UNIVERSITÀ DI PISA

Master Degree in Computer Science, ICT Solutions Architect curriculum

Peer to Peer and Blockchains

Final project: Pandemic Flu

Teacher:

Prof. Laura Ricci

Student:

Giulio Purgatorio

516292

June session

2019/2020

Contents

Chapter 1

Introduction

This is the relation of the Final Project for the Peer To Peer and Blockchains' course, Pandemic Flu.

The goal of this project is to implement a model describing the diffusion of a virus between smartphones which communicate through Bluetooth connections. The devices will represent the owner's movements and they can be infected only if they get in range of an infected device that shares the same operating system.
During these simulations, we will assume that every host will connect with any other node in its own range. In practice, this may happen with the *JUST_WORKS* access policy of the Bluetooth Low Energy, which connects two devices without requiring extra steps and preserving energy efficiency.

Hosts will simply walk around the simulation area, reaching predefined hotspots in a random order, creating high-density areas that will help spread the virus. Once reached a hotspot, which can be seen as a real-life café or any other type of *Point of Interest*, they may either just wait there (representing the act of sitting on the given café, etc.), explore around the location or just travel to another waypoint.
This will cause many nodes to eventually contract the virus, potentially all of them: for this reason, any node may from time to time install a patch that will prevent him from being infected or to infect others.

Now, I'll give a brief explanation for each chapter of this document.
In **Chapter 2**, I'll present the tools that I've used for this project.
Inside **Chapter 3**, there will be the implementation's history, so that it's possible to see what changes have been made and where to find them.
Chapter 4 shows the obtained results of the simulations, depending on various settings.
Finally, in **Chapter 5**, there will be conclusions of this document.

1.1 Starting the project

The only setup needed is to open the project and load the libraries that can be found in the *lib/* folder.
The main class can be found in *core/DTNSim.java* and simply running that will start the simulation. A default (and very basic) scenario is set just to show the correctness of the simulator, like node's exploring, hotspots, etc. To ease the checks, the default settings have very low values that help visualize the hosts' behavior, i.e. a low number of nodes.

Chapter 2

Tools used

To implement the environment scenario I've decided to use different already existing tools that I'll now present.

2.1 The ONE simulator

The ONE *(The Opportunistic Network Environment)* [1] is a simulator that is capable of generating node movement using different movement models. It can also produce a variety of reports from node movement to message passing and general statistics.

While extending the simulator, many choices have been made. First of all, each node in the simulation has its own identifier, and this way it represents unequivocally a person: this is an assumption that can be easily made just by imagining how smartphones or similar tools are frequent in real-life scenarios. This way, a simulation for the devices' movement implies a strict correlation with users' movements.

Just as a final note, this simulator allows many more things that weren't used, like *Message exchange* or *Mobility data import from real-world traces.*

2.2 NetworkX

NetworkX [2] is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

Chapter 3

Implementation

I'll now introduce the main changes I've made to the simulator in order to model the Pandemic Flu scenario. This chapter should be used to check my adjustments to the code in a fast way: to easily find *most of* the changes, I've put new methods at the end of each file, but methods that were changed are in their original spot.
All the specified parameters are under the **PandemicFlu** namespace in *default_settings.txt*, which will be introduced in Chapter 4.

3.1 Package *core*

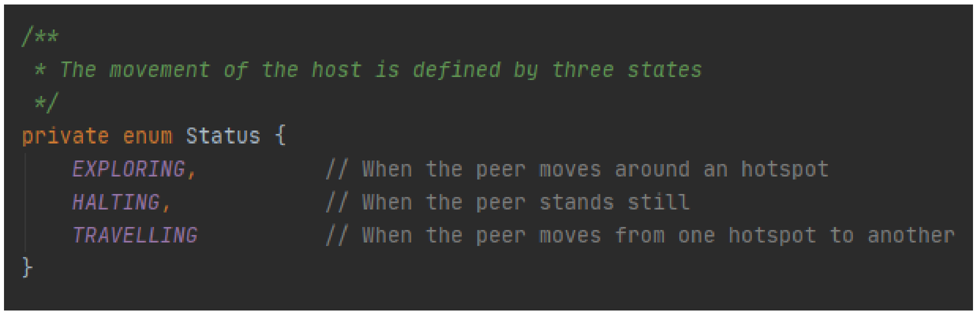
As the name suggests, this package contains core classes and interfaces of the simulator.

3.1.1 DTNHost

This class is the one that is used to represent the nodes in the simulation. The first change is the introduction of an operating system for each user, which is simulated by a simple String. Then, I've introduced the **Status enum**, used by the movement model to move around the map.
The three main states are labeled as the assignment requested and can be seen in Figure 3.1.

Going briefly over it:

- the **HALTING** state will cause the node to wait without doing anything for an amount



```java
/**
 * The movement of the host is defined by three states
 */
private enum Status {
    EXPLORING,          // When the peer moves around an hotspot
    HALTING,            // When the peer stands still
    TRAVELLING          // When the peer moves from one hotspot to another
}
```

Figure 3.1: The enum for the probabilistic automaton

of time specified through the initial settings *(parameter: waitTime)*.

