Reliability

Reliability will be consistant across games, and when errors occur they will be affect too much of the game. The spec does not include accoutns or complicated state keeping. Errors are most likely to occur during the playing of a game, when that happens it will be easy to restart a lobby and create a new game. I don’t see the process crashing too often and the service will be very reliable.

Performance

The architecture is optimized for a low amout of users. It wasn’t planned to scale for millions of users, I expect that it can handle a few thousand before we start seeing a lot of ram usage by the different lobbies being run at the same time.

Cost

We will be hosting this on a vps that will is faily cheep. For our use case, we won’t be hosting a lot of users, so not a lot of compute power is needed.

Compatibility

The main way to access the games will be through the browser, so it is compatable with most devices. The back-end should run on windows and linux server types, but we will be hosting it on a linux server and will optimized for that.

Deployment

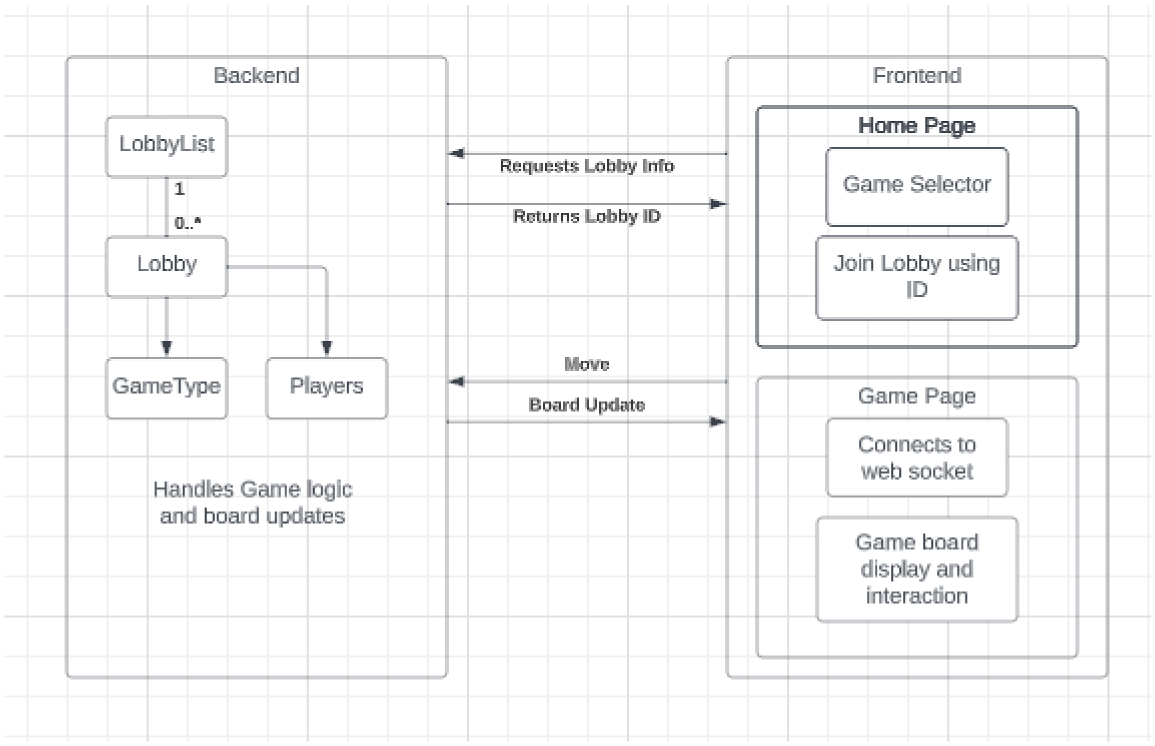
To deploy, you need the most recent version of go and either node or bun. You will also need a server that can host the appllication and a way to serve it. First ou build the front-end with node or bun, using bun build or npm buil. Place the built application so that it can be served by url/ and url/:gameString. The server is run by running ‘go build.’ This compiles a binary that you can run and is served on 8080. Make sure the desployment server has the proxy server set up so calls to url/api route to the backend.

Difficulties

The way that our architecture is designed, is that once we have the core way of making the connections and sending messages, that the games will be easy to insert as we go along. The core messaging part is already finished. Each game is its own difficulty because they are separate entities, we working on each one sequentially, we have finished tictactoe and are starting on checkers and other games.

