# Console-Based Checkers Game Requirements Specifications [*]

Belser, C.
cmb539@drexel.edu

Brennan, Z.
zab37@drexel.edu

Horsey, K.
kth37@drexel.edu

van Rijn, Z.
zwv23@drexel.edu

August 11, 2017

**Abstract**

The game *Checkers*, also known as *Draughts* [1], is a turn-based game playable by all ages. This document outlines the requirements for a console-based software implementation of the game, which allows players to enjoy the game from any Internet- connected computer.

## Contents

[*]CS451:002 Group 2, Drexel University
[1]https://en.wikipedia.org/wiki/Draughts

## Revision History

| Revision | Date | Author(s) | Description |
|---|---|---|---|
| 0.90 | July 22, 2017 | ZV | created LaTeX skeleton |
| 0.91 | July 22, 2017 | ZV | added platform specifications |
| 0.92 | July 30, 2017 | ZV | added remarks about Docker, others |
| 0.93 | August 1, 2017 | CB | added to Client Functionality & User Interface |
| 0.94 | August 4, 2017 | KH | added Glossary & updated user interface and introduction |
| 0.95 | August 5, 2017 | ZV | added pseudocode, server logic, others |
| 0.96 | August 5, 2017 | ZB | added to Non-Functional Requirements |
| 0.97 | August 5, 2017 | CB | added detail to Client Functionality & User Interface Requirements |
| 0.98 | August 6, 2017 | ZV | added charts |
| 0.99 | August 6, 2017 | (all) | finished documentation |
| 1.00 | August 6, 2017 | (all) | **final team approval** |

# 1 Introduction

## 1.1 Purpose of Document

This document shall serve as a reference for all current and future software developers and project managers affiliated with this project. These specifications are considered contractual, and all functionality must be implemented in accordance to this document. Additional or modified functionality is permissible only where explicitly stated. This document will also function as a reference for the evaluator.

## 1.2 Document Conventions

Content is organized in a sequential hierarchical manner. We aim to provide high-level descriptions and further elaborate in later sections. We may write console commands `like this`. We may refer to applications, tools, libraries, or other dependencies *like this*. These are often followed by some sort of citation, via which the reader may seek additional information.

Generally, we try to follow the IEEE Software Requirements Specification `ISO/IEC/IEEE 29148:2011` [2] guidelines, however elements of older standards may have made it into this document.

## 1.3 Intended Audience and Reading Suggestions

The intended audience of this document is computer scientists who have interest in seeing the components that go into the team's implementation of the game of Checkers in the C language. The document uses official tournament rules for the game of checkers therefore some rules might be unfamiliar for those who haven't played or observed checkers at a competitive level.

## 1.4 Scope of Document

We thoroughly detail the complete software requirements so that any software development team may follow, develop, and test the software. This document will detail the intended behavior and expected interactions between each component.

## 1.5 Overview of Document

Graphical figures, tables, and code listings are commonly used in addition to textual descriptions to convey the intent, logic, and desired functionality to a person or persons unfamiliar with the project. We therefore include such elements with the hope that it better conveys our requirements, however we cannot make any guarantee that such figures will be effective. If there are any questions, about any of the material contained within this document, please do not hesitate to contact one of the authors.

## 1.6 Project Scope

This document describes the high-level user requirements and specifications for a console-based Checkers implementation. The goal of the software is to provide an intuitive client where the users can comfortably play the game of Checkers. Specifically:

1. Automatic "move" validation, where player(s)' moves will be validated in realtime against a predefined set of official Checkers rules
2. Ability to play the game on several Unix-like systems
3. Provide exemplary source code quality that can be used as a teaching tool

We will consider the following to be *nice-to-have*'s, or features that we would like to implement but cannot guarantee to be included in the final deliverable:

1. Client Spectators section 4.3.2 to allow additional clients to connect to the game server for the sole purpose of watching an existing game
2. Web-based implementation, which may run natively as compiled WebAssembly[3] or via an intermediate server and accessed via a VNC [4] client

# 2 Overall Description

This document describes the complete requirements for a console- based implementation of the game *Checkers*. Slight changes may be present between the original game and the console-based implementation (hereforth known as **game**). Therefore, it is imperative that the product description be read, understood, and any confusion clarified, by all parties, before development begins.