- the **EXPLORING** state will cause the selection of a random number of coordinates around the hotspot *(parameter: exploreNum)*. The range for these coordinates is another specifiable argument *(parameter: exploreRange)*.
- the **TRAVELLING** state is simply when the node is moving from one hotspot to another

Finally, I introduced two main *boolean* attributes that were used for the infection logic:

- **isInfected** which is True if the node has been infected and so can infect others.
- **isPatched** which is True if the user installed the patch that makes it impossible to be further infected or infect others.

By combining these two values, it's simple to represent the values for the SIR model, which defines:

- **Susceptible** for nodes that aren't infected yet, but can be infected by others.
- **Infectious** for nodes that are infected and can infect others.
- **Recovered** for nodes that became immune.

3.1.2 World

This class contains a reference to all the nodes *(DTNHosts)* and is responsible for their updates. That's why here lies the patch check, that is run every *installCheck* ticks of the simulation. The

patch probability is as requested another specifiable parameter, which is *patchProbability*, and works like the precedent timer.
Similarly, general periodic statistics are placed here too: they're managed by a different parameter, precisely *statCheck*.

3.1.3 NetworkInterface

This class takes care of connectivity among hosts. The interface that I've used for this project is the Bluetooth one, which has many real-life relevant features that can be manipulated, like *transmitRange* which is the range in meters for the Bluetooth coverage.
Since the Bluetooth connections are handled here, I've put the code that allows the infection between hosts in the *notifyConnectionListeners()* method. Practically, this means that every time two nodes are in range of each other, they'll try to infect one another. For this specific reason I've also implemented an *HashSet* of connected users: whenever a user is in the range of another one, it'll first try to add him to its own set. The *add()* method returns True if and only if the item was successfully inserted in the set, which means that the node will be a new connection. Otherwise, nothing happens, because the infection was already handled in the past. On the other hand, when two nodes move too far away from one another, they'll *remove()* that other one from the set in order to allow future infections in case of reconnections.
Without the set, all nodes would keep trying to infect each other at every tick of the simulation.

3.1.4 SimScenario

Here it's possible to find most of the settings used for a given simulation run. So I've put the operating system logic here, like parsing the given number and name of operating systems through the parameters *nrOfOS* and *operatingSystem#*. In particular, it's also possible to set up different distributions of these operating systems, through their respective parameter *percOS#*.
Note: the distribution doesn't assure in any way that all of the given operating systems will appear, due to the nature of statistics itself. If a given OS will miss from the simulation, a little print will be shown in the stdout as a warning.

Finally, the last part of the infection's setup is to actually infect one host for each OS that

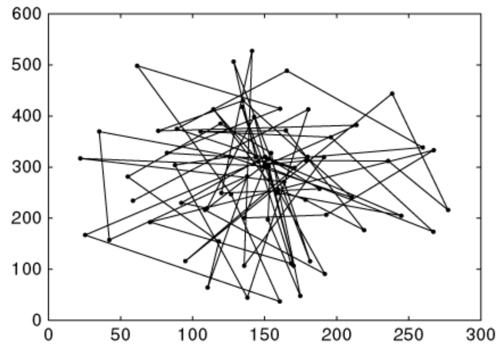


Figure 3.2: RandomWalk generated pattern

is present at runtime. This is done by shuffling randomly the list of peers and infecting the first that uses the corresponding OS.

3.2 Package *movement*

Movement models are born by the necessity to estimate where and when a user will be found in a given future moment.
This package contains the possible movement models that are used by a *DTNHost* to move around the simulation. There are many models already present in the downloadable version of the simulator, ranging from random models to more complex ones. Considering the requests for the project, like the presence of a rectangular map (so nothing that would limit movements like roads, etc.), the random movement models are the most coherent. These are divided into three main models: RandomWalk, RandomDirection, and RandomWaypoint. [3]

In the **RandomWalk** model we find the entity that is moving from its current position to another one, chosen independently from any current factor. Speed will also be randomly chosen and then the node will reach the new random destination, repeating its behavior indefinitely. In the case where the node would reach the edges of the map, it'd bounce off of it with a symmetric angle with respect to the delimitation, keeping its momentum. The randomness is guaranteed by the lack of information storing because this model aims to be stateless in all its features.
Figure 3.2 presents the general pattern created by this model.

