Console-Based Checkers Game
Software Design Specifications [*]

Belser, C.
`cmb923@drexel.edu`

Brennan, Z.
`zab37@drexel.edu`

Horsey, K.
`kth37@drexel.edu`

van Rijn, Z.
`zwv23@drexel.edu`

August 13, 2017

Abstract

This document is intended to be used in conjunction with the provided `requirements.pdf` document. **Note:** this document is *Proprietary and Confidential*; duplication or reproduction of any content herein must be explicitly granted in writing.

Contents

1 Introduction **3**
- 1.1 Purpose of Document . . . 3
- 1.2 Document Scope . . . 3
- 1.3 Definitions, Acronyms, Abbreviations . . . 4

2 Use Cases **4**
- 2.1 Actors . . . 4
  - 2.1.1 Active Player . . . 4
  - 2.1.2 Waiting Player . . . 4
  - 2.1.3 Spectator . . . 4
- 2.2 List of Use Cases . . . 4
  - 2.2.1 Active Player . . . 4
  - 2.2.2 Waiting Player . . . 5
  - 2.2.3 Spectator . . . 5
- 2.3 Use Cases . . . 5

[*] CS451:002 Group 2, Drexel University

3 System Overview **22**
3.1 Description of Software 22
3.2 Technologies Used 22
3.2.1 Docker 22
3.2.2 Alpine Linux 23
3.2.3 ncurses 23
3.2.4 VNC 24

4 Design Considerations **24**
4.1 Assumptions and Dependencies 25
4.1.1 Hardware 25
4.1.2 Software 25
4.2 Dependencies 25
4.3 General Constraints 25
4.3.1 End-User Environment 26
4.3.2 Availability and Volatility of Resources 26
4.3.3 Standards Compliance 26
4.3.4 Interoperability 26
4.3.5 Data Repository / Distribution 26
4.3.6 Verification, Validation, and Testing 26
4.3.7 Other Quality Controls 26
4.4 Goals and Guidelines 27
4.5 Design Rationale 27
4.5.1 Docker 27
4.5.2 Separate Client/Server Modules 27
4.5.3 1x32 Boardstate Storage 27

5 Architectural Strategies **28**
5.1 Choice of Language 28
5.2 Planning for Future Development 28
5.3 User Interface Paradigms 28
5.4 Error Detection and Recovery 28
5.5 Memory Management 28
5.6 Concurrency and Synchronization 29
5.7 Containerization and Deployment 29

6 System Architecture **29**
6.1 Core Module 29
6.1.1 Requisite Libraries 30
6.1.2 Game Logic 31
6.2 Networking Module 33
6.3 Server Module 34
6.3.1 Network Communication Component 34
6.3.2 Game Logic Component 34
6.4 Client Module 36
6.4.1 Graphical Display Component 36

6.4.2 Player Input Component 39
6.4.3 Move Assistance Component 40
6.4.4 Network Communication Component 40

7 Policies and Tactics 40
7.1 Choice of Compiler 40
7.2 Coding Conventions 40
7.3 Source Code Organization 42
7.4 Issue Tracking 42
7.5 Unit Testing 43
7.6 Continuous Integration 43
7.7 Build Instructions 43

8 Miscellaneous 43
8.0.1 Board Display 43

9 Glossary 46

10 References 47

Revision History

Revision	Date	Author(s)	Description
0.90	July 30, 2017	ZV	created initial LaTeX version
0.91	August 6, 2017	KH	added Glossary & updated Game Logic
0.92	August 6, 2017	ZB	updated Coding Conventions
0.94	August 11, 2017	ZV	added Docker notes
0.95	August 12, 2017	ZB	finished section 5.2 Networking Module
0.96	August 13, 2017	KH	Updated Sections 3 and 5
0.97	August 13, 2017	CB	Class tables, component descriptions, display elements
0.98	August 13, 2017	ZV	use cases, graphics, discussion
1.00	August 13, 2017	All	final team approval

1 Introduction

This document describes in detail all components of the Console- Based Checkers Game implementation outlined at a high-level in the provided `requirements.pdf` document. To recap, this application is a software implementation of the popular board game Checkers, otherwise known as Draughts.

1.1 Purpose of Document

This document is to describe the implementation of the Console- Based Checkers Game, as described in the Console-Based Checkers Requirements document. This game is designed to be played by two people on separate computers, connected to each other over a network.

1.2 Document Scope

The scope of this document shall contain the following:

1. Design and description of complete system architecture
2. Listing and design of all subsystems, with further descriptions and potential algorithms and details of their interconnectedness
3. Amendments, if necessary, describing any changes to to the design after it has been approved by the client

In particular, we divide the implementation details into distinct categories:

1. *Game logic* - represents the core logic of the game, including official rules, move validation, and other necessary checks to ensure the game is played in a fair and correct manner
2. *Server logic* - the code necessary to create a game session, to which clients can connect and play, as well as "spectator" clients, time permitting; the server logic is responsible for maintaining network connections to the clients, using the above *game logic* to further validate moves
3. *Client logic* - the client logic will control any visualization, user input, output as well as *game logic* to ensure the user is able to play the game; the client will be responsible for ensuring that it can connect to the server, and for handling any disruptions in the network connection gracefully

By implementing these components, we will have achieved a base level of functionality (minimum viable product) which should be sufficient to satisfy the client's needs. Further refinements or changes shall be made, described and agreed upon in writing, following the completion of the initial development phase.

1.3 Definitions, Acronyms, Abbreviations

1. *VNC* - Virtual Network Computing; a graphical desktop sharing system which allows remote users to connect to and potentially control a system which has a graphical user interface (GUI). In the context of this application, a future addition could be a web-based client that may utilize VNC to simplify the development/porting process of bringing the native game to the web browser
2. *Boardstate* - The data structure that contains the current state of the board, and the locations of each piece
3. *Board* - Refers to the visual representation of the *BoardState*
4. *Game* - A capitalized **Game** shall be used in place of the full name of Console-Based Checkers, where appropriate

2 Use Cases

2.1 Actors

2.1.1 Active Player

The **active player** is a player who is currently engaged in a loop with the server while they attempt to choose a valid move. Each time the user submits a move to the server it will respond with a message either acknowledging the move and provide the updated game state, or it will request that the user provide another attempt at a valid move, in which case the user is notified that their previous attempt was invalid.

2.1.2 Waiting Player

The **waiting user** is simply the other "main" player (the opponent) who is engaged with the first player in a game of Checkers. This user must wait until it is their turn in order to make a move.

2.1.3 Spectator

A **spectator** is the `3..n` connected client, who does not have any interactions with the server aside from the initial connection or receiving of notifications. If any such input is provided, the server will simply ignore the request(s).

