# Networked Game: “Fences”

**Lars Brestrich and Ivor Walker**

## 1. Overview

This submission achieves all basic and intermediate requirements. From the advanced requirements, it fully implements a maximum number of players and a login/logout system and partially implements different land values.

This report details our implementation of the Fences game, which is based on the original game by the same title (https://classicreload.com/win3x-fences.html). The game is played by two or more players via the internet using a server. In the game, players take turns placing fences on a 2D board, trying to enclose tiles for points and keeping their opponents from successfully enclosing tiles. There are special tiles that reward additional points, as well as a random map generator that allows the host to change map size and other attributes.

## 2. Code Architecture

The code is split into the following areas, each with their own classes:

**1. Client and UI**

Handle user input and user interface representation of menus and game.

**Client** – User initialization class. Handles user input in a loop, funnels communication from server messages to the user’s UI and local copy of the game. Has a lot of abstractions for navigating the game grid, selecting cells and menu elements, as well as functions for processing server messages.

**View** – represents menus and game representation. The game elements and scores are represented using player colours.

**UIMenu** – each menu is stored in its own object, which allowed us to easily modify/add new menus and navigate menu elements cleanly. Stores instances of UIElement in a list.

**UIElement** – object for UI elements. Allows us to get its position to navigate easily, set cursor position or highlight elements.

**Enums** - contains all client-side enums (e.g. UserState). Also contains some helper methods for converting curses output (ASCII) to strings and numbers, which helped us make the code more readable.

**2. Game**

Handles game mechanics and stores game state.

**Game** – stores the game grid, keeps track of players and player turns, and defines game parameters. Implements mechanics for placing fences, claiming land and deciding winners.

**Grid** – grid representation of the game board. Made up of cells.

**Cell** – storage structure for each individual grid cell, defining position, type, ownership and value.

**3. Client and Server Connections**

**Connection -** Specifies a protocol for reading and writing over a given socket, and provides methods for sending messages and listening & handling sent messages using this protocol. Requires some methods that need to be implemented by any extension interested in handling messages. Listening is done synchronously, with each Connection's listening loop running on a separate thread. Each message is a JSON containing three parts: a "category", denoting the type of message being passed e.g a login or a placed fence. a "status", indicating whether a message is a "request" (ask the other node to do something, e.g place a fence) or a responding "success" or "error", and a "message", containing the data to be sent.

**ClientConnection -** Client's extension of the above protocol which sends messages to Server and handles messages sent by Server by passing data to the Client registered as a listener. e.g Client wants to place a fence, ClientConnection sends info on this fence to Server, and passes a confirmation of success (or any error) it receives from Server.

**ServerConnection -** Server's extension of the communication protocol which modifies the Server's state and sends ClientConnection an appropriate reply, e.g Server receives a request from ClientConnection to place a fence, Server tries to place a fence on the Server's copy of the game, and sends information on the new fence to all ClientConnections playing this game.

**4. Server, Player & Leaderboard**

**Server -** Target location of ClientConnections. Once a ClientConnection connects to a Server, it instantiates a ServerConnection and passes it the newly opened connection (. Maximum of five concurrent connections due to the synchronous nature of Connection. Stores collections of currently running games and clients, and contains methods to

send messages to given batches of clients, e.g notify all clients in game that game has ended.

**Player** - Represents a single player of Game. Player representations in Client are hollow, only storing a username, whilst Player representations in Server stores their password and win/loss/draw statistics (updated by Leaderboard).

**Leaderboard -** Collection of all Players to have joined Server. Contains methods to add a new Player, search for a Player, and update the scores of given players with a win, draw or loss after a game ends.

## 3. Design Discussion

**1. Game on both ends**

We decided that our clients and server should each run their own copy of the game. This has several reasons: firstly, it allows us to send miniscule messages between server and client and thus save on bandwidth (i.e. to only send changes rather than the entire game state). Secondly, it ensures that the server validates every move the player makes. Thirdly, it is the easiest way to implement it.

