GO VIRUS, GO!

A visual simulation of a pandemic

Operating systems and process-oriented programming (1DT096)

Abstract

A virus being spread in a community is a process that might be hard for humans to grasp due to the size and scale of it. This project delivers a visual representation of this complex process to facilitate a more intuitive understanding of it. Not only can the user observe the virus spreading visually but can also affect its course and intensity. This is done by choosing from a variety of parameters that can be changed and adjusted. Statistics are collected and shown during a simulation run.

Johan Apelgren, Marcus Jansson, Aljaz Kovac, Pontus Ljungren, Måns Rönnberg, Alexander Sabelström, Victoria Täng, Marion Wallsten

Contents

1. Introduction and purpose

There is a common understanding in the field of IT that things often take much longer to implement than they were originally estimated to. A feature projected to take two months ends up taking six, much to the manager's dismay. Of course, this habit of under-estimating and over-promising is not unique to programmers. Humans in general are notoriously bad at estimating things, whether it be time, cost, benefit, risk [1], or something more mundane, such as the number of people in a crowd [2].

This is doubly true when dealing with events that occur on scales beyond what we can intuitively grasp. The ongoing coronavirus pandemic is an example of one such event. Given the obvious concurrency-related aspects of a virus moving through a population, we decided to create a visual simulation of a pandemic. A visual simulation could help us, and perhaps others, better understand how even small changes in the underlying conditions during a pandemic can create very different end results. For example, given that masks have a certain degree of effectiveness at preventing spread, what happens if we double the number of people who use them? Or how does the introduction of a vaccine affect the scale and intensity of the spread? Etc.

To allow the tweaking of results in this fashion, the simulation provides various parameters that can be adjusted to create different conditions for the simulation. Examples of these conditions might be the percentage and effectiveness of mask use, the degree and effectiveness of vaccinations, how many locations and individuals are simulated, and so forth.

1.1 Related work

A product that is similar to this project is the videogame *Plague Inc.* [3], released in 2012. In *Plague Inc.* the player creates its own virus, with the goal of infecting the entire world. Throughout the game the virus mutates and evolves and the spread is visually represented on a map of the world. The game has over 85 million downloads and several developments have been released, such as a post-corona game mode where you try to spread a vaccine instead [4]. Further development on our project could be made with inspiration from *Plague Inc.*

2. System architecture

The simulation is implemented through a concurrency system written in Go, coupled with a visualization system written using Gotron [5], an Electron.js implementation for Go. This allows us to build the user interface in HTML and CSS and use Javascript. Go's native concurrency features allow us to easily model the characteristics of the virus as well as the individuals who will be spreading it. Diagram 1 gives an overview of the system architecture.

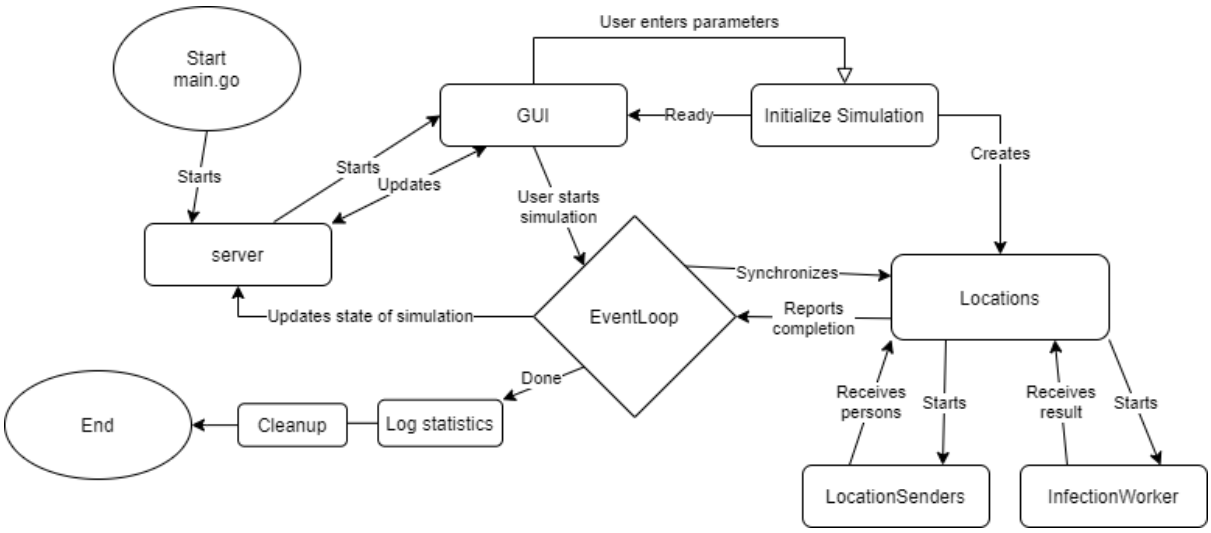

Diagram 1: The system architecture of the simulator.

The server and GUI represent the frontend, made through Gotron, while the remaining part of the system architecture is the backend, made in Go.

2.1 User interface

2.1.1. The technical implementation

2.1.1.1. Technologies used

The choice to build the user interface in Gotron provides access to all Electron.js [6] features in Go. As seen in diagram 2, Electron works by starting a server on your local machine and then connecting a basic Chromium-based web browser to it. This means that you can utilize HTML, CSS and Javascript to build cross-platform UI. Finally the UI connects to a websocket started by the Go program which is used to send messages between them. The

websocket is a standard TCP socket and allows messages of any kind to be sent through. This is used to update the UI and send settings to the simulation.

There is a clear divide between the UI and the backend, therefore it is easy to apply the Model-View-Controller design pattern [7]. Since a controller needs to parse all messages between the frontend and the backend, there is no way for the UI to affect the backend unintentionally.

To draw the simulation a library called p5.js [8] is used. P5 is a free and open source Javascript library that creates a canvas in HTML which can then be drawn onto. This allows easy creation of animations that the simulation can use, e.g., the movement of circles representing human beings can easily be achieved by using the lerp() function which uses the principle of linear interpolation [9]. The interactable items like buttons and menus are written in HTML.

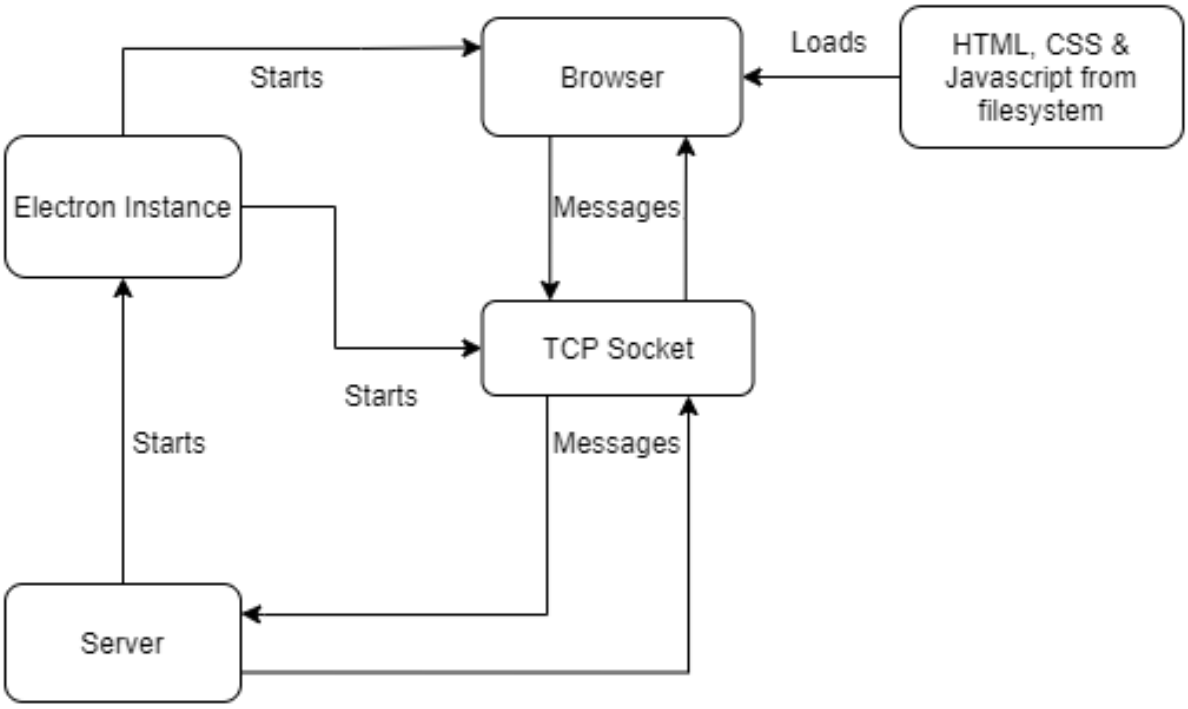

Diagram 2: The Server - Browser architecture for the UI.

