Peer to Peer Systems and Blockchains

Academic Year 2019/2020

Final project - Pandemic Mobile Flu: Virus Diffusion in Mobile Environments

Jacopo Massa - 543870

June 2020

Contents

Chapter 1

Introduction

The goal of this project is to implement a model describing the diffusion of a virus between smartphones which communicate through Bluetooth connections, only if the involved devices have the same operating system (OS). To follow the rules of the Bluetooth protocol, connections between nodes can happen within a short range, so couple of nodes must stay close enough so that a connection can take place. Connections simulate the infections of the model (better explained in the next section) and the probability of being infected is one of the parameter of the model. As in real cases, the infection can be eradicated by providing a cure for the nodes, in line with another model parameter that is the probability of receiving a patch.

1.1 Simulator and Plot Library

1.1.1 The ONE

The environment used to make the simulation has been built by using an already existing and very powerful simulator, which is `The Opportunistic Network Environment` (The ONE). As stated in its main guide, The ONE is a simulation environment capable of

- generating node movement using different movement models
- creating connections and routing messages among nodes

I've exploited these two characteristics of the simulator in order to implement the requested scenario. Another peculiarity of this simulator is the settings file, which is simply a list of *key,value* pairs divided hierarchically, where each level of the hierarchy is limited by a dot (see figure 1.1). In most of the snippets in the rest of the report, some `Settings` objects can be found: they're useful to access the settings file and pick the values of the several parameters needed for the simulations. Here a part of the `default_settings` file used by The ONE:

```
#
# Default settings for the simulation
#

## Scenario settings
Scenario.name = Pandemic Mobile Flu
Scenario.simulateConnections = true
Scenario.updateInterval = 0.1
# 43200s == 12h
Scenario.endTime = 43200

## Operating Systems settings
# Probabilities for each defined OS in enum 'OperatingSystem'
OperatingSystem.ANDROID = 0.5
OperatingSystem.IOS = 0.3
OperatingSystem.WINDOWS_PHONE = 0.2
```

Figure 1.1: Snapshot of `default_settings` file.

1.1.2 NetworkX

Data obtained by several simulations with The ONE, were subsequently processed using this very efficient Python package for the creation, manipulation, and study of the structure and functions of complex networks. The network at the base of the analysis was obtained by linking the nodes that were infected during the simulation.

Chapter 2

Models and Implementation Choices

In this chapter I'll explain the movement model and the compartmental model used in the project, and how I've implemented them using the tools listed in the section 1.1.

2.1 Movement Model

The requested model is based on the definition of several high density zones (or hotspots) where people, so the nodes of the simulation, move to. The ONE provides a similar approach among the different movement models, which is the *Random Waypoint* one. But this model doesn't take into account a larger number of hotspots, so I've extended the provided `RandomWaypoint` model, building the `RandomHotspot` one. The main difference is that when the model is instantiated, it generates randomly a given number of hotspots, obviously bounded in the limits of the simulation area (see figure 2.1).

Nodes of the network will always move among these hotspots, following a state machine made by three states:

1. `HALTING` a node can stop wherever it is, for a bunch of time defined in the settings (*waitTime*);
2. `TRAVELLING` a node is travelling to the next hotspot;
3. `EXPLORING` a node moves close to the last hotspot it has visited.

Each node can be in one of these three states following some probabilites defined in the settings as other model parameters (i.e. `HTE.HALTING` = 0.4). The states are collected into an enum (`HTEStatus`), and each node has an attribute typed with this enum, so I can manage and keep track of the status changes in the whole simulator.

```java
private void generateHotspots() {
    Settings settings = new Settings(HS_NS);
    hotspots = new ArrayList<>();
    try {
        int numberOfHotspots = settings.getInt( name: "nrofHS");
        if (numberOfHotspots < 0)
            throw new SettingsError( cause: "");

        for(int i=0; i<numberOfHotspots; i++)
            hotspots.add(randomCoord());

    } catch (SettingsError se) {
        System.err.println("Can't find '" + HS_NS + "nrofHS', or it has a value < 1");
        System.err.println("Caught at " + se.getStackTrace()[0]);
        System.exit( status: -1);
    }
}
```

Figure 2.1: Functions called in the `RandomHotpost` constructor to generate randomly some hotspots.

