CS1680 Final Project

Brandon Gong, Patrick Peng - “Generals”

1 Introduction

In this project, we set out to make a clone of the online multiplayer game generals.io. In particular, we wanted to learn how to work with WebSockets – both on the client and server side – to create interactive experiences where multiple people can interact. We successfully built a working mimic of the game by building a backend websocket server, a frontend, and a custom message protocol for communication between server and clients.

2 Design/Implementation

The project may be broadly broken down into 3 parts:

- The backend (C++), a server which all players connect to, which manages queueing players for games as well as running the games (taking player moves and computing the new board state),
- The message protocol, which specifies how the backend and frontend communicate – how moves and boards are encoded, how the user sends their name / receives other player names, how the user finds out if they won/lost, etc., and
- The frontend (TypeScript/React), which collects user input and relays it to the server and renders the board during the game.

2.1 Backend

The central abstraction for representing a game is the `Board` class (`board.h/.cpp`), which manages storing the *board*, computing the new board per tick, and communicates with clients.

A *board* is a 2-dimensional grid of *cells* with a coordinate system defined with the origin at the top-left and growing downward and to the right. That is, the cell at the very top-left has coordinates (0,0), the cell to its right has coordinates (1,0), and the cell below it has coordinates (0,1), and so on.

A *move* – which is always simply a transfer of troops from one cell to another vertically or horizontally adjacent cell – can thus simply be represented as a pair of points under this coordinate system, i.e. $((x_{src}, y_{src}), (x_{dst}, y_{dst}))$.

Cells (`cell.h/.cpp`) represent the basic building blocks of the game, and can be in a variety of different states:

State ID	Name	Description
0	`MOUNTAIN`	Impassable terrain features, cannot be captured by anyone.
1	`VACANT`	Empty cell not occupied by anyone. Does not generate troops.
2	`OWNED`	A cell that is occupied by a player. Generates 1 troop per 50 ticks.
3	`GENERAL`	Players lose if their general is captured. Generates 1 troop per 2 ticks.
4	`UNCAPTURED_CITY`	A city costs between 40-50 soldiers to capture. It does not generate troops while uncaptured.
5	`CAPTURED_CITY`	Once captured, cities generate 1 troop per 2 ticks for the owner.

Because per-state tick behavior for cells depends on the state the cell is currently in, the `Cell` class provides a `Cell::tick` method to decouple this logic from the `Board`. However, the `Board` manages transitions in the Cell state machine according to player moves as the game progresses.

A `Board` may be flexibly initialized using a `board_config` struct, which specifies the dimensions of the board and the locations of the generals. Based on this, locations of mountains and cities are randomly generated.

The `Board` waits to begin the game until enough players have joined, assigning each waiting player an ID. Once all players have joined, a thread is spawned which manages updating the board per tick and sending out this updated board to players. Meanwhile, the main thread continues to receive messages from players, responding appropriately by e.g. enqueuing moves.

Each player has their own queue of moves, and since there are two threads that operate on each queue (the main thread enqueues moves, and may occasionally clear the queue, while the board thread pops available moves), we define a class called `SafeQueue` which protects these operations with a mutex.

We handle multiple lobbies by checking after each new connection if a `Board` is full, and if so, instantiating a new `Board` to handle future connections. Since each instance has its own thread for running the games, they may all run in parallel without conflict.

2.2 Message protocol

Messaging between the backend and the frontend is achieved with WebSockets. Because our goal was not to implement the WebSocket protocol (which is extremely complicated), but rather to learn how to use it in the context of building applications, we elected to use a third-party library µWebSockets instead on the backend. On the frontend, we use the standard WebSockets API without any additional libraries.

Cells are encoded in one 16-bit short (in network byte order): the 3 highest order bits record the cell state (see the table in Section 2.1), the next 3 hold the current owner of the cell, and the remaining 10 bits hold the number of troops in that cell. Note that the owner field is only relevant if cell state is `OWNED`, `GENERAL`, or `CAPTURED_CITY`, and the last field is only relevant in those cell states plust `UNCAPTURED_CITY`. In other states, these fields are irrelevant and only there for the sake of consistency and ease of decoding.