2.2 List of Use Cases

2.2.1 Active Player

1. receive current game state from the server
2. request (again) current game state from the server

3. receive passive message notification
4. receive (blocking) message notification
5. send tentative valid move sequence
6. receive move validity confirmation or rejection
7. resign from game

2.2.2 Waiting Player

1. receive current game state from the server
2. request (again) current game state from the server
3. receive passive message notification
4. receive (blocking) message notification
5. resign from game

2.2.3 Spectator

1. receive current game state from the server
2. request (again) current game state from the server
3. receive passive message notification
4. receive (blocking) message notification

2.3 Use Cases

Use Case Name: Receive Game State		**ID**: RGS	**Priority**: critical
Primary Actor: Active Player	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the critical function of the system where the player receives the current game state from server			
Goal: The client receives the current game state from the server			
Success Measurement: The client and server have the same (byte-for-byte) game state representation, and the client renders the board correctly			
Precondition: The client is connected to the server **Action**: The server sends the current board state to the client **Postcondition**: The client and server have identical copies of the state			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client ready -¿ Server send -¿ Client receive			
Assumptions: The data is not corrupted during transmission			
Implementation Constraints: n/a			

Use Case Name: Receive Game State		**ID**: RGS	**Priority**: critical
Primary Actor: Waiting Player	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the critical function of the system where the player receives the current game state from server			
Goal: The client receives the current game state from the server			
Success Measurement: The client and server have the same (byte-for-byte) game state representation, and the client renders the board correctly			
Precondition: The client is connected to the server **Action**: The server sends the current board state to the client **Postcondition**: The client and server have identical copies of the state			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client ready -¿ Server send -¿ Client receive			
Assumptions: The data is not corrupted during transmission			
Implementation Constraints: n/a			

Use Case Name: Receive Game State		**ID**: RGS	**Priority**: critical
Primary Actor: Spectator	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the critical function of the system where the player receives the current game state from server			
Goal: The client receives the current game state from the server			
Success Measurement: The client and server have the same (byte-for-byte) game state representation, and the client renders the board correctly			
Precondition: The client is connected to the server **Action**: The server sends the current board state to the client **Postcondition**: The client and server have identical copies of the state			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client ready -¿ Server send -¿ Client receive			
Assumptions: The data is not corrupted during transmission			
Implementation Constraints: n/a			

Use Case Name: Request Game State from Server		**ID**: RGS	**Priority**: critical
Primary Actor: Active Player	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the critical function of the system where the player requests the updated game state from server			
Goal: The client receives the current game state from the server			
Success Measurement: The client and server have the same (byte-for-byte) game state representation, and the client renders the board correctly			
Precondition: The client is connected to the server **Action**: The server sends the current board state to the client **Postcondition**: The client and server have identical copies of the state			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client ready -¿ Server send -¿ Client receive			
Assumptions: The data is not corrupted during transmission			
Implementation Constraints: n/a			

Use Case Name: Request Game State from Server		**ID**: RGS	**Priority**: critical
Primary Actor: Waiting Player	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the critical function of the system where the player requests the updated game state from server			
Goal: The client receives the current game state from the server			
Success Measurement: The client and server have the same (byte-for-byte) game state representation, and the client renders the board correctly			
Precondition: The client is connected to the server **Action**: The server sends the current board state to the client **Postcondition**: The client and server have identical copies of the state			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client ready -¿ Server send -¿ Client receive			
Assumptions: The data is not corrupted during transmission			
Implementation Constraints: n/a			

Use Case Name: Request Game State from Server		**ID**: RGS	**Priority**: critical
Primary Actor: Spectator	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the critical function of the system where the player requests the updated game state from server			
Goal: The client receives the current game state from the server			
Success Measurement: The client and server have the same (byte-for-byte) game state representation, and the client renders the board correctly			
Precondition: The client is connected to the server **Action**: The server sends the current board state to the client **Postcondition**: The client and server have identical copies of the state			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client ready -¿ Server send -¿ Client receive			
Assumptions: The data is not corrupted during transmission			
Implementation Constraints: n/a			

Use Case Name: Receive passive message notification		**ID**: RGS	**Priority**: critical
Primary Actor: Acting Player	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the function where the player receives a message about their input from the server			
Goal: The client receives information about the current game state from the server			
Success Measurement: The client has received their message from the server			
Precondition: The client is connected to the server **Action**: The server sends a message to the client **Postcondition**: The client has received the message from the server			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client ready -¿ Server send -¿ Client receive			
Assumptions: The data is not corrupted during transmission			
Implementation Constraints: n/a			

Use Case Name: Receive passive message notification		**ID**: RGS	**Priority**: critical
Primary Actor: Waiting Player	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the function where the player receives a message about their input from the server			
Goal: The client receives information about the current game state from the server			
Success Measurement: The client has received their message from the server			
Precondition: The client is connected to the server **Action**: The server sends a message to the client **Postcondition**: The client has received the message from the server			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client ready -¿ Server send -¿ Client receive			
Assumptions: The data is not corrupted during transmission			
Implementation Constraints: n/a			

Use Case Name: Receive passive message notification		**ID**: RGS	**Priority**: low
Primary Actor: Spectator	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the function where the player receives a message about their input from the server			
Goal: The client receives information about the current game state from the server			
Success Measurement: The client has received their message from the server			
Precondition: The client is connected to the server **Action**: The server sends a message to the client **Postcondition**: The client has received the message from the server			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client ready -¿ Server send -¿ Client receive			
Assumptions: The data is not corrupted during transmission			
Implementation Constraints: n/a			

Use Case Name: Receive (blocking) message notification		**ID**: RGS	**Priority**: critical
Primary Actor: Acting Player	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the function where the player receives a message about their invalid input from the server			
Goal: The client receives information about the current game state from the server			
Success Measurement: The client has received their message from the server			
Precondition: The client is connected to the server and makes an invalid move **Action**: The server sends a message to the client **Postcondition**: The client has received the message from the server			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client ready -¿ Server send -¿ Client receive			
Assumptions: The data is not corrupted during transmission			
Implementation Constraints: n/a			

Use Case Name: Receive (blocking) message notification		**ID**: RGS	**Priority**: critical
Primary Actor: Waiting Player	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the function where the player receives a message about their invalid input from the server			
Goal: The client receives information about the current game state from the server			
Success Measurement: The client has received their message from the server			
Precondition: The client is connected to the server and makes an invalid move **Action**: The server sends a message to the client **Postcondition**: The client has received the message from the server			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client ready -¿ Server send -¿ Client receive			
Assumptions: The data is not corrupted during transmission			
Implementation Constraints: n/a			

Use Case Name: Receive (blocking) message notification		**ID**: RGS	**Priority**: low
Primary Actor: Spectator	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the function where the player receives a message about their invalid input from the server			
Goal: The client receives information about the current game state from the server			
Success Measurement: The client has received their message from the server			
Precondition: The client is connected to the server and makes an invalid move **Action**: The server sends a message to the client **Postcondition**: The client has received the message from the server			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client ready -¿ Server send -¿ Client receive			
Assumptions: The data is not corrupted during transmission			
Implementation Constraints: n/a			

Use Case Name: Send Tentative Valid Move Sequence		**ID**: STVMS	**Priority**: critical
Primary Actor: Active Player	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the critical function of the system where the client sends an expectedly valid move sequence to the server			
Goal: The server receives move sequences from clients			
Success Measurement: The server receives a move sequence from the client			
Precondition: The client gets a move sequence from the user **Action**: The client sends the move sequence to the server **Postcondition**: The server receives the move sequence			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client Move Ready -¿ Server ready -¿ Client send -¿ Server receive -¿ Server send -¿ Client receive			
Assumptions: The data is not corrupted during transmission			
Implementation Constraints: n/a			