The `Travelling` status was implicitly managed by the simulator itself after having defined the `RandomHotspot` model as movement model for the simulations. To implement the behaviour of the nodes in the other two states, I've modified a part of the `DTNHost` class, which represents a node of the simulation. In particular, the *move* method defines how and where a node has to go to, so I've updated the *move* function, adding two "if branches" providing the `Halting` and `Exploring` status behaviour (see figure 2.2).
The `Halting` one exploits the *waitTime* already defined by The ONE, used to set the bunch of time that passes between two steps of the simulated movement. The `Exploring` status generate a coordinate in the simulation area, close to the location of the node (which is always an hotspot when this code is invoked), and then builds a path to reach this new location.

2.2 Compartmental model

The compartmental model defines how the simulated virus grows and evolves among the nodes. The requested one is the SIR (Susceptible, Infectious, Removed) model, which is one of the simplest epidemiologic models, based on three states:

1. **S**usceptible: when a susceptible individual comes into contact with an infectious one, in contracts the virus and moves to the infectious state ($S \rightarrow I$)

2. **I**nfectious: individuals who have been infected and are capable of infecting

```
double randomNumber = this.random.nextDouble();

if (this.destination == null) {
    // EXPLORING
    if(!isExploring()) {
        if (randomNumber > hteProbabilities.get(HTEStatus.HALTING) +
                hteProbabilities.get(HTEStatus.TRAVELLING)) {
            Coord closePoint = exploreCoord();
            this.hteStatus = HTEStatus.EXPLORING;
            this.destination = closePoint;
            this.path = new Path( speed: 1);
            this.path.addWaypoint(closePoint);
        }
    }
    if (!setNextWaypoint()) {
        return;
    }
}

// HALTING
if(!isHalted()) {
    if (randomNumber < hteProbabilities.get(HTEStatus.HALTING)) {
        this.hteStatus = HTEStatus.HALTING;
        return;
    }
}
```

Figure 2.2: Implementation of `Halting` and `Exploring` states in the *move* function.

susceptible nodes.

3. **R**emoved: individuals who died or have recovered from the virus. They can no longer be infected. ($I \rightarrow R$)

In addition to this, as we are in the context of mobile networks, each node has an operating system and can infect only other nodes with the same *O.S.*. The distribution of the devices for each operating system is regulated by other probabilities defined in the settings.

Also this model is represented by an enum (`SirStatus`) which typifies an attribute of the hosts. The state transitions exploit one of the main functionalities of The ONE, which is estabilishing connections when two nodes are

one in the range of the other. In detalis, each `DTNHost` can have multiple `NetworkInterface` which take care of the node connectivity, notifying whenever there is a new connection or when an old one is going to end. The ONE offers a set of already pre-configured interfaces, including the Bluetooth one that was required. I've used it also to reflect a more "realistic" environment, where the range to spread a virus is pretty small. For this project, each host has only one Bluetooth interface and every time a new connection is detected, if the host is infected and falls in the probability of infecting another susceptible node, it tries to infect it (Figure 2.3).

```
case CON_UP:
    if(this.host.isInfected() && otherHost.isSusceptible()
            && this.host.getOS().equals(otherHost.getOS())) {
        double randomNumber = rng.nextDouble();
        if(randomNumber < SimScenario.sirProbabilities.get(SirStatus.INFECTIOUS)) {
            otherHost.infectNode();
            try {
                SimScenario.bw.write( str: this.host.toString() + ","
                        + otherHost.toString() + ","
                        + SimClock.getIntTime() + "\n");
            }catch(IOException e) {
                e.printStackTrace();
            }
        }
    }
```

Figure 2.3: Implementation of *infections* in the `NetworkInterface` listener manager.

