L-Università ta' Malta
Faculty of Information & Communication Technology
Department of Computer Science

CPS2008 – Operating Systems and Systems Programming 2

Practical Assignment for the Academic Year 2020/21

Xandru Mifsud

Typeset on June 27, 2021 using LaTeX and some patience.

∅ Introduction

The aim of this assignment was to familiarise ourselves with networking and multi–threading principles, building on top of the material covered in *Operating Systems and Systems Programming 1*. To this extent, we use whenever possible principles of signal handling, memory management, etc.

This document is divided into three principle components:

- Chapter 1 covers the outline of the implementation, outlining in particular the principles of networking and thread–safety used and the design decisions made in that regard.
- Chapters 2 and 3 outline the core component of the assignment, where the bulk of the work is concerning networking and multi–threading. We go into detail with regards to message passing, game session creation and P2P communication for the purposes of illustration, demonstration of understanding, and discussing the rationale behind certain design decisions. We also chose to outline P2P communication as that is the major area which we later identify as needing improvement, with the testing section demonstrating a number of shortcomings.
- Chapters 4 and 5 detail the testing carried out, along with any discovered limitations (both during implementation and testing) and possible future improvements. These go hand in hand with the video demonstration.

We choose to focus this report on *concepts* rather than in–depth specifics about the implementation, given that the project has a substantial amount of code. In this manner, we hope that this report conveys our level of *understanding* of the subject, complimenting the in-line documentation detailing the code (on an almost line–by–line basis).

1 Repository Details, Installation, and Usage Details

The code is intended to run on Linux platforms, in particular on Debian–based systems such as Ubuntu and Lubuntu, on which we have tested the implementation with relative success.

1.1 `CPS2008_Tetris_Server`

The repository is available at github.com/xmif1/CPS2008_Tetris_Server.

Requirements

For installation, `cmake` version 3.17+ is required.

Installation Instructions

Clone the repository, and `cd` into the project directory. Then run:

1. `cmake .`
2. `make`

Usage Instructions

Simply `cd` into the directory containing the compiled executable, and run `./CPS2008_Tetris_Server`.

1.2 `CPS2008_Tetris_Client`

The repository is available at github.com/xmif1/CPS2008_Tetris_Client.

Requirements

For installation, `cmake` version 3.17+ is required.

Installation Instructions

Clone the repository, and `cd` into the project directory. Then run:

1. `cmake .`
2. `sudo make install`
3. `sudo ldconfig`

where the last two instructions install the shared library on your system, ready for use in particular by any front-end implementation.

1.3 `CPS2008_Tetris_FrontEnd`

The repository is available at github.com/xmif1/CPS2008_Tetris_FrontEnd.

Requirements

For installation, the following are required:

1. `cmake` version 3.17+
2. `ncurses`
3. The aforementioned shared library installed on the client system.

Installation Instructions

Clone the repository, and `cd` into the project directory. Then run:

1. `cmake .`
2. `make`

Usage Instructions

Simply `cd` into the directory containing the compiled executable, and run:

```
./CPS2008_Tetris_FrontEnd <server_ip>
```

where `server_ip` is a required argument specifying the IPv4 address in dot notation of the server.

1 Implementation Outline

We begin by outlining the core components of our implementation, and how they interact with each other. The project is naturally partitioned into three components,

- A server–side application that handles, amongst other things, communication between clients via the chat functionality and setting up of new game sessions.
- A client–side library that handles communication with the server, while providing a public facing API for the implementation of a front–end.
- A client–side front–end making use of the aforementioned library.

The first major design decision concerns the interaction between the client library and the front–end: while the library provides thread–safety using mechanisms that will be outlined later on, at no point does it create, join or cancel a `pthread`. Rather, it exposes a number of mutexes along with thread–safe getters and setters for shared variables, such that the front–end implementation can implement and handle multi–threading. The reason for this decision was in relation to `pthread` operations being prone to undefined behaviour on, for example, raising of signals. By consolidating thread creation and handling to a single entity, this generally made debugging easier.

The second design decision is related to the front–ends graphical user interface. To this extent, we decided to use this opportunity to familiarise ourselves with `ncurses`. It quickly became apparent, however, that `ncurses` is not inherently thread–safe[1]. The terminal window is divided into two areas, one for the live–chat and another for game–play, all within the same terminal window.

It is worth mentioning from the get–go that when looking at the commit log for the front–end, one observes that initially we attempted to update the chat and game–play areas simultaneously using two separate threads. However, due to the aforementioned lack of thread–safety in `ncurses`, this lead to undefined behaviour[2]. After realising this, any updates to the `ncurses` `WINDOW` data structures were consolidated to the main thread, and carried out sequentially rather than simultaneously (without any noticeable impact of the game's refresh rate). The changes carried out between the multi–threaded and single–threaded variant of the front–end can be observed in commit c68ba16.

As expected, information about clients, game sessions, etc, are represented appropriately using structs. Information is shared between clients and the server using a *tagged* message. The type of message (eg. chat, command, etc), along with client status (such as whether they have joined a game, etc) and game mode are respectively represented as enum types, for convenience and more importantly consistency. These enums, along with the tagged message passing system, define the mechanism by which communication occurs, and will be outlined in the next chapter.

