# WLAN-Opp-based Game: Infectious

Group Project Report

Manuel Stalder
Janine Thoma

November 10, 2013

**Advisor**: **Sacha Trifunovic**
**Supervisor**: **Prof. Dr. Bernhard Plattner**

Computer Engineering and Networks Laboratory, ETH Zurich

**Abstract**

The goal of this group project was to design and implement a game for Android which uses opportunistic networks. Such networks do not depend on a fixed infrastructure, but are formed by dynamic connections between devices. Often these networks are highly regional, because of the limited range of the used communication link, which is usually WLAN. Inspired by the way information spreads in opportunistic networks, the idea of the game is to design and spread viruses with pandemic potential.

In an editor, the player can design and modify viruses, which can then infect other players in the same network and possibly spread even further. This way, every player both spreads viruses, his own but also those which he is infected with, and receives enemy viruses. Points are achieved by a player's own viruses' spreading, infecting and killing other players.

Besides implementing and testing the game, we also evaluated two key characteristics of the finished application. The considered aspects were the data traffic and the energy consumption.

The results of these considerations were very satisfying, the energy consumption of the game is not noticeable by the user and the data traffic caused is well within reasonable bounds.

# Contents

# List of Figures

# Chapter 1

# Introduction

Opportunistic networks are a communication concept which in contrast to traditional networks does not depend on a fixed infrastructure, such as stationary wireless access points. Instead the network is formed by dynamic connections between the participating users' devices. With the rise of powerful smartphones with extended communication capabilities, such as WLAN, Bluetooth or 3GS/LTE, these networks can revolutionize the way data is transferred and processed. Opportunistic networks can enable communication in remote areas without wide network coverage or become useful in disaster scenarios. Furthermore they can be used for entertainment purposes, such as local social networks or gaming platforms.

Opportunistic networks lead to a new class of possible applications. However, they are still small in number. One reason for this may be that, in order to fully profit from opportunistic networking, a certain number and density of participants needs to be reached.

Our goal for this project is to explore the possibilities of opportunistic networks in a playful manner. The starting point is an Android application called WLAN-Opp [1], which provides a communication platform using opportunistic networks. It was developed at ETH Zurich's Communication Systems Group.

Thinking about how data can spread in opportunistic networks, we saw an analogy to how germs spread. With some inspiration from the well known browser and iOS game *Pandemic* [2], we decided that this is what the game should be about: Creating your own virus and letting it spread to other players, and from them to others. This type of game fully capitalizes on opportunistic networks. It is also a potentially entertaining game, as it is time and location dependent and directly interacts with other players. It also has further values: If the viruses' spreading is regularly registered in a database, this allows recording of how data spreads in opportunistic networks. Thinking even further, if the game is designed strictly by scientific parameters, it can be used as a simulation for real germ spread.

For the time being, our application *Infectious* is meant to be a simple game, enabling players to create their own viruses and letting them spread, while maintaining a high score board to monitor the progress of each player. With these options implemented, the game can easily be expanded to either a full and commercially viable game or a useful research application.

# Chapter 2

# Related Work

For *Infectious* two different categories of related work are important. One the one hand there is the work previously done on opportunistic networks. It is described in section 2.1. On the other hand there are the games we drew inspiration on when coming up with the idea for *Infectious*. These games are described in section 2.2.

## 2.1 WLAN-Opp

Our game uses opportunistic networks in order to spread the viruses created by the players. For this purpose, our application uses the service application WLAN-Opp. WLAN-Opp was developed at ETH Zurich. It enables opportunistic communication between classical WiFi clients by using tethering and stationary open APs [1].

## 2.2 Comparable Games

At the point of writing there are several games which give the player the opportunity to take control of their own disease. Examples are the flash games *Pandemic* [2], *Pandemic: Extinction of Man* [3] and *Pandemic II* [4] as well as *Pandemic 2.5* [5] for iOS and *Plague Inc.* [6] for Android.

In these games the ultimate goal is to eradicate humankind. The player is usually presented with a map of the world, indicating the virus' dissemination. In order to keep the governments on this fictional world from developing a cure against one's virus or from segregating the disease, the player must constantly modify the virus. Any changes to the disease take effect on the whole planet immediately.

In contrast to the existing games our viruses' spreading is not calculated by an algorithm but happens in a way similar to the natural dissemination of diseases. Viruses spread from one player to the next over the opportunistic networks provided by WLAN-Opp whenever two or more players come into communication distance of each other. This way any changes to a player's virus or to their arsenal of viruses (we do allow multiple viruses) will have to spread too, and do not take place everywhere immediately.

In *Infectious*, one does not simply create a virus, but simultaneously also becomes a potential victim of other people's viruses. All potential victims are also players. This means that victims will live on after their death, and it is almost impossible for one player to eradicate all others. Therefore, the goal in our game is to infect and kill as many people as possible, but not all of them.

# Chapter 3

# Design

This chapter gives a general overview of *Infectious*. In section 3.1 an outline of the game is given and in section 3.2 the different views of the application are explained.

## 3.1 Brief Explanation of the Game

The game *Infectious* is about creating and spreading a virus or several of them. Viruses are spread via opportunistic connections between the different players. Therefore every player is both creator and victim at the same time.

Playing this game, one is faced with several strategic choices. When it comes to creating viruses, the player has to decide what kind and how many viruses they want. Building a virus, the user has to pick the symptoms they want and their probability to arise. The difficulty hereby is to find the best combination of symptoms. The player does not a priori know the effects of a symptom on a victim, but they can guess a symptom's strength by its name (seizures have a graver impact on a victims health than flatulence for example). To find out more about the symptoms, a player has to watch the effects of other people's viruses on themselves.

Choosing the number of viruses may prove to be a difficult task. If the player creates too many viruses they will start repeatedly killing any victim which is close by and will therefore never spread past their direct victims.

Another decision the player can make is whether to avoid or seek the proximity of other players. Any additional infection could be lethal, but the contact to other players also gives the possibility to spread ones own viruses.

## 3.2 Views of the Game

The game has four different activities and one service. Each of the four activities has its own view. These views are described in the following subsections.

### 3.2.1 Home Screen

The home screen as seen in figure 3.1 is usually the first thing the player sees when starting the game. In the following a list of the different elements on this

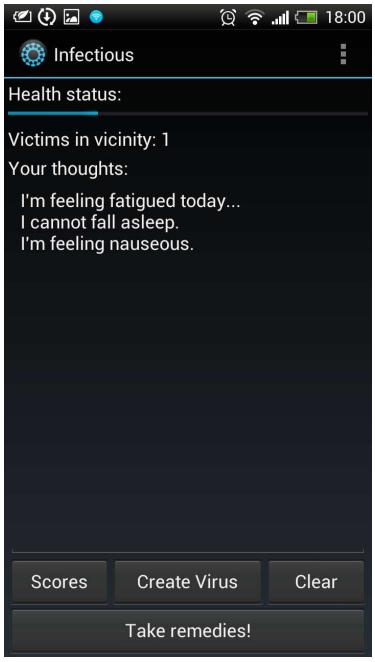


Figure 3.1: Home screen

screen and their purposes is given.