The **game** is a two player game designed to be played remotely (by two computers on joint or disjoint networks). When a player joins the game they are taken to a waiting screen until a second player has joined. Once two players are connected, they are taken to a checker game board and begin the game. Players then take turns making moves until either a draw is reached (neither play can force a win), when one player has no pieces remaining on the board, or if a player has no legal moves available on their turn. At this point a winner is displayed (or that it was a draw) and the game concludes.

The game will require a server and two or more connected clients where any additional clients become *spectators*.

## 2.1 Product Perspective

The **game** is typically played on a square checkboard of dimensions *8x8*, though several variations[2] exist, we will focus on the *American checkers* variety.

[2] English draughts (8x8) a.k.a. American Checkers, Russian draughts (8x8), International (Polish) draughts (10x10), Canadian checkers (12x12) [1]

## 2.2 Game Rules

This version of the game has the following rules [5], modified for formatting:

1. Checkers is played by two players. Each player begins the game with 12 colored discs. (Typically, one set of pieces is white and the other red.)
2. The board consists of 64 squares, alternating between 32 dark and 32 light squares. It is positioned so that each player has a light square on the right side corner closest to him or her.
3. Each player places their pieces on the 12 dark squares closest to them.
4. White moves first. Players then alternate moves.
5. Moves are allowed only on the dark squares, so pieces always move diagonally. Single pieces are always limited to forward moves (toward the opponent).
6. A piece making a non-capturing move (not involving a jump) may move only one square.
7. A piece making a capturing move (a jump) leaps over one of the opponent's pieces, landing in a straight diagonal line on the other side. Only one piece may be captured in a single jump; however, multiple jumps are allowed during a single turn.
8. When a piece is captured it is removed from the board.
9. If a player is able to make a capture, there is no option – the jump must be made. If more than one capture is available, the player is free to choose whichever they prefer.
10. When a piece reaches the furthest row from the player who controls that piece, it is crowned and becomes a king. One of the pieces which had been captured is placed on top of the king so that it is stacked twice as high.
11. Kings are still limited to moving diagonally, but may move both forward and backward. (Remember that single pieces, i.e. non-kings, are always limited to forward moves.)
12. Kings may combine jumps in several directions – forward and backward – on the same turn. Single pieces may shift direction diagonally during a multiple capture turn, but must jump forward (toward the opponent).
13. A player wins the game when the opponent cannot make a move. In most cases, this is because all of the opponent's pieces have been captured, but it could also be because all of their pieces are blocked in.

## 2.3 Product Features

As a complete system, our product is composed of several smaller subsystems.

### 2.3.1 Server

The server will act as the main driver for the game, controlling player connections, dictating player turns,updating the board, checking win-state, and providing updatedgame-states to the clients.

### 2.3.2 Client

The client will act as the player interface, displaying the game to the player, gathering player moves, and displaying messages from the server. A client can be instructed (by the server) to operate in a reduced-functionality mode, where such a client is unable to make moves, and therefore functions as a spectator to the current, active game.

## 2.4 User Classes and Characteristics

Typical users for **game** are the same who enjoy the traditional board game, and individuals who may not be familiar with the game. The rules are straightforward, and anyone with a computer and an Internet connection should be able to learn, play, and hone their skills in this competitive, thrilling new console environment. Some users may even find a bit of nostalgia in our rendition of the classic game.

## 2.5 Operating Environment

Docker [6] is a software platform that abstracts many aspects of the operating system in order to provide an isolated environment, called a 'container', which may contain only the bare minimum required libraries and packages to support a given application. The filesystem, network interfaces, and other parts of the host system are available to the container as needed, and while it shares a common kernel with other adjacent containers, it does not depend on the host system for any other libraries.

We recommend using Docker as a platform for both building and initial testing of the software. We have provided an image that closely resembles our internal computing envrironment, based on `alpine:3.6`, a minimal Unix-like environment. The image we provide has several tools and libraries preinstalled, which can be linked statically to produce a portable executable for distribution to end-users, further described in section 2.9.

## 2.6 Design Constraints

There are no relevant regulatory or financial constraints that should be considered at this time.