Lastly, we note that the Tetris logic and graphics implemented were (to a very large extent) derived from Stephen Brennan's implementation, available at https://github.com/brenns10/tetris. The major modifications required were with regards to the various game modes and what defines game termination. Whenever such changes were carried out, an appropriate comment was added, beginning with the prefix `@xandru`. Kindly feel free to search the front–end source for this prefix, to make it easier to identify modifications to Stephen's source. The original license is included along with the source.

[1] One valuable lesson learned here is to always check whether libraries are thread–safe, whether they provide a thread–safe specific API, etc.

[2] We suspect that this is due to having two separate threads trying to access the same file descriptor for writing to the console. Initially, a number of wrappers were written around the used `ncurses` functions, however this quickly became convoluted and difficult to debug.

1 Thread Safety Considerations

We expand further on our thread safety considerations, which we have already began discussing. Most of the data maintained is within arrays, with elements being modified by independent threads. We shall then discuss *atomicity* in the sense of implementing granular thread–safety across arrays, as well as in the sense of having threads carrying out atomic transactions[3].

Consider for example the `clients` array maintained by the server. Upon connection, a new `client` instance is created and stored at the first available index in the `clients` array. Upon doing so, a new thread is created, exclusively to handle communication received from the client, which in turn may require modifying the data held in the client instance.

Suppose that we maintained a `pthread_mutex_t` across the entire `clients` array. Upon, for example, completion of a game session, the internal data of each `client` instance that joined the game session must be updated with, in particular, whether they have won or lost the game. This will then create a situation where up to 8 separate threads (one for each respective client) instantaneously request a lock on the same mutex, resulting in unnecessarily less performant operations – this is because each thread will modify a *different* element from the `clients` array.

Instead then, we maintain a `pthread_mutex_t` for each `client` instance, allowing whenever possible such operations to occur simultaneously. We represent these mutexes as an array, with the mutex at index i corresponding to the client at index i in the `clients` array. The same principle is applied to other arrays (such as those maintaining game session information) whenever possible.

```
client* clients[MAX_CLIENTS];
pthread_mutex_t clientMutexes[MAX_CLIENTS];
```

An illustrated example of the above is given below. In this case, the snippet demonstrates thread–safe searching of the `clients` array for the next free index. It is taken from the `add_client` function defined in `server.c`.

```
int i = 0;
for(; i < MAX_CLIENTS; i++){
    pthread_mutex_lock(clientMutexes + i);
    if(clients[i] == NULL){
        break;
    }
    else{
        pthread_mutex_unlock(clientMutexes + i);
    }
}
```

There are also a number of instances where we may wish to cancel a `pthread` (say, due to the raising of a `SIGINT` – we illustrate later on how we handle graceful termination whenever possible). In this regard, we make use of `pthread_setcanceltype` and `pthread_setcancelstate`, which are outlined in the `manpages`. In particular, we occasionally make use of `PTHREAD_CANCEL_ASYNCHRONOUS` as appropriate. In this case, we optionally use `PTHREAD_CANCEL_DISABLE` and `PTHREAD_CANCEL_ENABLE` to restrict cancellation during the execution of only a particular portion of the threaded code.

This offers control over the atomicity of transactions, and acts as a guarantee for correctness. For a detailed example, kindly see the `accept_peer_connections` function in `client_server.c`.

[3]This is a principle typically applied in databases: either a sequence of instructions is executed in its entirety, or none at all, ensuring data correctness.

2 Networking Considerations

Networking was exclusively handled using `sockets`, with the *domain* set to `AF_INET` (to expose IPv4 communication protocols) and the *type* set to `SOCK_STREAM` (to ensure data arrives in the order that it is sent, especially important considering that events such as chat messages must appear in order).

As mentioned earlier, in the next chapter we shall delve into the manner by which we encode and decode data sent across the network, such that different systems running different OSes interpret the received data in the same manner. We shall also treat in later chapters the mechanisms by which game sessions and P2P communication are established. What is key in P2P communication is that we are able to allocate a unique port number to each client joining the game, and before any client attempts to join another client in the game session, a synchronisation mechanism is required to ensure first that each has successfully opened a socket on which to accept connections, at the allocated port. As we shall later see, this initial synchronisation is carried out via the server.

Effort was made such that server or client disconnection is handled as gracefully as possible by all parties. To this extent, we firstly carry out the necessary error checking on calls to `read` and `send` on a socket. If they return erroneously, with the error number indicating disconnection, suitable handler functions are called.

For example in `server.c`, the `remove_client` function is typically called when a call to `send` returns a negative number (indicating disconnection; a non–negative number indicates the number of bytes sent, with zero being no data sent which may possibly occur after eg. a timeout). It handles, amongst other things, ensuring that the file descriptor associated with the client's connection is closed, the thread servicing the client is cancelled (and consequently any associated mutex locks are released), and the respective entry in the `clients` array is freed.

