# Final report: Online Minesweeper

# System goals

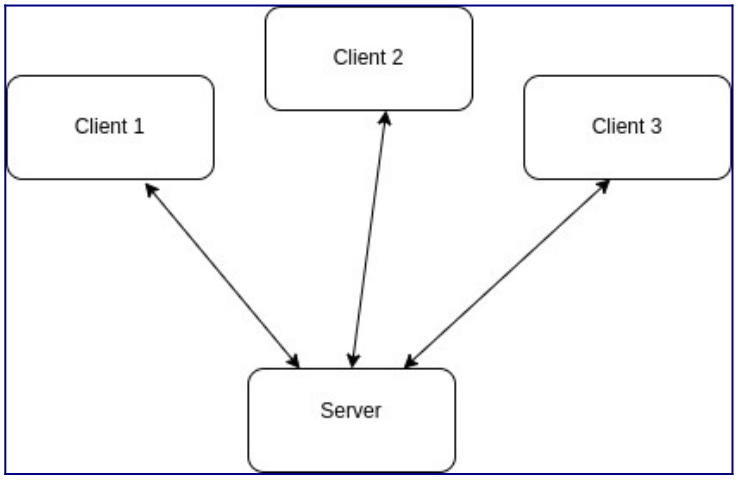
The purpose is to provide a minesweeper game that can be played collaboratively online via an internet connection. The game should have all the staple minesweeper features, such as appropriate game logic, flags and game customization.

One important goal is that everything in the game state syncs properly between clients. Clicks in the game grid should be registered and update the game state accordingly. Likewise, starting a new game should apply properly for all clients.

# Architecture

### Basic system logic

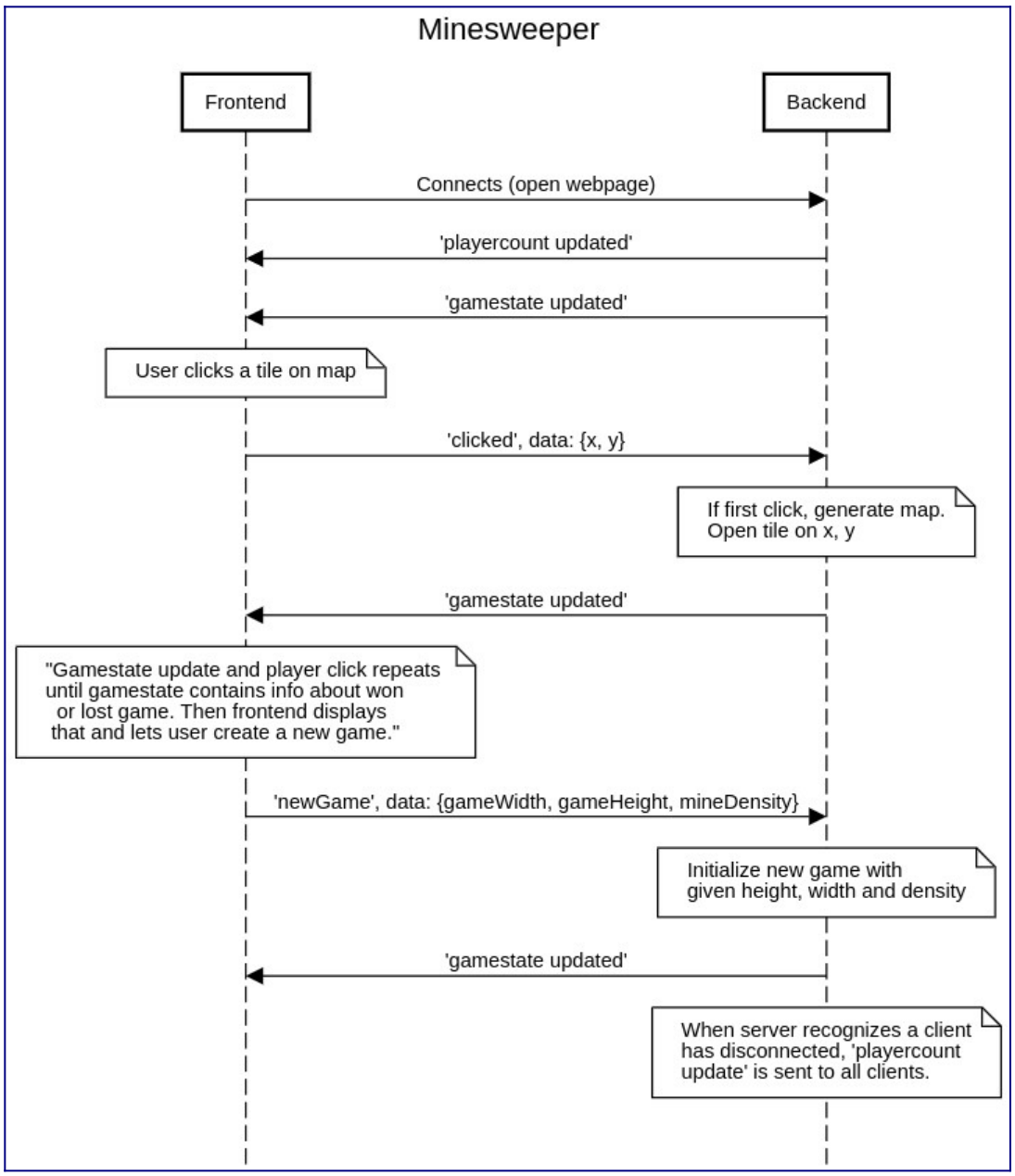
The server contains the game logic and serves many clients. When one client makes a click that causes server to update game state, the update is then sent to all clients at once. This keeps the game synchronized between clients.