Use Case Name: Receive Move Validity Confirmation or Rejection		**ID**: RMVCR	**Priority**: critical
Primary Actor: Active Player	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the critical function of the system where the client receives a validity response after sending a move to the server			
Goal: The client receives a validity response from the server			
Success Measurement: The client receives a confirmation for a valid move, and a rejection for an invalid move			
Precondition: The client sends a move to the server **Action**: The server sends the confirmation or rejection to the client **Postcondition**: The client receives and identifies the response			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Server ready -¿ Client send -¿ Server receive -¿ Server send -¿ Client receive			
Assumptions: The data is not corrupted during transmission			
Implementation Constraints: n/a			

Use Case Name: Resign from the game		**ID**: RGS	**Priority**: critical
Primary Actor: Active Player	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the critical function of leaving the game while it is in progress			
Goal: The game ends, with the opponent declared the victor			
Success Measurement: Only one player remains connected, and a win-state is entered, with that player declared the victor			
Precondition: Both players are connected to the server **Action**: The player drops connection, either intentionally or not **Postcondition**: The game enters a win-state, and the opponent is declared the victor			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client disconnects -¿ Server recognizes lost connection -¿ Opponent wins			
Assumptions: The server recognizes that the opponent has disconnected immediately			
Implementation Constraints: n/a			

Use Case Name: Resign from the game		**ID**: RGS	**Priority**: critical
Primary Actor: Waiting Player	**Source**: primary	**Use Case Type**: core	**Level**: overview
Interested Stakeholders: All			
Brief Description: This use case describes the critical function of leaving the game while it is in progress			
Goal: The game ends, with the opponent declared the victor			
Success Measurement: Only one player remains connected, and a win-state is entered, with that player declared the victor			
Precondition: Both players are connected to the server **Action**: The player drops connection, either intentionally or not **Postcondition**: The game enters a win-state, and the opponent is declared the victor			
Relationships: **Include**: n/a **Extend**: n/a **Depends On**: n/a			
Typical Event Flow: Client disconnects -¿ Server recognizes lost connection -¿ Opponent wins			
Assumptions: The server recognizes that the opponent has disconnected immediately			
Implementation Constraints: n/a			

3 System Overview

3.1 Description of Software

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 stalemate is reached (neither player can perform a move) or one player has no pieces remaining on the board. At this point a winner is displayed (or a draw in the case of a stalemate) and the game concludes.

3.2 Technologies Used

To aid in the development process, as well as provide a robust deliverable, we make use of several technologies and software libraries. Some of these are necessarily part of our design, and others are only used during the development and build process. In the latter case, it is possible to omit these components from the deliverable.

3.2.1 Docker

Docker [1] 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.

Within the context of our Checkers implementation, Docker is instrumental in providing consistent development and deployment environments, reducing development time and costs by allowing us to target one platform while supporting many. In particular, we utilize a static build toolchain with which we compile static binaries that can be distributed to and executed on almost any similar platform. The ideal scenario is that the user(s) would not need to deploy any build environment, compile any software, or otherwise perform any potentially time-consuming steps, and instead can simply copy a precompiled executable with which they can run.

If this is not desirable, or in the event that changes are made to the application source code or for security / trust concerns, a new build environment can be constructed with a trivial amount of work because this infrastructure is *entirely scripted*.

Our usage of Docker is particularly interesting. Imagine the following scenario:

1. A Docker image `alpine:checkers` is created based on the `docker/Dockerfile` file in the project tree, containing all of the necessary tools and development libraries. This image is built using the command `make image` (which does some further processing).

2. Any time a developer or user performs some action that requires this development environment, such as building the game, testing the source code, etc., they spawn an ephemeral instance of the image, called a *container*, which is run by default with a flag `--rm` which causes the container to be deleted when it is exited. Not to worry, the containers are designed to be temporary.

3. As soon as a container is spawned, the main project directory is mounted inside the container to a configurable location. Inside the container, the user now has full access to the project source. However, instead of dropping the user into a shell, which is possible via the `make envir` command, the top-level `Makefile` executes various commands inside the container *on behalf of* the user. The user doesn't even realize that these actions are being performed inside the container's consistent development environment.

4. The developer (or user, if desired) can simply run `make game` or `make check` to either build the game's static executable(s), or check the source code and memory for errors. Because we mount the local source tree to the container instead of copying the source code or cloning it from the development repository, the static `x86_64` executable is built directly into the `bin/` directory, ready for distribution.

Our main `Makefile` operates in the manner outlined in fig. 1.

3.2.2 Alpine Linux

The primary Docker image that we will use is based off of Alpine Linux [3], a secure, minimal Linux distribution that targets embedded systems and low-resource platforms in an effort to maximize efficiency. For context, the minimal Alpine image is just `5MB`, and it utilizes *MUSL* [4] and *BusyBox* [5] to provide efficient and correct C/C++ libraries and standard utilities.

3.2.3 ncurses

We make use of a software library *ncurses* [6] for the implementation of our graphical user interface, which is of course console-based, in other words running under a terminal emulator such is standard practice today for most Unix-based

```
    $ make usage
    Usage:

      make <target>

    Targets:

      usage         prints this text then exits
      image         builds the devel. Docker img.
      envir         spawns/enters devel. envir.
      clean         remove intermediate files
      check         runs   various tests on code
      game          builds the Checkers(tm) game
      run-loc       runs   the game on this mach.
      run-con       runs   the game in Container
```

Figure 1: Primary `Makefile`

systems. In effect, many of the platform-specific changes that we would need to support can be safely accounted for by relying on *ncurses* to handle these differences.

3.2.4 VNC

The software *noVNC* [8] is a utility that enables web-based connections to a VNC server, which is one way to share a graphical display remotely over a network. This will enable us to provide a web-based interface to our game in the event that we cannot natively compile the application into WebAssembly [9], which is to be determined.

4 Design Considerations

Upon our client's request, we must develop a networked Checkers implementation. This is not necessarily a problem per se, but by choosing to implement the application in the C programming language, we must be careful and thorough in providing a robust deliverable. Our solution must account for unexpected network troubles such as dropped packets or corrupted data.

As we are using Docker for our development and potentially the execution environment, we make the guarantee that our software will function correctly within this predefined environment, and we aim to design our software to be compatible with external platforms but cannot make such guarantees as to its efficacy. We do not plan to develop for other platforms, but we will account for the potential change of plan by allocating some slack time to the matter.

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 2: Generic System Requirements

4.1 Assumptions and Dependencies

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.

4.1.1 Hardware

fig. 2a 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.

4.1.2 Software

fig. 2b 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.

4.2 Dependencies

Most or all of the software dependencies are directly satisfied or satisfied by equivalent packages in the first stage of the project prototype. These are bundled into the base Docker image that we will use; all such discussion shall occur in separate Docker-related documents, if applicable.

4.3 General Constraints

To more comfortably fit our design and development of our software into the time constraints imposed by our client, we define a set of constraints on the project. In effect, detailing further the project scope in areas not yet covered.