In a P2P setting, clients acknowledge disconnection between each other in the same manner. For simplicity, the P2P implementation we consider is a fully–connected mesh, with no communication between clients via intermediary ones. It should be noted that if three clients (A, B and C) are connected in a fully–connected mesh, and A disconnects from B but remains connected to C, then C has no means of knowing that A disconnected from B. Albeit very unlikely (as typically A is either connected to all, or else the cause for disconnection is such that it disconnects from all clients in the mesh), this is still an implementation limitation which should be addressed in the future, as it may technically lead to unfair scenarios during game play (with one player playing against more clients than the other).

Here is an example illustrating these disconnection principles, from the `service_peer_connections` function in `client_server.c`. If a client fails to connect to another client (i.e. an invalid file descriptor is provided), then we close any file descriptors associated with that client (the file descriptor on which they act as a client, and that on which they act as a server).

```
1  if(client_server_fd < 0){
2      pthread_mutex_lock(clientMutexes + i);
3      gameSession.clients[i]->state = DISCONNECTED;
4
5      if(gameSession.clients[i]->client_fd > 0){
6          close(gameSession.clients[i]->client_fd);
7      }
8      gameSession.clients[i]->client_fd = 0;
9
10     if(gameSession.clients[i]->server_fd > 0){
11         close(gameSession.clients[i]->server_fd);
12     }
13     gameSession.clients[i]->server_fd = 0;
```

```
14
15      pthread_mutex_unlock(clientMutexes + i);
16  }
```

2 Message Tagging, Passing, and Handling

In this chapter, we illustrate the mechanism by which data is shared between clients and the server. This mechanism defines the only manner by which data is shared across a network, both for client–server and client–client communication, and irregardless on whether the data being shared is some non–public–facing state information (eg. flags to signal some event such as client disconnection) or public–facing data (eg. a chat message).

1 Representing Messages: `msg` struct & `MsgType` enum

We begin by first describing how we maintain messages throughout the project. A message is encapsulated in a `msg` struct which maintains a `MsgType` enum type and a `char` pointer to a textual representation of the data content.

The `MsgType` is used to represent the type of message, which consequently establishes how to handle the message. In some cases, the `MsgType` is simply used to flag some event, with the data part being discarded for example. We briefly outline these various flags below:

- `INVALID` (-2) : Indicates that the message is invalid (eg. due to disconnection while the message was being read). The data part is generally ignored in this case.
- `EMPTY` (-1) : A message with an empty data part. Typically the default return of a function with a `msg` return type (think of it as the null type equivalent for messages).
- `CHAT` (0) : A message whose data part is a message to be displayed in the live chat (if sent by the server) or was typed in the chat box (if sent by a client).
- `SCORE_UPDATE` (1) : A message whose data part is a score of a client during an active game session, sent by the client to the server.
- `NEW_GAME` (2) : A message which signals a client to setup a local game session, with the data part including all the game options and the networking configuration to setup the P2P network (in the case that the game is a multi–player one). This is outlined in more detail later on.
- `FINISHED_GAME` (3) : A message sent by a client to the server to signal that they have successfully completed and closed the local game session.
- `P2P_READY` (4) : A message sent by a client to the server to signal that they have successfully setup the specified socket on which P2P connections are to be accepted, and are indeed ready to accept P2P connections.
- `CLIENTS_CONNECTED` (5) : No longer in use but left for the sake of explanation. Kindly see **Chapter 5** *Section 1*.
- `START_GAME` (6) : A message sent by the server to all clients in a game session, whenever it has received a `P2P_READY` message from all the clients in the game session.

These last 5 flags in particular hint at the P2P setup procedure, which we shall outline in further detail later.

2 Sharing Information over a Network: Encoding and Decoding

While reviewing the literature, we observed that when sharing for example an `int` over a `socket`, what is sent is not necessarily what is received, in the sense that the receiving system may for example be using a different endianness. To this extent, we restricted ourselves to strictly communicating encoded strings over `socket` instances, with a header specifying amongst other things the number of bytes to follow. The header is such that it is of a fixed number of bytes – in this manner, we **always** know how many bytes are expected.

A message takes the following format,

`"<msg_data_len>::<msg_type>::<msg_data>"`

where:

- The data length is in bytes (i.e. characters in the string representation).
- We always ensure that the data string is always null terminated and hence the null character is always included in the data length.
- The number of digits that the message length may have is `MSG_LEN_DIGITS`, i.e. the maximum length of the data string is $10^{\texttt{MSG\_LEN\_DIGITS}}$ characters. If the textual representation of the length requires fewer digits than `MSG_LEN_DIGITS`, it is padded by the required number of zeros in the beginning.
- The message type requires only a single byte and corresponds to a flag defined in the aforementioned `enum` type.

The initial portion, `"<msg_data_len>::<msg_type>::"`, constitutes the header of the message and is always of length `MSG_LEN_DIGITS` + 5. We typically however consider the header as a separate string from the rest of the message, and hence we require an additional byte for the null terminating character. In this case we consider the header length to be `MSG_LEN_DIGITS` + 6, and is the definition used for `HEADER_SIZE`.