A board is simply one byte for number of rows, followed by one byte for number of columns, followed by its encoded cells, in the order of all cells in first row from left to right, followed by all cells in the second row, etc.

With this encoding in hand, we can define our message formats from server to client and vice versa. All messages start with a 1-byte tag which determines the message type.

2.2.1 Server-to-client messages

- Game start:

Length (bytes)	Name	Notes
1	Tag	Value always 0.
1	Player ID	The ID of the player this message is sent to.
1	Num players	The number of players, n.
$32n$	Player names	Player names, stored in 32-byte blocks, null-terminated.
$2 + wh$	Encoded board	Where w and h are dimensions of the board.

- Game end; you lost: just a single byte of value 1.
- Game end; you won: just a single byte of value 2.
- Player connected (used to update number in waiting room):

Length (bytes)	Name	Notes
1	Tag	Value always 3.
1	Total players	The total number of players now connected.

- Player disconnected:

Length (bytes)	Name	Notes
1	Tag	Value always 4.
1	Player ID	The ID of the player that was disconnected.

- Per-tick update:

Length (bytes)	Name	Notes
1	Tag	Value always 5.
$2 + wh$	Encoded board	Where w and h are dimensions of the board.

- Name update (sometimes a player name may come in after game starts):

Length (bytes)	Name	Notes
1	Tag	Value always 6.
1	Player ID	The ID of the player whose name to update.
32	Name	Their new name.

2.2.2 Client-to-server messages

- Enqueue move:

Length (bytes)	Name	Notes
1	Tag	Value always 0.
1	Initial x	
1	Initial y	
1	Final x	
1	Final y	

- Clear move queue: just a single byte of value 1.

- Send name:

Length (bytes)	Name	Notes
1	Tag	Value always 2.
32	Name	The new name.

The chief pieces of code used for communication are the `Board::unicast` and `Board::broadcast` methods on the backend and `useGameServer.ts` on the frontend. Parsing of messages in the backend occurs in `Board::process_message`.

2.3 Frontend

The frontend is implemented with React. A custom hook is implemented to delay initialization of the WebSocket until after the user has input their name, and to avoid spurious reinitializations of the socket on state changes. The frontend also handles cosmetic details such as fog and the leaderboard.

The frontend does no work in terms of game logic, besides prohibiting users from moving outside the bounds. The backend does all of the validation work (ensuring users don't capture mountains, or move squares that they do not own), discarding invalid moves.

3 Discussion/Results

We were able to successfully create a playable clone of Generals. Our version supports up to 7 concurrent players per game and multiple concurrent games. Below are screen captures comparing our implementation (Figure 1) to the actual site (Figure 2). For a live demonstration of the game running, refer to the demo video uploaded to the repository here.

Over the course of building the project, we encountered several challenges that were quite difficult to overcome.

One bug we faced in the backend was due to how µWebSockets is implemented as a single-threaded library. To send messages from sockets from a thread other than the thread that created the socket (in our case, we needed to send messages out from sockets that were accepted by the main thread), we must use the `Loop::defer`, which waits until the socket in the main thread is ready to send our message. This lead to tricky situations associated with deferring messages for

