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

Arnaud Casteigts
LaBRI, University of Bordeaux
arnaud.casteigts@labri.fr

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.

Keywords: Mobile ad hoc networks, Distributed Algorithms, Interactive Simulation, Time-Varying Graphs

1. INTRODUCTION

JBotSim is an open source simulation library 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). JBotSim can also be used to prepare live demos of an algorithm and show it to colleagues or students, as well as to assess the algorithm performance. It is not a competitor of mainstream tools like NS3 [3], OMNet [7], or The One [6], as 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.

JBotSim comes as a JAR package that can be found on the website [1]. Once in the classpath, one can start using the API. Please refer to the long version of this paper for more information regarding installation, first steps, or the API. An online javadoc is also available. Besides its features, the main asset of JBotSim is its simplicity of use. A basic program is shown on Listing 1. It results in a gray surface where nodes can be added, moved, or deleted using the mouse.

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());
    }
}
```

2. OVERVIEW

JBotSim consists of a small number of classes, the most central being `Node`, `Link`, and `Topology`. 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. Link can be directed or undirected, and wired or wireless – in the latter case, JBotSim updates the set of links automatically.

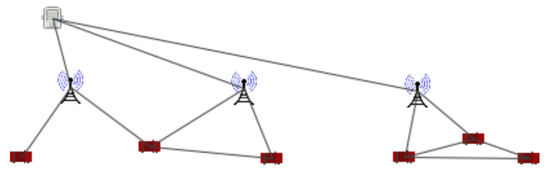

Figure 1: A highway scenario composed of vehicles, road-side units, and central servers.

[*]A longer version is available on arXiv (abs/1001.1435).

SIMUTOOLS 2015, August 24-26, Athens, Greece
Copyright © 2015 ICST
DOI 10.4108/eai.24-8-2015.2261067

Figures 1 and 2 illustrate two different uses of JBotSim. 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: part of the network is wired and static, the other part is wireless and dynamic. Figure 2 illustrates a purely *ad hoc* scenario, where swarms of UAVs and robots strive to clean a public park collectively. In this scenario, robots can clean wastes of a certain type (red or blue) only if these are within their *sensing range* (depicted by a surrounding circle). UAVs detect the wastes and provide their location to the robots.

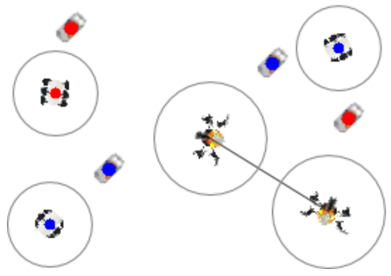

Figure 2: A swarming scenario, whereby mobiles robots and UAVs collaborate in order to clean a public park.

2.1 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 (and yet, rely on the API as needed).

Distributed algorithm.

JBotSim comes with a default type of node that is implemented in the `Node` class. This class provides general features for moving or exchanging messages, among others. Distributed algorithms are implemented through extending this class. Listing 2 provides a basic example in which the nodes are endowed with self-mobility. In this example the `onClock()` method is overridden in order to perform some action (here, moving) periodically. The rest of the code is responsible for setting a random direction at construction time (in radian).

Listing 2 Extending the `Node` class

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

Once this class is defined, new nodes of this type can be added to the topology either manually, e.g. using `tp.addNode()`, or by telling JBotSim that new nodes should, by default, be of this type (`setDefaultNodeModel(MovingNode.class)`). Several models can be registered, in which case JBotSim's GUI displays a selection list when a node is added.

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 start designing a distributed algorithm in a centralized way. Centralized algorithms must not inherit from a given class. They can simply use the API from their own class. Listing 3 gives an example of a central algorithm that records into a dynamic graph the traces of connectivity of the nodes (here, in a format close to that of [5])

Listing 3 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 fact, most events available to the nodes through overridding methods (e.g. `onClock()`, `onMessage()`, `onLinkAdded()`, `onSensingIn()`, `onSelection()`, *etc.*) are also available to other classes through the corresponding interfaces (`ClockListener`, `MessageListener`, `ConnectivityListener`, etc.).

2.2 Exchanging messages

The API for messages in JBotSim is simple. Messages are sent through calling the `send()` method on the sender node, and they are received through overriding the `onMessage()` method or by scrutinizing the mailbox manually. Listing 4 shows an example

Listing 4 Example of message-based broadcast

```
public class BroadcastNode extends Node{
    boolean informed = false;
    @Override
    public void onSelection() {
        informed = true;
        sendAll(new Message("MY␣MESSAGE"));
    }