2.1.1.2. Communication between backend and frontend

Before anything can be drawn to the screen the information on what and where to draw needs to be received. This information is kept in the backend, but how can it be communicated to the frontend? The communication process takes place in 3 steps. The first

step involves sending a message (in this case, an array of data) to the frontend and “tagging” this message with an event type. The second step is receiving the data by scanning for the right “tag” and, when this tag is found, calling a function that parses the message and handles the data sent. The parsing and handling of the message contents is the third step in the communication process. All these steps are done via **G**otron, with simple send and receive handlers.

It is important that the frontend and backend remain in sync otherwise the frontend is not particularly useful. To achieve this the messaging system is used to block execution until a specific sync message is received.

When a time step is done in the backend the frontend receives all the new information in one message and that means that it should update the UI to match the new data. After the UI update is finished, a message is sent from the frontend to the backend which allows the backend to proceed to the next time step. This enforces that the UI will always represent the latest time step that the backend has processed.

2.1.2 How to use the UI

2.1.2.1 Setting the parameters

When the program is started the user interface is opened. This contains the canvas upon which the simulation is drawn, real-time statistics of the simulation are shown on the right, and options for setting the simulation parameters can be chosen in the dropdown menus above the canvas. There are three dropdown menus for setting the simulation parameters available to the user.

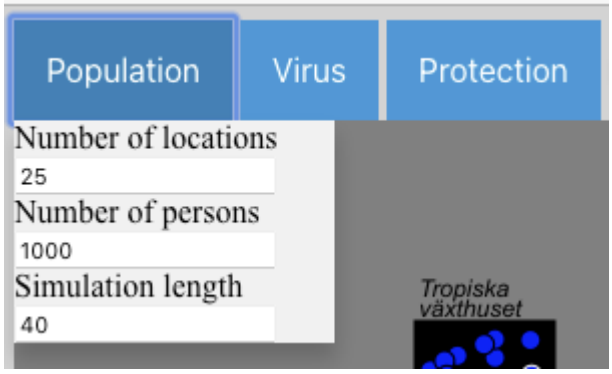

Diagram 3: Population parameters.

In the Population dropdown menu the user can set the number of locations, number of persons and length of simulation, where the length of simulation refers to the number of time steps the simulation will run. A time step is a time "slot" in which persons can be infected and moved from one location to another (see Diagram 3).

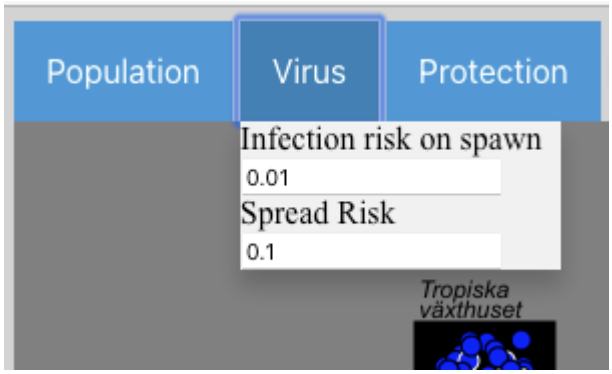

Diagram 4: Virus parameters.

In the Virus dropdown menu the user can set the infection risk on spawn. This is the probability that a person will already be infected on spawn, e.g., if the infection risk is 0.5 then roughly half of the spawned persons should be infected (see Diagram 4).

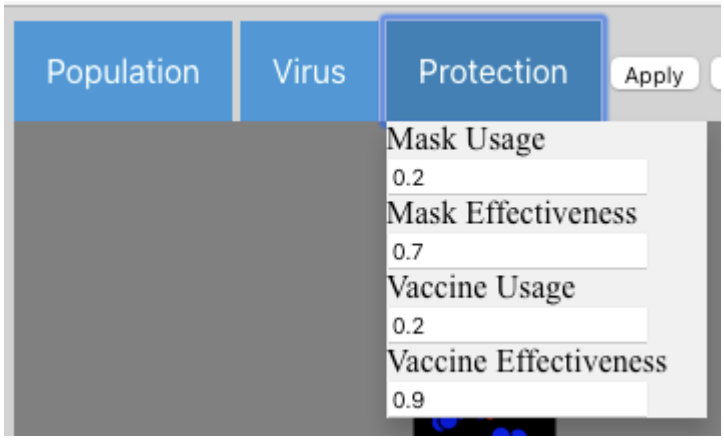

Diagram 5: Protection parameters.

In the Protection dropdown menu the user can set the following parameters: mask usage, mask effectiveness, vaccine usage and vaccine effectiveness. Mask usage is the percentage of persons that are wearing a mask. Mask effectiveness defines how effective the masks are. Vaccine usage is the percentage of persons that have been vaccinated, and vaccine effectiveness is how effective the vaccine is (see Diagram 5).

2.1.2.2 Initializing and running the simulation

Once the parameters have been set by the user, the *Apply* button activates the chosen parameters for the next simulation run and draws the state on the canvas (see Diagram 6).