The encoding procedure is fairly straight–forward, involving no more than the usual string manipulation operations typical to `C`. If `str_to_send` is the resulting string representing the encoded message and `str_to_send_len` is the corresponding string length, we ensure that the message is sent entirely and in the correct order as follows:

```
1  int tbs; // tbs = total bytes sent
2  int sent_bytes; // on each iteration, tbs += sent_bytes
3
4  // loop until total bytes sent (tbs) is equal to the number of bytes (chars) of the
   ↪  encoded message, or until disconnection
5  for(tbs = 0; tbs < str_to_send_len; tbs += sent_bytes){
6      if((sent_bytes = send(socket_fd, (void*) str_to_send + tbs, str_to_send_len -
       ↪  tbs, 0)) < 0){
7          break; // disconnection
8      }
9  }
```

Decoding a message is also fairly straight forward, knowing the header structure, however it is worth outlining the mechanism by which we receive messages such that we ensure that they are received in their entirety:

1. We begin by ensuring that we read `HEADER_SIZE` - 1 bytes from the respective socket by repeatedly calling `recv` with the required remaining number of bytes. If disconnection occurs at some point, this can be handled in a number of ways[1].
2. We then decode the header, converting the expected message data length to an integer.
3. Lastly, we repeatedly call `recv` once again, until the expected number of bytes are received, handling disconnection as before.

This is outlined in the snippet below.

```
int ret; // maintain the number of bytes returned by recv
int tbr; // tbr = total bytes read
int recv_str_len; // the expected message data part length
char header[HEADER_SIZE]; header[HEADER_SIZE - 1] = '\0'; // char array for header

// begin reading message starting with header
if((ret = recv(socket_fd, (void*) &header, HEADER_SIZE - 1, 0)) > 0){
    // if header not entirely read, continue calling recv until the entire message is
    ↪ received; note that since we are using TCP, bytes are streamed in the order
    ↪ they are sent
    for(tbr = ret; tbr < HEADER_SIZE - 1; tbr += ret){
        if((ret = recv(socket_fd, (void*) (&header + tbr), HEADER_SIZE - tbr, 0)) <
        ↪ 0){
            // handle disconnection
        }
    }

    // extract string representation of the data part length from the header
    char str_len_part[5]; strncpy(str_len_part, header, 4); str_len_part[4] = '\0';

    // convert to an integer, and extract the message type flag from the header
    recv_str_len = strtol(str_len_part, NULL, 10);
    int msg_type = header[6] - '0';

    // allocate enough memory for fetched data part
    char* msg = malloc(recv_str_len);

    // continously call recv until the entire data part of the message is received or
    ↪ until disconnection
    for(tbr = 0; tbr < recv_str_len; tbr += ret){
        if((ret = recv(socket_fd, (void*) recv_msg.msg + tbr, recv_str_len - tbr, 0))
        ↪ < 0){
            // handle disconnection
        }
    }
} // else{ handle disconnection}
```

[1]For example, in the client library this functionality is handled in the `recv_msg` function which returns a struct of type `msg`. In the case of disconnection, the message type is set to `INVALID` and then handled by the function caller (i.e. the front–end). On the other hand, in `server.c`, receiving and decoding messages is the primary task of the threaded function `service_client`, which handles a particular client's requests. In this case, disconnection leads to a call of the `remove_client` clean–up function, and termination of the `service_client` thread associated with the disconnecting client.

3 Message Handling

As alluded to before, messages can signal some internal event between connected clients, be a live–chat message, etc. The behaviour of the *recipient* depends primarily on the type tag of the message. In general, on receipt and decoding of a message,

- In a switch statement, whose cases correspond to the values of the `MsgType` enum, we determine which *handler* function to call to handle the message and its data contents.
- Execution of the handler function then determines how the behaviour of the recipient changes based on the message type.
- Each handler function must return to the caller whether it has finished in an erroneous state; if this is the case, this generally leads to disconnection.

In some instances, the message type on its own does not suffice. For messages sent by a client to the server via the chat box (i.e. those tagged as `CHAT`), we must establish whether the user has inputted one of the reserved keywords. In this case, we first call the handler for the `CHAT` message type, which determines what further handlers to call depending on whether the first word in the data part is a reserved keyword (eg. `!nickname`).

For the sake of brevity, we shall not detail the handlers here (as for the most part, many of them are self–explanatory, and do not necessarily offer us the opportunity to demonstrate networking and multi–threading principles learned). However they have all been well documented with in–line comments, and we encourage the reader to check them out. The top–level handlers (i.e. those that distinguish between message types) start with the prefix `handle_`, while any lower–level handlers in the case of `CHAT` messages start with the prefix `sfunc_` (short for *server functionality*).

4 Communication between P2P Clients

The mechanism described above is not exclusively used for client–server communication, but also for client–client communication over a P2P network. The same principles of tagging, encoding, decoding, and handling apply. The only difference is that the `MsgType` enum defined in the client library is extended over that defined in the server implementation, to support additional message types that may be exchanged only between clients.

There is in fact, so far, only one such additional message type:

- `LINES_CLEARED` (7) : A message sent by a client in a game of `RISING_TIDE`, to all the clients in the same game session, whenever the sender has cleared one or more lines from the Tetris board. The data part includes the number of lines cleared.

