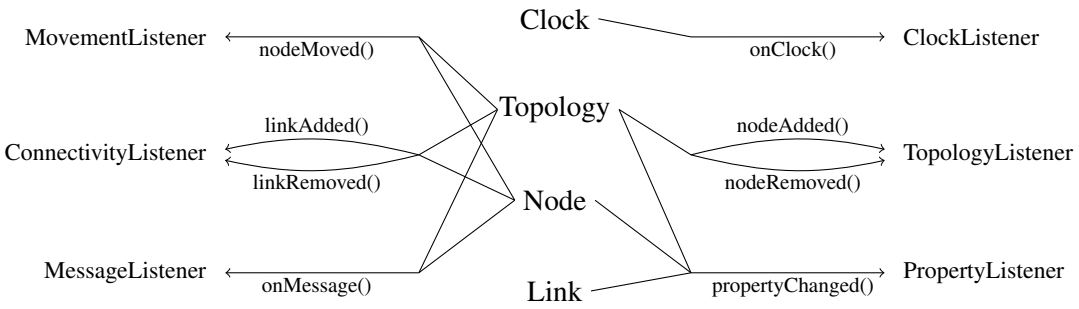


Figure 3. Main sources of events and corresponding interfaces in JBOTSIM.

wait period has expired (others remain asleep). They are notified in a random order. Hence, if all nodes listen to the clock at a rate of 1, they will all be notified in a random order in each round, which makes JBOTSIM's scheduler a 2-bounded fair one. (Other policies could similarly be realized by twicking the `onClock()` method. Eventually, we will add explicit support for some of them.) A simplified version of the scheduling process is depicted on Figure 4.

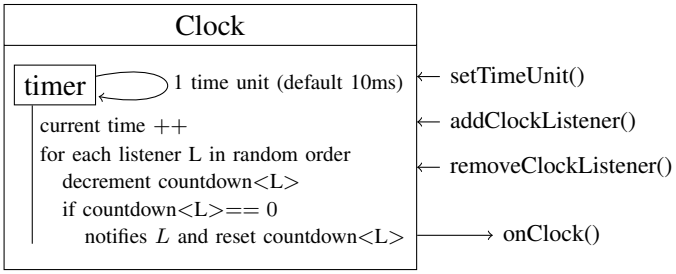


Figure 4. Simplified version of the internal scheduler.

One consequence of mono threading is that all computations – again, 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 should be careful, though, not to interfer with this thread from another user-defined thread. The canonical example is when a scenario is set up by program from within the thread of the `main()` method. If your initialization makes extensive use of JBOTSIM's API from within that thread, you will sooner or later get a `ConcurrentModificationException`. The easy way around this problem is to insert a `Clock.pause()` instruction before your initialization code, and `Clock.resume()` right after. This type of measure is what the GUI does in some cases to avoid conflicts.

E. Interactivity

JBOTSIM was designed with in mind a clear separation between GUI and internal features. In particular, it can be run without GUI – just omit creating the viewer in your `main()` method, things will work the same invisibly. Hence, 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 if graphical drawing is not required.

This being said, one of the most distinctive features of JBOTSIM remains *interactivity*, e.g., the ability to challenge your algorithm in difficult configurations by adding, removing, or moving nodes while it is executing. This approach also 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 makes extensive use of events related to nodes, links, and topology. The

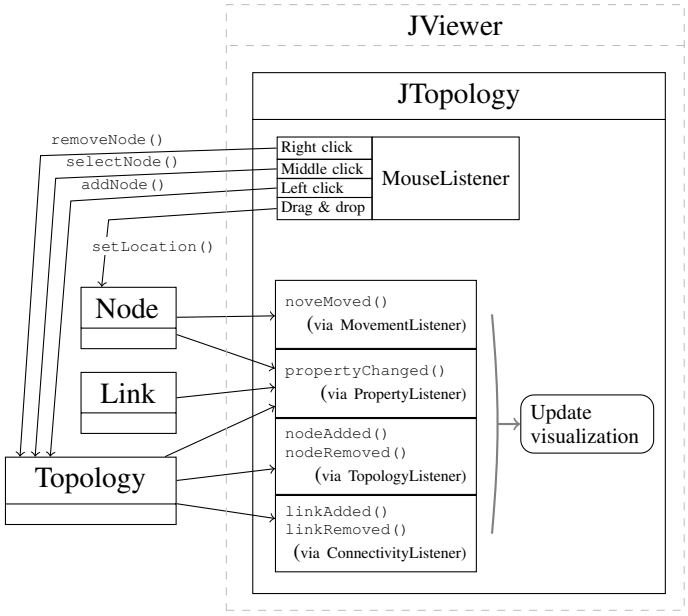


Figure 5. Internals of JBotSim's viewer

influence goes both ways, with mouse actions being assigned to 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. This class 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 as well be designed with specific uses in mind.

IV. A ZOOM ON SELECTED FEATURES

This section offers a zoom on some of JBOTSIM's features. We cover in particular the exchange of messages, fast prototyping of ideas at a graph level, and the realization of java applets to prepare online demos.

A. Exchanging messages

The way messages are used in JBOTSIM is independent from the technology considered. The API is quite simple, messages are sent by calling the `send()` method on the sender node. They are typically received through a `onMessage()` method that comes from the `MessageListener` interface. Another way to receive messages, not event-based, is for a node to check its mailbox manually – a relevant technique in rounded communication models.