The model requires also another transition, which is the one that occurs when a node dies or it has been recovered from the virus. To simulate this transition I've updated the `World` class of The ONE.
In this class we have an abstraction of the environment in which the simulation is running, and also references to the whole set of hosts. In particular, in the *update* method there is a periodic update of the status of each node. Here I've added also the probability to be patched, but to have a more realistic behaviour I couldn't try to patch nodes at each simulation tick, otherwise, even with a very low probability of recovery, after a few ticks all the nodes would have been patched.
So I've defined that a *simulation step* is equal to 1000 simulation ticks, and at each step an attempt is made to remove a node, with equal probability on all nodes (Figure 2.4).

```
// try to recover randomly some nodes
if(SimClock.getIntTime() >= this.nextPatchTime) {
    for (DTNHost h: hosts) {
        double randomNumber = rng.nextDouble();
        double sProb = SimScenario.sirProbabilities.get(SirStatus.SUSCEPTIBLE);
        double iProb = SimScenario.sirProbabilities.get(SirStatus.INFECTIOUS);
        if(randomNumber > sProb + iProb && !h.isRecovered()) {
            h.patchNode();
            try {
                SimScenario.bw.write( str: h.toString() + "," + SimClock.getIntTime() + "\n");
            }catch(IOException e) {
                e.printStackTrace();
            }
        }
    }
```

Figure 2.4: Implementation of *removal* of a node in the `World.update` method.

2.3 Other implementation choices

2.3.1 GUI changes

The ONE simulator can be executed also with a GUI, where the simulation can be seen step by step. To better understand what was happening during the experiments I've changed a bit the default GUI, associating different colors to each SIR status (see section 2.2). In the specific, the changed class is `gui.PlayField.NodeGraphic` which takes care of the visualization of a `DTNHost`. The following code shows the colors for each status:

```
// draw the "range" circle
if(node.isInfected()) {
    rangeColor = Color.RED;
}
else if(node.isRecovered())
    rangeColor = Color.GREEN;
else {
    rangeColor = Color.BLUE;
}
```

Figure 2.5: GUI changes for a better visualization.

2.3.2 Logging

Data obtained by each simulation have been processed due to analyze them. But first of all, these data have been logged into a file, populated with two types of informations, indicating also when, during the simulation, the event occurred:

- An infectious node infects a susceptible one.
  `< node, otherNode, time >`
- A node is removed (dies or patched).
  `< node, time >`

Collected data are then processed and plotted in a series of graphs that are well explained in the chapter 3. The Python script that plots the data uses the

NetworkX package, explained in section 1.1.2. The format used to log data was useful after the parsing, becuase I had already the set of edges fo the graph I wanted to build, and the information about the patched nodes were useful to indicate when the removal occurred.

```
def analyze_simulation(simulation_filename):

    # parse log file
    res = parse(simulation_filename)

    # number of nodes, list of infections, list of patched
    n, infections, patched = res

    # build and plot the network of infections
    g = plot_graph(n, infections, patched)

    # plot degree distribution for the graph
    plot_degree_distribution(g)

    # process parsed data
    sir_data = get_sir_data_per_cycle(n, infections, patched)

    # plot processed data
    plot_sir_per_cycle(sir_data)

    # plot percentage of infected per cycle
    plot_percentage_per_cycle(n, sir_data)

    # show all the plotted graphs
    plt.show()
```

Figure 2.6: Main function of Python NetworkX script.