**Health status** The health of a player is represented by an integer between 0 and 100. The user sees this information in form of a progress bar.

**Victims in vicinity** Here the number of current neighbours as seen by WLAN-Opp is displayed.

**Your thoughts** Whenever a symptom arises, is cured or causes the death of the user a written message is added to this field.

**Scores** Pressing this button will get the player to the scores view.

**Create virus** Clicking this button will get the player to the first of the two views used to create a new virus.

**Clear** Pressing *clear* clears the messages in the *your thoughts* field.

**Take remedies!** Taking remedies causes the player's health to increase by 1. With a probability of 1% pressing this button causes side effects and decreases the player's health by 20.

Next to these fields listed above, the home screen also has several options in the menu, reachable by pressing the options menu icon in the action bar or using the corresponding button on the phone. Choosing any item on that menu will cause a dialogue to pop up. In figure 3.2 the *immunity* dialogue can be seen. A complete list of the options menu items can be found below.

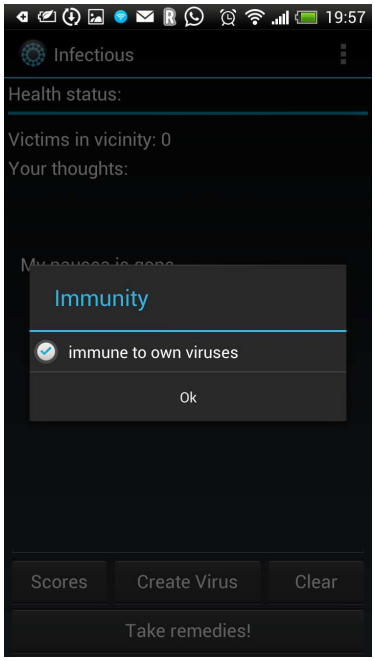


Figure 3.2: Immunity dialogue

**Change Google account** Every player using *Infectious* is linked to a google account. In this dialogue this account can be changed if the player has linked several google accounts to their Android device.

**Toggle service** The service, which will continue sending and receiving viruses when the player has closed the game, can be turned on and off with this option.

**Immunity** Especially for debugging purposes it can be handy to turn off the immunity of a player to their own viruses. This possibility is given here.

**Reset** If a player is no longer satisfied with their own collection of viruses, it can be deleted here. It is also possible to delete the list of viruses one has either survived or died of.

**Help** This option displays a short text explaining how the game works.

In addition to those listed above, there are three other dialogues that may show when starting the game, or when the user is returning to the home screen from another activity of this game.

**Loading data** This dialogue with a spinning wheel and the message *please wait...* shows up when the player has to be loaded from the database.

**Select a Google account** This dialogue shows when the application is started for the first time or after deleting the preferences belonging to the application. It asks the player to select the Google account they want to use to play *Infectious*.

**Enter your name** The player gets to choose a name which will be displayed to other users that are infected with one of the player's viruses. This dialogue only shows up if the player has not set a name before.

### 3.2.2 Scores

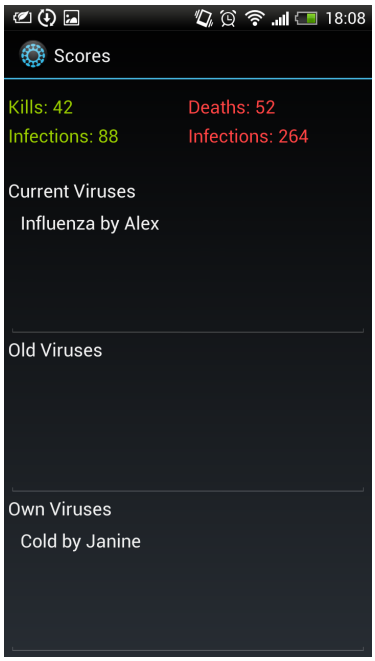


Figure 3.3: Scores view

The view belonging to the scores activity is shown in figure 3.3. At the very top of this view, right below the action bar, the user can see their scores. Displayed in a green colour is what is positive for the player: The number of players they killed and the number of players they infected. Displayed red are the number of times they were infected themselves or died.

Below the red and green numbers, there are three fields. Under *current viruses* the player can find any viruses they are currently infected with and which will eventually either be cured, end up killing them or be removed in case another virus kills the player first. Any virus that the player ever survived or was killed by is added to *old viruses*. The player is immune to viruses which are on this list. Any viruses the player creates are listed under *own viruses*.

### 3.2.3 Virus Editor

Clicking the *create virus* button leads the user to the virus editor, which allows the creation of a customized virus. The editor consists of two screens: one to enter a name for the virus and choose up to five symptoms the virus may cause (figure 3.4), and another to determine how the symptoms should behave (figure 3.5).

The second screen consists of an expandable list view with a row for each timestep of the virus, labelled as day 1 to 5. For each timestep, the player can distribute 75 percentage points to the chosen symptoms. This happens through seek bars, where a fully extended bar indicates 75 percent (this will set all other bars for this timestep to zero). It is not possible to distribute more than 75 percentage points. The percentage allocated to a symptom determines how probable it is that this symptom arises in this timestep. This way it is possible to pursue different tactics: a high number of symptoms arising with low probability, or fewer, more likely occurring symptoms. Some thought should also be given to which symptoms should be used for which strategy, as they differ greatly in their behaviour and severity. There is no explanation added to the different symptoms, because at this stage of the game, it is part of the challenge to find combinations that work well.

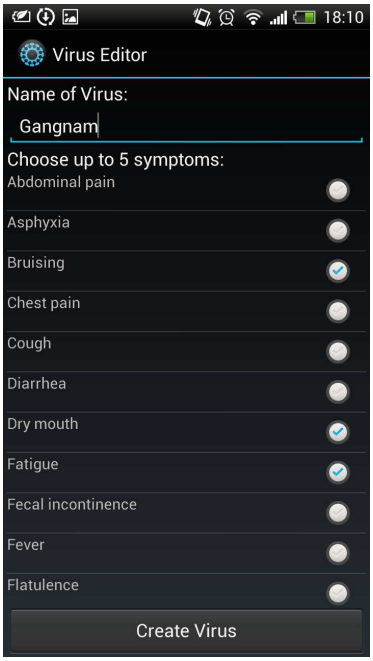


Figure 3.4: First virus editor view

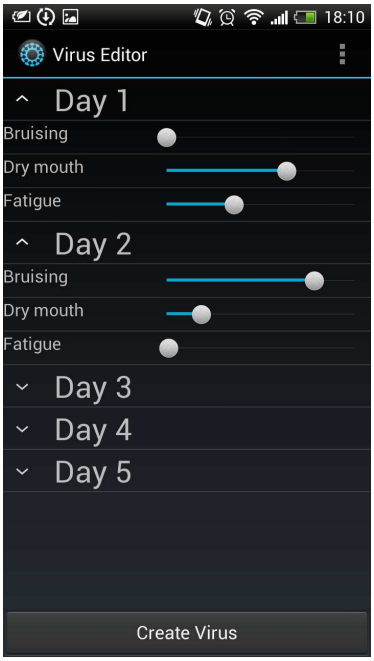


Figure 3.5: Second virus editor view

# Chapter 4

# Implementation

In this chapter a detailed description of the implementation of *Infectious* is given. The first section explains the internal representations used for players, viruses and symptoms. The second section is dedicated to the different mechanisms taking place whenever the player is actively using the application. The third section treats those mechanisms which are still active whenever the application is not running in the foreground. The last section describes the database used for *Infectious*.

