ANET TME 01 - Delay Tolerant Networks

Carlo Segat

student number 21115378

Carlo.Segat@etu.sorbonne-universite.fr

Question 1

Describe the random waypoint mobility model

When the nodes are initialised a random point within the bounds of the canvas is calculated.

Just before the clock pulse, the node direction is set to be the chosen random point and the node advances one step towards the set direction. When the node reaches the destination a new random point is chosen (more precisely, when the delta between the node position and the target destination is less than the step size, the node is moved at the target destination and a new random target destination is computed).

We should note that the acceleration is fixed (*step = 1*) but we may want to use a random value for the step variable as well to achieve a varying acceleration.

Question 2

Implement sending only to sink node

We just need to check if the node we met is the sink:

```
private boolean doWeTransmitToNode(Node node) {
    return node.getProperty("sink") != null;
}
```

If the node we have encountered is the sink then we transmit.

Question 3

Improving the simplest solution

An improvement is to send the message to a node we meet that is directed to the sink.

Assumption: the nodes are aware of the sink position. For this assumption, I have added a method to query a node destination.

In practice we only send when we meet the sink or a node that is directed to the sink:

```
private boolean doWeTransmitToNode(Node node) {
    if(isOtherSink(node)){
        return true;
    }
    if(doIMeetSink()){
        return false;
    }
    if(doesOtherMeetSink((WayPointNode) node)){
        return true;
    }
    return false;
}

private boolean doesOtherMeetSink(WayPointNode node) {
    return node.getDestination().distance( x: 300.0,  y: 300.0) < 40;
}

private boolean doIMeetSink() {
    return destination.distance( x: 300.0,  y: 300.0) < 40;
}

private boolean isOtherSink(Node node) {
    return node.getProperty("sink") != null;
}

public Point getDestination(){
    return destination;
}
```

Results with the time taken by some runs of this strategy:

Time units
6942
9479
8453
16101

Question 4

Use *distanceToSink* to improve the data aggregation time

The AdvisedWayPointNode node computes a number of random destinations until it finds one that will allow it to transmit to the sink, i.e. the last computed random destination will be in the range of the sink.

With all the random destinations computed in advance, we can establish the distance that the node must travel before being able to transmit to the sink.

When we meet a node we compare our *distanceToSink* with their's *distanceToSink*: the node with the shortest *distanceToSink* will receive the data from the other node and will be responsible for delivering all messages to the sink.

```
private boolean doWeTransmitToNode(Node node) {
    if(node.getProperty("sink") != null){
        return true;
    }
    return (double) node.getProperty("distanceToSink") < this.distanceToSink;
}
```

Results with the time taken by some runs of this strategy:

Time units:
4329
2484
8410
8042

Question 5

How can you improve again your algorithm

One problem with the distanceToSink is that often we are left with a handful of nodes that have aggregated some messages and are now waiting to reach the sink.
We could reduce the convergence time by allowing a green node to transmit to a node that has already sent its data (a red node) if that other node is guaranteed to reach the sink sooner.

The downside of this approach is more energy consumption as there are more transmissions.

To implement this we need to modify the *onClock* method:

```
public void onClock() {
    //If we don't have data, we cannot do anything
    if (!(boolean) this.getProperty("data")) {
        return;
    }

    java.util.List<Node> neigList = getNeighbors();
    for (Node node : neigList) {
        if ((boolean) node.getProperty("data")) {
            if (doWeTransmitToNode(node)) {
                setProperty("data", false);
                setColor(Color.red);
                nbTransmission++;
                break;
            }
        } else {
            if (doWeTransmitToNode(node)) {
                setProperty("data", false);
                setColor(Color.red);
                //nbTransmission++;
                node.setProperty("data", true);
                node.setColor(Color.green);
                break;
            }
        }
    }
```

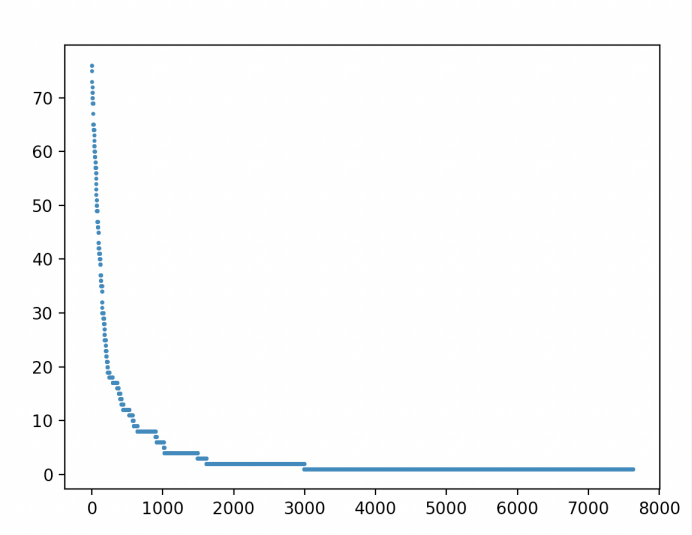
Results with the time taken by some runs of this strategy:

Time units
2210
3650
3343
1419
2978
2391

Question 6

record the evolution of the number of nodes that have transmitted

For the solution given in question 4, we can see below a plot of the number of nodes that still didn't transmit on the y, time on the x:

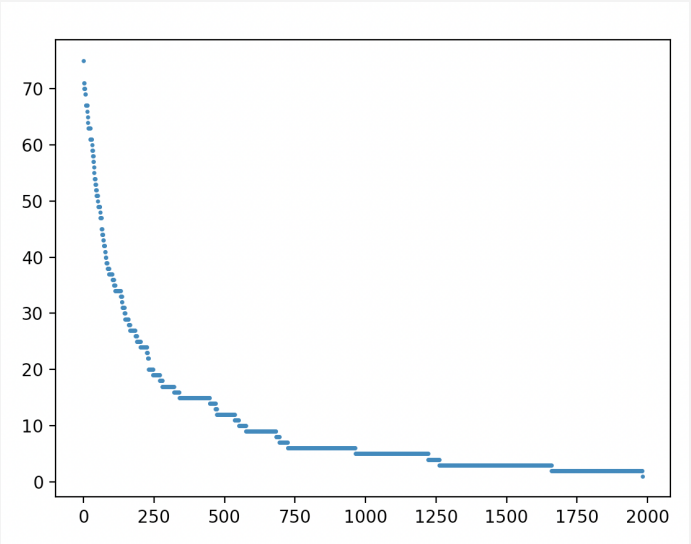


As time goes by more and more nodes have either transmitted to the sink or aggregated with a neighbour, this is perfectly normal.

More than 90% of the nodes transmitted in the first 1000 time units. This is normal as well as there are more chances to aggregate with the neighbours at the beginning.

Roughly after the 1000 time units timestamp, it takes longer for the remaining nodes to transmit as there are fewer available neighbours to aggregate with; those lingering nodes have to wait for the lucky random destination that will bring them to the sink.

On the other hand, we can observe a better behaviour for question 6 where we allowed aggregation with nodes that already transmitted:



Here we have overall fewer time units for all the nodes to transmit.
Here the last nodes take significantly less to transmit. For example, the last 3 nodes took roughly 400, 320 and 5 time units to transmit to the sink. In question 5 the last 3 nodes took roughly: 100, 1300 and 3500 time units respectively.