The product may have external dependencies, which are described in section 2.9. These dependencies have been captured in a "frozen" state (stable releases) to avoid future complications and compatibility issues. Copies of these

| Component | Min. | Ideal |
|---|---|---|
| CPU | 200 MHz | 1.0 GHz |
| Memory | 64 MB | 256 MB |
| Disk | 500 MB | 1.0 GB |
| Display | 800x600 | 1024x768 |

(a) Hardware

| Component | Min. | Ideal |
|---|---|---|
| kernel | 2.6.32 | 4.4.0 |
| make | 3.82 | 4.1 |
| gcc | 4.8.4 | 5.4.0 |
| bash | 4.2.46 | 4.3.48 |

(b) Software

Figure 1: Generic System Requirements

dependencies are stored on a common project server to mitigate issues in the event that those projects become unavailable.

## 2.7 User Documentation

Complete user documentation, including usage manuals and setup guides will be available at the following URL at the conclusion of the project: `https://git.zv.io/me/CS451-Checkers`. We may publish final PDF versions at a different URL, which may be found in one of the Appendices of this document, if available.

## 2.8 Assumptions

In order to guarantee that the software builds and runs on an agreed-upon generic platform, and operates correctly, we define the minimum expected software and hardware configuration that must be supported by the application. In other words, the game should be playable with the minimum specifications, but it can be assumed that most users will have machines that at least meet or exceed the "ideal" hardware and/or software.

### 2.8.1 Hardware

Figure fig. 1a contains minimum and ideal specifications of a generic platform that should be able to operate the game. This does *not* necessarily mean it needs to be able to compile (build the game), but the ideal specifications should be sufficient to do so.

### 2.8.2 Software

Figure fig. 1b contains minimum and ideal specifications of a generic platform that should be able to operate the game. This does *not* necessarily mean it needs to be able to compile (build the game), but the ideal specifications should be sufficient to do so.

### 2.9 Dependencies

Most or all of the software dependencies are directly satisfied or satisfied by equivalent packages in the first stage of the project prototype.

#### 2.9.1 Essential Libraries

These libraries are automatically pulled into the project and used during the compilation of the game. They should be built and linked statically, such that the final executable can be transferred to any supporting system for immediate use.

1. *musl*, a lightweight libc implementation [7]
2. *ncurses*, a text-based user-interface API [8]
3. *check*, a unit-testing framework for C [9]
4. *freetype*, a font parsing library for C [10]

#### 2.9.2 Supplemental Tools

Some additional tools can be built, optionally, to aid in the process of debugging and testing to ensure the application works in a safe manner.

1. *cppcheck*, a static analysis tool for C and C++ [11]
2. *valgrind*, a suite of debugging tools [12]

## 3 External Interface Requirements

### 3.1 User Interfaces

The user will be presented with an 8x8 Checkerboard initially containing all of the pieces for a game of Checkers. The user will interact with the environment through the use of a cursor that highlights the currently selected square. The squares with legal moves for that player will be highlighted and once the user makes their selection they will confirm their move using the `enter` key.

### 3.2 Hardware Interfaces

*No special hardware interfaces are required.*

### 3.3 Software Interfaces

*No special software interfaces are required.*

If time permits, game move validator API could be standardized to serve as a standalone tool or simple network service that can be used by the general public for any Checkers-related purposes.

```
:: move connected client to a dedicated port
:: players with id > 2 will be spectators

procedure AcceptClient(connection , start_port , id)
    send(connection , id , start_port + id)
    if receive(connection , start_port + id) == READY
        return BeginGame() :: not shown
    end if
    return false :: kills the thread
end procedure

:: max_clients: at least 2 ; higher to accommodate spectators
:: start_port : $1024 <$ start_port $< 65535$

procedure Listening(max_clients , start_port)
    id $\leftarrow$ 0
    while listen_tcp(start_port)
        if connection and id $\neq$ max_clients
            if AcceptClient(connection , start_port , id + 1)
                id++
            else
                send(connection , false , 'You must be ready')
                close_tcp(connection)
            end if
        else
            send(connection , false , 'Server is full')
        end if
    end while
end procedure
```

Figure 2: Server Logic for New Connections

## 3.4 Communications Interfaces

The interface for client-server communication shall consist of the following pseudocode:

## 4 Functional Requirements

### 4.1 System Features

The system will give users the option to create a new game, allow them to join a game, or allow them to participate as a spectator in an ongoing game. The system will also have an options menu that will allow users to change aspects