## 4.1 Data Structure

*Infectious* needs several different entities of data structure. These entities are players, viruses and symptoms. In section 4.1.1 the solution used for the implementation of the data structures is introduced. In section 4.1.2, section 4.1.3 and section 4.1.4 a detailed explanation for each of the data structure types listed above is given.

### 4.1.1 Protocol Buffers

The data structures needed for this game require a multitude of different classes for their implementation. The player, viruses and symptoms also have to provide functions which allow conversion to and parsing back from byte arrays. The direct implementation of all the classes and parsers using Java would be error-prone. Luckily there exists an off-the-shelf solution for such cases. It is called protocol buffers.

Protocol buffers are a mechanism for serializing structured data. The desired structure has to be defined as a so called message in a *.proto* file and can then be compiled to the desired programming language. The returned source code allows the generation of instances of the predefined data structure. These instances (also called messages) can be read and stored in a byte array. By using a corresponding builder, the messages can even be modified [7].

Using protocol buffers also simplifies a potential extension of our structure. Additional variables to a virus, such as coordinates of creation for example, only have to be added in the protocol buffer file. After recompiling, the necessary method to set this variable at the creation of a new virus is available.

### 4.1.2 Player

All important data belonging to a player is ordered in a class created with protocol buffers. The corresponding protocol buffer message can be seen in figure 4.1. It includes fields for the player's name, account, scores and current health status (see lines 2 to 8). There are also three lists, each of them containing viruses (lines 10 to 12). The list of current viruses contains all the viruses with which the player is currently infected. The old viruses are viruses that the player either survived or was killed by. The list of own viruses contains the player's virus creations.

```
1  message Player {
2          optional string name = 3;
3          optional string accountName = 10;
4          optional int32 kills = 4;
5          optional int32 infected = 8;
6          optional int32 infections = 5;
7          optional int32 deaths = 6;
8          optional int32 health = 9;
9
10         repeated Virus currentVirus = 1;
11         repeated Virus oldVirus = 2;
12         repeated Virus createdVirus = 7;
13 }
```

Figure 4.1: Player protocol buffer message

### 4.1.3 Virus

This subsection introduced the virus data structure used in *Infectious*. Firstly a general account of the design decisions, made when creating the virus' structure, is given. Secondly the actual virus structure is discussed. Finally an outline of the creation of viruses is given.

#### General Design Decisions

When designing the virus' structure our chief aim was to create a large number of different possible viruses. For this purpose a virus is made up by several symptoms which the user picks from a list of symptoms.

By allowing the user to choose any number of symptoms from 1 to n this leads to

$$\sum_{i=1}^{n} \binom{\text{Number of available symptoms}}{i}$$

different possible symptom combinations. In *Infectious* we limited n to five and for the time being provide 25 different symptoms. This leads to a total amount of 68'405 different symptom combinations.

The amount of different possible viruses is even larger than that. Because the course of a disease is something that varies with time, we implemented our viruses' structure in a way which allows for their symptoms to change over time.

To achieve the desired time variability, the structure in figure 4.2 has been chosen. A virus consists of several timesteps, each of them possesses a list of

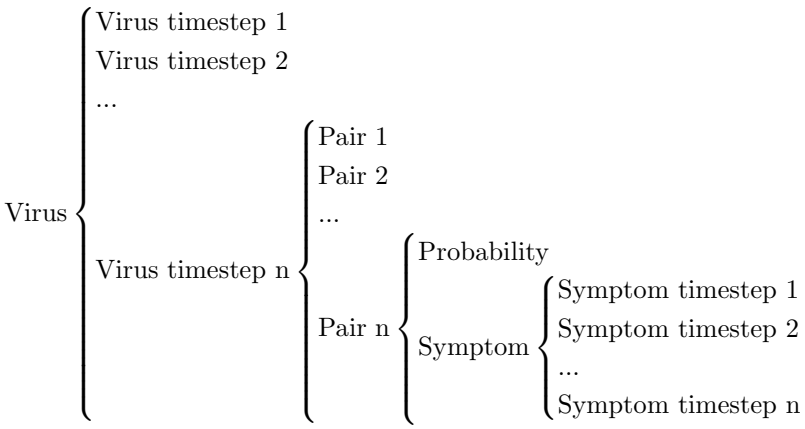


Figure 4.2: Virus structure

pairs. Each of these pairs consists of a symptom and a probability (given in percent) that this specific symptom arises during the timestep to which the pair belongs.

Symptoms themselves, too, do have timesteps. The probability for a symptom's severity to increase or decrease is a property which varies over time.

An example of a virus with the structure described above can be seen in fig. 4.3. The example virus only has two timesteps. During each of which both of the virus' symptoms (nausea and cough) have a certain probability of breaking out. The timesteps within each of the symptoms will be needed to determine the momentary effect of the symptom on the player's health at each point of time during which the symptom persists. The exact mechanism of how viruses and symptoms affect a player's health is described in section 4.2.2.

### The Virus Protocol Buffer Message in Detail

As seen in figure 4.2, a virus possesses several timesteps. The corresponding list can be seen on line 9 in figure 4.4. The *repeated* at the beginning of the line stands for a list with arbitrary length rather than a single item.

There are also other elements to the virus. From line 2 to line 7 there are several strings and integers to be found. The strings are used to save the virus' name, its creator's name and Google account and an id which is used to distinguish viruses from each other. The timestep counter field on line 6 is used to keep track of the amount of timesteps a virus has already aged. The probability of infection field might be used in future versions of *Infectious* to specify with what probability a virus infects a victim when it is received (at the moment this probability is always 100%).