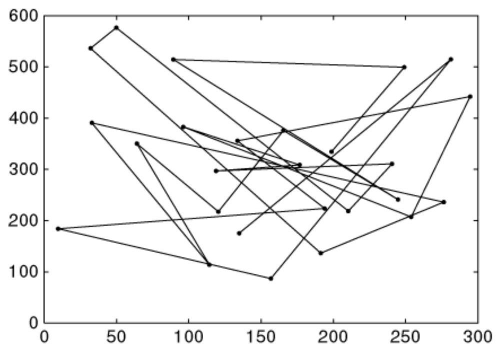


Figure 3.3: RandomWaypoint generated pattern

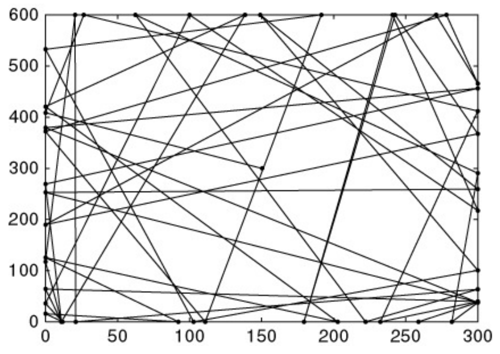


Figure 3.4: RandomDirection generated pattern

Differently from the previous one, in the **RandomWaypoint** model there are predefined destinations that will be uniformly randomly chosen. Once the path has been loaded, with a given speed that is limited by an upper and lower bound, the host will start moving towards the new location. Then, it'll repeat over and over this process. Figure 3.3 presents the general pattern created by this model.

Finally, the **RandomDirection** model was created because of a problem caused by the *RandomWaypoint* model. With that one, the hosts would all cluster in specific parts of the map: this phenomenon is called *"density wave"*. It has been proved that most of the time the overloaded part of the map is around the center of it because the nodes are more likely to pass through it than on borders. With *RandomDirection* instead, only direction and speed are chosen: the node will keep going straight forward until it'll find the map's boundaries. Figure 3.4 presents the general pattern created by this model.

3.2.1 PandemicFluRouting

This movement model is the one I've created appositely for the Pandemic Flu project taking into consideration the project's requests. It uses as a base the *RandomWaypoint* model because we had to define a set of hotspots on the map that were defining high-density zones. The first extension of this movement model is already in the precedent definition because the original RandomWaypoint model found in The ONE doesn't define a set of hotspots: it just gets a random valid point in the map area and moves towards it. This would prevent the "group behavior" of many hosts, that have to group in specific locations representing a *point of interest* (PoI), like a bar, a cinema, etc. For this reason, I've put in the Settings a new parameter, *nrOfPois*, which specifies the number of random hotspots to create.

3.3 Package *gui*

This package is GUI-related and will be shortly reviewed because it's more for user-friendliness than usual implementation.

3.3.1 playfield/NodeGraphic

To have a visual representation of the situations, nodes have different colors depending on their status, referring to the SIR model.

- When a node is **Susceptible**, it's represented as the default The ONE node. A blue rectangle representing the node and a blue circle around him representing its own Bluetooth range.
- When a node is **Infectious**, both its rectangle and circle color are red.
- Otherwise, when a node is **Recovered**, he's displayed as yellow and its range color is restored to the default one.

3.3.2 playfield/PlayField

The ONE simulator is usually used with preloaded real-world maps. In this project the map wasn't required, so the rectangular area had to be displayed: for this reason, a black rectangle delimiting the area of the simulation is now being drawn. Furthermore, for clearness, I've also decided to draw all the hotspots that were created: their range and background color are defined as usual in the Settings, respectively through the parameters *HotspotsRangeIndicator* and *HotspotsHexColor*. This view can be toggled by the Menu in the upper-left corner **PlayField options** and is by default set at *True*. The added menu item is viewable in the *gui/SimMenuBar.java* class.

3.3.3 DTNSimGui

The application may use very high dimension maps, depending on settings. For this reason, now the simulator will start in fullscreen mode.

3.3.4 EventLogPanel

Many checkboxes would make the simulator less user-friendly: for this reason, all message related checkboxes were removed, causing the Log messages to be lighter.

3.4 Package *Util*

A simple parser to use for the networkX plotting system was added, grouping some relevant informations like the number of infections per user, etc.

Chapter 4

Experimental results

4.1 Settings

Every settable setting is defined in textual files. The default one is called *default_settings.txt* and it will always be loaded. If one wanted to load custom file settings, they can be imported through the command-line executing the main class *core/DTNSim.java* and then the given file's name: settings that overlap on others that may be present in *default_settings* will be overwritten.

In the simulator's base version there are already many specifiable parameters that are useful for this project, like the map dimensions, the group of node's behaviors, etc. They all have a comment above them explaining what they do and most relevant ones are put on the top side of the textual files.

Speaking about introduced settings for this project, I've already given a quick overview of them in Chapter 3, but I'll quickly recap them here:

- **nrOfOS:** The number of operating systems that have to be parsed during the setup.
- **operatingSystem#:** The name of the #-th operating system. *Android, iOS, etc.*
- **percOS#:** The probability distribution for the #-th operating system. The sum of all these percentages must be exactly 1.
- **propagationProb:** The probability that an infected user can infect another one when

they connect.