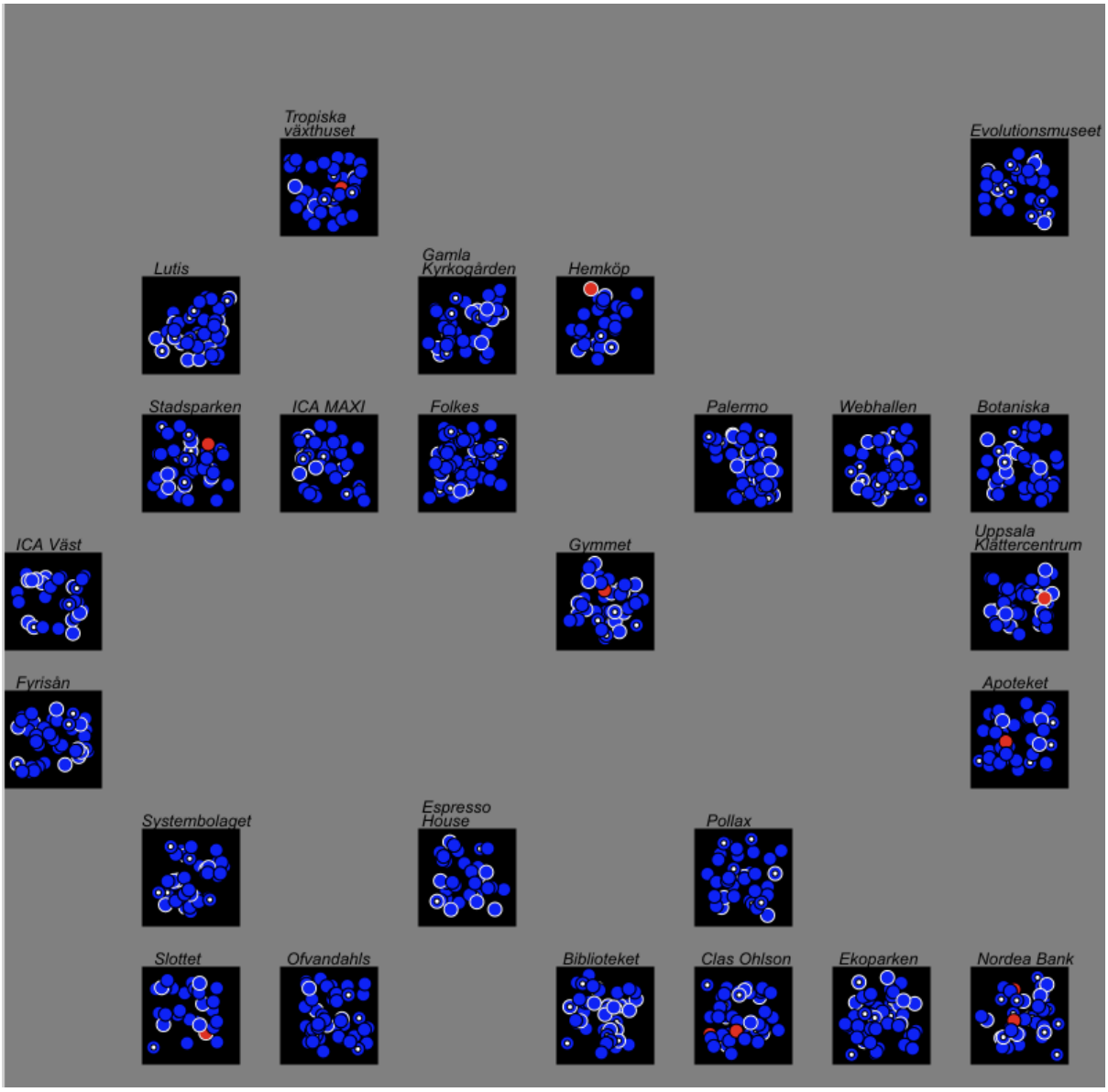

Diagram 6: An initiated but not-yet running simulation.

Once the *Apply* button has been clicked, the chosen amount of locations and persons are drawn on the screen. Locations are represented by black squares and they are placed on random locations spread out inside the canvas. Each location also gets a name assigned to it which is written above the square. The persons, represented either as blue or red circles, depending on their infection status (red for infected, blue for healthy, white dot inside circle

if vaccinated and a white-lined circle if the person is wearing a mask), spawn at random locations. If the user is unsatisfied with the parameters, new parameters can be simulated by altering the drop-down menus and pressing *Apply* again. The *Simulation* window then reinitiates and draws a new canvas with the updated parameters.

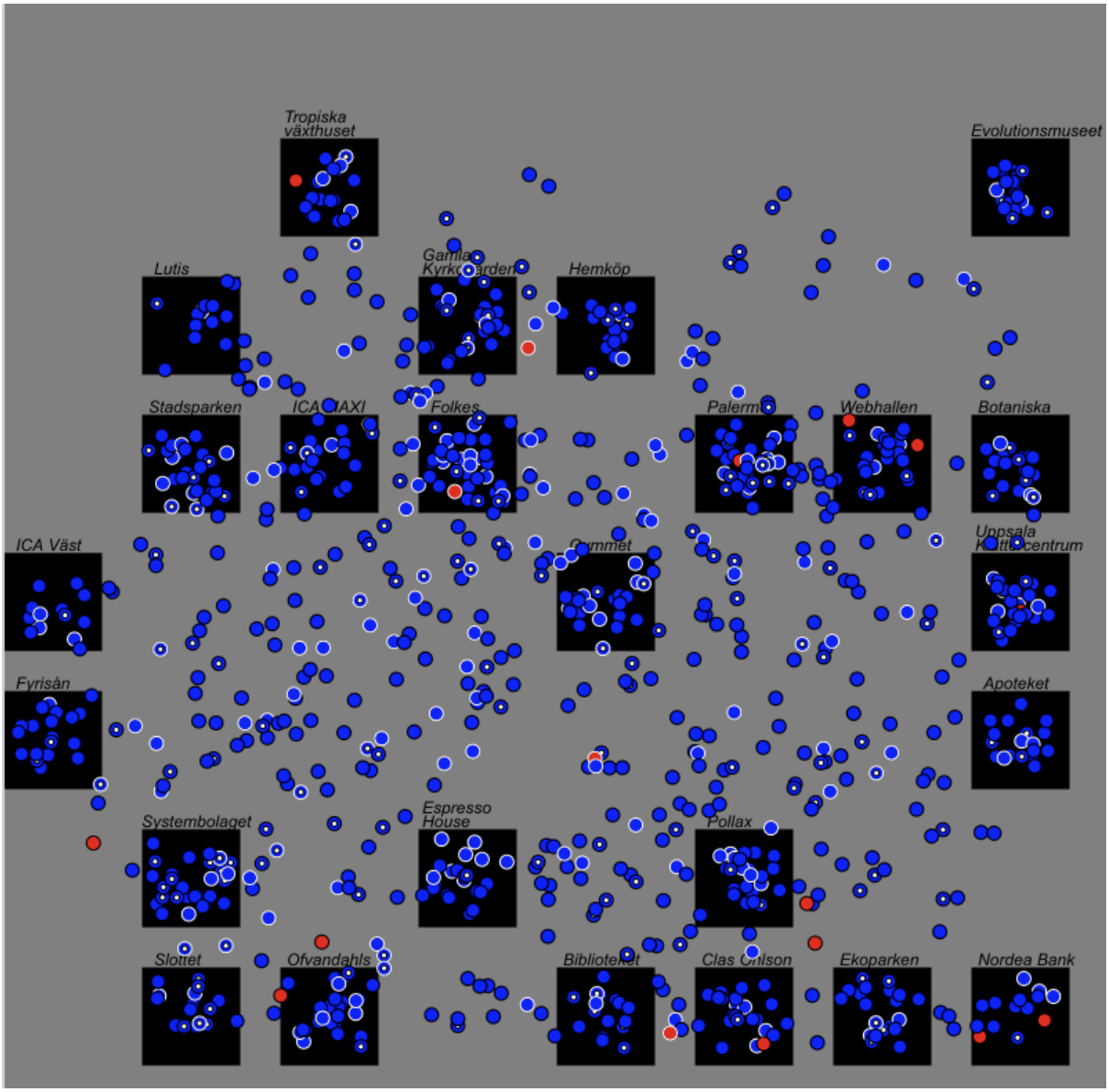

Diagram 7: A running simulation.

Once the simulation starts (by pressing *Run*), the persons are either moving inside a certain location randomly (while waiting to be moved to another location), or are on the move to the next location on their predetermined path (see Diagram 7). If a healthy individual becomes

infected they change their color from blue to red. An infected individual can never become healthy during a simulation run.

The section to the right of the canvas (see Diagram 8, 9, 10) shows statistics about the simulation during runtime. There are 3 statistics options to choose from: statistics for total infected population, statistics for newly infected, and overall statistics where one can also find detailed statistics for every location the mouse hovers over.

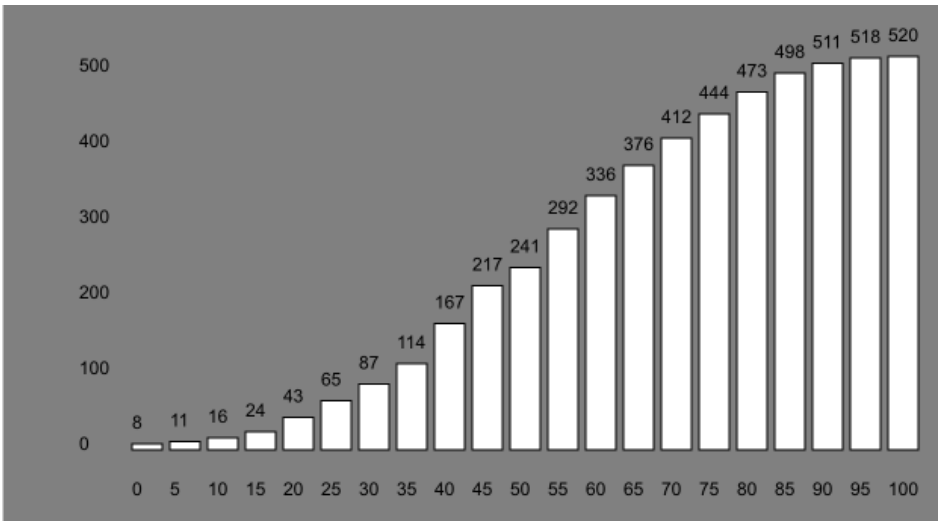

Diagram 8: Example UI statistics showing total infected per 5 ticks.