The list in line 10 is necessary to keep track of the virus' current symptoms. The current symptoms determine the virus' impact on the user's health. Because a symptom cannot arise more than once in the course of a user's infection with a specific virus, it is added to the list of old symptoms (line 11) once its timesteps run out. Every time a new symptom would arise, this list and also the list of

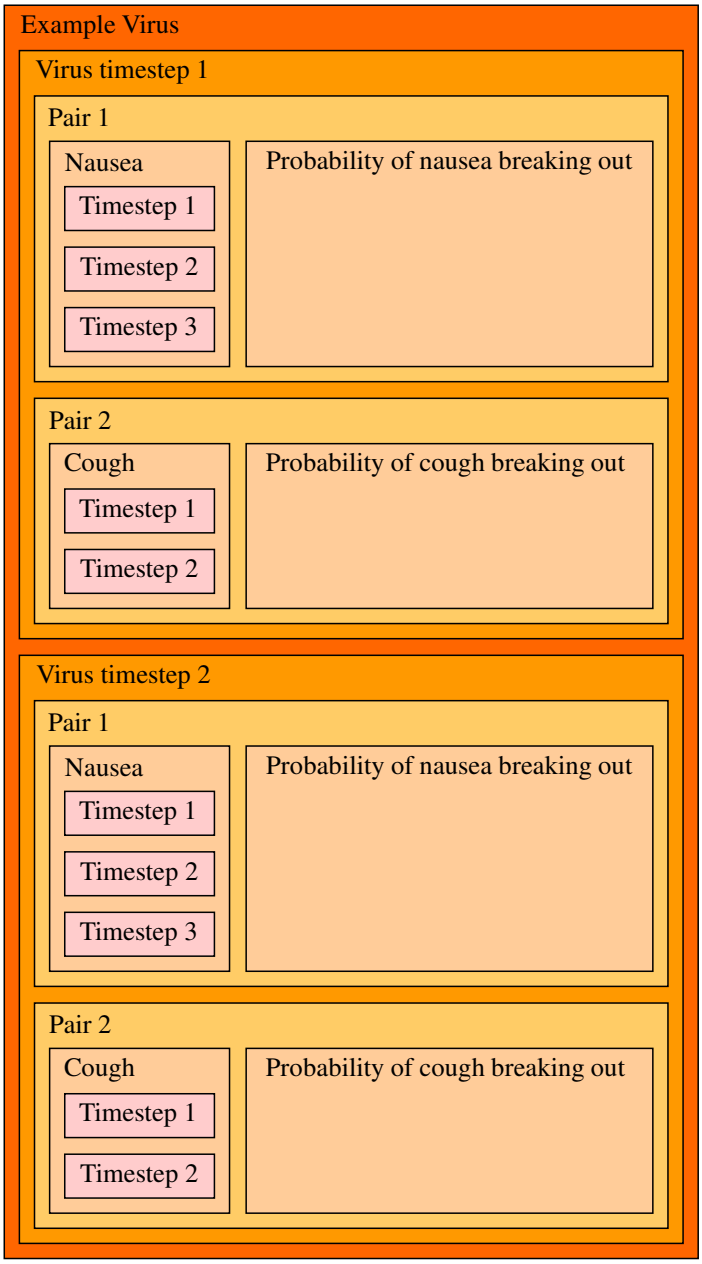


Figure 4.3: Example virus

current symptoms are consulted, to check whether the symptom is not already present on one of the lists. Only new symptoms are added to the current symptoms list.

On lines 13 to 15 in figure 4.4 it can be seen that the virus timestep message, the pair of symptom and probability message as well as the symptom message are all nested in the virus message. The virus timestep message, as seen in figure

```
message Virus {
        optional string name = 1;
        optional string creator = 2;
        optional string id = 3;
        optional string creatorAccount = 11;
        optional int32 timestepCounter = 6;
        optional int32 probabilityOfInfection = 10;

        repeated VirusTimestep virusTimestep = 7;
        repeated Symptom currentSymptoms = 8;
        repeated Symptom oldSymptoms = 9;

        message VirusTimestep {...}
        message SymptomProbabilityTouple {...}
        message Symptom {...}
}
```

Figure 4.4: Virus protocol buffer message

4.5, only consists of one repeated field: A list of symptom and probability pairs.

```
message VirusTimestep {
        repeated SymptomProbabilityTouple symptomProbabilityTouple = 1;
}
```

Figure 4.5: Virus timestep protocol buffer message

The symptom and probability pair message is shown in figure 4.6. It contains nothing but a symptom and its probability to arise. The message standing for a symptom is more complicated. It is described in subsection 4.1.4.

```
message SymptomProbabilityTouple {
        optional Symptom symptom = 1;
        optional int32 probability = 2;
}
```

Figure 4.6: Symptom and probability pair protocol buffer message

#### Creation of a Virus by the User

To prevent the virus editor from becoming overly vast the number of virus timesteps is limited to five. The number of possible symptoms for one virus is also limited to five.

In the first virus editor activity, the symptoms chosen by the player are saved into a newly created virus with five virus timesteps. Each timestep has one to five symptom and probability pairs, one for each symptom the user has chosen. Initially, all probabilities in each pair of every timestep are set to the following value:

$$\frac{75}{\text{Number of chosen symptoms}}$$

The player can modify the probabilities in the second virus editor view. By dragging the slider of one symptom to the very right, this symptom will get a

probability of 75% of breaking out during the according timestep. This maximum of 75% was chosen on purpose. In order to reach a probability higher than 75% for a symptom to arise, the user will have to maximize this specific symptom in several timesteps.

### 4.1.4 Symptom

The symptom protocol buffer message is shown in figure 4.7. In analogy to the virus protocol buffer message, the symptom message contains several optional fields (lines 2 to 9). The fields' names are chosen to be self-explanatory. The influence of the parameters symptom level and fluctuation speed on a player's health status is described in subsection 4.2.2.

```
1  message Symptom {
2          optional string name = 1;
3          optional string accountName = 2;
4          optional int32 timestepCounter = 3;
5          optional string messageAtInfection = 4;
6          optional string messageAtDeath = 5;
7          optional string messageWhenHealed = 6;
8          optional int32 symptomLevel = 8;
9          optional int32 fluctuationSpeed = 9;
10
11         repeated SymptomTimestep symptomTimestep = 7;
12
13         message SymptomTimestep {...}
14 }
```

Figure 4.7: Symptom protocol buffer message

As seen in figure 4.2, symptoms also have timesteps. These timesteps are represented by the symptom timestep message presented in figure 4.8. There is a repeated field for symptom and probability pairs which could be used for secondary symptoms (the current version of *Infectious* does not support this feature). Of high importance are the fields *probabilityOfSevertiyIncrease* and *probabilityOfSeverityDecrease*. Their purpose is described in subsection 4.2.2.

```
1  message SymptomTimestep {
2          repeated SymptomProbabilityTouple secondarySymptom = 1;
3
4          optional int32 probabilityOfSeverityIncrease = 5;
5          optional int32 probabilityOfSeverityDecrease = 6;
6          optional string message = 4;
7  }
```

Figure 4.8: Symptom timestep protocol buffer message

The symptoms in *Infectious* are created by us. There are no symptoms designed to be superior to others. All symptoms have both assets and drawbacks. We found inspiration for our symptoms by browsing the *List of medical symptoms* on Wikipedia [8].