3 Game Session Creation and P2P Setup

Having outlined the mechanism with which data is shared, we are now in a position to discuss how a game session is created. We shall concern ourselves with multiplayer instances, as this offers the best opportunity to outline the threading and networking principles discussed in class.

We begin by outlining the procedure for starting a new game session, before discussing in further detail certain aspects of the implementation. Kindly refer also to the flow chart overleaf, in conjunction with the outline below.

1. First, a user types into the chat box the appropriate command along with the parameters.
2. The library encodes this message, tagging it as `CHAT`, and sends it to the server.

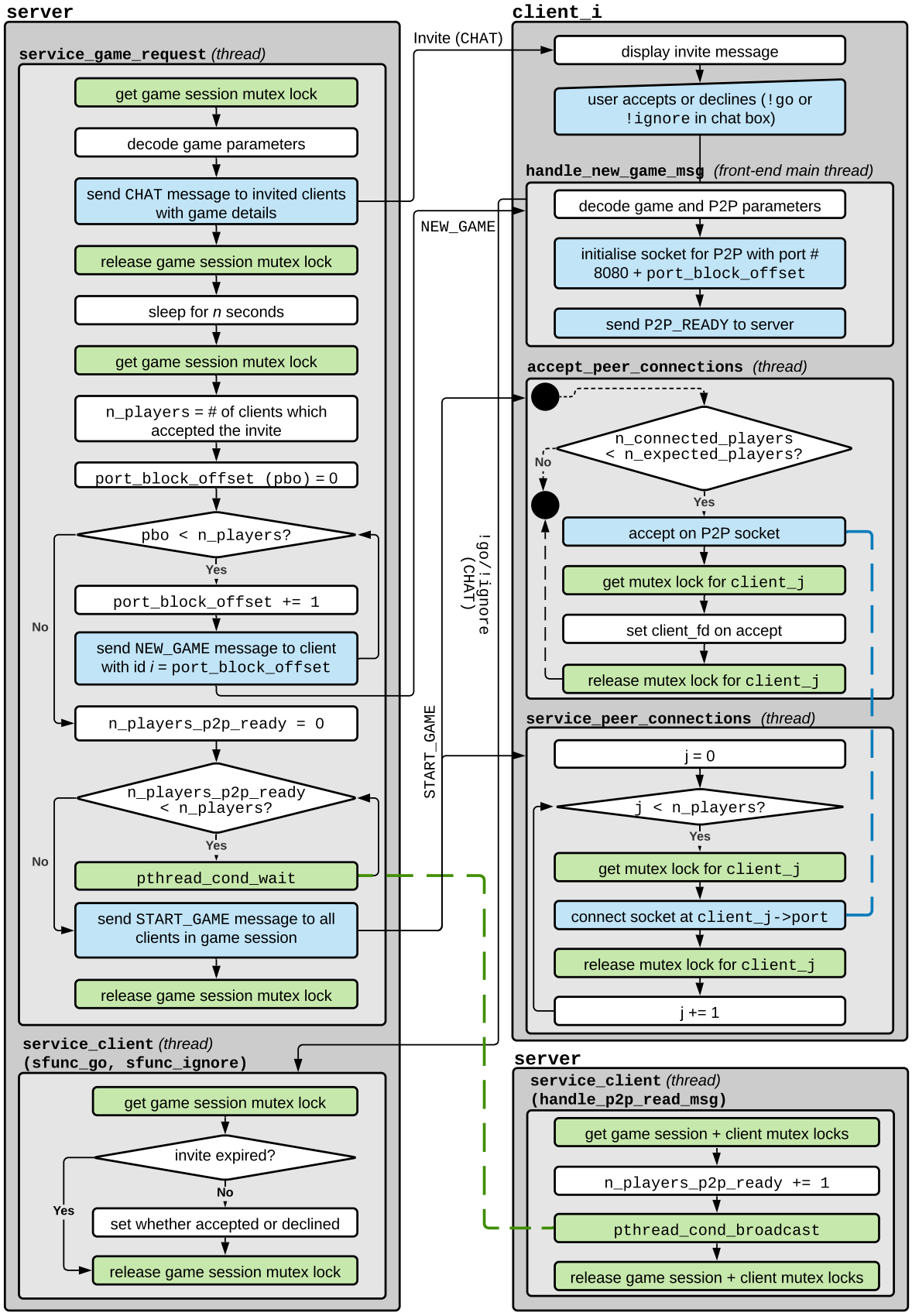

Figure 3.1: Flow chart diagram illustrating interactions between the server and the clients while setting up a game session and a P2P network. Note that only the relevant excerpt is shown of the `accept_peer_connections` function thread (the black dots denote the start and end of the relevant flow). Processes shaded in blue denote some *networking* operation, while those in green denote *threading* operations.

3. The server decodes the message, recognises that it starts with one of the game session commands, and calls the appropriate message handler. This is all handled by the `service_client` thread associated with the client.

4. The handler in particular decodes the game settings from the message, along with the usernames of invited clients, and checks the validity of everything. If some option or username is invalid, an appropriate message is sent to the client.