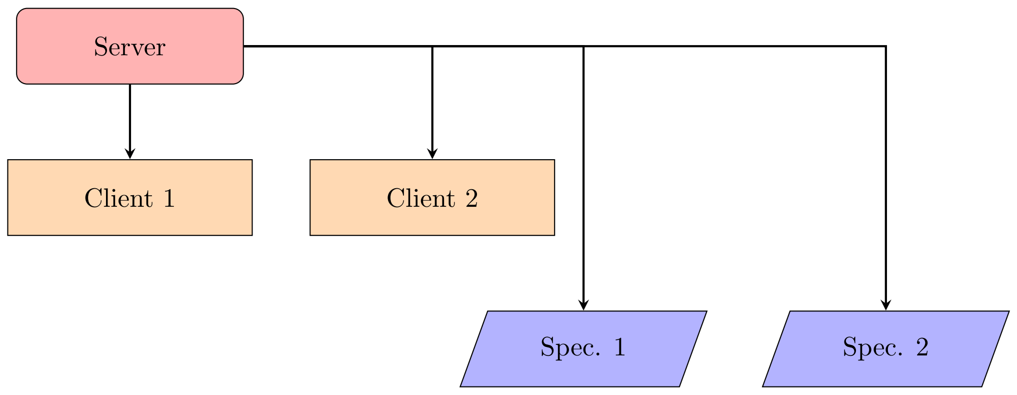

Figure 3: Server-Client Hierarchy, Spectators Optional

of the game such as the length of each turn.

## 4.2 Server Functionality

High-level logic for how the server component should behave is defined in fig. 2. Implementation details are intentionally omitted; they will be part of the design document, `design.pdf`. The server should always accept new connections, but reject connections if it is "full" or if the client does not respond appropriately.

### 4.2.1 Client Connections

1. **Accept connections from clients**

   The server shall enter a *listening* state in which it listens for TCP connections on a specified port. Usage might be as follows:

   ```
   ./server --base-port 9000 --max-clients 5
   ```

2. **Pair first two client connections into a game**

   While the server is listening for new connections, it shall treat the *first* successfully-connected client as *Player 1*, and the *second* as *Player 2*. Further connections, if allowed, will become *spectators*. Each player will be notified of their *id*, an integer in the range `1 < id < max-clients`. Each client will be redirected to a new port, which is calculated in the following manner:

   ```
   port = base_port + id
   ```

   Once the first two clients have connected, the server initiates a new game, constructing an internal board state and performing other necessary steps.

3. **Premature client disconnection**

   In the event that a client disconnects prematurely, defined as:

   (a) the network connection is dropped for any reason
   (b) the client chooses to close the application before the game is completed

   The server will notify the opponent that they are the winner of the game, then perform the following steps:

   (a) clean up the current game state
   (b) ask the winner if they would like to continue playing the game (as in, start a new game)
   (c) ask the first-connected spectator (if available) whether they would like to play a game, moving to each successive spectator until an agreement has been reached
   (d) if no spectators are available, the game will wait for a new connection, in similar fashion to how it starts initially
   (e) if the winner chooses *not* to continue then all client(s) are immediately disconnected

4. **Player refuses to provide input**

   Sometimes, a client may leave their computer for an extended period of time, which can be frustrating to the other player. A configurable *timeout* should be implemented such that if the server determines it has not received any input from the active player within some period of time, that player is disconnected and the opponent is declared the winner. To avoid frustration, each client will be notified of the timeout limit when they first connect to the server, and will also receive notifications at some reasonable time before they are disconnected (e.g., 5 mins, 1 min, and 30 seconds).

#### 4.2.2 Game State

1. **Send board state to clients**

   Both the *server* and *client* are compiled from the same source tree, which should include common header files describing data structures, game logic, and other non-specialized code. When a game is in progress, the server will have some internal representation of the *board state*, which is to be described by some agreed-upon data structure. Since the *client* is also compiled with this description, it should be possible to communicate the current board state between the client and server.

2. **Send messages to client (for display)**

   While communication between clients (whether directly or via the server) is forbidden, the server should be able to send string-encoded messages to

```
struct GENERIC_COMMUNICATION
{
    time timestamp;    /* or unique identifier       */
    int action;        /* mode of action required    */
    board state;       /* current board state        */
    int msg;           /* msg. length; 0 if not used */
    char[MSG_SIZE];    /* msg. is stored here        */
};
```

Figure 4: Sample Server-Client Communication, Pseudocode

each client. Such messages may include notices that their move needs to be made soon, or that another client has disconnected or that they have won/lost the game.