On a related note, we also decided to only draw the game on client when it is updated rather than running a loop, which should be great for performance.

**2. Model-View**

We split our client into the Client and View classes, where the Client handles input and server messages, whilst any visual representation happens in the View. This modular approach ensures separation of concerns, which is great for the incremental implementation/changing of features as well as debugging.

**3. Breakdown of Game and UI classes**

We decided to represent the game board in a grid, which is made up of cell instances. The abstraction of these elements into classes allows for very readable code and saved us from having to concern ourselves with game navigation or accessing raw array elements at every level of the hierarchy.

**4. Synchronous versus asynchronous server**

We need a part of the server that constantly listens for new messages over the socket. We chose to do so synchronously (sync) - the entire Connection is blocked until it receives a new message. We used thread-based concurrency to circumvent this limitation, where each Connection's reading loop is in its own thread so it doesn't block the Connection. One disadvantage is that the number of threads we need, thus the computation cost, increases with the number of connections. We needed to add a cap

of five connections per server to avoid the number of threads overwhelming weaker CPUs.

An alternative would be an asynchronous (async) solution, where only the read loop is blocked until a new message is received. This approach removes the need for multithreading, scaling in computational cost far better than our sync solution. This is at the cost of readability, as a sync solution requires less code and is concentrated in the initialisation whereas an async solution would also require more code in the handle_message loop. The need to scale computationally is quite limited in our case, as evaluating the solution is unlikely to require more than five clients, but readability is a priority during integration as both developers need to quickly understand all parts of the code when locating integration bugs.

**5. Extending one protocol for two separate use cases**

I chose to write a central Connection method to handle sockets and got ClientConnection and ServerConnection to inherit from it because both Client and Server required the same communication capabilities and the same function names to interpret messages.

A common source of bugs was the Client and Server differing in what message they needed or expected from the other node, e.g ClientConnection's placeFence expected "success" or "error" from ServerConnection whereas ServerConnection expected a JSON of locations from ClientConnection. I refactored this approach to pass the all JSON attributes of a Message as arguments to the respective function (**message).

## 4. Testing

We mostly tested our application by running it repeatedly under different conditions, much like a game tester would perform a smoke test. We ran through different combinations of inputs at each menu to try to break the system and did the same for the game as well, notifying each other of bugs we encountered. We did catch a few bugs this way and repeated this process after fixing them.

Bugs which occurred solely in a singleplayer component or a multiplayer component were easy to fix in isolation, but we encountered bugs mostly in the integration between these two systems.These bugs were difficult to fix because they could occur anywhere in the program, e.g the program could crash in ClientConnection but the bug could be in Game. In testing, we used a "single-player" mode which connected to an in-memory Server to remove any network-related concerns, and added a "debug" mode to print any changes in a Connection. These measures helped narrow the location of our search for integration bugs.

## 5. Team Working

Before the git repositories were released and we could start work, we held two in-person meetings to high-level design the entire solution together.

One member was unable to immediately start work due to a deadline clash, so we split the work into two sections. The server requires an implemented game, and by creating the client first we had a responsive user to enable testing of our server-client communication. Therefore, the immediately available developer implemented a single-player version of the game (game logic and state) and client (UI input/output). Afterwards, the delayed developer extended this to multiple players by outlining the communication protocol (conn/connection.py), and implementing it for a server (server, conn/server_connection, Leaderboard) and a client (conn/client_connection, Player).

We integrated these sections by pair-programming, and any bugs that emerged became the responsibility of the developer responsible for that section. We communicated via WhatsApp every day to update each other on our progress and held frequent pair-programming in the lab. The git log is not reflective of our individual contributions, as the results of a joint pair programming sessions were committed by one developer and work done by one person outside the lab was sometimes committed onto our git by the other developer (e.g the first server implementation). We wrote the group report together, each providing details on our sections and splitting the rest between us.

## 6. Libraries & Resources

We used socket for synchronous connections over network, and threading to create threads for new listeners.