4.3.1 End-User Environment

The **Game** is being developed with the intention of running inside of Docker, so that the actual User Environment is always controlled, and will function as expected. Thus there are no constraints on the user system, other than an ability to run the Docker software.

4.3.2 Availability and Volatility of Resources

The resources that are utilized for the project are all open source and thus are available for anyone to use. These resources are thus susceptible to being changed causing future releases of the tools to no longer work with our program. Release version specifications are contained in the requirements documentation.

4.3.3 Standards Compliance

The **Game** will meet all IEEE's standards for software of this nature.

4.3.4 Interoperability

The **Game** will not have the ability to interact with external programs or systems.

4.3.5 Data Repository / Distribution

The data is currently hosted within a Git repository. Once the project reaches completion, users will be able to find and receive the code and documentation from a public portion of the Git repository. The way users will run the code is through Docker a platform agnostic program that will let user run to the code on any machine that supports the program.

4.3.6 Verification, Validation, and Testing

We will utilize a unit-testing framework for the C language called *check* [1] and a memory checking tool called *valgrind* to ensure that our code is safe and correct. Our top priority beyond implementing all of the agreed-upon features is to do so without creating the possible of a software, hardware, or network crash.

4.3.7 Other Quality Controls

To maintain high quality software, after every major revision, the group redoes all the testing for the sections to make sure components that have been working, stay working as well and to verify the status of previously nonworking portions of the software.

[1]more details in `requirements.pdf`

4.4 Goals and Guidelines

We aim to achieve the satisfaction of our client by providing the highest quality service possible, focusing on the following areas:

1. The product must look, feel, and behave similarly to how a typical person would play the classic Checkers board game. This means, avoiding unnatural language or pop-up dialogs that could detract from the game experience.

2. Since the players will be the bottleneck in the game, it is not critical to design for speed beyond being able to support the minimum hardware and software specifications outlined in section 4.1.

4.5 Design Rationale

4.5.1 Docker

Our team has decided to make use of the Docker program because it allows users on the range of operating systems from Windows, macOS, and Linux to all be able to have an identical experience when they are interacting with our Checkers program. Another consideration is that with a singular platform testing can be done much easier as their is no potential for unique errors to appear.

4.5.2 Separate Client/Server Modules

Our team has decided on implementing separate Client and Server programs, as opposed to making a singular program that could function as both Client and Server. This decision was made in order to keep the codebase clean and concise. Another consideration was the ease of use for the Player. They do not need to have access to any of the Server functions, and giving them more options to choose from, and making the game more complex to set up, and less intuitive to use goes directly against our outlined ideology for this system.

4.5.3 1x32 Boardstate Storage

There are many different ways to represent the boardstate, but in the end, we chose to use a 1x32 array. In this array contains a single *Piece* in each space, and each index of the array represents a single location on the board. There are 32 spaces accessible by the pieces, which are numbered from left to right, and top to bottoms, starting with 1, and going to 32. Keeping the array this size allows us to avoid excessively long calls to different 2D locations, and instead keep everything accessible with a single integer index, which is much simpler and easier to read. It also allows us to only use the minimum amount of locations, so it is more difficult for a piece to get to somewhere that it shouldn't be able to.

5 Architectural Strategies

5.1 Choice of Language

As a reasonable software engineering firm, our team has decided, with the approval of our client, to implement our solution using the C programming language. Grateful for the flexibility that our client is giving us, we hope to gain a better understanding of the tools, software libraries, and design principles involved in using such a language, in that we can apply these techniques and experiences to future work.

5.2 Planning for Future Development

Our implementation of the Checkers game aims to be functionally complete, with only minor possible future enhancements. We will need to take into account the feasibility of porting our product to mobile devices, web browsers, and potentially other platforms but should not focus on creating an ideal one-size-fits-all product.

5.3 User Interface Paradigms

Console-based games typically use a computer keyboard for input, though may additionally accept mouse input. We will restrict our product to accept only keyboard input. Common interfaces, or a wrapper, could be written to make it easier to adapt the product to accept other forms of user input, but this is outside of the scope until further amendments to this document exist.

5.4 Error Detection and Recovery

Due to the short-lived nature of a typical Checkers game session we will avoid worrying about corrupted disks, memory, or other hardware components. The critical aspect of our product is to ensure that at least two connected clients (players) have a seamless experience during their gameplay. As such, it is imperative that we develop straightforward, clean, and robust networking logic and code to handle any unforseen troubles gracefully.

At the end of the day however, as this is not a mission-critical application, we need not spend an unreasonable amount of effort in this area if it means sacrificing the quality of other portions of our product.

5.5 Memory Management

On a similar note as in section 5.4, our product is not inherently a memory-intensive application, and within our guaranteed Docker-based testing platform any memory that is “leaked” is automatically cleaned up upon the termination of the application. The third-party libraries we use may also leak memory beyond our control.

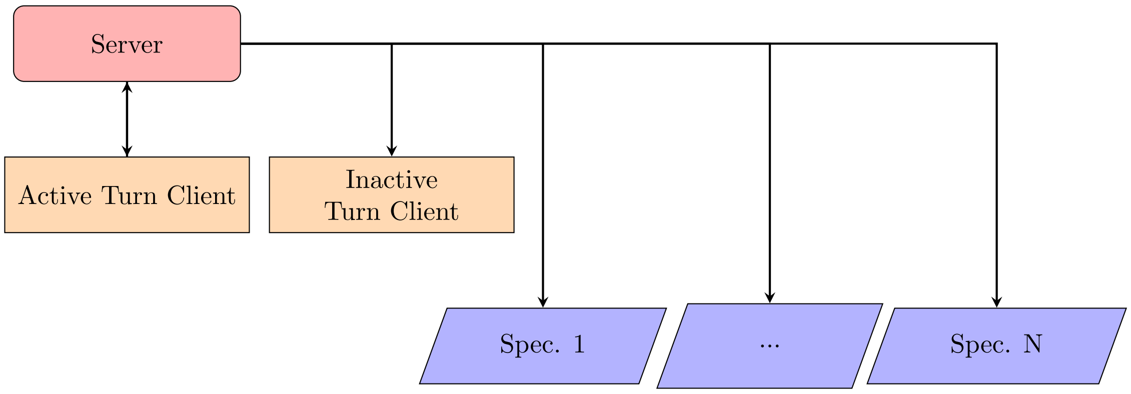


Figure 3: Server-Client Hierarchy, Spectators Optional

We will make every reasonable effort to account for memory leaks but cannot guarantee that our product will be free of memory leaks.

5.6 Concurrency and Synchronization

We will employ a thread-based model to ensure that any remote client is able to connect without the connection dropping due to an unlikely but possible scenario where the server is processing a previous request in that exact moment. By taking this approach we can also accept, rather trivially, additional (third, fourth, etc.) clients to serve as *spectators*.

Outside of the networking module, concurrency need not be used.

5.7 Containerization and Deployment

Deployment of our product (including, but not limited to testing and automated builds, distribution of the compiled executable(s) and other critical components of the software development process) shall be guaranteed with a core set of constraints.

