NORTHEASTERN UNIVERSITY - 2018

Project Report

Multi-Player Checkers Game

VEERESH MAHASHETTI | KOUSTHUBHA BELUR SHESHASHYEE

3/27/2018

AIM: To create a multi-player CHECKERS game supporting multiple game tables simultaneously

Table of Contents

1. Introduction and Game Description

The aim of this project is to build a multi-player computer game with a web browser interface. The client side logic and user interface is implemented using React.js and JavaScript. The server side computations are carried out using Elixir and Phoenix channels. Keeping the above aspects in mind we have chosen to implement a *Checkers* game.

A Checkers game is a strategic board game played on an 8*8 square board. The game is designed for two players and is turn based. Each player has twelve pawns, one of them has dark pieces and the other has light pieces. The pawns that belong to each player are placed in the first three rows closest to the player. The goal of this game is to remove all the opponent's pawns from the board by following the legitimate moves explained later in the report.

1.1. Features and Game Rules

It's a turn based game where each player gets to play alternatively. Actions by one player affect the game play and strategy of the other player as well. This game supports multiple game tables where players can meet and play; it also separates games being played at multiple tables at the same time. The game differentiates between players and spectators.

A player can lose or win in number of different scenarios. The first player to lose all pawns loses the game. If a player is put it into a state where, the player cannot make a single legitimate move, then that player loses the game. If players are put into a situation where both the players cannot make a single legitimate move, then the player with the most number of kings wins the game. A pawn becomes the king if it makes a touchdown in the opponent's nearest First Square. The result of the game can also be a Draw when the two players have same number of pawns and same number of kings.

The players can move the pawn only if it meets the below described conditions. A move is valid when a pawn is moved one space diagonally forward to an adjacent unoccupied square in. If the adjacent square contains an opponent's pawn, and the square immediately beyond it is empty, the opponent's pawn may be removed from the game by 'jumping' over it. Only the dark squares on the board are used by the pawns. A pawn as a 'King' can move one space diagonally in the forward or backward direction. The 'jumping' behavior also applies to king in the same way as applied to other pawns.

2. User Interface Design

2.1. Board and Pawns

The board is designed to occupy 800 * 800 pixels on the screen. The board consists of 64 equally sized squares arranged symmetrically in eight rows and eight columns. The board is created by writing a component in the React.js called 'Square' component. This component renders 64 squares and also adds attributes to each square. Each square occupies 100 * 100 pixels on the board. The component adds attributes such as 'sqaure-id', 'pawns', 'onclick'. Inside each square component, pawn is added as part of the <div> tag inside the <div> tag containing the square component. The pawn is designed to occupy 80% of the space inside each square component. The styling for each square and pawn is added as part of the class names which is assigned dynamically based on the type of pawn present in the square. The respective CSS for those class-names are defined in 'app.scss' file. *Bootstrap* classes are used to make the UI more colorful and elegant.

2.2. Square Highlighting and Pawn Movement

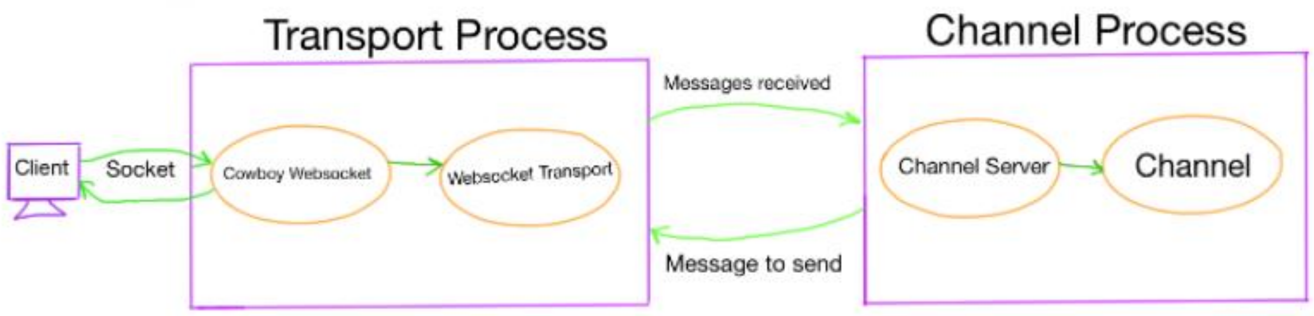
The square highlighting feature is provided to show the players all the valid squares that a pawn can be moved at that point of the game. When a player clicks on the pawn, the valid squared are highlighted giving hint to the players. The borders of the square are highlighted dynamically when by adding an extra styling attribute to the class name. The required border width is updated in the CSS.