- **patchProbability:** The probability that a user will install the patch.
- **nrOfPois:** The number of hotspots that must be generated for the PandemicFluRoutine movement model.
- **haltingProb:** The probability of standing still on a hotspot when arrived.
- **travellingProb:** The probability of moving to another hotspot.
- **exploringProb:** The probability of moving around the current hotspot. The sum of these 3 last probabilities must be exactly 1.
- **exploreNum:** The number of times the node will move around the hotspot will be chosen randomly between 1 and *exploreNum*.
- **exploreRange:** How far the exploring nodes will move from the current hotspot.
- **installCheck:** Every how many ticks in the simulation the hosts may try to install the patch.
- **statCheck:** Every how many ticks in the simulation some stats are retrieved, like SIR calculations, etc.
- **HotspotRangeIndicator:** The range to show for each hotspot.
- **HotspotHexColor:** The background color for each hotspot.

All precedent probabilities must be numbers between 0 and 1, sometimes inclusive, sometimes not, depending on the context.

4.2 Set of experiments

This section will be divided into subsections that represent the different simulation runs merged together with different seeds just to avoid statistical anomalies. All the relevant stats will be resumed in curly brackets at the beginning. A simulation will last up to 3 simulated days and

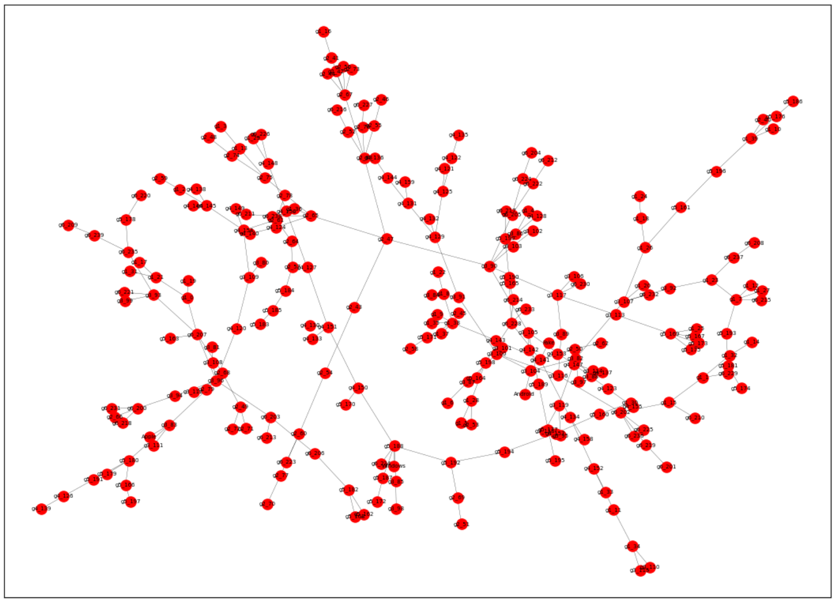


Figure 4.1: 233 nodes were infected in just 12h of simulation

will be interrupted when the number of infected has reached close to the 100%, because from that moment the only thing that will happen is that nodes will randomly patch at a given percentage.

Note: all nodes are using the PandemicFluRoutine movement model.

4.2.1 First simulation

{*240 hosts, mapX = 9000, mapY = 3400, propagationProb = 0.7, patchProbability = 0.05*}

For this experiment, a total of 240 hosts (6 groups of 40 each) were simulated in a map of dimensions close to $30km^2$. This means that this simulation is supposed to have very far away hotspots from one another, which should in theory lead to making it hard for the infected nodes to reach groups that aren't infected from the start.

For this reason, the probability of infection is set to a high rate, 70%, and results of the given infection can be seen in Figure 4.1.

This picture shows how, even with the precedent assumptions, in just 12 hours all the groups were infected, leaving just 8 nodes out of it. This picture doesn't show infected nodes that then patched becoming immune, but all the connections that went on while infected: in