6 System Architecture

fig. 3 shows an overview of the components and their relation to one another. Details for each component will be discussed further.

6.1 Core Module

The *core module* of our product will be included in all major components of the final deliverable(s), including but not limited to: *client*, *server* modules. The purpose of the *core module* is to consolidate necessary elements and to avoid code repetition, documentation redundancy, and reduce the potential for future

```c
#define N_ROWS      8

/* total pieces */
#define N_PCES      ((N_ROWS - 2) * (N_ROWS / 2) / 2)

typedef
struct game_piece
{
    int player : 1;
    int kinged : 1;
} piece;

typedef
struct game_board
{
    piece *board[N_ROWS][N_ROWS / 2];
} board;

```

Figure 4: `board.h`

debugging troubles. The core module of Checkers will contain all game logic, graphical components, and all logic that links the Server and Client modules. This is to ensure that the program will remain in sync and in the case that the program gets out of sync allow for easy correction to occur at that the time of the issue occurring. The primary role of the Server will be to serve as the mediator between the clients and will serve as the final as the final confirmation in making sure moves are being handled properly through the use of the core module. The clients themselves will be the way the players and spectators interact with and monitor the game board respectively. For the players the core module does move validation to show what moves they have available and handles all of their inputs.

6.1.1 Requisite Libraries

Using Alpine Linux `3.6` as a base Docker image, several additional packages are required in order to build the libraries mentioned in the `requirements.pdf` document.

Once these dependencies are satisfied, we can build the following libraries and tools, described in `requirements.pdf`:

- ncurses-6.0
- check-0.11.0
- freetype-2.8

```
RUN apk update && apk add --no-cache \
    bash      \ # shell
    cmake     \ # build tool
    coreutils \ # provides 'nproc' for parallel
       builds
    g++       \ # C++ compiler
    gcc       \ # C   compiler
    make      \ # build tool
    sed       \ # for text replacement
    upx       \ # binary packer; not currently used
    vim       \ # text editor for debugging
    wget        # generic CLI web client
```

Figure 5: Primary `Fragment of Dockerfile`

- valgrind-3.13.0
- cppcheck-1.79

6.1.2 Game Logic

The **Game** will store the *Boardstate* as a 1x32 array of pieces. This matrix represents the 32 spaces on the standard checkerboard that are reachable by the pieces. By using basic addition or subtraction (depending on the type of piece, and which way it can move) we can calculate the locations that each piece can move to, and where they are able to jump to. This allows us to take that information, and compare it with the current *Boardstate*, to generate all of the available moves for each piece. In order to do this efficiently, we take the index (i) of the piece that we want to check for moves on, and perform $i\%8$.

This is done because there are 2 distinct row types; those that start with a blank space on the board, and those that start with an occupied space. Luckily, the first two rows represent both of these types, and therefore can be used as a basis to calculate moves for the rest of the board as well. We know that a piece j in the first row can move to either $j + 4$ or $j + 5$, and a piece k in the second row can move to either $k+3$ or $k+4$. Using this information, we can add these integers to the original indices, before modulation, to obtain the moves available for any space on the board. For moving in the other direction, the same method is used, but the distances for the first row are -3 and -4, and for the second row are -4 and -5.

After the possible moves are found, the game checks which pieces are in each space that the selected piece can move to, and determines whether or not it is able to make that move, in the current gamestate.

The way that jumps work, is by first finding the available moves and then taking the available move in the same direction as the first move. The game

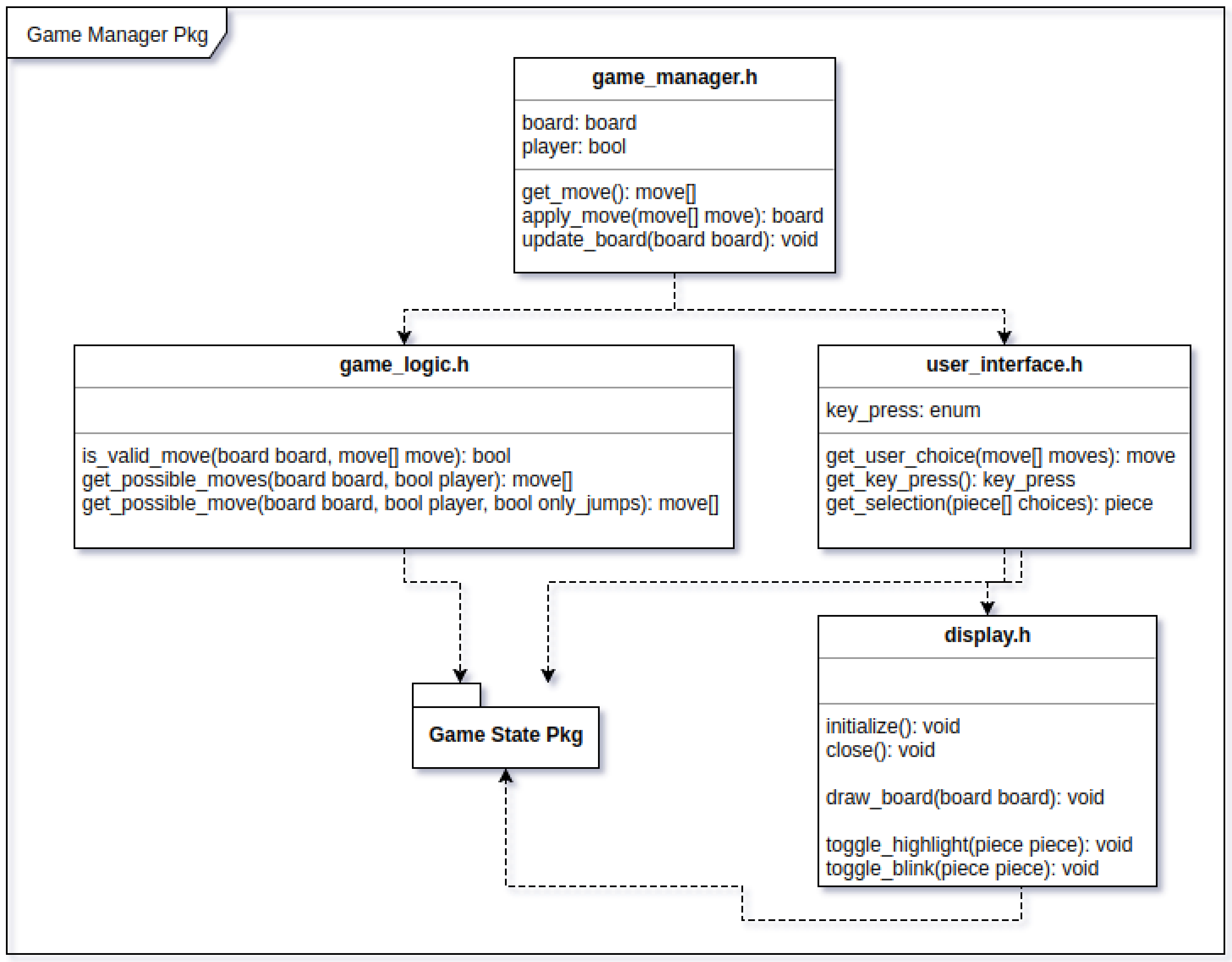

Game Manager Pkg
game_manager.h
board: board
player: bool
get_move(): move[]
apply_move(move[] move): board
update_board(board board): void
game_logic.h
is_valid_move(board board, move[] move): bool
get_possible_moves(board board, bool player): move[]
get_possible_move(board board, bool player, bool only_jumps): move[]
user_interface.h
key_press: enum
get_user_choice(move[] moves): move
get_key_press(): key_press
get_selection(piece[] choices): piece
display.h
initialize(): void
close(): void
draw_board(board board): void
toggle_highlight(piece piece): void
toggle_blink(piece piece): void
Game State Pkg