The other hardest part of the design that will take some effort is the chat feature. This will be the first feature that is dropped if we run out of time. This is not a core feature to our application, but we plan on focusing on it when we get to the second demo.

Connection Chart



App Architecture

```
packages:
- "fmt" -- allows for printing to stdout and
- "net/http" -- go's base http library
- "github.com/gin-gonic/gin" -- http package, makes it easier to organize and send
information
- "gopkg.in/olahol/melody.v1" -- websocket package
- "encoding/json" -- allows for converting structs to and from json


# Http endpoints

## GET /
- gets the index.html, or endpoint that gets the homepage

## GET /:game
- generates lobby id
- creates a Lobby for that game type and puts it into the LobbyList
```

- returns a lobby id

### Get /:game?lobbyId={lobbyId}
- check if there is a query
- returns the starting board state for the selected game

## GET /ws?lobbyId={lobbyid} (ws://{url})
- checks if the request has a lobbyId, if not it returns error
- check if the lobby is already playing, if so return an error
- else upgrades the http connection to a websocket

## GET /lobby?lobbyId={lobbyId}
- checks is the lobbyId exists
- returns gameType
- else returns "unknown lobbyId"

# Websocket handler functions

## HandleConnect
- gets the lobbyid from the query
- adds user session to the lobby struct from the LobbyList

- creates string channel with buffer size of 2 for the lobby

- if there are enough players for the game to start
  - send a "starting" message to each ws session with their player id (or the index that their session is in the slice)
    - set each session id to this index
    - set each session lobbId to the lobbyId
  - create a go routine, passing in a pointer to the lobby
    - this starts the function that initializes the game, in this instance it will call "checkers" but can be expanded to other games

## HandleMesssage
- get lobby id from session
- change message received from []byte to a string
- send mesage into lobby channel

## HandleDisconnect
- gets lobbyId from session
- check if the lobby exists
  - sends the exit code to through the lobbies channel
  - and deletes the lobby from the lobby map
- when closing the socket, use .CloseWtihMsg to inform the client why it closed

# Game

## GenericGameFunction (*lobby,)
- calls ValidateMessage

- if exit is true, exit the go routine
  - close one or all of the player sockets

- if receives a move
  - checks player id to make sure the correct player is moving
  - check if the move is valid

```
  - makes changes to a board
  - sends the board to the other player/players
    - send the updated board state to everyone in the game?

- runs until someone wins, someone disconnects, or the game is reset

## ValidateMsg (\*Lobby) (Move, bool)

- reads from channel for the input

- check if the quit channel received something
  - if it did return "", true

- checks if there is input from the data channel
  - if true,
  - unmarshall the data
  - return Move, false

- put the above in a loop, checking if there is anything it returns


# Data types

## LobbyList
- params:
  - lobbies: map[string]\*Lobby
  - mutex: sync.Mutex

- functions:
  - GetLobby(string) (Lobby, bool)
    - locks the mutex and returns lobby if exists
    - bool is true if exists, false if not

  - SetLobby(string, Lobby)
    - locks mutex
    - sets the map at string with Lobby

  - RemoveLobby(string)
    - locks mutex
    - removes a lobby from the list

## Lobby
- params
  - GameType: string
  - Players: []\*melody.Session
  - C: chan string
  - Quit: chan struct{}

- functions:
  -

## board
- params
  - board: []int
    - 0 = empty, 1 = red, 2 = black
    - only uses a 1d slice, uses math to find correct indicies
  - row: int
  - column: int
- functions:
  - Set (x int, y int, value int) error
    - sets the piece on the board

  - GetPiece (x int, y int) (int, error)
    - gets a piece from the board

  - GetBoard () []int
    - returns the board

## Move
- params
  - Player: int
  - Reset: bool
  - To: Point
  - From: Point

## Point
- parms
  - X: int
  - Y: int

## BoardUpdate
- params
  - ValidMove: bool
  - PlayerMoveId: int //the person who made the last move
  - PlayerTurn: int //the id of the person who is currently going or able to go
  - isOver: bool
  - Board: []int

- functions:
  - Reset ()
    - sets Opponent to 0
    - makes Board empty
      - the idea is you don't have to recreate this struct everytime, you can make sure that the struct is empty


# Websocket json information
- front-end to back-end
  - Move

- back-end to front-end
  - BoardUpdate
```