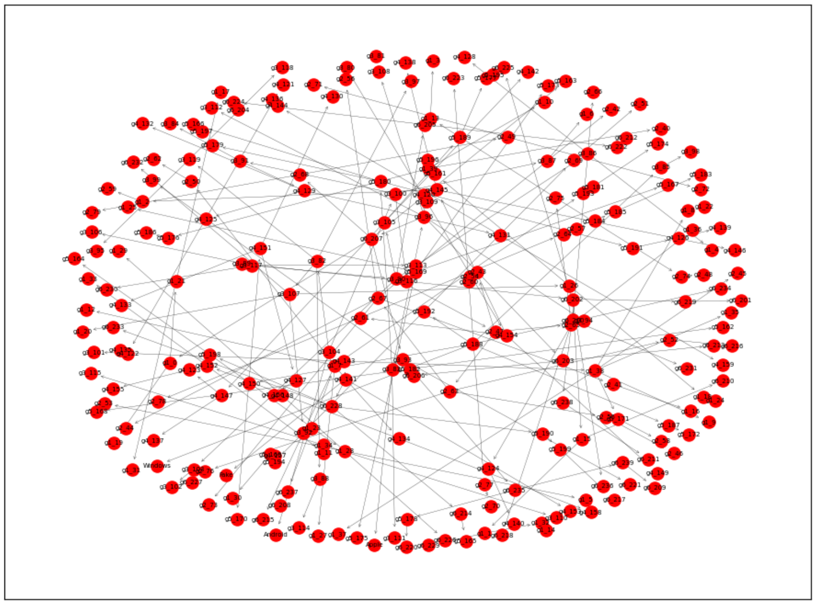


Figure 4.2: Directed Graph of infections

other words, people patching way too late after contracting the virus will still infect others of course, and this a crucial aspect to observe. On the other hand, when the simulation stopped, there were a total of 19 patched hosts.
So this simulation shows how a high infection probability will still exponentially explode, even if groups are meant to be far away from one another: just one case may start a whole new group of cases. This is visible in the directed graph presented in Figure 4.2. Centered nodes are the first infected ones: they were able to infect through these paths all the ones in the ring around the circle.

4.2.2 Second simulation

{*Same as previous one [4.2.1], but propagationProb = 0.1*}

I've tried lowering the patch probability to 10%, which is still enough to reach a similar result to the previous one but simply needs more time, which was nearly 18 hours as opposed to the 12 ones. This supports the observation made in 4.2.1, where nodes follow an exponential-like curve for infecting, requiring always less and less time to infect another group of nodes. This also happens because the virus is a point of no return unless the random patch occurs,

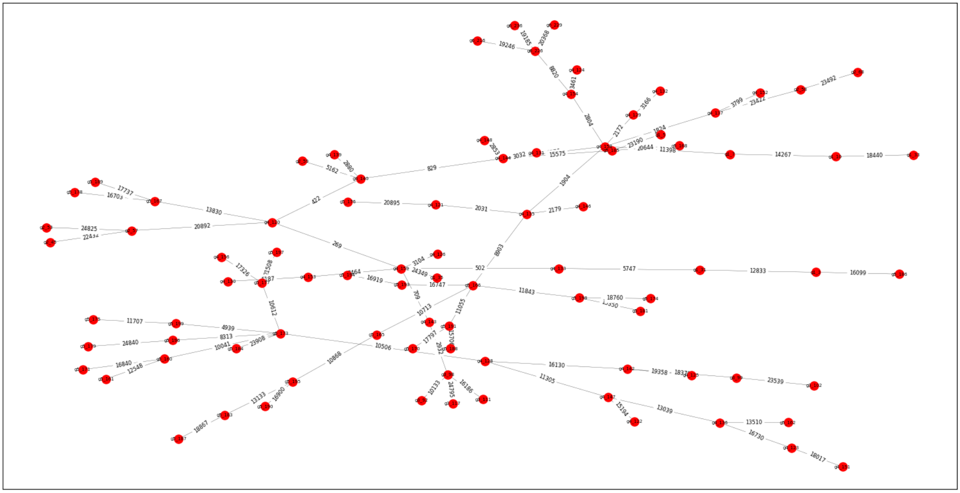


Figure 4.3: Infections with labels (12h of simulation)

so it suggests that the propagation probability isn't the main value to consider. In Figure 4.3 there's the infections with labels representing the time of infection: this won't be shown in the *Main Simulation* section [4.3] because of the huge number of nodes that would make it impossible to read anything considering the resolution.

4.2.3 Third simulation

{*Same as 4.2.1, but only 3 groups and 2 hotspots*}

The main aspect that leads to infections is nodes that find each other: so I aimed to check if hotspots were the principal factor in this type of simulations. By lowering the number of groups down to 2, the number of Points of Interests decreases linearly with them, causing fewer possible movements and so decreasing the probability of having cross-links with another group that may be infected. Out of all the simulations done, we can divide them into two main categories: the one that had crossed links and the one that had at least one group not crossing others' paths.

No cross links

In Figure 4.4 we can see how the hotspots' distribution doesn't allow cross links. So what happened is that the virus for the i-th operating system was limited to that group only, not