Figure 6: Game Manager

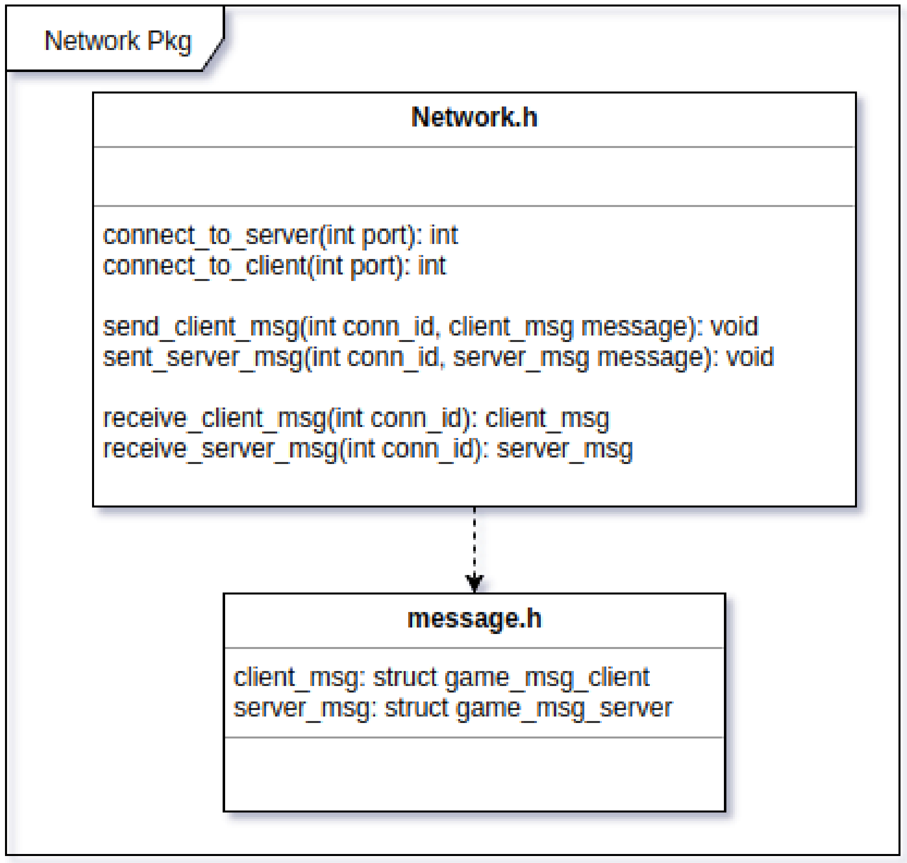

Network Pkg
Network.h
connect_to_server(int port): int
connect_to_client(int port): int
send_client_msg(int conn_id, client_msg message): void
sent_server_msg(int conn_id, server_msg message): void
receive_client_msg(int conn_id): client_msg
receive_server_msg(int conn_id): server_msg
message.h
client_msg: struct game_msg_client
server_msg: struct game_msg_server

Figure 7: Network Module

will then check the first move location to see if it contains a jump-able piece, and then the final jump location to confirm that it is empty.

6.2 Networking Module

The `Networking` module contains all functionality required for the client and server applications to communicate with each other. This includes methods for establishing and closing connections, as well as the sending and receiving of messages.

network.h	
connect_to_server(int port): int	Connects to an existing server using the provided port.
connect_to_client(int port): int	Accepts a connection request from a client using the provided port.
send_client_msg(int conn_id, client_msg message): void	Sends a `client_msg` to the server associated with the `conn_id`.
send_server_msg(int conn_id, server_msg message): void	Sends a `server_msg` to the client associated with the `conn_id`.
receive_client_msg(int conn_id, client_msg message): void	Accepts a `client_msg` from the server associated with the `conn_id`.
receive_server_msg(int conn_id, server_msg message): void	Accepts a `server_msg` from the client associated with the `conn_id`.

6.3 Server Module

The `Server` module contains all functionality required to interface with multiple clients, manage a master *Boardstate*, and ensure that the game is played without error, according to the rules detailed in the requirement document. This module will be broken down into three main components:

6.3.1 Network Communication Component

This component will provide all functionality required to accept communications from multiple clients, and place them into a game together. It will also handle sending and receiving *sequences of moves* from the clients that are in the game, and sending *Boardstates* and messages to these clients, to communicate necessary actions with the Players.

6.3.2 Game Logic Component

This component will manage the **Game**, including the storage of the *BoardState*, and validation of Player moves. This will contain all of the functionality required to validate moves given to it by the Client. Although the Player will only be able to select from a list of available moves that are pre-validated by the Client, re-validating these moves on the Server allows us to make sure that everything is synchronized between the Players, and adds a security measure against any possible cheating.