5. The server then creates a new thread to handle the game session setup. This is primarily done so as not to hang the client's thread while for example waiting until the 30 second invitation expiration time passes, or waiting while the clients setup a P2P network between them. This allows the client to continue sending chat messages in the meantime.

6. For all those clients that accepted the game invite, the server sends a `NEW_GAME` message with the necessary information for the client library to setup the game session and open the specified socket on which P2P connections are to be accepted. Once this is done, a `P2P_READY` message is sent by the client to the server.

7. Once the server has received a `P2P_READY` message from all clients in the game, a `START_GAME` message is sent to all the clients in the game. The front–end handles this appropriately (in this case, by initialising a game of Tetris – but the server and client library are agnostic of this). Moreover, the thread created by the server to handle the game session terminates.

8. During the game, clients communicate between each other via the P2P network. For the most part, the only data that is actually shared between clients is the number of lines cleared per move in a game of Rising Tide.

9. Every second, each client in the game sends a `SCORE_UPDATE` message to the server, which is handled by the respective `service_client` thread.

10. When a client finishes the game, a `FINISHED_GAME` message is sent to the server and any of the currently connected clients in the P2P network.

1 The `service_game_request` Thread

As mentioned, whenever the server receives a request to setup a new game session, once all the parameters have been verified (including player availability), a thread is launched to handle the game session setup separately, such that the client's thread does not hang while waiting for the game invite to expire, the clients to set up a P2P network, etc. This is the `service_game_request` thread.

Before launching the thread, we ensure that a new game session can be started, and if yes we initialise a new `game_session` struct in the `games` array. The corresponding index is passed to the `service_game_request` thread, so that not only can it access the struct, but also obtain the necessary mutex from the `gameMutexes` array (using the locking mechanism on arrays outlined earlier).

In the case that the game type is a multiplayer game session, an appropriate message (`request_msg`) specifying the game parameters is sent to each client invited to the game session. Once this is done, the mutex lock is released and the thread sleeps for a specified amount of time, during which clients may either accept or decline the request. Accepting or declining a request involves modification to the `game_session` struct by a client, hence why the mutex lock is important:

- Not all game request messages are sent to the invited clients simultaneously, hence you could have the game struct changing while invitations are still being sent.
- Moreover, as we shall see, whenever each client accepts or declines, we must first obtain the mutex, since each accepts or decline is handled by the client's respective thread i.e. we could otherwise have simultaneous reads and writes on the same struct which could lead to data corruption without the mutex.

After the invitation expires, the `service_game_request` thread continues execution, by first obtaining the mutex lock on the `game_session` struct as before. A check is carried out to ensure that a sufficient number of clients have accepted the invite, and if so a `NEW_GAME` message is sent with the game parameters and P2P settings. If n clients are to join the game (including the client that initiated the session), then the invite takes the following form (excluding the header of the message):

```
"<game_type>::<n_baselines>::<n_winlines>::<time>::<seed>::<port_block_offset>::<client
_1_ipv4>::<client_2_ipv4>:: . . . ::<client_n_ipv4>"
```

Most of the parameters above are self–explanatory. The `seed` parameter is a randomly generated integer value which is sent to all clients in the game session and will be used as a seed to a linear congruence generator. In this manner, all clients generate the same sequence of tetrominoes.

The `port_block_offset` is an integer which is added to the default port `8080` so as to obtain a distinct port number for each client. We allocate this by enumerating the clients in the game session from $1 \leq i \leq n$, and giving client i the port offset i, i.e. the resulting port number on which they will open a socket to accept P2P connects will be $8080 + i$. The list of IPv4 addresses that follow specifies the clients to which to connect in the P2P network. The offset is also used to prevent a client from connecting to itself (the IPv4 address at position i for the client with `port_block_offset` i is the IPv4 address of the client itself). This prevents any *'feedback loop'* situations that bombard the client with messages.

1.1 Maintaining the Number of `P2P_READY` Messages Received

The `START_GAME` message is only sent once all the clients joined in the game session have successfully opened a socket on which to accept P2P connections by other clients, and hence have sent to the server a `P2P_READY` message.

To maintain the number of such messages received, the `game_session` struct maintains the `int` variable `n_players_p2p_ready`, which is initially set to 0. Before continuing, the `service_game_request` thread must wait until `n_players_p2p_ready == n_players`, by continuously checking the value in a while loop.

```
1 games[game_idx]->n_players_p2p_ready = 0;
2 while (games[game_idx]->n_players_p2p_ready < n_players) {
3     pthread_cond_wait(&(games[game_idx]->p2p_ready), gameMutexes + game_idx);
4 }
```

Simultaneously, clients may send a `P2P_READY` message to the server, in which case their respective `service_client` thread will increment `n_players_p2p_ready` by 1. We must then ensure thread–safe access to `n_players_p2p_ready`. In order to achieve this, the `game_session` struct maintains a `pthread_cond_t` type conditional variable called `p2p_ready`. With each iteration of the while loop above, we call `pthread_cond_wait` on this conditional variable and the mutex associated with the `game_session` struct. This results in a block until a `pthread_cond_broadcast` call is made on the conditional variable.