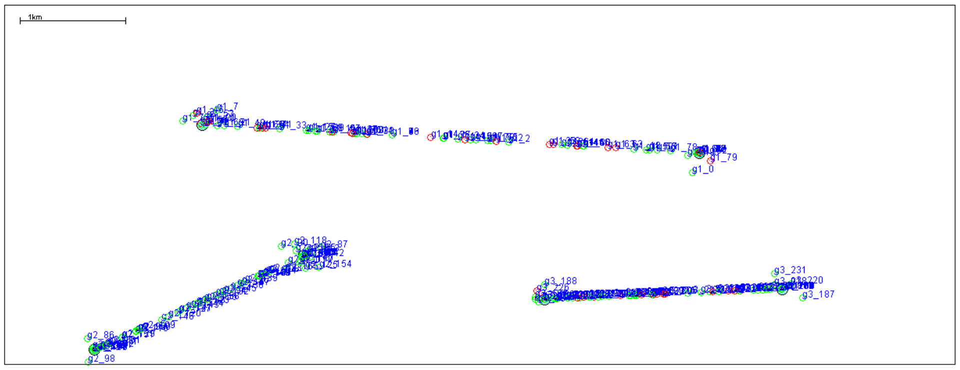


Figure 4.4: A safe group because it's out of the infection's range

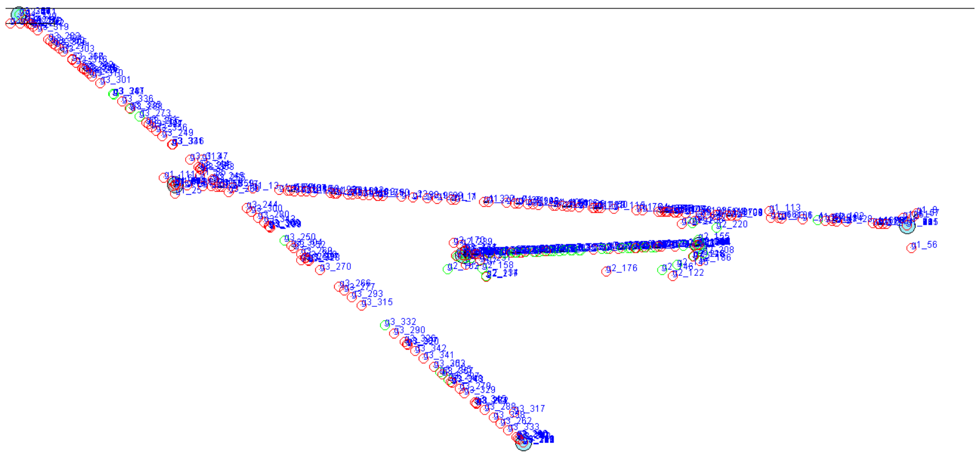


Figure 4.5: Snapshot of simulation with crossed links

being able to infect the other 2.

Cross links

Contrarily, with cross-links, we return to the previous results: a special case I wanted to show is the one in Figure 4.5. It's easily observable that *group1* and *group3* are crossing each other, but *group2* got infected because of the explorations around the hotspot further to the right. This way, all nodes reached again in less than a day a total infectious (or patched) state.

4.3 Main simulations

While the first three simulations were more of an introduction to gathering information on the hosts' behaviors, these are more complex ones, aiming to remove (most) biases that my settings may cause.

First of all, I've also put the other 3 random movement models, in their base form *(RandomWalk, RandomWaypoint, and RandomDirection)*. This way I'm sure that, even if no crossing links are present, everybody is still in danger because of these random walkers. I've then put the probability of infection to 1%, because of the long term simulation that had to last up to 6 days. Each group was formed by a random number between 5 and 10 hosts, similarly as what happens in a generic real-world group. There were a total of 200 groups, where 180 shared the PandemicFluRoutine movement model. Finally, many groups shared common points of interests, again because I was trying to relate with real-world scenarios: different groups may come in contact with others just because they share a common interest or visit the same place regularly, even without knowing each other.

The patch happened regularly every 10000 ticks (6 days = 518400 ticks) of simulations with a probability of 2% for each peer.

All the presented results are an average of 5 different runs with the same parameters but different seeds.

4.3.1 Results of experiments

Every simulation ended up with the same pattern, which is presented in Figure 4.6.

The simplest curve to explain is the green one, about patched nodes: they had a specific percentage to be patched from time to time, so their increase is linear as it should be.

Looking at the other two, we can see a very interesting correlation: around the whole first day of the simulation, infections would grow inversely proportional to susceptible hosts. This pattern is interrupted only by the lack of an infinite pool of users, where at some point the number of infected hosts will be larger than the ones they can actually infect. Considering this and the steady increase of patched peers, the infection is limited to an upper bound of around half the nodes. Without the patch, the orange curve would just be the inverse of the blue one,