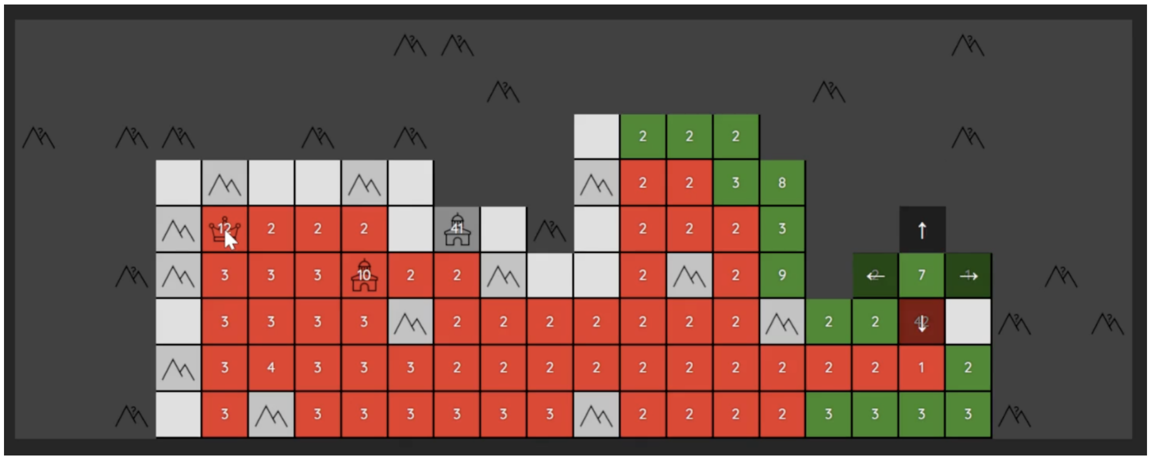

Figure 1: Our implementation. Note for practical purposes, we made our main testing board a smaller board for two players, but our implementation supports more players and a configurable board size.

sending right before the socket is closed, and we found that game end messages we sent right before closing the socket were lost. Given more time, we may have been able to get around this by having the deferred function set a boolean flag once sending has completed. For now, we simply delay closing the socket.

In the frontend, a particularly difficult bug we faced arose from linking state-updating functions as callbacks triggered when certain messages are received through the WebSocket; in particular, we used to send a game start message from the backend and without delay send a game update message from the backend. This resulted in two callbacks on the frontend triggering very shortly one after the other, both with state updates; the second callback would be called so soon that the state update had not fully taken place yet, so when the second callback executed its state change, it overwrote its cached old values into the state. This was resolved by adding a natural one tick delay before sending the first game update, but still begs the question on how to handle rapid messages while avoiding such race conditions.

4 Conclusions/Future work

In working on this project, we learned how to build clients and servers that use WebSockets to communicate, and how WebSockets are very natural extensions of the TCP sockets we implemented in this course. We are very happy with how the project turned out, as it is completely usable in its current state, being the first live interactive web app either of us have made.

If we could continue working on this project, there are many different directions we could pursue. The first would be to add new message formats that enable e.g. chats between users or splitting troops for more complex strategies. We would also move fog of war computations into the backend, since in the frontend the data is easily exploitable by users who wish to cheat. Finally, a major iteration would be to explore replacing per-game ticks with UDP (WebRTC) instead of TCP (WebSocket), and see how that affects user experience particularly over unstable connections.

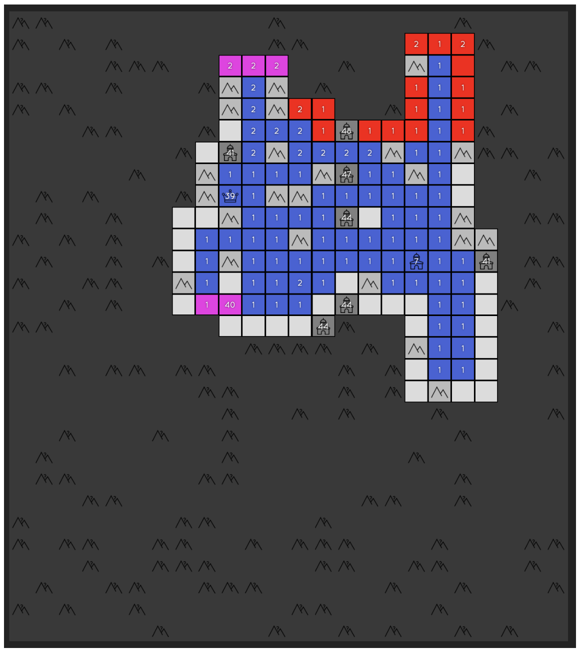
Figure 2: The reference implementation.