Algorithm 7 shows a message-based implementation of the flooding principle, whereby informed nodes retransmit periodically. None of the nodes are informed initially. Assume an external entity sets the information at one of the nodes at some point. This node,

Algorithm 7 A message-based flooding algorithm

```
public class CommunicatingNode extends Node
                implements ClockListener, MessageListener{
    String information = null;
    public CommunicatingNode(){
        this.addMessageListener(this);
    }
    public void onClock() {
        if (information != null)
            send(null, information);
    }
    public void onMessage(Message msg) {
        information = (String)msg.content;
    }
}
```

and subsequently all informed node will send this piece of information periodically based on clock pulses (`ClockListener` interface). The information, here a object of type `String`, is sent to all neighbors at once. The choice of addressee is specified by the first parameter of the `send` method: `null` to send the message to all neighbors, or a reference of type `Node` to send the message to that node only. Any object can be used as message *content*. This object will be received as a generic object by the receiver (be careful, this is not a copy!), in the `content` field of the received `Message`. Thus, it has to be explicitely cast. Finally, delivery is not immediate, it occurs after a specified number of clock pulses (one, by default). The message will successfully be received if and only if the link is still present at delivery time.

B. Working at the graph level

Implementing an idea in the message passing model can reveal fastidious or even sometimes impossible. Whether you want to visualize your idea, play with it, or show it to others, a full implementation is uncessary effort. JBOTSIM makes it possible to try an idea at a graph level before switching to message-based implementation.

Consider the following example: we want to design a type of node, the `SocialNode`, who likes the company of others. Such a node is happy when it is in the vicinity of other nodes, unhappy otherwise. Algorithm 8 proposes a graph-based implementation of this node that is pretty concise. Here the nodes can directly react to topological

Algorithm 8 A graph-based distributed algorithm

```
public class LonerNode extends Node
                    implements ConnectivityListener{
    public LonerNode(){
        setColor("red");
        addConnectivityListener(this);
    }
    public void linkAdded(Link l){
        setColor("green");
    }
    public void linkRemoved(Link l){
        if (!hasNeighbors())
            setColor("red");
    }
}
```

events irrespective of messages. These events are notified through the `ConnectivityListener` interface. In this example, the nodes get only notified for those links that are *local*, due to calling `addConnectivityListener()` on their own instance rather than on the topology – both nodes and topologies implement this method, the latter doing it network-wide.

Of course, from a message passing perspective, this implementation is cheating. Similarly, methods like `hasNeighbors()` or `getNeighbors()` should be used with caution as they work at a graph level. Note that this level of abstraction is relevant in its own right and

has led many interesting studied in the field of distributed computing, see for instance *graph relabeling systems* [7] and *population protocols* [1].

C. Turning your demo into an applet

One of the features of JBOTSIM's viewer is to create a windowed frame automatically for your topology. This is the default behavior of `JViewer`'s constructor when a single parameter of type `Topology` or `JTopology` is given as argument. Other behaviors can be obtained by using other 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 leisure, as well as it enables the creation of java applets.

By way of example, Algorithm 9 shows an applet version of the HelloWorld program from Algorithm 1. Here, the `JTopology` object is created manually from a new 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. Omitting this step may cause it to keep running in the background, even after the applet page was left.

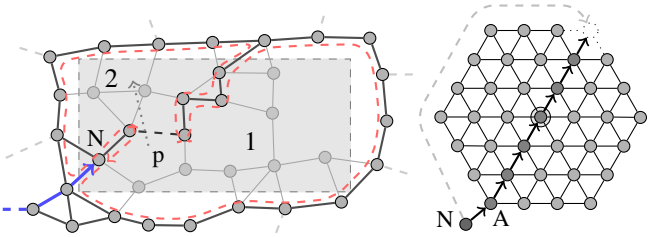
Algorithm 9 Turning your demo into an applet

```
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{
    public void init(){
        JTopology jtp = new JTopology(new Topology());
        new JViewer(jtp, false);
        this.add(jtp);
    }
    public void destroy(){
        Clock.pause();
    }
}
```

V. CONCLUDING REMARKS

I 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 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 allows one to export the current topology as a Ti*k*Z picture – a powerful format for drawing pictures in LaTeX documents (among others). Figure 6 illustrates this with two pictures



(a) Geocasting in sensor networks (b) Robot relocation

Figure 6. Two examples of pictures that were first generated using JBOTSIM and the Ti*k*z extension, then twicked manually to fit the need of a particular illustration.

that I had to generate for another paper. 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 developped (by myself and others), some of which are already available. For instance, the `EMEGPlayer` takes as input a birth rate, death rate, and an underlying graph (given as a `Topology`), and generates an edge-markovian dynamic graph behavior based on these parameters, the events of which can be listened to in your algorithm in a classical way.

Contributions are most welcome, as well as suggestions to add some particular feature or extension which you think could be useful.

REFERENCES

[1] 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.

[2] 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.

[3] Collective Authors, "The NS-3 network simulator," http://www.nsnam.org/, 2009.

[4] B. Derbel, "A Brief Introduction to ViSiDiA," USTL, Tech. Rep., 2007.

[5] 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.

[6] 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.

[7] 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.

[8] M. Mosbah and A. Sellami, "Visidia: A tool for the visualization and simulation of distributed algorithms," University of Bordeaux, Tech. Rep., 2003.

[9] A. Varga *et al.*, "The OMNeT++ discrete event simulation system," in *Proceedings of the European Simulation Multiconference (ESM'01)*, 2001, pp. 319–324.