Chapter 3

Experiments And Plots

This chapter describes the whole set of experiments done by using The ONE, and some comments on the obtained results plotted with NetworkX.
I've made three main experiments:

- The first one has a few number of nodes but is useful to show in the best possible way the plotted results that I couldn't have shown on simulations involving more nodes.
- The second one increases a bit the number of nodes and changes the probabilities of node infection and removal, to show how these parameters are fundamental for the evolution of the simulation.
- The latter works on a larger number of nodes, and it was made to have a more realistic example of how the spread of a virus works.

Each simulation consists in different runs of the simulator with the defined parameters. For each run, which lasts two simulated days (172800 seconds in the settings), I've changed the seed used by The ONE for random choices (hotspot position, next hotspost ...) and a node for each defined operating system was infected.

3.1 Experiment 1

$\#nodes = 200$ $\#hotspots = 6$

$P(infection) = 0.8$ $P(removal) = 0.05$

The first thing to show is the structure built by the infections (Figure 3.1): we can notice how in the middle of the net there are a few nodes which are the first ones that were infected. The other hosts on the net boundaries may have been infected later or they're completely detached from the rest of the network. In the former case, it's important to notiche how, from a bunch of nodes a large infection can still begin.
In the latter case, nodes were patched (removed) in the first steps of the simulations or anyway before getting infected, so they can't have no more "infection contacts" with the others.
Edges are also labeled with the cycle (multiple of 1000, as I specified at 2.2) in which a node has been infected.

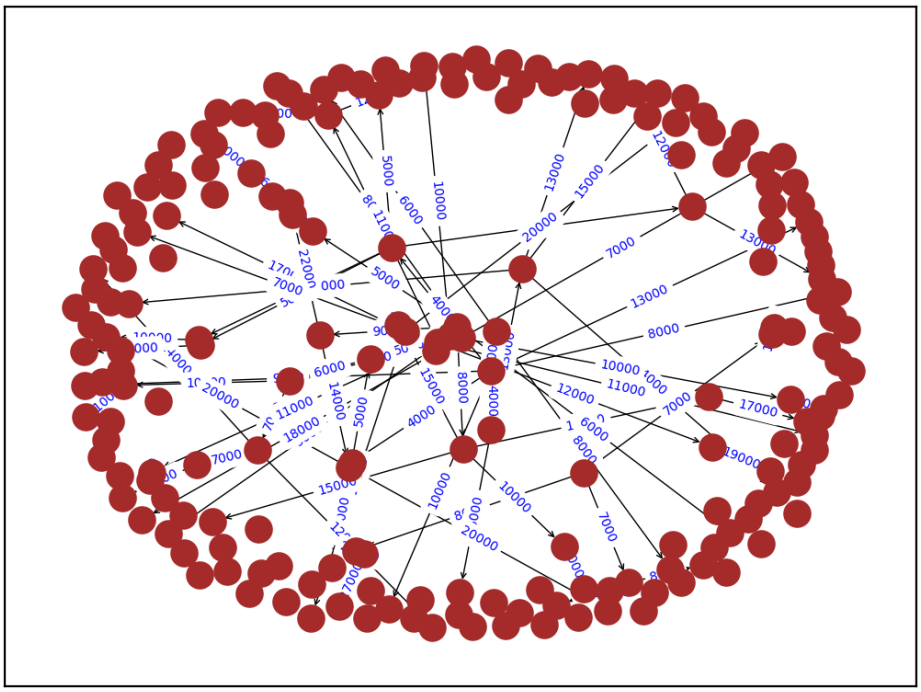

Figure 3.1: Directed graph of infections.

What I said can be found also in the nodes' degree histogram (Figure 3.2) which shows how a lot of nodes were removed without having contacts ($degree = 0$), because of the removal probability (5%) that is applied to each node at every step.