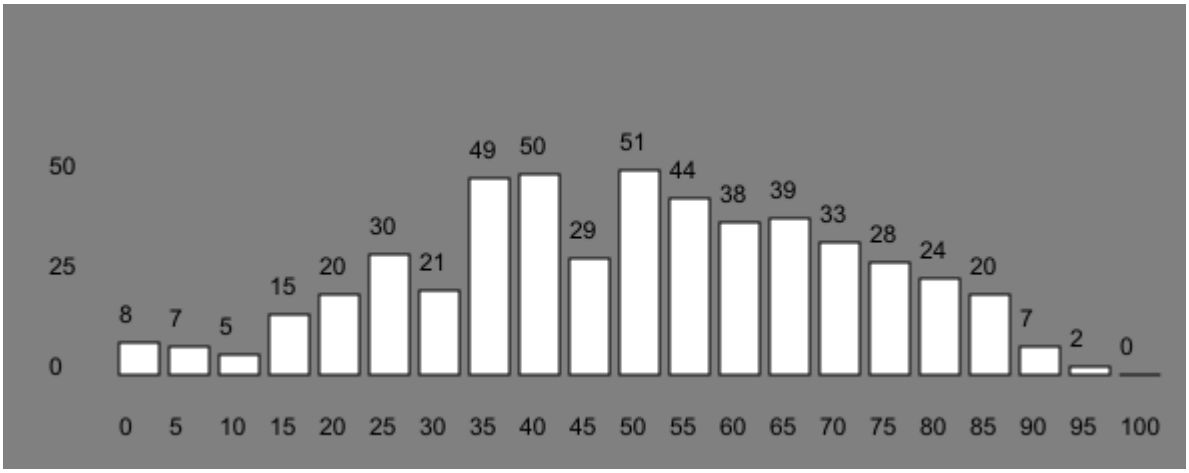

Diagram 9: Example UI statistics showing newly infected per 5 ticks.

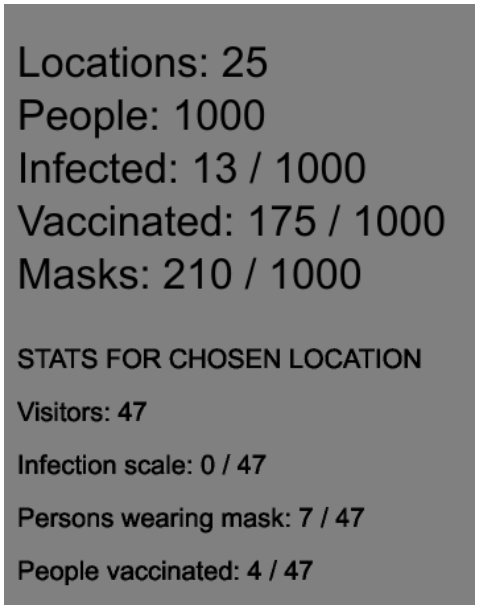

Diagram 10: Example UI statistics showing overall statistics, with stats for a hovered-over location at the bottom.

2.2 Backend

2.2.1 Choice of programming language

The Go language was chosen for this project primarily due to its built-in concurrency features. The primary feature is the so-called “goroutines”, which are concurrent subroutines that are automatically managed by the language during runtime. The goroutines run in the same address space, and communicate through the use of “channels”. These channels do not make use of memory sharing, which means there are no issues with data races.

Other types of race conditions can still be present depending on the design of the system, but these can be addressed through the use of synchronization tools, the simplest of which is the fact that receiving from a channel, as well as sending to an unbuffered channel, are both blocking operations. These features, combined with the relative simplicity of Go’s syntax and its ease of use, made it a perfect fit for the project.

2.2.2 Choice of Data Structures for Locations and Persons

Since the simulation represents persons moving between different locations in a small city, one of the first design decisions made was how to represent these persons and locations. There were two fairly obvious options: either represent each person as a goroutine that holds information on (among other things) their current location, or represent each location as a goroutine that holds information on (among other things) their current visitors. Since we

would have a lot more persons than locations, having persons be goroutines would result in a larger overhead. Additionally, representing locations using permanent “places” that accept visitors aligns closely with reality, which makes it easier to intuitively grasp how the system works. For these two reasons we use goroutines to represent locations, and structs to represent persons.

2.2.3 How a person is represented

One person struct is generated for each individual person. These structs contain various attributes necessary for the simulation or for the collection of statistics. These attributes include the ID of a person, whether the person is infected or not, which locations the person will visit, whether the person is vaccinated or uses a mask, how many locations the person has visited, the times they departed from different locations, whether they have finished travelling, and so forth. Once a person has been created, they are sent to their initial location to await the start of the simulation. This is done by sending the struct to that location's dedicated channel for receiving visitors.

2.2.4 How a simulation is initialized

Once parameters have been sent from the GUI, the *InitializeSimulation* function is run to apply these parameters to the simulation. This includes creating all necessary channels for communication between locations, for collecting statistics, and for communication between the backend and the GUI. It is also responsible for starting the goroutines that handle statistics and GUI-communication. *InitializeSimulation* then creates the locations by starting the appropriate number of *LocationWorker* goroutines, and the persons by generating the appropriate number of structs. Once this function is complete, the simulation is ready to start.

2.2.5 How locations work

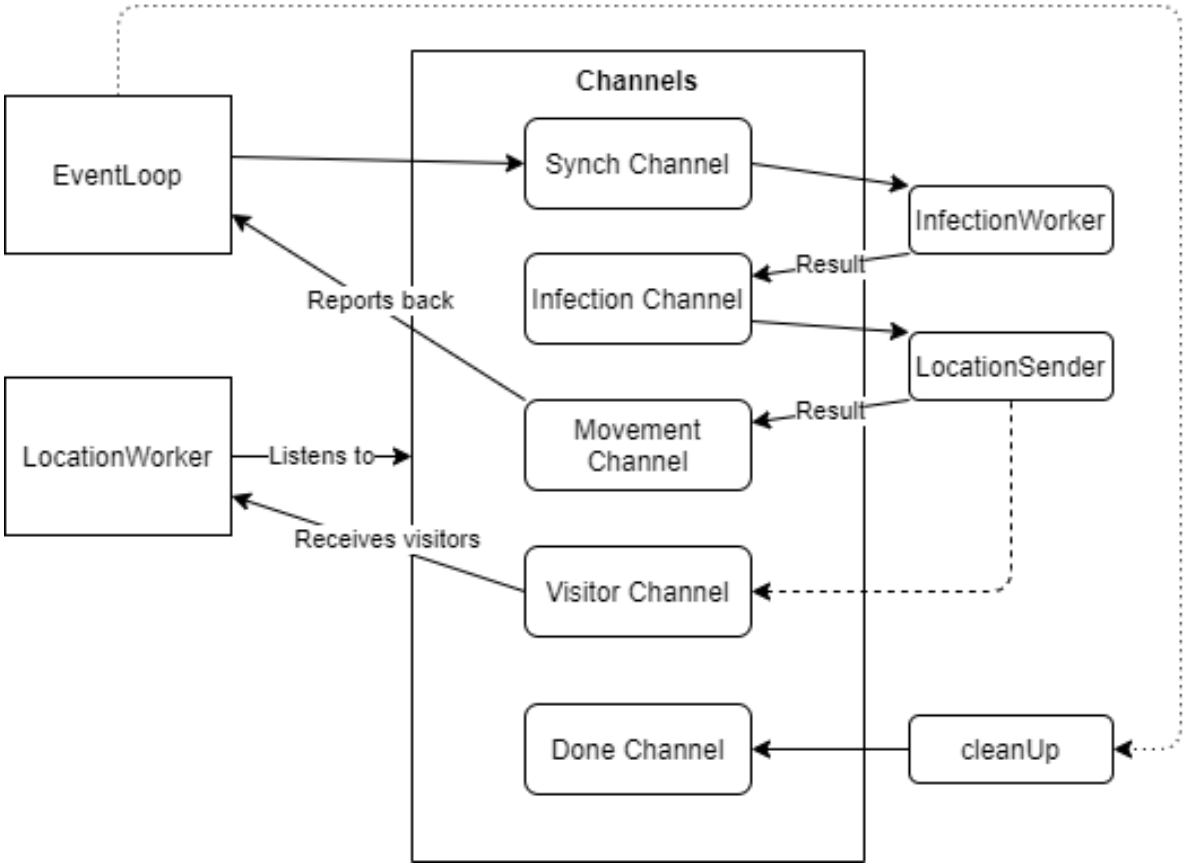

Diagram 11: LocationWorker flowchart

As mentioned, locations are represented by the *LocationWorker* goroutine, of which one instance is started for each location by the *InitializeSimulation* function. The *LocationWorker* remains active for the duration of the simulation, constantly running in the background, and during this time it will run an endless for-loop over a Select statement. A Select statement is a built-in feature in Go, used to listen to multiple channels when you do not know the order in which these channels will receive messages.