In doing so, another thread (in this case, a `service_client` thread whenever a `P2P_READY` message is received) may obtain the mutex to the `game_session` struct, increment `n_players_p2p_ready` by 1, then call `pthread_cond_broadcast` on the conditional `p2p_ready` and release mutex. In this manner, the while loop in `service_game_request` thread may continue execution.

```
1 int handle_p2p_read_msg(char* chat_msg, int client_idx){
2     pthread_mutex_lock(clientMutexes + client_idx);
3
4     // some code that carries out a check that the client that sent the P2P_READY
      ↪  message is indeed in the game session
```

```
5
6       (games[clients[client_idx]->game_idx]->n_players_p2p_ready)++;
7       pthread_cond_broadcast(&(games[clients[client_idx]->game_idx]->p2p_ready));
8
9       // some more code
10
11      pthread_mutex_unlock(clientMutexes + client_idx);
12
13      return 0;
14  }
```

2 Client Game Creation

Once the `NEW_GAME` message is received by a client, the library function `handle_new_game_msg` is called. Firstly, the function decodes the string above by means of tokenisation and carrying out the necessary type conversions. The respective `game_session` struct[1] is populated, in particular the port for each client on the P2P network is calculated during decoding, as described above. Each client instance in the game session has an associated state, the possible values of which are outlined below:

- `WAITING` : Waiting for the P2P network to be set up (i.e. client has not connected to all other clients in the mesh).
- `CONNECTED` : Client has connected to all other clients in the P2P network.
- `FINISHED` : Client has finished the game.
- `DISCONNECTED` : Client has unexpectedly disconnected and hence left the game session.

Initially, every client instance in the game struct is set to `WAITING`. Once the game struct is initialised, the socket on which P2P connections are to be accepted is created. If socket initialisation is successful, a `P2P_READY` message is sent to the server.

3 P2P Setup: The `accept_peer_connections` and `service_peer_connections` Threads

The client library provides two threaded functions for P2P setup, one intended to accept incoming connections and subsequently handle messages received, and the other intended to request connections in order to setup a bi–directional P2P mesh network. These two threaded functions are the `accept_peer_connections` and the `service_peer_connections` respectively.

Note that the front–end in fact runs the `service_peer_connections` thread in the main thread, since at the moment there is no need to run it in a separate thread. However, in case we add further handshaking requirements for P2P setup, it may be the case that the function would have to run in its own thread, hence it was designed with that in mind.

3.1 Requesting P2P Client Connections

The `service_peer_connections` threaded function is fairly straight forward: for each pair of IPv4 addresses and port specified in a `NEW_GAME` message, a call to `client_connect` is made. The `client_connect` function is a utility function provided in the API for opening a new TCP socket for a given IPv4 address and port.

[1] Here we are referring to the `game_session` struct as defined in the client library, and not the server implementation – the two differ slightly in the information they maintain.

For each client we attempt to connect with, we check if a valid socket was returned, and if not we disconnect gracefully in a thread–safe manner. In fact, in **Chapter 1** *Section 2: Network Consideration*, we outlined graceful disconnection, using a code snippet from the `service_peer_connections` threaded function. Kindly refer back to that section as necessary.

Since no handshake mechanism is in place between the clients, as outlined in the limitations section later on, this may result in a one–directional connection instead of a bi-directional one, as one party may not successfully manage to connect with the other.

3.2 Accepting and Handling Communication from P2P Clients

The `accept_peer_connections` threaded function is responsible for accepting connections on the socket opened by the client for accepting P2P connections, and subsequently handling messages from all the clients connected. Note that we do not handle received messages from clients until all clients expected to join the game session have successfully connected.

In this case, unlike the server, we do not handle messages received from clients by having a separate thread for each client. The reason for this is two–fold:

1. Client systems may not cope well under load from multiple threads, unlike a server which is typically designed to service multiple clients. Hence we try to maintain the number of spawned threads to a minimum on client machines.
2. Little to no messages are actually passed between P2P clients during game play. Indeed, in all game modes the only messages passed are disconnection or completion messages. The only exception is the `RISING_TIDE` game mode, where whenever a client clears a line from the board, a `LINES_CLEARED` message is sent to the P2P clients.

Rather, we used this as an opportunity to familiarise ourselves with the `select` operations on sockets and how to use a `timeval` struct for a timeout on a `select` statement, to avoid indefinite waits.

For each iteration of the indefinite while–loop, which terminates on disconnection or game completion, we begin by constructing an `FD_SET` consisting only of the file descriptor associated with the socket on which we are accepting P2P connections. We then iterate through all the clients registered in the game session (but not necessarily connected yet to the client in question on the P2P network), and if connected we add the associated file descriptor to the `FD_SET`.

This is followed by a call to `select` on the `FD_SET`, with a 1 second timeout. If the timeout expires, we simply begin a new iteration. In doing so, this offers the opportunity to cancel the thread in an atomic manner (refer to our discussion on atomic transactions in the opening chapter), since within the while–loop we make use of `PTHREAD_CANCEL_DISABLE` (i.e. if we were to use a blocking `select`, this could potentially lead to the thread indefinitely hanging and not allowing for graceful termination).