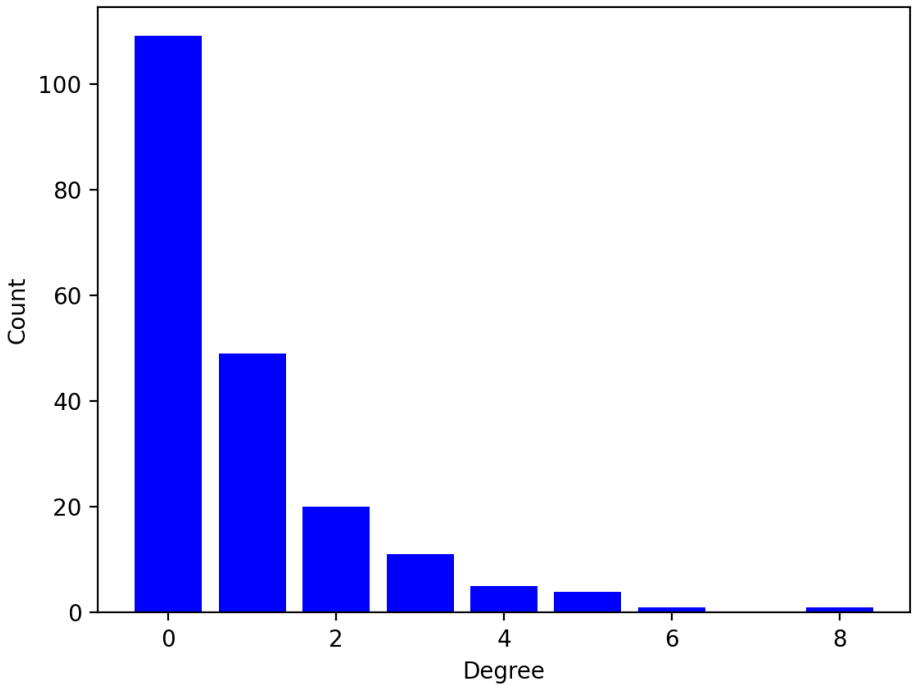

Figure 3.2: Node degree histogram.

Maybe the small number of nodes and the too high probability of removal for a node brought to data inconsistent with the initial probabilities. In fact, for this experiment I've obtained a maximum peak of 15% of infected nodes, against the 80% defined as probability. Maybe another factor that influenced this trend was the size of the map, too large for the small number of nodes.

3.2 Experiment 2

$\#nodes = 300$ $\#hotspots = 6$

$P(infection) = 0.4$ $P(removal) = 0.005$

This experiment was more successful than the first one. I've obtained the same structure for the directed graph of infections, but the results were consisent with the initial probabilities. As shown in Figure 3.3, more or less the 40% of infections has been reached in the first 50 cycles, which means in the first 13 hours of the simulation.

The next figure 3.4, shows how the very low removal probability leaves the number of infected nodes almost unchanged until the end of the simulation.