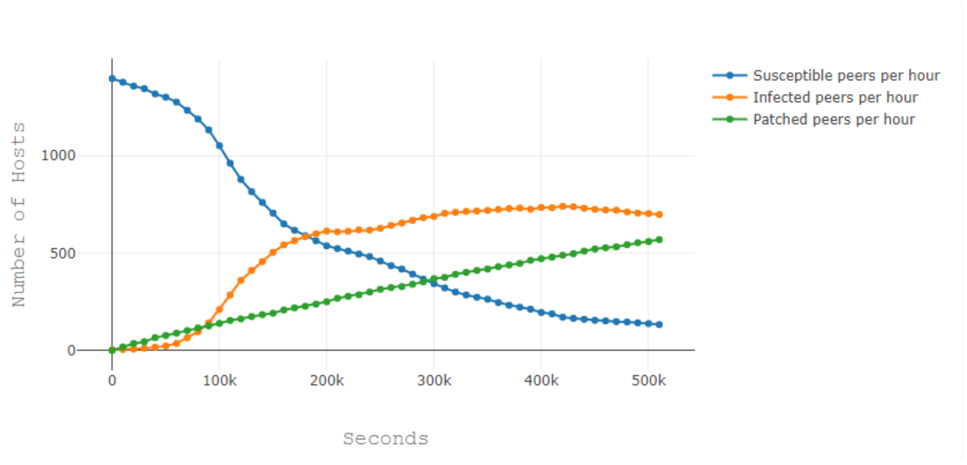


Figure 4.6: Graph for peers' states over time

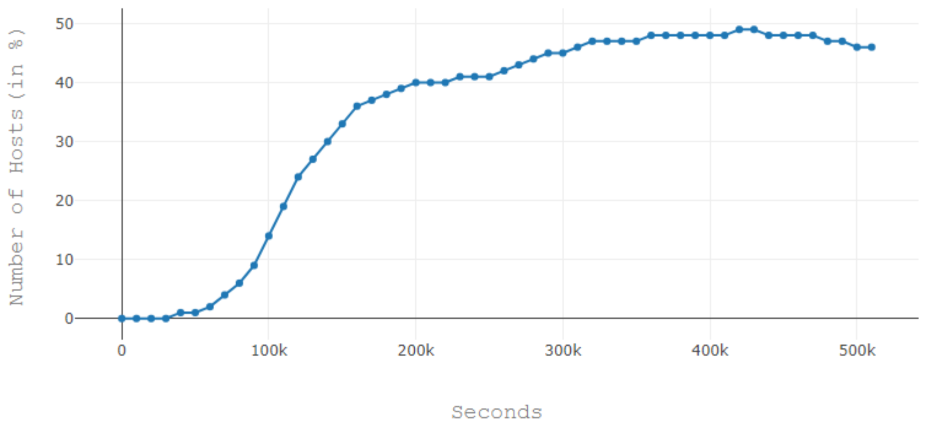


Figure 4.7: Percentage of infected per second

causing all nodes to be infected. But with the possibility that anyone can at any time install it, the number of infected people stabilizes at around 3 days and finally starts decreasing at the end of the run, where the number of infectable people is approaching zero.

A better view of the infectious rate can be seen in Figure 4.7, that shows a percentage representation of the number of infected with respect to the total number of nodes.

This suggests that a virus that can infect anyone, even under some specific conditions like in our case, is just a time-bomb ready to explode as soon as the ”right” node is infected. Once that one is infected, all the others will in a short time be too, and it stabilizes around its maximum level very quickly: in this case, the 3rd day is where the values attenuated.
Another interesting property to analyze, related to the graph represented in 4.7, is the number

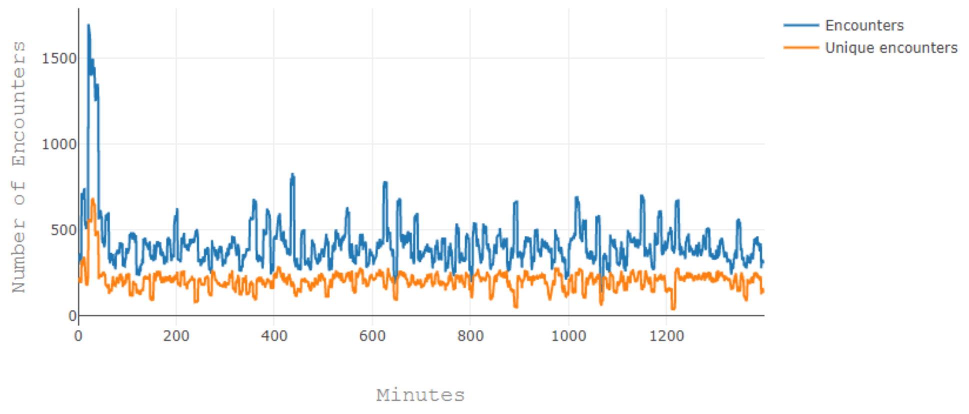


Figure 4.8: Percentage of infected per second

of encounters and the number of unique encounters. The graph has been interrupted after the first day because the unique encounters would just be around zero by that day, while encounters followed nearly the same pattern all the time: this way, a zoomed version makes it easier to visualize the picture.
What we can observe is that there was one big spike at the beginning because obviously nodes weren't connected, but then each node started doing its own routine finding from time to time new nodes.

Finally, a last Figure [4.9] representing how the infection went on in a specific simulation of a precise operating system. Putting them all together would just be impossible to understand what's going on considering the number of nodes.