The *LocationWorker* listens to five different channels: the synch channel, the infection report channel, the movement report channel, the receive channel and the done channel.

The 'synch' channel is used by the event loop to synchronize the simulation for each time step. How this synchronization works is explained in more detail in section 2.2.8.

Receiving a message through this channel indicates that the *EventLoop* wants to proceed to the next time step, and causes the location to initiate the steps that must be completed before doing so.

The first of these steps is to start an *InfectionWorker* goroutine, which is explained in detail in section 2.2.6. Briefly, the purpose of the *InfectionWorker* is to calculate which of the current visitors will become infected by the virus during the current time step. Once this goroutine has been started, the *LocationWorker* goes back to listening to the other channels.

The 'infection report' channel is used to receive the return value from the *InfectionWorker* goroutine, and begins the second step each location must complete before continuing to the next time step. The return value from the *InfectionWorker* contains the number of persons infected during the previous round, as well as all visitors currently at the location, with their infection status possibly altered by the *InfectionWorker*. Once this information has been received, the *LocationWorker* saves the number of infected persons for statistics purposes, and starts the *LocationSender* goroutine, sending it the updated list of visitors it just received from the *InfectionWorker*. The purpose of the *LocationSender* is to determine if any of the visitors are leaving the location, and will be explained in detail in section 2.2.7.

The 'movement report' channel is the third and final step towards continuing to the next time step. This channel receives the return value from *LocationSender*, i.e. the list of persons that did not move on to another location. Once this has occurred, the Location is finished with all the steps it needs to take for the simulation to proceed to the next time step. To indicate this, the *LocationWorker* performs the Done method on the current timestep's WaitGroup (explained in detail in the EventLoop section, 2.2.8).

The fourth channel is the 'receive' channel, which is used to receive new visitors coming from other locations. When a person is received by a location, the location will run the *handlePerson* function on the person, which will perform necessary checks (e.g. whether this location is the person's final stop) and update the relevant values in the struct. After this is done, the *LocationWorker* will add the person to its list of current visitors.

The fifth and final channel is the 'done' channel. This channel is used by the *EventLoop* to indicate the end of the simulation. This causes the *LocationWorker* to send its final data to the *locationString* function, which collects the data and formats a string for the statistics log. Once done, the *LocationWorker* terminates.

2.2.6 How the infection algorithm works

The purpose of the *InfectionWorker* is to check whether the infection status of any visitor should be updated during the current time step. The *InfectionWorker* is given the list of all visitors for the parent *LocationWorker*, and checks the ratio of healthy to infected persons. This ratio is then used to calculate what the risk of infection is for the location together with *locationModifier*. The *locationModifier* describes how infectious a location is, for example, an outdoor has a low *locationModifier* but a small store has a high *locationModifier*. Furthermore, the infection spread is regulated by vaccine and mask usage and how effective they are. Diagram 12 shows the spread of the virus with two different sets of parameters, the top ones are when masks and vaccines are used at 40% and the lowest is with no masks or vaccine use.

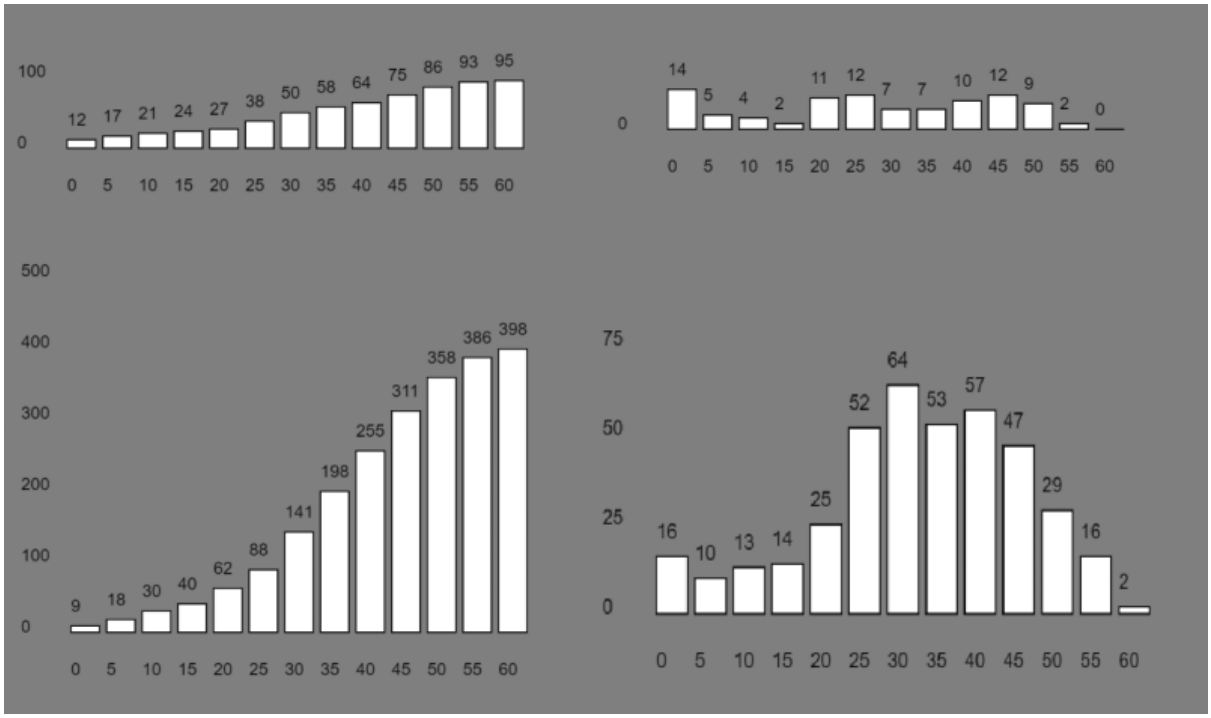

Diagram 12: Results after running Go virus, GO! with different parameters.

2.2.7 How persons are moved to other locations

As mentioned in 2.2.5, persons are sent from one location to another goroutine is to check which persons, if any, are scheduled to move on to a different location at a given time step. If the person is scheduled to move on, *LocationSender* sends this information to the UI communication goroutine. If the person has finished travelling altogether, it sends this information both to the UI communication goroutine and to the person statistics goroutine,

by way of the *personString* function (which formats a string containing the appropriate information). Once it has checked all persons that were given to it, it sends those that did not move on back to its parent *LocationWorker* and terminates.