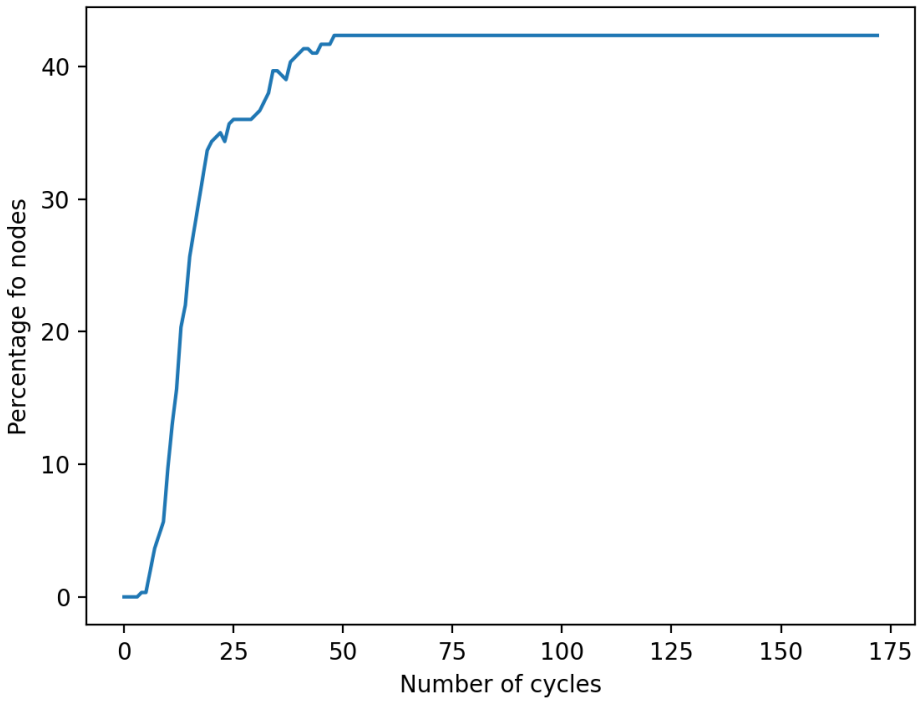

Figure 3.3: Percentage of infected nodes after a certain number of cycles/steps.

3.3 Experiment 3

This has been the larger experiment, in terms of number of nodes and size of the map.

$\#nodes = 1500$ $\#hotspots = 8$

$P(infection) = 0.5$ $P(removal) = 0.04$

The structure and the flow of the different graphs were almost the same of the previous experiments. I used this scenario to better characterize and understand the structural parameters of the obtained graph (I can't show it becuase of the resolution of the image and the huge number of nodes). I decided to calculate, using NetworkX, the following characteristics:

- **Clustering coefficient**: its value in all the experiments was **zero**. This is exactly the expected result, because of the "limitation" on the infections: only two nodes with the same operating system can infect each other. This means that some clusters will be implicitly created, because hosts with different o.s. will never meet. So, since the clustering coefficient is related to the connections among the neighbours of a node, most of the nodes' coefficents were null.

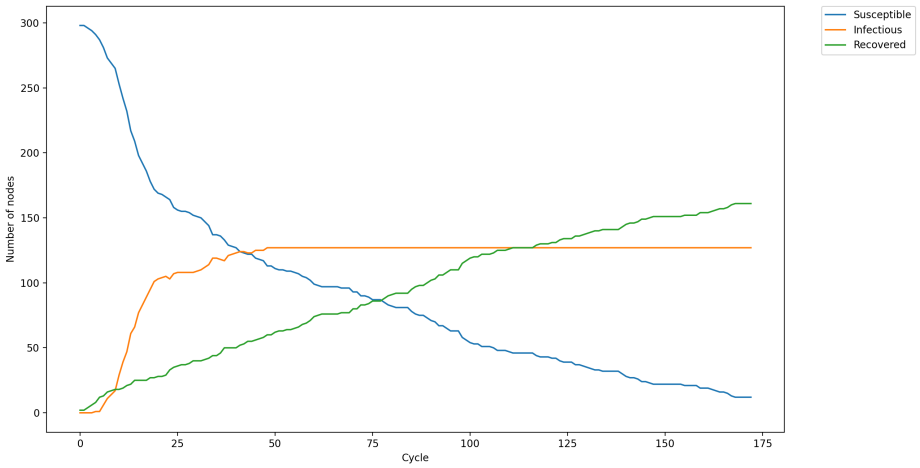

Figure 3.4: SIR data step by step.

- **Diamater**: for all the experiments it was **infinite**, and the reason is the same one of the clustering coefficient. The diameter is defined as the maximum distance between pair of vertices, so, so since the digraph (Figure 3.1) is not strongly connected, the diameter can be only infinite.

- **Density**: the density of a graph is the ratio of the number of edges and the number of possible edges. For the last experiment (the more significant) I obtained a value equal to **0,0004** that reflects a lot the SIR model, where an host that has been already infected won't receive other infections (so a lot of edges will not be there), and also removed nodes won't have interactions with other hosts.