Pawn movement is mainly handled as part of the CSS dynamically. If the pawn is to moved from one square to another, the pawn in the former case is made hidden and the pawn in the later case is made visible. The logic regarding the pawn movement and its placement is handles as part of the server side logic.

2.3. Join Game

Whenever the user hits the URL containing the game, screen containing the board is not shown to the user. The initial page consists of two fields which takes user's name and the game room's name as the input. Since this game supports multiple table functionality, it's important for the user to decide the game room to be joined to play the game. So, when the user provides the required inputs, player is taken to the actual game screen, where the screen also displays the game room name, player's name and whether the user is a player or a spectator.

3. User Interface to Server Protocol



The entire game logic is broadly divided into client side and server side, communication between the two happens via UI to server protocols described below.

The above diagram gives an idea of phoenix architecture and how it communicated with the client to process the requests and send the required response. The entire UI to Server protocols consist of the following elements.

3.1. End Points

The end point is created when we run the phoenix web server using the mix.phoenix.server command. The receiver of the HTTP requests becomes the end point and it passes the request to the router. It is mainly responsible for dispatching the requests to the correct router.

3.2. Phoenix.Router

The router receives the request from the endpoints. It is mainly responsible for generating the routes that dispatch to specific controllers and actions. The routers also provide helpers for generating paths to map to a set of resources that we create with controllers/actions.

3.3. Phoenix.Controller

Controller is the core of the Phoenix applications. These provide the necessary actions that handle the requests. These controllers prepare the data and send it to the views that are rendered by the browser. It also provides functions for manipulating the connections and rendering the templates.

3.4. Views/Templates

These are the presentation layer of the Phoenix application and it serves as the base for other views and templates. The templates contain the index page that is the initial page that is rendered to the user at first. In our application, template also contains game.html.eex file which is the base of the application.

3.5. Phoneix Channels

Channels provide bi-directional communication with the socket connections. These are mainly responsible for managing socket requests mainly in real time applications. The users join a channel which implements a ‘join’ method that authorizes the socket. When the user has joined successfully, the incoming requests are processed through the ‘handle_in’ functions defined in the channel

3.6. Sockets

Socket connections are necessary to route the events to the channels. In our application, sockets params are passed from the client and can also be used to authenticate the user information. Once the connection is verified, ‘assign’ method will be set for all channels. Sockets are also important to broadcast a joining event and also to broadcast a ‘disconnect’ event.

4. Data Structures on the Server

In our application, mainly *Lists and Maps* are used to store and handle the data. Initially for **Pawn Creation** an array holding the pawn positions are created. These positions are used as *IDs* to identify each pawn in a unique way. Each pawn is treated as object in which it contains different attributes as {key, value} pairs. Each pawn object is put in a list. So list contains list of pawn objects where each object comprises attributes. To access each pawn object from the list, we run a map by which its attributes such as ID can also be accessed. In this way, two lists for each *red* and *black* color are created. A parent Map is created containing key as the pawn color and value as the list of pawns of that color.

The pawns are created as part of the game state. In the same way, other elements of the states are players, next player's chance, and score of a player and so on. When a user initially clicks on the pawn, the player should be shown the **valid squares** in which the player can make a legitimate move. These valid square IDs are put into a map where key is the square id and value is a Boolean value indicating true or false. So, based on this the valid square involving the legitimate moves can be highlighted accordingly.

The task of capturing **Pawn movement** is also crucial. To set the pawn position, we iterate through the Enum.map function provided by the Elixir. We first get the clicked pawn object, valid square in which the pawn can be moved legally and then, set the pawn object to move to that particular square id. Enum.map function returns again a list of pawns but with updated position of the desired pawn.

5. Implementation of Game Rules

5.1. Legitimate Pawn Moves

Pawns are allowed to move diagonally forward to the next adjacent empty square. To implement this rules, initially when a player clicks on a pawn, the adjacent squares to which a pawn can move are highlighted and shown to the user. In our code base this is handled as part of the server logic in Game.ex file. 'getNextPos' function is called once the player clicks on a pawn. Current game state, pawn_id that is clicked, square_id and color of the pawn are passed to the function. So, using this info, clicked pawn is selected from the list of pawns using Enum.map and valid square Ids are calculated for that pawn. Using the square id, the valid squares are highlighted and shown to the user. The user selects one of the valid squares and then the position of the pawn is changed to the clicked square id. 'Position' is an attribute in the Pawn object indicating the residing square id on the board.

5.2. Jump Feature

Valid squares for that a pawn can move is obtained as above. If one of the valid squares has opponent's pawn, then the valid square contains the next square adjacent to the opponent's pawn. In this case, the player is allowed jump on the opponent's pawn. In our implementation, if the player decides to jump on the pawn, then that particular opponent's pawn is removed from the list of the opponent's pawn using Enum.filter in the 'selectPawnToRemove' function. When the jump happens, score of the player is increased by 10 points and game state is updated with the new set of opponent's pawns and new positions of the current player pawns.