## 4.2 Game Mechanisms

This section describes the game mechanisms relevant whenever the player is actively using the game. The continuance of certain mechanisms in the background after the application has been left is handled by a service described in section 4.3.

### 4.2.1 Spreading of Viruses

Every player both receives and sends viruses. Both actions are handled in a separate thread. Both threads are started in the main activity's *onResume* method. The process of sending viruses is described under *exhale thread*. The code necessary to receive viruses is documented under *inhale thread*.

#### Exhale Thread

The exhale thread is passed a runnable with the code used to send viruses. Within the *run* function of this runnable the code is surrounded by a while-loop which is executed as long as the main activity's global boolean variable *exhaling* is set to true. Using this variable, the sending process can be stopped.

Within the while-loop the thread first creates an array list which contains all the viruses from the player's created virus list. This array list is passed on to the *sendViruses* function which is specified in the main activity. This function iterates over all neighbours and over all viruses in the array list passed as argument, sending each of the viruses to every neighbour.

The sending code executed during each cycle of the two nested for-loops (neighbours and viruses) is shown in figure 4.9. Within the *sendViruses* function, these lines are surrounded with a try catch block to catch any socket or IO exceptions.

```
1 virus.toBuilder().clearCurrentSymptoms().clearOldSymptoms().build();
2 outSocket = new DatagramSocket();
3 outSocket.connect(n.getIPasInetAddress(), PORT);
4 outSocket.send(new DatagramPacket(virus.toByteArray(),virus.toByteArray
      ().length));
5 outSocket.close();
```

Figure 4.9: Java code for sending viruses

Before a virus is sent, its current and old symptoms must be cleared. Because a protocol buffer message cannot be edited directly, it has to be transformed into a builder first, then edited and then built back into a message again. This can be observed in figure 4.9 on line 1.

The sending is achieved with a datagram socket. To send the virus using the socket, it has to be parsed into a byte array. Protocol buffers provide a function *toByteArray*, which turns any message into a byte array. This is the function used on line 4 in figure 4.9.

Once the *sendViruses* function has sent all the created viruses of a player, the exhale run function creates an array list with the player's current viruses and passes this one on to the *sendViruses* function.

#### Inhale Thread

The inhale thread is passed a runnable, which inside a while-loop repeatedly tries to first create a datagram socket *inSocket* (if there is not one already) and then to execute the code in figure 4.10. First a new byte array (*buf*) is initialized. Then a new datagram packet called *packet* is created using this byte array. The in-socket is used to receive data. Any received data is saved in *packet*.

```
1 byte[] buf = new byte[8000];
2 DatagramPacket packet = new DatagramPacket(buf,buf.length);
3 if (inSocket != null && !inSocket.isClosed()) {
4 inSocket.receive(packet);
5 }
```

Figure 4.10: Java code for reciving viruses

If any data is received, the code shown in figure 4.11 is executed. The method *parseFrom* is used to reconstruct the received virus. The byte array passed to *parseFrom* has to have the exact same length as the array which was created when calling the method *toByteArray* on the original virus before it was sent to our receiver. If there are any appended zeros at the end of our byte array, the virus can not be reconstructed. *Packet.getData()* however has the length of *buf* (the byte array used to create *packet*). Therefore a new byte array *packetContent* is necessary. By creating this new array with the length *packet.getLength()* (unlike *packet.getData().length*, this will return the actual length of the originally sent byte array representing a virus) and passing it to *parseFrom*, a virus message can be reconstructed from the received bytes.

```
1 byte[] packetContent = new byte[packet.getLength()];
2 for (int i = 0; i < packet.getLength(); i++) {
3         packetContent[i] = packet.getData()[i];
4         }
5 try {Virus recivedVirus = VirusProto.Virus.parseFrom(packetContent);
6 } catch (InvalidProtocolBufferException e1){
7         e1.printStackTrace();
8 }
```

Figure 4.11: Java code for parsing viruses

Once a virus has been received, it is compared to any viruses in current, old and, if the player has set the option *immune* to true, also the created viruses. If the virus' ID is not the same as any of the viruses in these lists, its timestep counter is reset and it is added to the list of current viruses.

After a new virus has been added to the current virus list, the player's number of infections is incremented. There is also a message sent to the database causing the virus' creator's infected count to be increased by one.

### 4.2.2 Progression of Time

In the game's main activity there exists a runnable called heartbeat. This runnable is posted to the UI thread's handler in the main activity's *onResume* method. After every cycle of its execution it will post itself to the same handler

with a delay of one *HEARTRATE* (currently set to 10 seconds). When main activity's *onPause* method is called, any callbacks for heartbeat are removed.

In the following one full heartbeat cycle is described. Heartbeat iterates over all viruses in the player's current virus list. For every virus it will first check if any new symptoms arise. For this purpose it will check the virus' timestep counter and find the corresponding virus timestep (unless the timestep counter is larger than the number of timesteps). For any pair in this timestep it will call the function seen in figure 4.12 with the probability given in the pair as function argument. If the method returns true, then the symptom is added to the list of current symptoms (unless it is already present on this list or the list of old symptoms).

```
1  private Random rand = new Random();
2
3  public boolean probability(int percentage) {
4          int val = rand.nextInt(100);
5          if (val < percentage) {
6                  return true;
7                  } else
8                          return false;
9          }
```

Figure 4.12: Probability Java method

When a new symptom is added to the current symptom list, its timestep counter is set to zero and the *messageAtInfection* belonging to the symptom is displayed. Also the player's health is decremented by the symptoms' symptom level. If this reduces the health value to a number smaller than or equal to zero, the player dies.

When a player dies, the killing virus is added to the old virus list (this prevents the player from being killed by the same virus over and over again). Also, the player is presented with an according dialogue (which includes the name of all the current viruses and their creators) and the *messageAtDeath* belonging to the symptom giving the death blow is displayed in the *your thoughts* field of the main view. Furthermore, the current virus list is cleared, the health is reset to 100, the death count is increased by one and a message is sent to the database, leading to the kill count of the virus' creator being increased by one. All this (except the addition the old virus list) is handled by a function called *die*, which requires a symptom message as function argument. In case of a player's death, the iteration over all viruses is broken using a labelled break statement.

After having gone through all the pairs in the timestep, the virus timestep counter is increased by one. If the timestep counter had already been larger than the number of timesteps, it is checked if there are any symptoms in the current symptom list. If the list is empty, the virus is moved to the old virus list.

Still in the same loop of the iteration over all viruses, we now iterate over all symptoms in the current symptom list. For every symptom it is first checked if its timestep counter is larger than the number of symptom timesteps. If yes, then the symptom has been survived. In this case the symptom is added to the old symptom list, the players health is incremented by the symptom level (this becomes visible to the player by also updating the progress bar) and the

symptom's *messageWhenHealed* is displayed.

