Autonomic networks - S1

TME0 -- Installation and test environment TME JBotSim

During all the practical work sessions (TMEs), you will use the simulator *JBbotSim* developed by Arnaud Casteigts. [*Arnaud Casteigts, Remi Laplace.* ***JBotSim: a Tool for Fast Prototyping of Distributed Algorithms in Dynamic Networks.*** *In Proc of SIMUTools 2015]*

Note that *JBotSim* requires *Java* on your system, Release 8 or later. The exercises are supposed to be performed in a terminal, under *linux*. However, they also work (and have been tested) under MacOS/X. They were not tested under Microsoft Windows.

First, in a source directory (for instance ANET-TME), create the following directories:

- lib
- TME0_HelloWorld
- TME1_DTN

Then, enter the lib directory and download the attached file *jbotsim-standalone-1.2.0.jar*. You are ready to execute the 3 TMEs.

The documention is available at https://jbotsim.io/javadoc/1.2.0/.

The remainder of this first session is directly adapted from the exemples provided at https://jbotsim.io.

Hello World

- First set TME0 as the current directory.
- Next, download the attached source file *HelloWord.java*.
- Compile the file to create the corresponding class file:

```
javac -cp .:../lib/jbotsim-standalone-1.2.0.jar HelloWorld.java
```

- Then, execute HelloWorld as follows:

```
java -cp .:../lib/jbotsim-standalone-1.2.0.jar HelloWorld &
```

You should see an empty window in which you can add new nodes (left click), delete them (right click on the node), or move them around (drag & drop). (*Note that on MAC OS/X, click or tap with one finger to emulate the left button, and click or tap with two fingers to emulate the right button.*)

In this basic example:

1. We create a `Topology`, which is the main object of *JBotSim*. This object centralizes information about the simulation; it also manages the nodes and links, and organizes the inner life of the system (timing, messaging, etc.).
2. We pass it to a `JViewer`, which will display the simulation elements and allow the user to interact with it.

Right now, the nodes are basic nodes provided by default by *JBotSim*.

Evolved Nodes

Node algorithms (*a.k.a* distributed algorithm) are created in three steps:

1. Create a class that extends `Node`
2. Write the algorithm through overriding methods
3. Tell *JBotSim* to use your type of nodes

Consider the following skeleton of a node algorithm where four methods are overridden.

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

import io.jbotsim.core.Node;
import io.jbotsim.core.Color;
import io.jbotsim.core.Message;


public class EmptyNode extends Node{
    @Override
    public void onStart() {
        // JBotSim executes this method on each node upon initialization
    }

    @Override
    public void onSelection() {
        // JBotSim executes this method on a selected node
    }

    @Override
    public void onClock() {
        // JBotSim executes this method on each node in each round
    }

    @Override
    public void onMessage(Message message) {
        // JBotSim executes this method on a node every time it receives a message
    }
}
```

Coding an algorithm boils down to insert code in the above method. Most algorithms can be specified using 2 or 3 such methods. Other methods exist, and you can add your own.

This is done through calling `setDefaultNodeModel()` on the topology object, passing it your class of node. Typically, this is made in the `main()` method, itself being located either in the same class (not very clean) or in a dedicated class, as below:

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

public class Main{
    public static void main(String[] args){
        Topology tp = new Topology();
        tp.setDefaultNodeModel(EmptyNode.class);
        new JViewer(tp);
        tp.start();
    }
}
```

Let us now write a node algorithm that effectively does something, namely a *broadcasting* algorithm using messages.

Basic Message Passing Distributed Model

Now, we will create a simple message-passing algorithm, in which a selected node disseminates a piece of information to all the other nodes (through multi-hop). Most often, the algorithm is coded through the three following steps:

1. Create a class that extends `Node`
2. Override some methods, namely `onStart()`, `onSelection()`, `onMessage()`
3. Tell JBotSim to use this class

Using the above skeleton *EmptyNode*, create a class *BroadcastNode* (in a new source file *BroadcastNode.java*) and override the methods *onStart()*, *onSelection()*, and *onMessage()* as follow:

```
    public void onStart() {
        boolean informed = false;
        setColor(null);
    }

    public void onSelection() {
        informed = true;
        setColor(Color.RED);
        sendAll(new Message("My message"));
    }

    public void onMessage(Message message) {
        if (! informed) {
            informed = true;
            setColor(Color.RED);
            sendAll(new Message(message.getContent()));
        }
    }
```

Before executing this algorithm, we tell the topology that this class is the default node model. We also set the duration of a round to 500ms (for the sake of visualization) via `setTimeUnit()`.