Communication from the server to a client or client(s) shall be performed in this manner: clients are always listening for new input. Clients receive, as part of the fig. 4 data structure, an `int`, referring to some action that they must perform (i.e., move, re-check validity, wait for next turn, display message to user).

3. **Request move from clients**

   Clients are expected to acknowledge the server's message. The server will wait for the client to send a followup message (with the user's move) while it rolls the timeout counter. If the server does not receive acknowledgement within **5 seconds** it will repeat the request every **30 seconds** until the timeout is reached.

4. **Create, store, and update game state**

   In the first iteration of this software application, just **2 clients** need be supported. If time permits, the server should be able to handle multiple simultaneous games (where more than one pair of users can play the game without needing to run multiple instances of the server). This is *optional*.

#### 4.2.3 Game Move Validation

Every time a client submits a move, the server shall confirm that the client is *actually* able to make the move, or if the game rules are somehow violated. While each client is compiled to contain the game logic (mostly to hint to the user that they are dis/allowed to make such a move), the server is considered the referee and has the final word. The server is not required to notify the client that their move is invalid, but it should ask that player to move again, sending the previous valid board state. The client is expected to refresh the board and allow the player to attempt their move again.

The server and client should both implement the logic required to ensure that all game rules in section 2.2 are followed.

1. **Validate move from client**

   When it is one player's turn, the server engages the client in a loop, similar to the following:

```
:: repeatedly request move from client until valid

procedure GetMove(client)
    while !valid(move)
        RequestMove()
    end while
end procedure
```

2. **Check for win state**

   Internally, the server shall check whether any player has won the game, as defined in section 2.2. The server may attempt to "look ahead" to determine if no further moves by any player will lead to a winner or loser.

   The server shall notify all connected clients that a winner has been determined, and the game ends.

3. **Check for draw**

   A *draw* is possible if both players make "perfect" moves throughout the game. Because players are forced to jump when jumping is possible, such a condition occurs much less frequently.

   In the event of a draw, both players are notified of the condition and the game ends.

4. **Player cannot make a valid move**

   When a player can no longer make a valid move it will trigger the win state for the opposing player in accordance to the official tournament rules.

5. **Player refuses to provide input**

   Please refer to section 4.2.1.

#### 4.2.4 External Interface

The interface players and spectators will see displays basic information such as whose turn it is, board state and the valid move of the current player as well as any notifications that the players need to see.

### 4.3 Client Functionality

This section details the functions that the clients will be responsible for during a standard execution of the game of checkers. The clients are further broken into two cases being the player client and the spectator client.

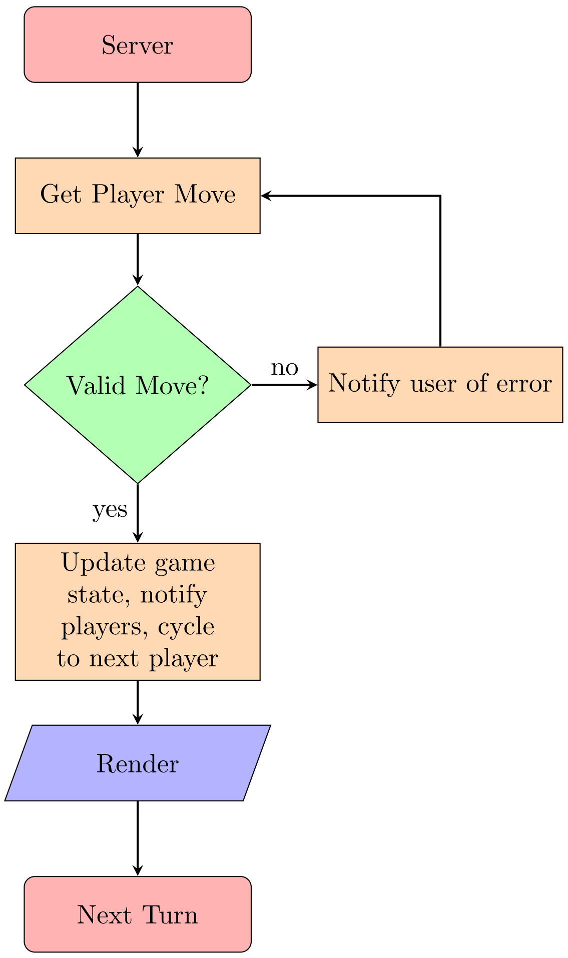

Figure 5: Game Move Validation Logic

### 4.3.1 Player

1. **Connect to server**

   The client shall attempt to make a TCP connection to the server on a specified port. Once a connection has been established the client will enter a `listening` state in which it listens for data from the server to begin a game.