If the symptom has timesteps left, we look at the timestep belonging to the value of the timestep counter, and the symptom level is saved in a local variable. Next the probability of severity increase given in the symptom timestep is passed to the *probability* method. If it returns true, the symptom level is incremented by the value of *fluctuation speed*. Then the same thing is repeated with the probability of severity decrease, only the symptom level is decremented this time. If this would result in a negative symptom level, the symptom level value is set to zero.

The player's new health level is determined by first adding the local copy of the symptom level to it and then subtracting the new one. It has to be checked if this change has been fatal. If so, the player dies the same way as described above. If not, the symptom's timestep counter is increased by one.

After going trough all current symptoms, the for-loop for one virus is completed and the function moves on to the next virus.

The code for the heartbeat runnable is quite long. What adds to its length is the fact that we refrained from using required fields for our protocol buffers, since deleting such fields in the future could lead to incompatible versions. Because all our fields are optional, we always have to check if a field is set before accessing it.

Another complicating factor is that any for-loop over a list of messages or builders (such as a virus message), during which we change or delete any items on the list, cannot be realized with the simple code shown in figure 4.13. The for-loop will not execute the loops over a second or third element in the list, if the first has been deleted or modified. A solution to this problem is the code shown in figure 4.14. A new array list is generated, turned into an iterator and after the loop, the modified list is used to overwrite the original one.

```
1 for(Virus virus:player.getCurrentVirusList()){
2         ...
3 }
```

Figure 4.13: Malfunctioning for-loop example

```
1 ArrayList<Virus> virusList = new ArrayList<Virus>(player.
      getCurrentVirusList());
2 Iterator<Virus> virus_iterator = virusList.iterator();
3 while (touple_iterator.hasNext()) {
4         ...
5 }
6 player.clearCurrentVirus();
7 for (int l = 0; l < virusList.size(); l++) {
8 player.addCurrentVirus(l, virusList.get(l));
9 }
```

Figure 4.14: For-loop example

### 4.2.3 Mutation

A symptom's symptom level can potentially change every time *heartbeat* is called. If all the probabilities of severity increase and decrease belonging to one

symptom are balanced, the level is expected to stay around the initial symptom level. There are however symptoms, where either the probabilities of increase or decrease prevail. This means that there are symptoms where the symptom level increases or decreases over time. This increase or decrease respectively is continued in a new host, should the virus spread. The symptom levels are not reset before or after transmitting a virus. This means that some symptoms will become harmless after a certain amount of time whereas others will become lethal. The latter will eventually inhibit a virus' spreading because a host that immediately dies of a disease does not spread it to new victims.

## 4.3 Service

To increase the possibilities for the viruses in our game to spread, it is essential that the game does not only send and receive viruses when it is actively being played, but that both, sending and receiving, continue when the game is closed. For this purpose a service is used.

It is also necessary that the in-game time keeps progressing when the service is running. Otherwise the player would be overwhelmed by a huge load of new viruses when they return to the game after a long absence.

To provide these necessary functionalities, the service periodically invokes *breathing cycles*. Each cycle consists of a *starting task*, a *respiration task*, a *stopping task* and a *heartbeat task*. All of these tasks are implemented as runnables.

The breathing cycles are synchronized in time over all devices. In the current version of *Infectious* cycles always start on a full five minutes. That is XX.00, XX.05, XX.10 and so on.

Any time the main activity's *onPause* method is called, the service is started (unless the player enters the edit virus mode). In the service's *onCreate* method, the first *starting task* is posted to start at the next desired starting time for the cycle. The last command of every task posts the next task in the cycle to the handler of the service's main thread. Since *heartbeat task* is the last task of a breathing cycle, it will post *starting task* again with the needed delay for it to start on the next desired cycle starting time. The purpose of each of the tasks is described in the list below.

**Starting task** This task starts WLAN-Opp.

**Respiration task** If WLAN-Opp has detected any neighbours, this task will start both an inhale thread and an exhale thread. These threads are implemented analogously to their counterparts in the game's main activity.

**Stopping task** This task stops the inhale and the exhale thread, as well as WLAN-Opp.

**Heartbeat task** The heartbeat task runnable looks very similar to the main activity's *heartbeat* runnable. The only difference is that *heartbeat task* cannot post to any user interface. Therefore it saves any arising messages. The main activity's *onResume* method reads them out and posts them under *your thoughts* once the user opens the application.

The service's *onDestory* method removes any callbacks for the different tasks, closes WLAN-Opp (if it is not closed already) and stops the inhale and exhale threads if they are running.

If a user does not want the service to start once they close the game, it can be turned off in the options menu belonging to the main activity view. The service will also stop itself after 1000 cycles (that is about three and a half days).

## 4.4 Database

Other vital parts of the application are the web application and the database. The database is used to store all the symptoms available for the creation of the viruses and all the players.

The web application displays a high score board and enables the game administrators to add and modify the symptoms in the database through a user interface (which is only visible to those administrators).

The platform used for this purpose is Google App Engine. GAE is an easy way to develop web applications without worrying about servers or databases. The application is built on the same scalable system Google uses for their applications (e.g. gmail, youtube). It also provides a NoSQL schemaless object datastore, with a query engine and atomic transactions [9].

We decided to use GAE because it is a simple all-in-one solution. With a plugin for the eclipse IDE, which we also used for the Android programming, it made writing, managing and uploading the code relatively easy.

The web application consists of two JavaServer Pages and six small Java Servlets. The two pages display the high score table and the form to edit or add symptoms, the servlets handle the interaction with the datastore, which is part of GAE.

There are two entities stored in the datastore: symptom and player. The symptom entity has two fields, one for the name and one for the data, which is the binary representation of the built protocol buffer message. This way, symptoms can be stored easily and sent immediately, but it requires the server to handle protocol buffers to create and update them. The player entity works in a similar way, only it has two more fields: one for the nickname, one for the data, and two integer fields in order to count the infections and kills the player caused. The data is again the binary of the built player message, containing all information defined by the protocol buffer message. The counters are used to register all infections and kills caused by any virus of the player. As a virus spreads, every player who receives it will increase the *infected* counter of the virus' creator and do the same with the *killed* counter if killed by it. The standing of these counters are copied into the player's data whenever the player entity is updated in the datastore or requested by the application. So these two counters exist twice, once inside the player data, and once in the datastore entity, where they can be addressed directly by other players using our Android application.

In the following a list of the servlets and a short description of their functionality is given.

- *InjectSymptom* creates or modifies a symptom in the datastore. It can only be addressed by administrators (whose Google account must be registered accordingly in the servlet) using the online user interface and serves.
- *DeliverSymptom* answers the request from the virus editor in the Android application and delivers all symptoms currently in the database.

- *SetPlayer* modifies an existing player in the database or creates a new one. This servlet is only addressed by the Android application. It updates the player data and the nickname, and while doing so, it gets the most recent *killed* and *infected* counts and writes them to the player before building and saving it.
- *GetPlayer* delivers the requested player data from the datastore and also updates the two counters.
- *AddKill* and *AddInfect* increase the counters mentioned above. If a player is infected or killed by a virus, the Android application addresses these servlets providing the necessary information about the virus' creator. The addressed servlet then increases the respective counter of the virus' creator.