2.2.8 EventLoop and synchronization

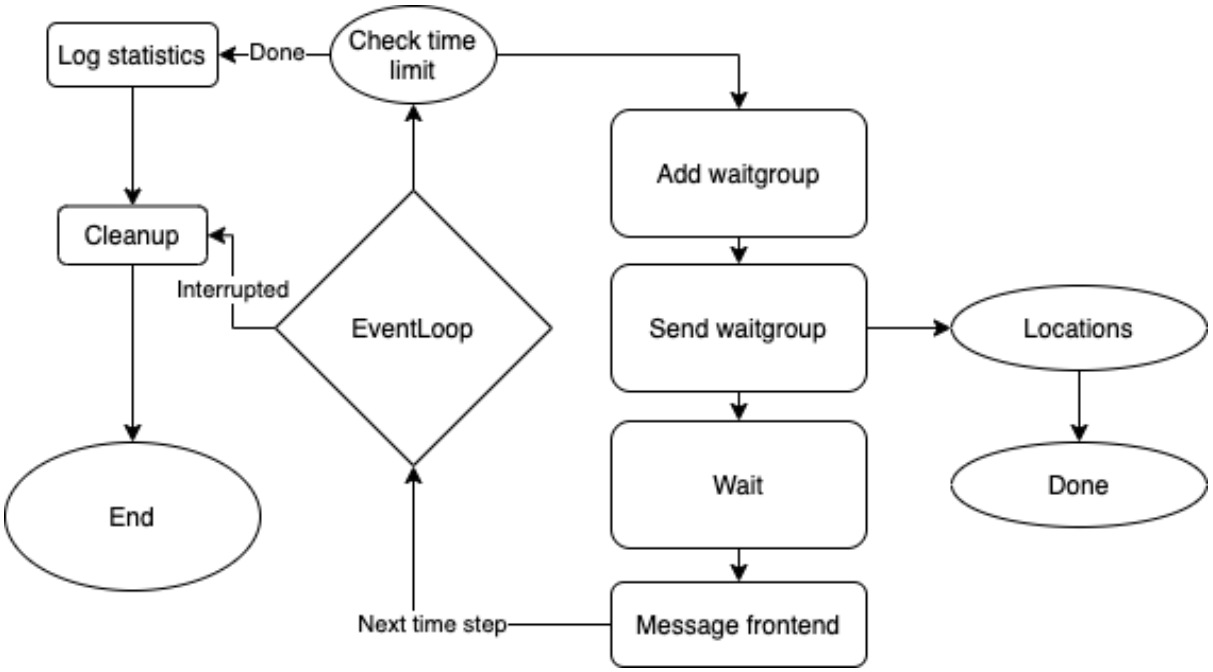

Diagram 13: Eventloop flowchart

The *Eventloop* function is responsible for making sure the simulation runs properly, meaning it is the function which keeps track of the time, and makes sure that all necessary operations have been completed before moving on to the next time (i.e., the next loop).

This primarily includes synchronizing the execution of each *LocationWorker*, which is done through the use of a WaitGroup, a feature of Go's sync library. A WaitGroup is roughly equivalent to a counting semaphore. It contains a value, represented by an integer that can be incremented and decremented by calling the Add and Done methods. The Add method increments the value by an amount equivalent to the argument given to Add, while the Done method decrements the value by one. Since the *EventLoop* needs to wait for all *LocationWorkers* to finish running both the *InfectionWorker* and the *LocationSender*, it will perform Add on the WaitGroup with an argument equal to the number of locations.

A pointer to this WaitGroup is then added to a synch struct, which contains said pointer as well as an integer representing the current time. This struct is then sent to every location through their synch channel, after which the *EventLoop* performs the Wait method on the WaitGroup, which causes it to become blocked until the value in the WaitGroup reaches zero.

As explained in section 2.2.5, a location receiving a synchronization message causes them to begin the process of infecting and moving persons. After this process is complete and the final result is returned to the Locations, each Location will perform the Done method on the WaitGroup, decreasing the value by one each time. Once the value hits zero, the *EventLoop* is unblocked, and can continue execution knowing that all goroutines have finished their necessary steps.

At that point, the *EventLoop* receives information containing which persons have been infected and which have changed location. This information is transmitted to the UI, so that the visuals can be updated to match the new state of the simulation. To allow the UI time to do this, it waits for a message from the UI before continuing, after which it increments the time and moves on to the next loop.

At the beginning of each loop, the *EventLoop* will check if the simulation time limit has been reached. If so it will break out of the loop, perform a cleanup, and then terminate.

2.2.9 How statistics are collected

One important part of the simulation is that it should be possible to analyze the results after the simulation without manually recording what the UI presents. This is achieved by the backend which generates a human readable log file with detailed statistics on what happened during a simulation run.

At the top of the log file the values of the active parameters given to the simulation by the user are shown, followed by statistics about infection spread, locations and persons. All information is gathered during simulation runtime by a goroutine *StatisticsHandler* which is started when the simulation is initiated. The *StatisticsHandler* concurrently listens for messages sent on global statistics channels and the different types of statistics channels are grouped inside of a struct to make it easier to keep track of all channels related to statistics. The different types of messages the *StatisticsHandler* can receive are parameters, infection, location, person, exit and done.

2.2.9.1 Parameter statistics

The parameters are set upon simulation initiation which means it is the first message the *StatisticsHandler* will receive. Once the parameters are set, a function which formats the information into a readable string will be called (see Diagram 14) and send it over the 'parameter' channel. The *StatisticsHandler* receives the string and stores it in a variable.

```
Simulation parameters:
    Number of locations: 25
    Number of persons: 1000
    Infection risk on spawn: 0.010
    Infection spread risk: 0.100
    Simulation time limit: 30
    Mask usage: 0.200
    Mask effectiveness: 0.700
    Vaccine usage: 0.200
    Vaccine effectiveness: 0.900
```

Diagram 14: Parameter statistics.

2.2.9.2 Infection statistics

The infection statistics report which persons got infected at which location at each individual time step, see Diagram 15. For each time step at each location, *InfectionWorker* is called, and a number of persons possibly get infected (explained in section 2.2.6). After each call the result is formatted into a string and sent to the *StatisticsHandler* in a struct along with what location and what time step. The *StatisticsHandler* saves the information in a map in order to keep the information organized.

```
Infection Statistics:
    Time 1
        Location #1 (ICA): No persons were infected.
        Location #2 (Apoteket), the following persons were infected:
            #40
            #68
        Location #3 (Webhallen): No persons were infected.
        Location #4 (Biblioteket): No persons were infected.
        Location #5 (Uppsala Klättercentrum): No persons were infected.
    Time 2
        Location #1 (ICA): No persons were infected.
        Location #2 (Apoteket): No persons were infected.
        Location #3 (Webhallen), the following persons were infected:
            #49
            #78
```

Diagram 15: Infection statistics.

2.2.9.3 Location statistics

Location statistics show how many persons have visited and how many have been infected at each location (see Diagram 16). All locations contain this information in their struct, therefore the information needs to only be sent once a location will no longer receive any more persons, which is when the simulation is done. When the *StatisticsHandler* receives the message Done (see more in section 2.2.9.6), it starts to listen for location messages. It stores the location information in a map to later be able to make a formatted string where all locations are in the correct order.

```
Location Statistics:
    Location #1 (ICA MAXI):
        Total persons visited: 622
        Total persons infected: 26
    Location #2 (Apoteket):
        Total persons visited: 622
        Total persons infected: 61
    Location #3 (Webhallen):
        Total persons visited: 568
        Total persons infected: 32
    Location #4 (Biblioteket):
        Total persons visited: 580
        Total persons infected: 10
```

Diagram 16: Location statistics.

2.2.9.4 Person statistics

Person statistics shows all the information about each specific person (see Diagram 17). As in location statistics, all information about a person is saved in the struct, therefore the formatted string for each person can be made once a person has completed their path (see more at 2.2.3). The *StatisticsHandler* receives the information together with the persons ID and stores it in a map to later sort the persons in the right order.

```
Person Statistics:
    Person #1:
        Was infected on spawn.
        They started at location #4 (Biblioteket),
        then moved to location #2 (Apoteket) at time 2,
        then to location #10 (Lutis) at time 3,
        then to location #6 (Clas Ohlson) at time 4,
        then to location #9 (Ofvandahls) at time 5,
        then to location #3 (Webhallen) at time 6,
        then to location #7 (Pollax) at time 7,
        then to location #11 (Folkes) at time 9,
        then to location #8 (Palermo) at time 11,
        then to location #5 (Uppsala Klättercentrum) at time 12,
        then to location #1 (ICA) at time 13,
        then to location #12 (ICA Väst) at time 15,
        which was their final destination.
```

Diagram 17: Person statistics.