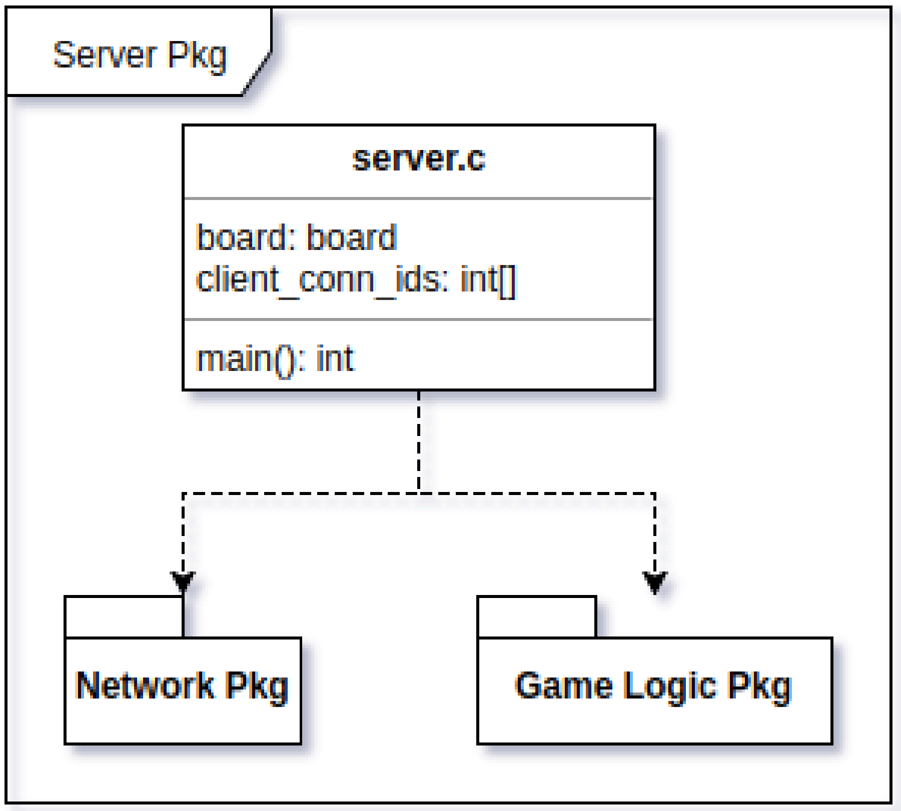


Figure 8: Server Module

game_logic.h	
is_valid_move(board board, move[] move): bool	Evaluates a sequence of moves against the board. `move` will contain a single move in the case of a normal move, and a sequence of moves in the case of a sequence of one or more jumps.
get_possible_moves(board board, bool player): move[]	Evaluates each piece of the provided player for available moves, returning a list of all 1^{st} generation moves. If a jump can be made, only jumps will be returned.
get_possible_move(board board, piece piece, bool only_jumps): move[]	Evaluates all moves an individual piece can make. If `only_jumps` is `true`, then only moves consisting of a jump are returned.

6.4 Client Module

The `Client` module contains all functionality required to interface with the player, and relay content to and from the server. This module will be further broken down into four main components:

6.4.1 Graphical Display Component

This component will provide the functionality to display information to the user. It will use the *ncurses* library to create a visualization of the *Boardstate*, and show messages from the Server.

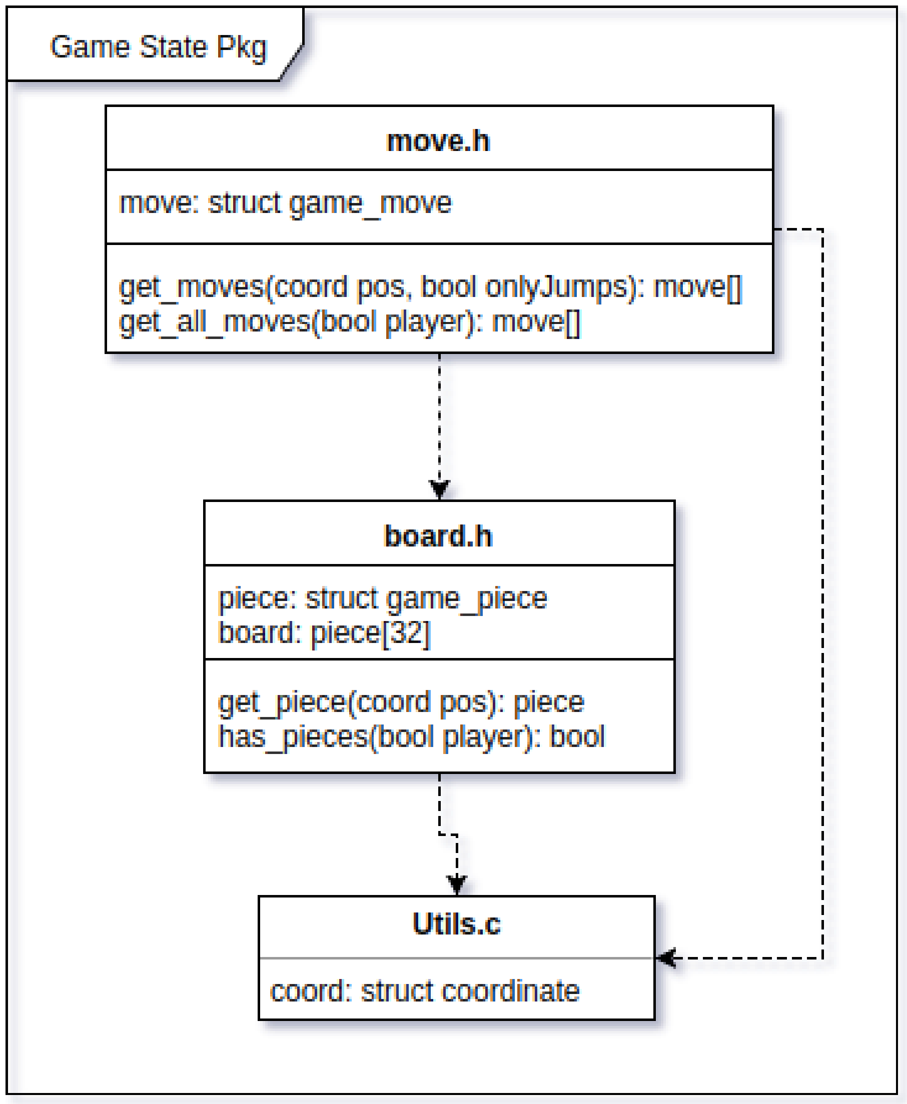

Game State Pkg
move.h
move: struct game_move
get_moves(coord pos, bool onlyJumps): move[]
get_all_moves(bool player): move[]
board.h
piece: struct game_piece
board: piece[32]
get_piece(coord pos): piece
has_pieces(bool player): bool
Utils.c
coord: struct coordinate

Figure 9: Game State Module

```c
#define MAX_MOVE    12

typedef
struct coordinate
{
    int row;
    int col;
} coord;

typedef
struct game_move
{
    coord from;
    coord dest;
} move;

typedef
struct move_sequence
{
    move moves[MAX_MOVE];
} mseq;

```

Figure 10: `move.h`

display.h	
initialize(): void	Configures the terminal to `ncurses` display mode.
close(): void	Frees the terminal from `ncurses` display mode.
draw_board(board board): void	Renders the board to the screen.
display_msg(string msg, bool is_blocking): void	If `is_blocking` is true, renders the message as a popup over the board, requiring the enter key to dismiss. If `is_blocking` is false, renders the message at the top of the window, over any previous message.
toggle_highlight(piece piece): void	renders or de-renders an inner border on the space containing the piece.
toggle_blink(piece piece): void	Renders or de-renders an inner blinking border on the space containing the piece.

6.4.2 Player Input Component

This component will provide the functionality required to receive input from the Player, in order to play the **Game**. This includes the logic behind the selection of moves that are generated from the **Move Assistance** component.

user_interface.h	
key_press: enum	Enum for the possible keys the user may press to advance the game.
get_user_choice(move[] moves): move	Utilizes the `get_selection` method to get the initial piece of a move. Repeats this process to get the destination of the move.
get_key_press(): key_press	Helper method to get keyboard input from the user.
get_selection(piece[] choices): piece	Displays available pieces the player may move, allowing the player to use the `left` and `right` arrow keys to change the selected piece, until the user confirms with the `enter` key. Upon confirmation, the selected move is returned.

6.4.3 Move Assistance Component

This component will contain all the necessary features to determine which moves and jumps the Player can (or must) make.

game_manager.h	
board: board	Stores the current `BoardState`
player: bool	Identifies which player the client is playing
get_move(): move[]	Utilizes `game_logic` and `user_interface` components to generate possible moves and get user's selection
apply_move(board board, move[] move): void	Applies the given move sequence to a board.
update_board(board board): void	Updates the stored `BoardState` with the provided board

6.4.4 Network Communication Component

This component will handle all interaction between the Client and Server modules. This will include receiving Boardstates, sending pre-validated moves, and receiving messages for the Player to view.