## 7. Self-Evaluation

The teamwork on this project was very engaging, especially our pair-programming. We each were responsible for our own sections of the program, and both held up our end, delivering a final product that we are happy with.

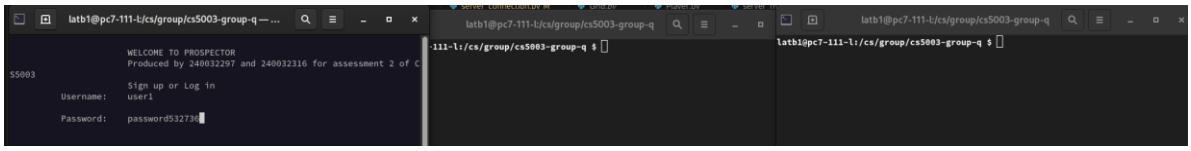
We spent a lot of time ironing out bugs and implementing error checks, which should ensure a smooth player experience. Given most of the bugs happened during integration, we could have written integration tests when designing our program or after completing the first component. These would specify what was expected of the other solution, reducing their likelihood, and pinpoint exactly where integration was failing, further narrowing our search space. However, writing tests would have been too time consuming and integrating through pair programming meant we could communicate expectations better than inferring them from a test.

One thing that could be improved is that the model-view is not entirely separate and could have implemented the listener pattern to fully decouple them from each other. One could also argue that the game view could be prettier, but beauty is in the eye of the beholder.

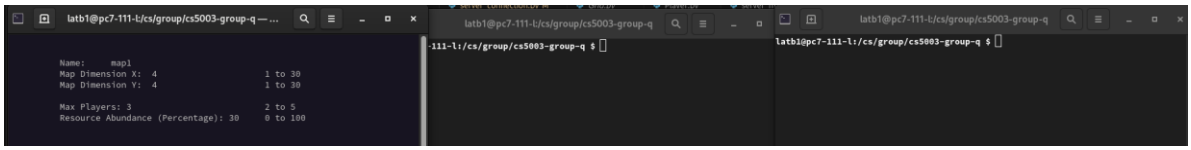
We managed to implement most of the features. If we had more time, we would implement the few additional features.

## 8. Screenshots

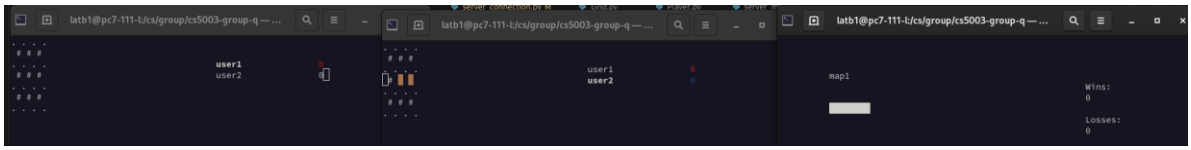
1. Login or register with password.



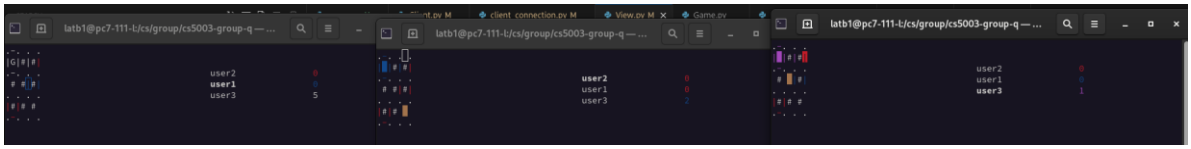

2. Map Creation Screen. Shows possible value ranges and will force the player to stay within these ranges.




3. Left/Middle: Game View, showing map and player scores. Right: active games list on the left, player statistics on the right.




4. Fences and land tiles are shown based on player colours (mostly). Resources are shown in bronze, but unfortunately not synchronized between clients.




5. End game screen. Shows player scores and winner. In this case, it was a draw, so there was no winner.