2.2.9.5 Exit

If the simulation is interrupted during runtime the *StatisticsHandler* will receive a message on the ’exit’ channel which will terminate the goroutine.

2.2.9.6 Done

Once a message on the ’done’ channel is received it means the simulation has ended and it is time to write all statistics to the log file. However, first all information about the locations must be received (see more at 2.2.9.3). All maps must be iterated to format ordered strings and then the different strings are concatenated. A file log.txt is created with the concatenated string as its content and that is the result of the *statisticsHandler*.

3. Testing the system

In order to test the system, Go’s own test library has been used. Functions that return or send values through channels are easy to test, these often include small help functions. Tests for functions not returning any value are more complicated and are often implemented using channels and other functions.

Testing in Go starts by creating a test file where all the tests will end up. If the file being tested is called ’backend.go’, for example, the test file needs to be named

’backend_test.go’. In this test file Go’s own testing library needs to be imported. This is all that is needed for testing in Go and the test implementation can be started.

All tests are implemented in a way that checks if a condition is true or not. If the condition does not hold, the test fails. There can be multiple checks in a single test.

After all tests have run, a summary is printed in the terminal which gives a clear overview over which tests failed and why. A smart way to test different inputs for the same function is by using for loops in the tests. A list of structs containing inputs and corresponding outputs is iterated in a for loop. By doing this the amount of code is reduced significantly. If any of the outputs do not match the expected output, the test fails and an error message is printed.

3.1 Test implementation

In the system, many functions send messages through a channel that they receive as an argument. By keeping track of that channel, we can check that everything is executing as planned. For example, some functions send messages through the channels they receive as an argument. By calling that function from the test, the function will send back messages through the channel that was created in the test, instead of sending it back through its original channel.

Testing functions that return random values proved to be a challenge. One way this was tackled was by using average and expected values. By calling the same function many times a certain value could be expected, based on the infection algorithm. These tests calculate a delta value by taking the absolute value of the subtraction of the average value and expected value. If the delta value does not exceed a specific value the test passes. However, there is a risk, albeit small, that the tests fail because the returned value might fall outside the interval of the expected values. The expected values are chosen depending on how precise the test should be. A bigger delta leads to a higher pass-rate, but could potentially hide errors in the system. A smaller delta leads to a lower pass-rate, but will immediately showcase any errors in the system. This has to be taken into consideration while using the tests.

Once unit tests were written, continuous integration was implemented through Github actions. This made all tests run automatically each time a new pull request was made. This makes for quicker, more consistent and more efficient debugging.

4. Discussion

One of the primary design goals of this project was to create a visual simulation that efficiently portrayed the relevant information to the user. Simply put, we wanted our simulation to be easy to digest, in the hopes that this gives the user more insight into how such a spread can occur. Several design choices were made to accomodate this goal:

Infected persons are colored red, a choice of color that makes it clear which persons are infected and which are not, given the common connotations between red and danger [10]. While the color green is perhaps the color most often associated with safety, it was instead decided that healthy persons would be colored blue, to avoid confusion if the user happens to be red-green colorblind. Masked persons have a white border, and vaccinated persons have a white dot in the middle. While it was difficult to find a visual representation of all the combinations of states (infected/not infected, vaccinated/not vaccinated, masked/not masked) that didn't clutter the visuals, the white border/white dot indicators felt like it gave the required information while still making it possible to easily discern different states from one another.

In a further effort to keep the visual impression clean, we decided to represent locations using a simple square on a mono-colored background, rather than something more graphically cluttered like e.g., overlaying the locations on a map, which was an option we considered given that we named the locations after places in Uppsala. While naming the locations does add some noise to the design, it provides a more intuitive way of keeping locations distinct than simply numbering them would. It also felt significantly better than having no labels at all.

In addition to the visual simulation, there are real-time statistics displayed to the side. These were added to enable the user to get a more accurate picture of the current state of the simulation. Despite efforts to make the visual simulation easy to grasp, actually counting the number of infected vs uninfected persons at a given location is virtually impossible, especially when the simulation is run at larger scales (i.e. thousands, or tens of thousands, of persons). The choice of what statistics to display here was not obvious. The aim was to give the user as much information as possible, without giving the user so much information that it would confuse and distract rather than inform. Hopefully, we came reasonably close to striking this balance.

Finally, another design choice made to give the user easily digestible information was the creation of the log. This is not a part of the visual simulation, and thus does not help the user. However, given the significant amount of data that the simulation churns through, it

felt appropriate to give the user an optional way of digesting this information, should they be interested in doing so. The log should be considered a supplement to the visual simulation, rather than an integral part of the experience. Here, no effort was made to avoid overwhelming the reader with information, since the expectation was that if a user seeks out the log, overwhelming amounts of information is what they are looking for.

One last feature to mention when it comes to user-friendliness is the Help feature, clearly visible and easily accessible in the application. This part of the system contains detailed instructions on how to use the program and read its output. A helpful point to start from if you are a new user.

A significant challenge encountered during development was how to effectively test the system. In concurrent programs, especially programs written in Go where the programmer does not need to use manual locks, it might be harder to know what is executing and at what time. A common way to debug a program is to use breakpoints to pause execution, and from there check that everything executes as planned. Unfortunately, this is not as straightforward in concurrent systems, since the program might encounter a breakpoint in one goroutine, but from that thread you cannot access the variables of the other concurrent parts, nor know where they paused their execution. Functions that use concurrency in this program usually do not return anything. This makes it harder to test their functionality. Additionally the system has a lot of randomness which further complicates testing capabilities. This was discussed in more detail in section 3.1.

One of the limitations of the system is that it is not a simulation based on real-world data. In other words, the formula for the infection algorithm is based on our intuition and driven more by the goal of achieving a visually pleasing result than the goal of providing a realistic simulation of a particular virus in a particular setting.

It was difficult to arrive at a formula that was not either overly aggressive (e.g. going from 0.1% infection rate to 50% infection rate in just a few ticks) or passive to the point that it barely registered. Many different alternatives were considered before arriving at the final formula. In the first variation, everyone got infected within just a few ticks, and in the second version of the algorithm, the infection depended on how many persons were in the simulation and changed drastically if the user changed the number of persons.

5. Conclusions

The goal of this project was to create a concurrent simulation of a virus spreading through a population, coupled with a visual representation of the simulation. More specifically, the goal was to create a simulation that enabled the user to grasp the end result in an easy and intuitive way.

Compared to the initial goal, the project group is satisfied with the end result. The application is visually clean, easy to use with clear instructions, and provides an easy way to observe the spread of an imagined virus in a population, with the corresponding changes in scale and intensity as parameters are adjusted. Despite the project's limitations, i.e. the fact that the virus simulated does not represent a real virus, we believe that it might provide some entertainment value to an interested user. A possible use-case for this application could perhaps be as an educational tool in the lower grades of primary school, for example in a biology or a mathematics class.

6. Future work

The possibilities to expand this project are abundant. More parameters could be added, especially parameters related to virus spread and interpersonal interaction. Locations could be expanded to be actual "physical" locations where persons move around (not just visually in the GUI), and the infection algorithm could be changed so that persons infect each other upon physical contact. The population could be more "individualized", i.e. you could give each person attributes such as age, interests, etc., and these parameters could influence what locations a person visits as well as their susceptibility to the virus.

A more visually pleasing GUI could be implemented, with a more cartoon-like (or more realistic) approach. A day/night cycle could be implemented, and individual behavior be given to persons. Some could go home at night and reappear the next day, while others come out later in the day and stay out late in the evening, instead of the current system where all persons spawn at the beginning of the simulation and leave at roughly the same time.

7. Reference List

[1] Kahneman, Daniel and Lavallo, Dan. “Delusions of Success: How Optimism Undermines Executives’ Decisions.” Harvard Business Review, July 1, 2003. https://hbr.org/2003/07/delusions-of-success-how-optimism-undermines-executives-decisions. Retrieved May 11, 2021.