Figure 4.15 shows a graphic summary of the situation described in this section.

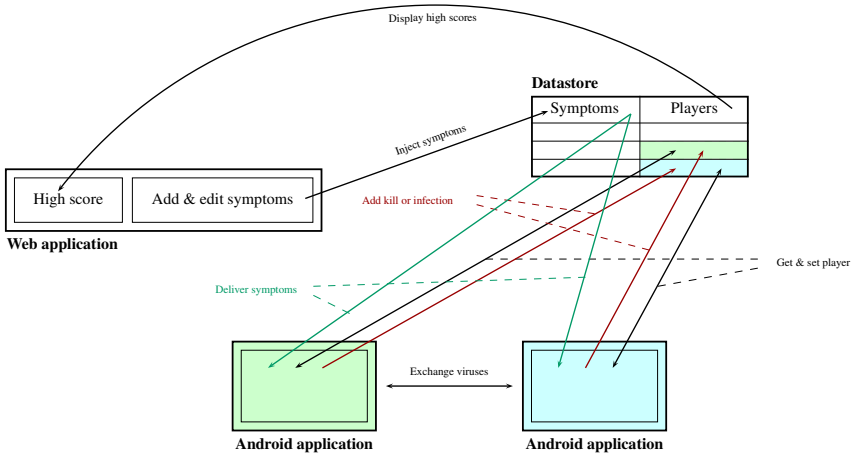


Figure 4.15: Communication overview

# Chapter 5

# Evaluation

When it comes to mobile applications, the energy consumption and also the generated data traffic are important factors. This chapter provides an analysis of both these aspects for *Infectious*.

## 5.1 Energy Consumption

Since our application is using a background service, it is important to test how much energy the application consumes. Ideally, it should be possible to run the service continuously in the background without impairing the battery lifetime noticeably.

We used two methods to observe the energy consumption. Combined, they should give a good estimate of the impact the app and its service have on a phone's battery life.

The first method utilizes a tool called Little Eye. Little Eye is a Windows program which allows to observe the energy consumption of single apps in real time. This program is used to evaluate and optimize the energy consumption and data traffic, as well as the memory usage of android apps.

While Little Eye provides very detailed momentary information on energy consumption, we were not sure if it could accurately predict the service's effect on battery life over a longer period of time. Therefore we also tested if the prediction made with Little Eye could be confirmed by just running the service over a long time.

All measurements were conducted twice, once on a Samsung Galaxy S3, and once on a LG P880.

### 5.1.1 Measurements with Little Eye

Because Little Eye allows monitoring in real time, it shows what happens when the service starts (both while the phone is in sleep mode or when it is actively being used for other purposes than playing *Infectious*).

After the service starts, WLAN-Opp runs for approximately 30 seconds. If the phone is in sleep mode, several high, but short power peaks can be observed in this time. This is shown in figure 5.1. The dotted line represents the overall

energy consumption of the device, which was in sleep during this test, while the red line depicts the consumption caused by the service.

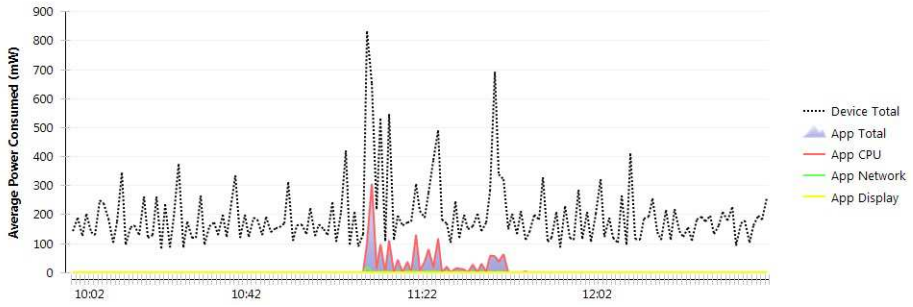


Figure 5.1: Energy consumption

These peaks elevate the consumption to about twice the value of the regular standby consumption over these 30 seconds on both phones. As the service starts every 5 minutes, this would sum up to an overall increase of energy consumption by 10%. This may seem like a considerable increase. However, when tested with the phone running, i.e. the screen turned on, the peaks were in the order of the regular fluctuation.

As mentioned above, these figures here only are a prediction. A field test was conducted to verify them.

### 5.1.2 Field Test

To test the prediction based on the Little Eye measurements, we observed the runtime of fully charged phones with the service running in the background and while being regularly used in everyday life. The goal of this measurement solely was to see how the increased energy consumption would be subjectively noticed by the user.

When compared to the runtime without the service running, we were not able to detect any remarkable differences. This is in agreement with our prediction based on the measurements by Little Eye.

### 5.1.3 Conclusion

The results of the evaluation of our application's power consumption lead us to the conclusion that the user will not find the battery lifetime noticeably impaired by *Infectious*. This is especially true if the phone is also readily used otherwise and not always kept in sleep mode.

## 5.2 Data Traffic

The core aspect of *Infectious* is the dissemination of viruses via opportunistic networks. This requires an exchange of viruses between players. This section provides an analysis of how large the exchanged viruses are and also on how often they are sent.

### 5.2.1 Packet Size

To determine the size of the data packet content that is transmitted from one player to another all possible viruses were analysed. Below, the minimum, maximum and also the arithmetic mean of the sizes of all possible viruses are listed:

$$\begin{aligned} \text{minimum virus size} &= 638 \text{ bytes} \\ \text{average virus size} &= 4753.08 \text{ bytes} \\ \text{maximum virus size} &= 8508 \text{ bytes} \end{aligned}$$

To obtain the values above, an iteration over all possible combinations of symptoms was used. The different symptoms vary greatly in size. Their minimum, maximum and average size is listed here:

$$\begin{aligned} \text{minimum symptom size} &= 105 \text{ bytes} \\ \text{average symptom size} &= 189 \text{ bytes} \\ \text{maximum symptom size} &= 606 \text{ bytes} \end{aligned}$$

There are also four more fields to a virus that can influence its size. However, since all of these fields are usually much smaller in size than the average symptom, for the measurement of the virus' sizes above, they were just kept to a random, but constant value. The influence of changes to the size of these fields is described in the following.

**User name length** For the virus size measurement this field was set to a String with 10 characters. Increasing the name length by one character leads to a virus that is one byte larger. Decreasing the value will lead to a smaller virus respectively.

**Virus name length** This field was set to a length of 10. Changing this length by one character will result in a change in virus size of two bytes. This is because the ID includes the virus name and therefore increases in length as well, if the virus name length is increased.