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

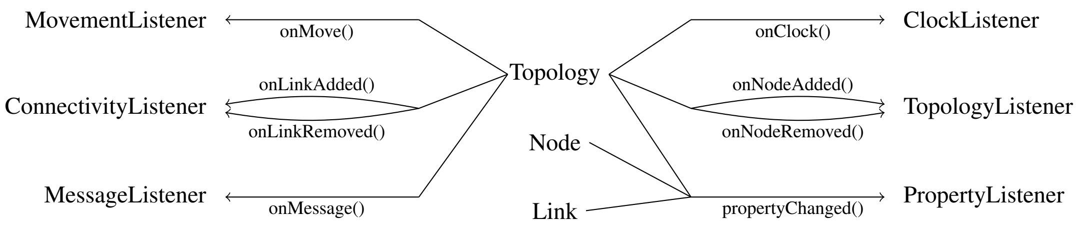

Figure 3: Main sources of events and corresponding interfaces in JBOTSIM.

of broadcast algorithm, where the initial emitter is selected by the user (middle click in the GUI). Any object can be inserted in a message (here a String). By default, messages take one time unit to be transmitted. This can be tuned in a number of ways.

2.3 Working at the graph level

In addition to messaging, JBOTSIM makes it possible to work at the (more abstract) graph level. Consider an example 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. Listing 5 shows a possible implementation of this principle at graph level.

Listing 5 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);
    }
}
```

2.4 Interactivity

I designed JBOTSIM with a clear separation in mind between GUI and underlying logic. As such, JBOTSIM can be used without GUI and works just the same (except for interaction, of course). Hence, JBOTSIM can be used to perform batch simulations of unattended runs, e.g. to log the effects of varying some parameters.

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 the viewer is depicted on Figure 4. As one can see in the figure, the main component is actually the `JTopology`. It is noteworthy that this component can be embedded in other containers than a `JViewer`. As a result, JBOTSIM's GUI can be embedded in other software, such as simulation environments or even java applets (see the long version for details).

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.

3. CONCLUDING REMARKS

I my view, JBOTSIM is a *kernel* that focuses on generic features whose purpose is to be used or extended by others. The current

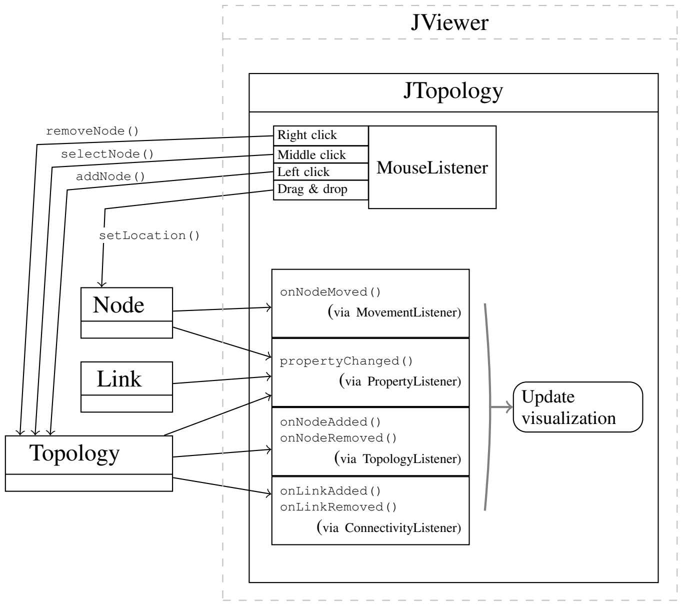

Figure 4: Internals of JBotSim's GUI

distribution includes an extension package called `jbotsimx`, in which more specific features could be found, such as algorithms for connectivity testing or manipulation of time-varying graphs. Contributions are most welcome, as well as suggestions of improvement and feature requests.

References

[1] JBOTSIM's website: `http://jbotsim.sf.net/`.

[2] M. Bauderon and M. Mosbah, "A unified framework for designing, implementing and visualizing distributed algorithms," *Electr. Notes on TCS*, vol. 72, no. 3, pp. 13–24, 2003.

[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," In Proc. of *EPNACS*, 2007.

[6] A. Keränen, J. Ott, and T. Kärkkäinen, "The one simulator for dtn protocol evaluation," in Proc. of *SIMUTools*, 2009.

[7] A. Varga *et al.*, "The OMNeT++ discrete event simulation system," in Proc. of *ESM*, 2001.