2. **Initial connection fails**

   In the event a connection to the server fails, the client will inform the player that a connection issue occurred, and attempt to connect again. After a specified number of failed attempts, the client will inform the user that a connection cannot be established, and the client will close.

3. **Receive board state from server**

   The client will accept the board state as part of the data communication format shown in fig. 4

4. **Display board state to player**

   The client will display the board, described in section 7.2.1

5. **Handle case where server disconnects**

   In the event that the server disconnects prematurely, defined as:

   (a) the network connection is dropped for any reason

   The client will notify the player in the form of a **Game State Message**, described in section 7.2.4

6. **Receive messages from server**

   The client will accept messages as part of the data communication format shown in fig. 4

7. **Display messages to player**

   The client will display messages as **Action Messages**, described in section 7.2.4.

8. **Get move from player**

   In the event the server requests a move from the client, the client enters a `move selection` state. The client utilizes the user interface described in section 7.2.3 to get a move. If a jump is made, additional jumps are checked.

9. **Player refuses to move**

   Please refer to section 4.2.1.

10. **Validate move**

    Once the player selects a move sequence the client will check the validity of the move. Should the move be found invalid, it is rejected, and a new move is requested from the player.

11. **Send move to server**

    The valid move sequence is sent to the server using the data communication format shown in fig. 4

12. **Display winner to player**

    The client will display the winner as a **Game State Message**, described in section 7.2.4.

13. **Restart match at user request**

    The client will query both active players, to determine if they want to start a new game.

#### 4.3.2 Spectator

**Like in section 4.3.1, except**:

1. Spectators cannot make any moves or in any way communicate with the other players
2. If a spectator is disconnected for any reason, the two normal clients shall continue gameplay uninterrupted
3. No clients or spectators shall be notified upon the disconnection of any spectator
4. The server does not wait for any input from the spectator

## 5 Non-Functional Requirements

### 5.1 Performance

#### 5.1.1 Server

The server software must be able to handle the following:

1. **Receive frequent messages from remote client**

   Connected clients will send move sequences as requested by the server

2. **Send frequent messages to remote client**

   The server will send the board state to all connected clients, as well as request moves from the designated client's turn

3. **Validate a given move in reasonable time**

   The server will evaluate a move sequence provided by a client, containing at least one move but possibly a sequence of moves comprising a sequence of jumps.

### 5.1.2 Client

The client software must be able to handle the following:

1. **Receive frequent messages from remote server**

   The client will accept the board state, move request, and messages, in the format shown in fig. 4.

2. **Send frequent messages to remote server**

   The client will send move sequences to the server when requested by the server

3. **Validate a given move in a reasonable time**

   Before sending a move sequence to the server, the client will evaluate the move sequence, consisting of at least one move but possible a sequence of moves comprising a sequence of jumps.

## 5.2 Safety

This program shall not be able to move any physical devices, to prevent damage to the system, users, or surroundings. It shall also be free of memory leaks, so it does not compromise the systems that it is being run on.

## 5.3 Security

In order to keep relative security of the integrity of the software, both the client and server shall validate moves that are input by the user.

## 5.4 Software Quality

Both the server and client software shall be implemented using a set of standards that is agreed upon by all developers. This shall keep code standardized for other developers to maintain.

## 5.5 Limitations

Unless explicitly defined in this document, no additional features or functionality should be implemented.

## 6 Other Requirements

We make the assumption that the terminal window is at least `80x24` columns, with the preferred size being at least `150x50` columns.

## 7 User Interface

### 7.1 Initialization Screen

The initialization screen will be shown to the user when their client is launched, changing to the game screen once a game begins. This screen will display the game logo, as well as a message detailing the current task the client is performing:

1. Connecting to server
2. Waiting for game

### 7.2 Game Screen

The game screen is shown the duration of the game, until the client closes. This screen will display the board to the user, showing the board itself and the checker pieces. It will also display messages to the user, and facilitate move selection. The size of the display will be dynamically sized based on the size of the terminal.