### Game logic and communication between frontend and backend

Game logic happens in a NodeJS Express backend. The game is rendered on a HTML/JavaScript frontend. The Express server first serves the game.html file, which contains JavaScript code, which then creates a websocket connection to server using socket.io.

The following diagram explains how the basic flow of logic happens with socket.io. Notice that because there can be many frontends connected to the same backend, all the messages sent by backend to client are sent to **all clients** at once. The following sequence diagram simply shows one frontend.




Some notices of the sequence diagram above:

- When server receives click and runs the game logic, it may happen that the click is on a tile that is already opened before, perhaps due to a click by some other client. In this case nothing happens, no gamestate update is sent to clients, since it has been sent when the previous successful click was processed.
- Server initializes map size when new map is generated, but mine locations are randomed only after the first click is made, to make sure that the first click cannot be a mine.

# Modules and components

As described in the architecture-document, the main modules of the system are client (frontend), which is a webpage, and a backend, which is a node/express server. The server serves the webpage via http, and the client then creates a websocket connection to the server, using socket.io JavaScript library.

The interface exposed between the modules are the functions that are listening for specific messages via websocket. These messages are identified by a string and may hold data, architecture-section has a sequence diagram explaining the messaging.

# Express

Express is a widely used NodeJS-based webserver library. In our project, it simply serves the game.html file to the user. The project as it is could quite simply be written without express, so strictly speaking express is not necessary, but it simplifies the code and offers a lot of potential for further development, for example in the case that user authentication is added.

# Socket.io

Socket.io is a JavaScript library used for connecting client and server via websocket.

It simplifies the websocket connection and communication process. Both frontend and backend use it.

# WebSocket

WebSocket is an application layer protocol supported by modern browsers. It provides a communication channel through a single TCP connection. It allows the server to push data to the client, as opposed to HTTP which only works on request-response basis. Therefore the client does not have to poll the server for updates, which would create unnecessary network traffic and delay of updates.

# Pros and cons

The Express and Socket.io libraries do not have many downsides, at least at the current scale of the project. They serve to simplify writing the code.

Because the backend is built on NodeJS, it is single-threaded. If the project were to be scaled to handle larger amounts of players or several simultaneous game sessions, this could become a problem. However, worker threads are possible, which allows parallel JavaScript execution.

The chosen system architecture where the game logic is ran on the backend, has some pros and cons which are evaluated in the architecture-section.

## Which of the fallacies of the distributed system does your system violate, and how?

“Latency is zero”. Our program violates this one because it assumes that all players have similiar latency which could couse syncronization problems for players who have high latency. Since request are processed with first-come-first-served basis this latency issue should not have any impact on the program itself.

## What needs to be added to your system be used to be integrated/extended by another system?

Because the project has a separate frontend and gamelogic backend, integration to other systems should be relatively straight forward. For example lets say that someone is developing a game site which has multiple games on it. If these games are hosted separately then adding our backend to their frontend would be relatively straight forward and the same could be said about our frontend.