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

public class Main{
    public static void main(String[] args){
        Topology tp = new Topology();
        tp.setDefaultNodeModel(BroadcastNode.class);
        tp.setTimeUnit(500);
        new JViewer(tp);
        tp.start();
    }
}
```

When the program starts, you should see an empty window in which you can add nodes and play with them.

As you can see, the nodes are eventually colored in red. This is due to the execution of `onStart()` on each node after it is added to the topology. You can select the source node by Middle-Clicking (or Ctrl+Clicking) on it, causing `onSelection()` to be executed on that node. You should now see the propagation occurring through the network.

Test 1 : Implement and test the above objet *BroadcastNode*.

Test 2 : Modify your program so that each node sends its ID and output a trace of message exchanges.

Basic movement and timing

This example illustrates how to write a node algorithm that:

- Uses time as a source of event (by overriding `onClock()`)
- Makes the underlying node move in each round

Download the attached file *MovingNode.java*.

- Upon initialization (`onStart()` method), the node chooses a direction at random
- In every round (`onClock()` method), the node moves in this direction by one unit

Another option for moving a node is to call `setLocation()` with given coordinates (works in 2D or 3D). Depending on the cases, one might prefer one or the other. Note that `setDirection()` will cause the icon of the node to rotate.

Movements can be specified in many ways. In this example, we set an initial direction; then the node moves by one unit in this direction in each round.

- The `move()` method argument specifies the distance to be moved (equivalent to speed). It also exists without argument.
- The `wrapLocation()` method re-locates the node in a toroidal fashion when it gets out of bounds.

Message Passing *vs* Graph Models

When designing a node algorithm, an important question is what communication primitives a node can use? For example, does the node know its neighbors? can a node detect new incident links? can a node read and/or write into the memory of its neighbors? should it rather exchange messages for all purposes?

While a large number of models exist, it is common to distinguish between two families of paradigms:

- *Graph-based* algorithms

- *Message passing* algorithms

This tutorial gives an example for each paradigm, solving the same initial problem, called the *Loner* problem, where a node turns *red* when it has at least one neighbor, *green* otherwise (the node is a *loner*).

Graph-based Model

At the graph level, a node is directly aware of its incident links and adjacent neighbors. Technically, we allow the node to use the three following methods that allows, which one should consider as forbidden in message passing algorithms.

1. `onLinkAdded()`: override this method to specify what a node does when an incident link is added
2. `onLinkRemoved()`: override this method to specify what a node does when an incident link is removed
3. `getNeighbors()`: retreives the list of nodes with whom the current node shares a link

An implementation of node in the graph-based model is given in *LonerGraphBased.java*.

Message passing algorithm

In the message passing version, the nodes are not immediately informed when a neighbors arrives or leaves. Instead, they rely on sending messages repeatedly to detect the presence of other nodes. A possible implementation of this principle is provided in *LonerMessageBased.java*.

In both cases, think of telling the topology which of these classes is your default node model. You may want to slow down the execution by calling `setTimeUnit()` on your `Topology` object (in the main method).

Mixing Both Models

We presented above two possible ways of solving the same problem, depending on whether we consider a graph-based model or a message passing model. In many cases, the actual model lies between these two extremes. For instance, it is common to assume that the nodes can identify their neighbors (or communication channels towards them). In this case, looping over `getNeighbors()` to send each neighbor a particular message seems reasonable. Likewise, manipulating directly the node object of a neighbor is *cheating* in general, but the particular case of calling `getID()` on a neighbor when we assume that its identifier is known is reasonable too.

In summary, the polyvalence of JBotSim makes it extremely permissive. An algorithm designer can cheat in about all imaginable ways. We do not attempt to prevent this; rather, we consider that it is your responsibility to identify which primitives you can allow yourself, depending on the considered model.

Test 3 : Write a program that test the Loner node objets for each model.

Test 4 : Write a program that mix the Loner problem with moving nodes.

Test 5 : Write a program that mix the Broadcast paradigm with moving nodes.

Test 6 : Write a program that mix both models, i.e., each node sends messages to neighbors (`getNeighbors()`) only.

Centralized algorithm

Contrary to node algorithms (distributed algorithms), you do not need to inherit from a particular class in order to write a centralized algorithm. Still, you may want to react to some of JBotSim's events. Typical purposes for centralized algorithms are to code adversaries (also called *deamons*), which may perform global operations in between rounds (e.g. add or remove links or nodes), while the distributed algorithms on the nodes keep executing. Another typical use of centralized algorithms is to implement classical "precomputed" global and fixed data strauctures graph algorithms (*e.g.* a BFS or a DFS), although in this case no particular event is required, except perhaps for updating the computation when the graph changes.

Listening to the events (from outside)

As you know, node algorithms react to various events by means of overriding methods from the `Node` class. These methods are not available in other Java classes. However, you can listen to these events from within an arbitrary class, via the following interfaces:

- `TopologyListener`: be notified when new nodes are added or removed, through overridding `onNodeAdded()` and `onNodeRemoved()`.
- `ConnectivityListener`: be notified when new links are added or removed, through overridding `onLinkAdded()` and `onLinkRemoved()`. These methods are called differently from the ones in a node, because a node is only notified for incident links, whereas central listeners are notified for all links.
- `ClockListener`: be notified in each round, through overridding `onClock()`. By default, this method is called *after* being called on the inner components of the system (including nodes). This behavior can be redefined by replacing the inner scheduler of JBotSim, see Scheduling for more (advanced users only).
- `MessageListener`: be notified whenever a message arrives at a node, through overridding `onMessage()`. This interface is particularly useful for complexity analysis (or to keep logs of an execution).
- `StartListener`: be notified when the topology starts (or restarts). This is usefull, for instance, for (re)initializing global aspects of the topology.
- `MovementListener`: be notified when a node moves, through overridding `onMovement()`.
- and some others interfaces...

Concretely, what you have to do for `ClockListener` (say), is to:

1. add `implements ClockListener` in the declaration of your class
2. add the corresponding `onClock()` method (filled as desired)
3. register your class through calling `addClockListener` on the topology object

Another option is to define an anonymous listener from anywhere as shown in *CentralizedAlgo.java*.

Test 7 : Mix the Loner Node program and the Central Algorithm so that the current number of links is outputed each time a link is added or removed.

- CentralizedAlgo.java
- HelloWorld.java
- jbotsim-standalone-1.2.0.jar
- LonerGraphBased.java
- LonerMessageBased.java
- MovingNode.java
- TestCentralizedAlgo.java

Télécharger le dossier

◄ Lecture Slides

Aller à...

TME1 -- DTN ►