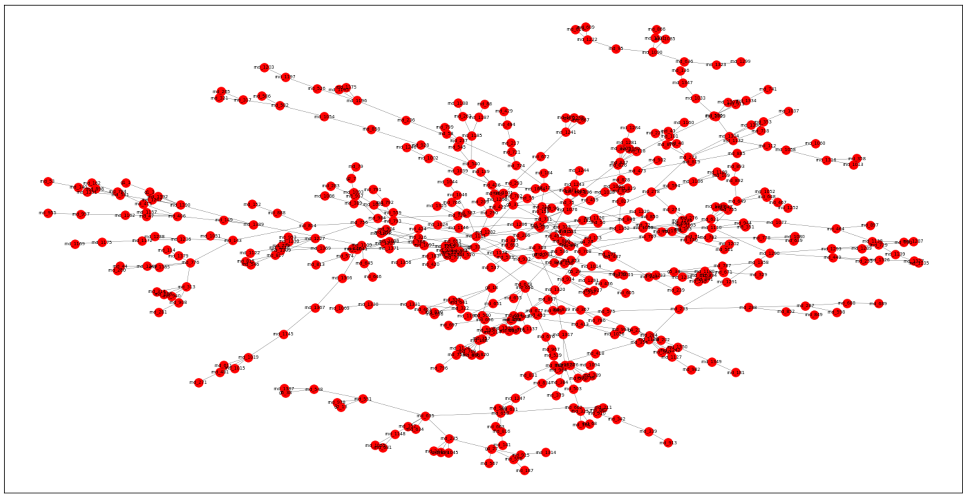


Figure 4.9: How a single OS infection went like

4.3.2 Structural properties of the graph

4.3.3 Infections graph

Node Degree	[0, 237, 122, 59, 35, 16, 4, 2, 1, 1, 0, 0, 0, 0, 0, 1]
Diameter	∞ (or 22 with one OS)
Clustering Coefficient	0
Density	0.004157785321438909

The **Diameter**, which is the maximum eccentricity and so is calculated with the maximum distance between two nodes of in G. This will be infinity as long as there are at least two different operating systems because they cannot interact with each other.

The **Density**, which is calculated as

$$d = \frac{m}{n(n-1)},$$

with n the number of nodes and m the number of edges in G, is consequently very small too.

The **clustering coefficient** will not be different from 0 in any simulation: infections happen from one node only, so when we check neighbors and relationships between them they'll have none. Formally speaking, it's a tree, not a graph.

The most important result is the one returned by the **Node Degree**: it's an array of *max-weight* positions, where the weight is the number of nodes connected to a given one: in this

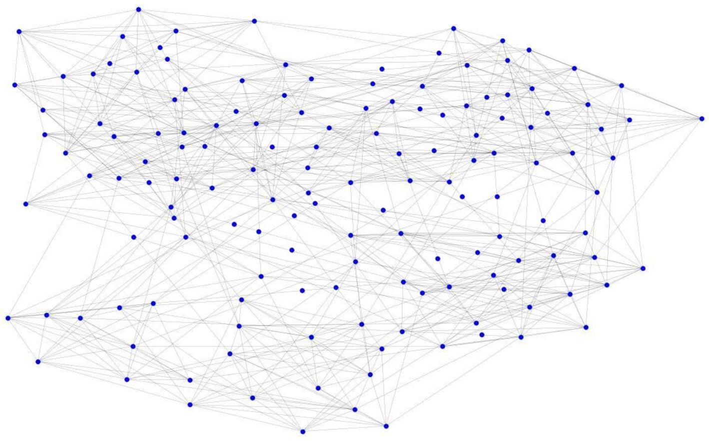


Figure 4.10: Connections in a hotspot

case, we can see that most of the infections happened with a 1 to 1 relation, meaning that there wasn't a plague-spreader: hosts would meet someone, get infected and then infect someone else themselves. The only exception's the last index, which infected 15 different peers. In total, there were 478 users sharing the same operating system *(this specific case was Android)* that got considered for this plot.

4.3.4 Contact graph

Considering what happens with the requested statistics, I've re-calculated them with a connected graph [4.10]: I've chosen a single hotspot at a specific time and considered a single Operating System. These are the results for the 199 nodes meeting these requirements:

Node Degree	[0, 131, 32, 14, 7, 2, 1, 2, 3, 1, 3, 1, 0, 1, 1]
Diameter	16
Clustering Coefficient	0
Density	0.011060151236231219

In this case, the graph isn't disconnected and so the diameter can be calculated: it gets up to 16 hosts, but as previously mentioned the clustering coefficient doesn't change because of the way the scenario is built. Finally, **Density** is way higher than the previous one but is still low.

Chapter 5

Conclusions

The goal of this project was to model how a virus would spread between smartphones in a simulated environment.
Taking and analyzing the results visible in Chapter 4, we can understand that dense zones are the most dangerous ones when we talk about pandemic viruses, which is coherent with the current pandemic situation we're living in. Grouping together is the main activity for people's social life, and we call this *Clustering*.
The ratio of infection or patch is of course relevant, but not as relevant as how much people cluster: for this reason, understanding the concept of *Point of Interest* is crucial.
Anyway, a simulation is an abstraction and can't comprehend all the constraints that the real world has, so for this reason every result has to be contextualized with the respective settings and biases introduced while extending the simulator.

Bibliography

[1] The Opportunistic Network Environment simulator. *https://www.netlab.tkk.fi/tutkimus/dtn/theone/*

[2] A Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. *https://networkx.github.io/*

[3] T. Camp, J. Boleng and V. Davies. A survey of mobility models for ad hoc network research.

[4] All the predefined functions implemented by the NetworkX package. *https://networkx.github.io/documentation/stable/reference/functions.html*