7 Policies and Tactics

7.1 Choice of Compiler

In accordance with the minimum software specifications outlined in fig. 2b, we can make certain guarantees about the tools required to build our product. Using the Docker platform to provide a consistent build environment, we assert that we will prioritize support for the GNU C Compiler (GCC) using a certain set of compiler flags and standards.

7.2 Coding Conventions

Naming Conventions:

- Constants in all capital letters
- Underscores to delimit words in struct, function, and variable names
- Pointers will be declared with the * near the variable name, and not the pointer type
- Global variables will be prepended by a 'g_'

```
# In this case, $CC is the default inside container;
# configured within the Dockerfile, if desired.

# FLAG - linker options (use static and PIC)
# LIBS - required link line to build the project

FLAG := -static -fPIC -flto
LIBS := -lncurses -lmenu -lc

game:
    $(Q)$(RUNNER) \
        -v $(PROJECT_ABS):$(MOUNTS) \
        -it $(IMAGES) \
        $(CC) $(FLAG) \
        -o $(MOUNTS)/$(OUTFILE) \
        $(SRCFILE) \
        $(LIBS)
```

Figure 11: Game `Makefile` Target

Commenting standards:

- All functions will have a comment describing the input, output, and any parameters that are being passed in
- Comments will be placed with any code that may not be immediately obvious in their functionality

Overall formatting guidelines:

- Code will be kept to 72 char width max
- Braces will done in the Allman style [2]. This style puts the brace associated with a control statement on the next line, indented to the same level as the control statement. Statements within the braces are indented to the next level
- Four(4) spaces will be used as a level of indentation
- Global variables will be avoided
- All If, While, and Do statements will use braces, even if there is only a single line in the braces
- There will be a single space between keywords (such as If, and While) and parentheses

```
CS451-Checkers
├─ docker
├─ docs
│  ├─ cleveref
│  ├─ datetime2
│  │  └─ samples
│  └─ hyperref
│     ├─ doc
│     └─ test
├─ spec
└─ src
   ├─ fonts
   ├─ game
   └─ network
      ├─ common
      ├─ client
      └─ server
```

Figure 12: Source Code Organization Tree

- Constants shall be placed on the left side of equalities and inequalities
- Each line shall contain one statement
- Each variable declaration will have it's own line

7.3 Source Code Organization

The source code is organized as shown in fig. 12

7.4 Issue Tracking

Using GitLab for our repository gives us the ability to use the built-in issue tracking system. This system allows us to create Issue tickets, that contain a title and description, to fully document the issue that is observed, and allow others to recreate the issue for testing purposes. These tickets can be assigned to a specific person on the team, or left open for anyone to work on, as they are available. A ”Due Date” is also able to be set, for critical systems that needed to be fixed by a certain time. Once a ticket is submitted, it is available to view by everyone associated with the project, and can be placed into a ”To Do” or ”Doing” list, depending on the current status of the issue.

While Issues are still open, it is possible for other members of the project to leave a comment on the ticket, so that the team can collaborate on fixing the issues, without everyone having to work on it at the same time.

From the Issue screen, a merge request can be created, so that the code to fix the problem is directly associated with the ticket, and easy to search for. Once the problem is fixed, the ticket is closed, and placed on the ”Closed” list, where it is still available to view. This allows us to keep a log of what went wrong, how it was fixed, and the exact code that was used to fix it.

7.5 Unit Testing

The methodology behind our tests is treating every board interaction like it is occurring on a physical checkerboard. The goal is to test every situation that a player can expect to find themselves in a standard game of checkers, as well as any situations that occur specifically when someone is playing a virtual game of checkers like a player trying to move pieces off the virtual game board or leaving the game idle for extended periods of time. The way we can ensure that the tests are good is by making sure that we do identical tests that are completely equivalent for both players to make sure we don’t run into a case where one player can do something the other player would be unable to do.

7.6 Continuous Integration

The way that the we are automating testing is by creating different board states in separate files for testing. By utilizing these files, the group will be able to see what items are currently working and which are currently broken through our unit testing.

7.7 Build Instructions

Simply run the command listed in fig. 13.

8 Miscellaneous

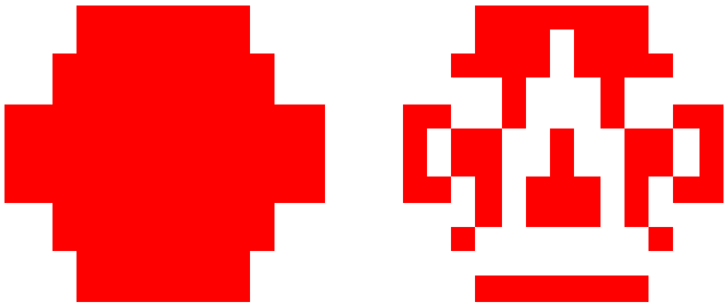
Most of the components within section 6 should be discussed in more detail, which may include additional low-level subcomponents. This section will provide additional details and/or follow-up discussion on these matters.

8.0.1 Board Display

The checker pieces are 13 pixels wide, by 6 pixels tall (keeping in mind that the rows of a terminal window are twice the size of the columns). fig. 14 shows the

```
$ make game
make[1]: Entering directory '/CS451-Checkers/docker'
sudo docker run --rm \
        -v /CS451-Checkers:/home \
        -it alpine:checkers \
        cc -static -fPIC -flto \
        -o /home/./bin/out \
        ./src/fonts.c ./src/main.c ./src/game.c \
        -lncurses -lmenu -lc
[sudo] password for me:
make[1]: Leaving directory '/CS451-Checkers/docker'
```

Figure 13: Game `Makefile` Target



(a) Piece (b) Crown

Figure 14: Checkers Piece Design

Figure 15: Game Logo

design of the basic and `Kinged` pieces. Given these sizes, the board size can be calculated. There will be a one pixel border around the edge of the window. The top row will be reserved for the display of messages. Therefore the terminal size needed to display the **Game** is `106px by 102px`.

9 Glossary

Below is a comprehensive list of some of the terms and language that we use within this document, knowledge of which will lead to a more effective understanding of our product's design and aid in future communications regarding our product.

- **Checkerboard** NxN (typically 8x8) game board composed of alternating black/red (or other) squares on which game pieces *Checkers* reside
- **Piece** standard Checkers disc 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
- **Crowning**, the act of changing a standard game piece to a *King*

10 References

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

[2] Allman Indentation style, available online at
`https://en.wikipedia.org/wiki/Indent_style#Allman_style`

[3] Alpine Linux Development Team, *Alpine Linux*, available online at
`https://alpinelinux.org/`

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

[5] Anderson, E., et al., *BusyBox: The Swiss Army Knife of Embedded Linux*, online at
`https://busybox.net/about.html`

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

[7] Information on coding standards, and reasoning behind each standard.
`https://users.ece.cmu.edu~enocodingCCodingStandard.html#classnames`

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

[9] 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`

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