#### 7.2.1 Board

The board will be displayed as a grid filling a majority of the terminal of color alternating between red and white, with bottom-left and top-right spaces being red. We avoid using black because the standard terminal is also black, which could confuse the user.

#### 7.2.2 Pieces

The pieces will be displayed as letters:

1. Basic - **C**
2. King - **K**

The letters will be colored according to the player they belong to. Should time permit, the pieces will be made to be colored circles, with the crowned pieces having a crown icon to distinguish them from the standard pieces.

#### 7.2.3 Move Facilitation

1. The game screen will use highlighting to denote which pieces are capable of being selected for a move, placed on the inside border of the grid square
2. Selected pieces will have a blinking highlight in the same location as above highlighting
3. The left and right arrow keys will be used to switch which piece is selected, moving the blinking highlight
4. The `enter` key will confirm the selection

#### 7.2.4 Messages

Strings (of messages) are encoded in the general data structure that the server sends to clients with the current game state, etc. Clients should passively receive these messages, displaying them to the player. Messages will be handled in two different ways, depending on the priority of the message.

1. **Action Messages (Notifications)**

   Messages relating to actions required of the player should be displayed above the board, at the top of the screen, not disrupting gameplay. Messages shall remain on the screen until a new message is to be displayed, overwriting the previous.

   (a) You must perform a jump
   (b) Another jump is available

2. **Game Status Messages**

   Messages relating the the state of the game should be overlayed on top of the board. These messages will require the `enter` key be pressed to acknowledge and close the message.

   (a) Player Red is the winner!
   (b) Player White has disconnected
   (c) A Draw has been reached

## 8 Use Cases

### 8.1 Joining a game

Running the client, while an accessible server is running on the port that the client is directed at will cause the client to connect. Once two clients have connected, they will be placed into a game together.

| | |
|---|---|
| **Precondition**: | A server is being run on an accessible network |
| **Action**: | A handshake occurs |
| **Postcondition**: | The server and client have established a connection |

## 8.2 Making a move

A player selects the **C** piece they would like to move from the list of pieces that have the ability to move. All possible moves become highlighted once a piece is selected. They then select the destination of that piece, based on available valid moves. The **C** pieces have the ability to move forward along the diagonals of the board.

| | |
|---|---|
| **Precondition**: | At least one piece has valid move |
| **Action**: | Player selects a piece, and its destination |
| **Postcondition**: | The piece is moved to the destination and the board is updated for all players |

## 8.3 Crowning a piece

A piece becomes a King when the player successfully moves it to a space along the border of the board, on the opposite side from where the piece started the game.

| | |
|---|---|
| **Precondition**: | A piece has a valid move that places it against the border of the board, on the opposite side from where it started the game |
| **Action**: | Player selects that piece to move and the border location as its destination |
| **Postcondition**: | The piece is moved to the destination and transforms from a **C**, to a **K**, while remaining the color associated with the player that owns it. The board is then updated for all players |

## 8.4 Moving a King

A player selects the **K** piece that they would like to move from the list of pieces that have the ability to move. All possible moves become highlighted once a piece is selected. They then select the destination of that piece, based on available valid moves. The **K** pieces have the ability to move forward or backward along the diagonals of the board.

| | |
|---|---|
| **Precondition**: | At least one piece has valid move |
| **Action**: | Player selects a piece, and its destination |
| **Postcondition**: | The piece is moved to the destination and the board is updated for all players |

## 8.5 Jumping a piece

If it is possible for a piece to jump an opposing piece, that move must be taken. The opposing piece that is jumped is then removed from the game.

| | |
|---|---|
| **Precondition**: | A piece has a valid move involving performing a jump |
| **Action**: | Player selects that piece to move and the location that it would finish its jump as the destination |
| **Postcondition**: | The piece is moved to the destination and the opposing piece that was jumped is removed from the game. The board is then updated for all players |

## 8.6 Multi-Jumping

If it is possible for a piece to jump an opposing piece, that jump must be made. If the same piece is able to jump another opposing piece from the end location of the first jump, that jump must also be made. A piece may make any number of such jumps per turn. All of the opposing pieces that are jumped are then removed from the game.

| | |
|---|---|
| **Precondition**: | A piece has finished at least one jump during its turn, and is positioned such that another jump would be a legal move |
| **Action**: | Player confirms the mandatory jump |
| **Postcondition**: | The piece is moved to the destination and the opposing piece that was jumped is removed from the game. The board is then updated for all players. This process is then repeated, if possible |

## 8.7 Winning the game

To win the game, the player must remove all of their opponents pieces from the board, or force them into a situation where they can not make a legal move.