[2] Still, G. Keith. “Crowd Science and Crowd Counting.” Impact 2019, no. 1 (January 2, 2019): 19–23. https://doi.org/10.1080/2058802X.2019.1594138.

[3] Plague Inc., Website. https://www.ndemiccreations.com/en/22-plague-inc

[4] Plague Inc., Wikipedia. https://en.wikipedia.org/wiki/Plague_Inc

[5] Gotron (https://github.com/Equanox/gotron). Retrieved May 11, 2021.

[6] Electron.js (https://www.electronjs.org/). Retrieved May 11, 2021.

[7] P5.js (https://p5js.org/). Retrieved May 11, 2021.

[8] P5.js (https://p5js.org/reference/#/p5.Vector/lerp). Retrieved May 11, 2021.

[9] Broberg, Niklas. 2016 “Model – View – Controller.” https://www.cse.chalmers.se/edu/year/2017/course/DIT952/slides/6-1a%20-%20Model-View-Controller.pdf. Retrieved May 11 2021.

[10] Tham, Diana Su Yun, Paul T. Sowden, Alexandra Grandison, Anna Franklin, Anna Kai Win Lee, Michelle Ng, Juhyun Park, Weiguo Pang, and Jingwen Zhao. “A Systematic Investigation of Conceptual Color Associations.” Journal of Experimental Psychology: General 149, no. 7 (July 2020): 1311–32. https://doi.org/10.1037/xge0000703.

Appendix – Team Reflection

Our general impression

Since the project was supposed to be focused on concurrency we picked a good idea and also a good language to implement it in. Our system has a lot of different parts that have the opportunity to operate concurrently but there is also a lot of need for communication between these parts which is a good challenge for learning more on how to write a good concurrent program. And because we chose Go which is a very good language for communicating between concurrent tasks we think it was overall a good project.

What did you learn?

As Go was a new language for everyone we all obviously learned a lot about the language, such as syntax, how to organize files into packages and modules and especially how concurrency works with channels, wait groups, select-statements and so forth.

We had some earlier experience of forking a process and doing minor concurrent programs before, but we hadn't built a big system like this to be concurrent, and we learned a lot about how to set up the system architecture and decide which parts of the system should operate concurrently and which shouldn't.´

In the courses we have had before we have only worked in smaller groups, thus it was an educative and new experience to work in a group of 8 members. When there are many members we learned that there is a bigger need for dividing tasks so that everyone has something to do. A very helpful thing for this was the project board and issues on Github.

The Github workflow has been a nice learning experience. Not that we were completely new to it, but there were still some new aspects, such as linking issues to pull requests, and using github as both a trello board and a code repository at the same time. Rebasing was also something that only one member had even attempted before, but that now several members have done multiple times and feel reasonably confident with.

In terms of the strictly technical part of the project, something that has been completely new to several members of the group was the combination of several programming languages in one project, as well as the combination of frontend and backend. Using multiple languages was a design goal for our project, which we ended up fulfilling by writing the backend in Go and the frontend in HTML, CSS and Javascript.

In the past, projects have been handed to us with clear instructions on what to do. We all came up with several project ideas and then voted on what to make as a project. After deciding on the project and some basics (e.g. that we wanted to use Go), we had to start making a plan on how everything was supposed to be put together. Here we got stuck for quite some time, since we had difficulties finding a good way of creating graphics in Go, or in another language and integrating it with Go. After testing several options (creating graphics in C++ that we could then run through Go, using Qt bindings for Go, and the Ebiten go graphics library) we eventually found the possibility of using Gotron, and therefore the ability to make the graphics the same as a normal website. While this was not optimal from a graphics performance standpoint, it made things significantly easier and allowed us to create the system with minimal friction. It also separated entirely the frontend from the backend which meant that we could have changed the backend language if we disliked Go for some reason without redoing much of the frontend.

What has worked well?

As all of us who have worked on a group project know, communication can break or make a project. In this case, the communication has flowed nicely and has been up to par when it comes to both level and clarity. The tool used for communication during the project was Discord, and a channel was set up at the very start to make it possible to set up different audio and text channels for different types of work. As most work was done in pair programming, the audio channels in Discord were ideal to use as medium. Different types of text channels have been used to keep the communication organized, to ensure that all information was easy to find when needed.

The group set up clear rules from the start and respected them. There have been daily meetings and eventual absences were announced on Discord. There has been dialogue and discussion and it was possible for people to express their opinion.

Following the group contract was not hard. When we wrote it everyone agreed on what was in it so that meant that everyone knew what to do. It also wasn't particularly restrictive so you didn't have to work hard to follow it. One thing that was missed from the group contract during this project was taking notes on our meeting for those who weren't able to participate. This didn't cause that much of a problem, usually all of the team members were at all meetings, and if someone wasn't there it was easy to just tell that person the next day what had been said.

Something we changed from the original group contract was to switch from bi-daily meetings to daily meetings. This was a positive change that made it easier to work continuously on the project and have better communication throughout the two months.

As for the actual coding, we have generally worked together in the same file using collaborative programming tools. This worked well and meant that we could work together on the same code similarly to how we did during the IOOPM course.

For each new feature of the program the main idea was to create a new branch to implement it on and then create a pull request to merge it into main. It was decided that in order to merge a branch into the main branch, at least two other members of the group would have to approve the pull request. This has been an effective safety mechanism that has prevented disastrous changes being merged into the main branch, while also making sure that at all times at least 3 group members have been aware of any changes.

What was more challenging than expected?

1. Testing

One goal for the tests from the start was to test that the system executed concurrently, which became more difficult than expected. Finding a way to prove that every process executes concurrently was later not a priority. Tests for functions not returning anything, or functions depending on random inputs/outputs was later realized to be a challenge as well.

2. Realistic infection algorithm.

For people to be infected during the simulation, there was a need for an algorithm that decided which people would become infected. Finding an algorithm that was somewhat realistic and did not give exaggerated results was difficult. We went through many different versions until we settled for the one we have now. It was a process that started after milestone 2 and was finished right before milestone 4.

3. UI challenge

When developing the UI we had to decide what frameworks we were going to use. This was challenging because we had a lot of false starts with different ideas. For example we started with trying C++ but it was hard to run from GO and then we tried ebiten which worked in go (most of the time) but was extremely cumbersome to develop for. After trying and then discarding QT we finally found Gotron which just worked and we stayed with that but it meant the real UI development started later than the backend and a lot of development effort went into things that are not in the end product.

4. Presentations

Fitting all appropriate information into the presentation time limits has been hard. Since the time limit for the presentations was 25 minutes but in our test presentations we would easily pass 30 minutes it was hard to remove things from the presentation and still be understandable, especially when some of the feedback we received was to add more (technical) information.

5. Report writing

Writing these types of technical reports isn't something that a lot of us have done much of before. There was some confusion on what to write where, and how to make the report come together in a nice way. We wanted the report to follow a natural flow, and make the reader feel like the section he/she was reading made a nice transition to the next section. We had some disagreements about what and where some things in the report were supposed to be, but managed to find a middle ground that everyone was satisfied with.

What would you do differently if you were to start over?

In the beginning of the project we were very indecisive regarding what programming languages to use. We wanted to write the project in a language that we didn't have much experience in, as we saw it as a great learning opportunity. We knew we also wanted to do the project in two different languages, so we had to find two that worked well together. There were several options to choose from, but the ones we tried initially didn't work very well, or not at all. This indecisiveness probably cost us a few working days, as if we had put in a bigger effort to look at different options right away, we could have also had a better idea of our options faster and made a quicker decision, instead of durdling around because the first things we tried weren't perfect.

Github has been used both as a planning tool and a tool to share the code between group members. However, we did not use github to its full potential, which made our collaboration a bit less effective than it could have been. We should have been more rigorous about creating Issues when problems came up, and better about assigning (or self-assigning) issues to people when they didn't have much to do. Failure to do this led to much of the work being done by the same people, i.e. the ones that discovered a problem, rather than being more evenly divided. This also meant that some features were delayed or derailed since as problems were discovered, we focused on fixing those problems, rather than making issues for the problems and focusing on implementing the thing we were originally supposed to.