**ID length** This value's contribution to virus size is the most difficult to predict, since it is a combination of the virus name and a hashed version of the account name. However, it is possible to put an upper bound on this value. The hashed part's length will not exceed 11 ciphers. Therefore the maximum contribution of the ID to the virus size is the length of the virus name plus 11. For the measurements this field had a length of 25 characters.

**User account name length** The length of this field was set to 20 characters for the measurement. An increase of this value will affect the virus size in two ways. Firstly, there will be a direct increase of one byte per additional character and secondly, there can also be a change in the length of the ID.

### 5.2.2 Sending Frequency

To determine if the data traffic caused by *Infectious* is within an acceptable frame, it is necessary to not only know how large the sent packets are, but also how frequently they are sent.

When the application is active the exhale thread is responsible for sending out all the viruses in the lists of own and current viruses. Between every sending cycle the thread goes to sleep for at least three seconds.

When the application is running in the background, the three seconds from above are shortened to a random duration of time between one second and two and a half seconds. However, since the service is only woken up every five minutes and the phase where viruses are exhaled is only very brief, the effective data rate is much lower in this scenario. The exact numbers cannot be calculated for either scenario because they depend largely on the amount of viruses a player has. As long as this number is within reasonable bounds, there should be no overly large data traffic caused by *Infectious*.

# Chapter 6

# Future Work

The current version of *Infectious* does function as a game. However, there are still numerous possibilities to improve it. These possibilities are described in section 6.1. With an improved version of *Infectious* a release might become an option. Under 6.2 the possible difficulties that come with such a step are described. Finally, in section 6.3 it is described what could be done with a successful version of *Infectious*.

## 6.1 Extensions and Optimizations

One way to create a more entertaining gaming experience, would be to address the game's lack of mechanisms to defend oneself (other than the very simple *take remedies*). One possibility to tackle this problem and to also give the game commercial potential is an item store. With the introduction of an in-game currency and a collection of different items (e.g. health potions, vaccines), the game would obtain a new dimension. The implementation of this feature would be a time consuming task, as the items need much thought and testing in order to be balanced. However, it would not be very difficult, as the framework was designed with the possibility of such an extension in mind.

Further possible improvements to our game include the optimization of parameters such as the frequency of breathing cycles in our service, the number of allowed virus timesteps, the *HEARTRATE*, or the probability of infection. Such an optimization requires large scale testing or the modelling thereof by replaying real-world contact traces. However, testing might not be an option at this stage. It is unlikely that a large enough number of people will be willing to test a game which has not been optimized yet. Therefore simulations and calculations for optimization purposes become inevitable.

Another important aspect of improvement is to find and resolve any possible weak points in the creation of viruses, their structure and the properties of the game mechanics which might lead to an easily found optimal strategy which is realizable without too much effort. For example, one such possible weak point could arise, if the number of virus timesteps has to be increased significantly. In combination with the current implementation of our virus editor this would allow viruses with a long incubation period followed by an almost certain death. Such a weak point could be countered with further additions and limitations to

the virus editor, potentially linked to the in-game currency suggested for the item store.

## 6.2 Release

Unfortunately, even if we realize all the points listed in section 6.1, publishing a game like *Infectious* poses a challenge, as it needs a certain user density to work properly.

One scenario leading to the necessary density would be to first spread the word about the game amongst a medium sized group of people with frequent encounters among its members (i.e. our year of electrical engineering students) and ensuring that the game is entertaining enough for them to recommend it to their friends.

Any small imperfection that keeps the players in this first phase from recommending the game could keep *Infectious* from reaching the necessary density of players fast enough. In this case the already won players might lose interest as well. Therefore it is essential to ensure the absence of any potential flaws. For this purpose extensive testing is necessary.

Unfortunately the testing poses the same problem as an actual release. The game already needs a certain degree of completion, before it has the potential to attract enough volunteers for testing. To reach such a degree, we would have to extensively test and further improve the game ourselves. This might prove to be an exceedingly time-consuming task.

## 6.3 Data Gathering

If the suggested improvements in section 6.1 and the necessary steps described in section 6.2 lead to popularity of *Infectious*, an analysis of the viruses' spreading could be pursued. This would allow further insight into the way information diffuses in opportunistic networks. The possibility of collecting such information was one of the reasons why we decided in favour of a centralized database. In order to gather the desired information on viruses' spreading, every infection would have to be registered in our database together with additional metadata, such as the infection's time and location. The necessary extensions to our framework (i.e. the protocol buffer messages and the datastore) are of minor complexity. The more difficult part in gathering meaningful information would be the interpretation of the metadata. However, simply displaying a virus' infections on a map with timestamps might be interesting for the players of *Infectious*.

# Chapter 7

# Conclusion

We designed and implemented the game *Infectious*, which utilises communication over opportunistic networks as a central aspect in its game mechanics.

All the features we planned are implemented and work well, enabling the game to be played and evaluated.

Since this game's appeal largely depends on the number and diversity of fellow players within communication range, a detailed evaluation of the gameplay is only possible with a large number of participants. However, the game was played by a few people in a much smaller setting when testing the game for stability. Regarding stability, the results were very satisfactory. The game runs stable and does not show any unwanted behaviour.

The evaluation of the application regarding its power consumption and network traffic also showed positive results. The observed impact on the battery lifetime was minimal and the network traffic well within reasonable boundaries.

Concerning the quality of the gameplay it soon became evident that in order to maintain the user's interest the game must provide more means to influence one's health. A possible way to achieve this is the system of medication items we describe in section 6.1.

In its current state, *Infectious* is playable and a solid building block for future improvements.

# Bibliography


[1] Sacha Trifunovic, Bernhard Distl, Dominik Schatzmann, and Franck Legendre. Wifi-opp: Ad-hoc-less opportunistic networking. In *ACM MobiCom Workshop on Challenged Networks (Chants 2011)*, Las Vegas, NV, USA, September 2011.

[2] Pandemic. `http://www.darkrealmstudios.com/games.php?game=1`. [Online; accessed 1-July-2013].

[3] Pandemic: Extinction of Man. `http://www.darkrealmstudios.com/games.php?game=2`. [Online; accessed 1-July-2013].

[4] Pandemic II. `http://www.darkrealmstudios.com/games.php?game=4`. [Online; accessed 1-July-2013].

[5] Pandemic 2.5. `https://itunes.apple.com/us/app/pandemic-2.5/id483737492?ls=1&mt=8`. [Online; accessed 1-July-2013].

[6] Plague Inc. `https://play.google.com/store/apps/details?id=com.miniclip.plagueinc&hl=en`. [Online; accessed 1-July-2013].

[7] Protocol Buffers. `https://developers.google.com/protocol-buffers/`. [Online; accessed 2-July-2013].

[8] List of medical symptoms. `http://en.wikipedia.org/wiki/List_of_medical_symptoms`. [Online; accessed 2-July-2013].

[9] App engine. `https://developers.google.com/appengine/`. [Online; accessed 3-July-2013].