# Evaluation

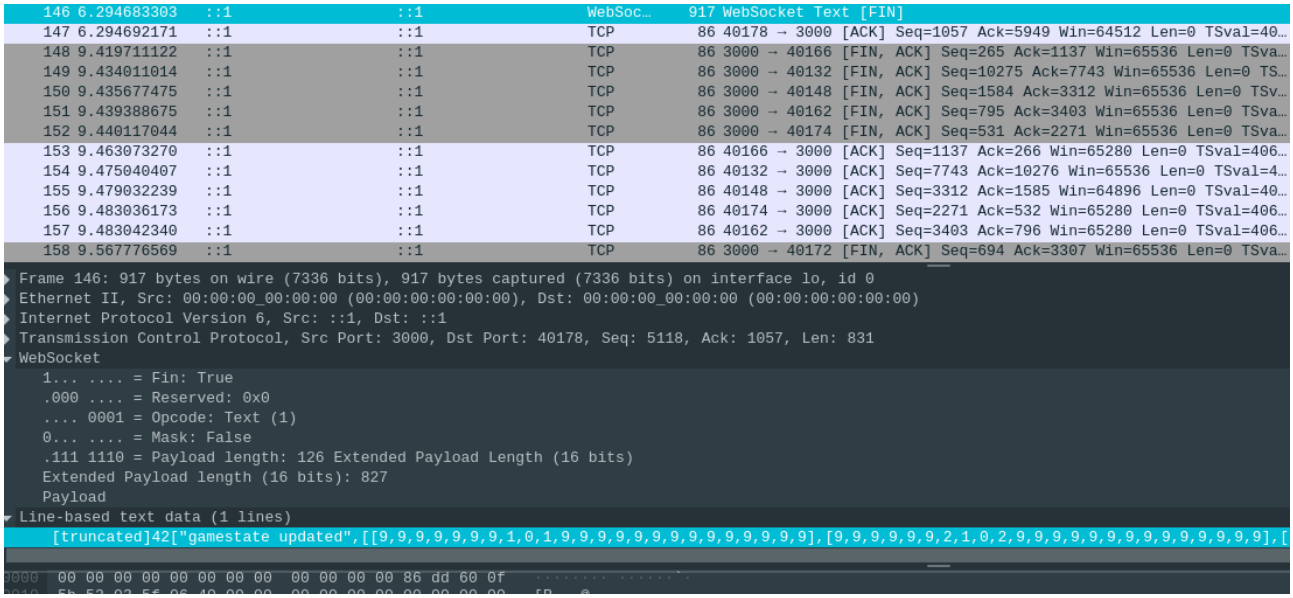
## Latency

The latency of the system is dependent of the users own internet connection, through their ISP to the server where the game backend is running. When ran locally, the latency is unnoticable. The game logic itself will take only a few milliseconds to run, when viewing the time log via wireshark between a “clicked” and “gamestate updated” packets.


In the above image from wireshark, the packet number 505 is a packet notifying server about player clicking a tile, and packet 506 is a packet updating the gamestate to the client, taking only ~3 milliseconds to run the game logic.

With modern internet connections, the game should basically always be playable, if the internet connection is working properly from the client to the server. The game is not particularly vulnerable to latencies in any case, because the game is not based on reactivity and fast action.

## Network load




In the above picture, we can see a gamestate update being sent from server to client. With the default map size the update packet is in total 917 bytes, so just below a kilobyte. This is completely feasible for modern internet connection, but on the other hand, the size could be compressed to much smaller. The transferred map is currently simply a javascript array which is transferred as a string. Websocket transfers text in UTF-8 format. This data could be represented in some format that takes less space, if the system would be scaled for big number of users.

# Evaluating the architecture

We have decided that the game logic runs on backend. This way the game can be synchronized simply, as it runs on the server and the clients only render the game and inform server of clicks that user has made. Another important reason is that the secret game map, containing the mine locations, does not have to be sent to the clients.

Alternative to this architecture could be, for example, one where the server functions only to inform the other clients of the clicks that a client has made. Client would send its click to the server, which then would send the click to all clients, including the one that made the click. A client would only run the game logic and re-render map after it has received the click from the server, even if it made the click by itself. This would keep the game in sync between clients, and it would remove the game logic work from the server. If the game was modified to have many sessions, running the logic on clients would remove the workload from the server. Another benefit would be reduced network load, since the system would only have to send simple coordinates of x and y of the click, instead of the current method, where the entire map array containing thousands of cells has to be sent on every gamestate update.

However, the alternative method would require the full map including mine locations to be sent to clients. This would make it possible for the user to view the mine locations and cheat. In a simple game like this, the possibility of cheating probably would not matter. However, it is important to note that when designing games like this in general, the system design has to take into account that any data sent to the client can possibly be viewed by the user.

# Future work

Currently there is only one game session and generating another one would require a new server. Implementing multi-session hosting in one server could be a good future goal. It could work by automatically assigning people into game sessions, or letting people create their own games, and invite people via link for example. This would probably require implementing worker threads in NodeJS, so that one sessions game logic would not impede other game messages, creating latency for other gamers.