Once either state is reached, the game is over, and the players may start a new match.

| | |
|---|---|
| **Precondition**: | The game is in progress |
| **Action**: | A player removes all of the opponent's pieces from the game, or forces the opponent into a situation where they are unable to make a legal move |
| **Postcondition**: | The game ends, and the server sends a message to both players declaring the winner to be the player. Both players are queried to see if a rematch will be played |

## 8.8 Losing the game

If all of the players pieces are removed from the game, the player has lost, and the game is over. Once this state is reached, the game is over, and the players may start a new match.

| | |
|---|---|
| **Precondition**: | The game is in progress |
| **Action**: | The current player has all their pieces removed from the board |
| **Postcondition**: | The game ends, and the server sends a message to both players declaring the winner to be the opponent. Both players are queried to see if a rematch will be played |

## 8.9 Unable to move piece

If a player is unable to move a piece on their turn the game will end with them losing. Once this state is reached, the game is over, and the players may start a new match.

| | |
|---|---|
| **Precondition**: | The game is in progress |
| **Action**: | The opponent makes the current player unable to move |
| **Postcondition**: | The game ends, and the server sends a message to both players declaring the winner to be the opponent. Both players are queried to see if a rematch will be played |

## 8.10 Playing a rematch

Once a game has been completed, both active players will be asked if they would like to play again. If both players agree, the game state is reset, and play restarts. If one player does not agree, and there are spectators present, the specatators shall be asked if they would like to join, in the order in which they connected, until either a new game is created, or the list of spectators is exhausted.

| | |
|---|---|
| **Precondition**: | The game has ended, with any outcome |
| **Action**: | Both players agree to play another game |
| **Postcondition**: | The game is reset, and play restarts as a new game, with the same active players and spectators |

## 9 Glossary

- **Checkerboard** NxN (typically 8x8) game board composed of alternating white/red (or other) squares on which game pieces *Checkers* reside
- **Piece** Standard Checkers piece with limited movement, specifically only forward-diagonal motion
- **King** Checkers piece that can move along any diagonals, forward or backward
- **Move** the act of changing the location of a piece on the board when it is that player's turn
- **Jump** the act of removing an opposing player's piece from the board, occurring in a straight diagonal fashion – e.g., "hopping over" the opponent
- **Multi-jump** the act of chaining multiple *Jumps* together, with one piece, in a single turn
- **Crowning** the act of changing a standard game piece to a *King*

## 10 References

[1] Wikipedia, *Draughts*, online at
`https://en.wikipedia.org/wiki/Checkers`

[2] IEEE Computer Society, *29148-2011 - ISO/IEC/IEEE International Standard - Systems and software engineering – Life cycle processes – Requirements engineering*

[3] Haas, A., et al., *Bringing the Web up to Speed with WebAssembly*, DOI: http://dx.doi.org/10.1145/3062341.3062363
online at
`https://github.com/WebAssembly/spec/raw/master/papers/pldi2017.pdf`

[4] Martin, J. et al., *noVNC: HTML5 VNC Client*, online at
`https://github.com/novnc/noVNC`

[5] The Spruce, *How to Play Checkers: Standard U.S. Rules*, online at
`https://www.thespruce.com/play-checkers-using-standard-rules-409287`

[6] Docker Inc., *Docker*, available online at
`https://www.docker.com/`

[7] Felker, Rich, et al., *musl*, online at
`https://www.musl-libc.org/`

[8] Free Software Foundation, Inc., *ncurses*, online at
`https://www.gnu.org/software/ncurses/ncurses.html`

[9] Malec, A., et al., *Check: A Unit Testing Framework for C*, online at
`https://libcheck.github.io/check/`

[10] Turner, D., Wilhelm, R., et al., *FreeType*, online at
`https://www.freetype.org/`

[11] Marjamki, D., *Cppcheck*, online at
`http://cppcheck.sourceforge.net/`

[12] The Valgrind Developers, *Valgrind*, online at
`http://valgrind.org/`