5.3. Becoming a King

As soon as the one of the player’s pawns reached the first row on the opponent’s side, that pawn becomes the king. To implement this, we keep track of the square id of the first square (temp_id) second row in the opponent’s side. Once the user selects the square id to make a move, that id is compare with the temp_id. If the selectd square Id is less than temp_id, then the pawn moved becomes the king. This is shown by changing the image of the pawn accordingly. ‘is_king’ is an attribute in each of the pawn object.

Once, the pawn becomes king, ‘is_king’ attribute is made true. Also, the king will have additional advantage of moving diagonally both in forward and backward direction. This extra feature is handled during valid squares generation when the user clicks on pawn to move. If the selected pawn’s ‘is_king’ attribute is true, then diagonally adjacent square Ids in the backward direction is also considered during computation.

5.4. Win/Draw Condition

The first player to capture the entire opponent’s pawns becomes the winner. At every instant of the game, number of pawns removed of both the players is tracked. So, when one of the player’s 12 pawns is removed, the other player wins. The function ‘getWinner’ in the Game.ex implements this feature. Pawns of both the players are part of the list. Red list belongs to player 1 and black list belongs to player 2. Length of both the lists is determined to decide at the winner.

The player can also win if the opponent is not able to move any of his pawns in his turn and the player can make at least one valid move in his next turn. This is also computer in the ‘getWinner’ function. At each turn, valid squares possible for the all the pawns of both players is computed and added to a list. If any of the player’s lists of valid squares is empty, then the other player wins. But, if the list of valid squares of both the players is empty, number of ‘King’ pawns of both the players is calculated. This is computed in ‘Kingcount’ function. In this function number of pawns that have the value of ‘is_king’ attribute as true is counted. The player with more number of this count is declared winner.

The last condition is draw. If the count obtained in the above function, that is the number of kings of the both the players is same, then the match is a Draw.

6. Challenges and Solutions

6.1. UI Design

The first challenge we came across was to decide on appropriate UI which is appealing and at the same time easy for the user to navigate in the page. To decide on this, we asked few suggestions from different users as in what kind of design would make the design look appealing to them. Once we designed the UI, implementation of the same was also a challenge. Getting the initial checkers board with pawns was a difficult task. With many trials, we came up with a solution of putting a square component inside a <div> and generating it in a loop for 64 times. We generated 24 pawns of color red and black. Each pawn object has attributes such as Pawn_id, Pawn_color, Pawn_position and so on. Each Pawn can be identified uniquely by the Pawn_id attribute.

The second challenge in the UI design part was to bring up the chat component. We got the solution by using Bootstrap class to bring up the chat interface and used jQuery for event handling during the chat process.

6.2. Implantation of Win/Draw conditions

As described in the above section, there were three conditions that were to be tested for the game to end. First condition when the player with more number of pawns on board wins the game was straight forward to implement. But, the next condition when player runs out of all the moves was a challenging task to implement. So we came up with a solution of finding possible moves for all the pawns at each instant of the game. This gave us a way to find out whether a player ran out of all the options to move pawns. The third challenge was to find out the winner when both the player ran out of options to move their pawns. In this case, we decided to count the number of kings each player has. We have implemented this in the function ‘king_count’. In this function, the pawn that has the value as true for ‘is_king’ attribute is counted and compared. The player with more number of this count is considered the winner.

6.3. Multi-Player functionality implementation

The implementation of this functionality was one of the major challenges that we faced. In this functionality, two or more people should be able to join the same game. Only two will be players, rest of them will be spectators. So, we found a way to form a URL which includes the game name and the name of the player joined. In this way, we found a way to get the name of the player joined using ‘window.userName’. In this way, we were able to restrict one player from not moving the opponent’s pawn.

One more challenge was, once the player moved the pawn, the other player was not able to see the change in his screen instantly without refreshing the page. So, live reload was required. To accommodate live reload, we found a solution of broadcasting the game state to all the players joined using phoenix channels. Using the statement, ‘broadcast! socket, "new_msg", %{uid: uid, body: body}’ we were able to broadcast the game state instantly to all the people joining the game. Also, ‘channel.on’ method is used in react file to receive the broadcasted game state. In this way, we implemented live reload in multiple screens.

7. Bibliograpghy

- https://elixir-lang.org/getting-started
- https://hexdocs.pm/phoenix/
- https://www.w3schools.com/
- https://getbootstrap.com/
- https://simple.wikipedia.org/wiki/Checkers