Two cases may then occur,

1. If the file descriptor associated with the socket on which we accept P2P connections is set and the number of connected P2P clients is less than the expected number, there is a new P2P client waiting to connect and hence we accept the connection. In the meantime, we continue ignoring any messages received from the connected P2P clients.
2. Else when the number of connected P2P clients is equal to the number of expected clients, we begin to accept messages from the P2P clients whenever a client file descriptor is set after the call to `select`. In this case we fetch, decode, and handle the message in a similar manner to that outlined in the previous chapter.

4 Testing

We outline here a number of tests carried out, in conjunction with the video–demonstration. The video–discussion is important in this regard, as it demonstrates some of these tests in action (which we believe makes them come across better). The tests are grouped into classes, based more or less on what concept is being tested.

1 Abrupt Disconnection

In this section we investigate disconnection of either the server or a client, with a test passing if there are no crashes and state across all connected clients is correct.

Test 1 Disconnection of a client (while *not* in a multi–player game).

Expected Server and the other clients should continue operating normally, with the other clients receiving a notification in their live chat window that a client has disconnected (including the client's nickname). The disconnecting client's terminal session should be restored to its initial state (whether the client was simply chatting or playing a single player game), i.e. `ncurses` clean–up routine should restore terminal correctly.

Pass/Fail **Pass**

Actual As expected. See video demonstration as well.

Test 2 Disconnection of a client (while in a multi–player game).

Expected What holds for Test 1 must also hold here. Furthermore, the multi–player game session should continue as normal (indicating that the clients in the game session have successfully disconnected on the P2P network from the disconnecting client). The disconnecting client has not finished the game, and hence should not send a `FINISHED_GAME` message i.e. should not be included in the final leaderboard.

Pass/Fail **Pass**

Actual As expected. See video demonstration as well.

Test 3 Disconnection of all the clients in a multi–player game session.

Expected What holds for Test 2 must also hold here for each individual disconnecting client. Furthermore, the server should recognise that the game has finished, but not due to completion, without any `FINISHED_GAME` message received. Consequently, the game ID (corresponding to an index in the `games` array) should be available for re–use.

Pass/Fail **Pass**

Actual As expected. See video demonstration as well.

Test 4 Disconnection of the server.

Expected All clients should disconnect, with their terminal sessions restored to its initial state and any P2P networks dismantled.

Pass/Fail **Pass**

Actual As expected. See video demonstration as well.

2 Connection and Re–Connection

In this section we investigate whether clients can successfully connect to the server and with each other over a P2P network (and possibly re–connect if starting another game session between the same clients), with a test passing if there are no crashes or indefinite waiting for connections.

Test 5 Connecting a client to the server (given sufficient server resources available).

Expected Front–end should display the live–chat window, with the server sending a welcome message including the client's randomly generated nickname. Typing `!players` should display the client as available to join a game session, while typing `!playerstats` should display that the client has 0 wins and losses and no high score (remember that we do not have any data persistence – see Limitations and Future Improvements section).

Pass/Fail **Pass**

Actual As expected. See video demonstration as well.

Test 6 Connecting a client to the server (given **in**sufficient server resources available).

Expected Server should disconnect immediately from the client, prompting the client's front–end to terminate.

Pass/Fail **Pass**

Actual As expected.

Test 7 Server initialisation of network connectivity.

Expected Server should successfully open a socket on its public IP at port `8080`, ready to accept client connections. A successful initialisation message should be displayed.

Pass/Fail *(Partial)* **Pass**

Actual The test sometimes fails in a specific scenario: Whenever the server crashes, if we attempt to re–start it immediately afterwards, the operating system might not have released the port `8080`. Hence during initialisation, a socket binding error is thrown. Note that this scenario is not always realisable. See the Limitations and Future Improvements section for more details on the attempts to mitigate the issue. This is also discussed and highlighted in the video demonstration.

Test 8 Client P2P setup and connection.

Expected Each client in a game session should open a socket at their public IP and the specified port by the port block offset allocated in the `NEW_GAME` message. A full mesh network should then be setup between the clients in the game session. If this is achieved correctly, then in a game of *Rising Tide* between say 4 players (as tested), whenever a player clears a line, the others have their playing field shifted up by one line.

Pass/Fail *(Partial)*[1] **Pass**

Actual The test sometimes fails in one of two specific scenarios:

- Whenever a client joins a multi–player game session shortly after they have finished another multi–player game session, if the server happens to allocate to the client the same port block offset as before, a socket binding error is thrown or the clients hang indefinitely attempting to setup the P2P network. Note that this scenario is not always realisable under any tested OS.
- Otherwise, all clients might setup their P2P network successfully and send a `P2P_READY` message, which are subsequently received by the server. However the server, even if the required number of `P2P_READY` messages is received, doe not send a `START_GAME` message in the client–server handshake. We believe that this issue is related to a deadlock

[1] We are considering this test to be a *partial pass* only because consistent failure was only replicated on Mac OS X, which is not the target system. Hence beyond our fragile implementation, there might also be some portability issues in that case.

scenario arising from our use of `pthread_cond_wait` and `pthread_cond_broadcast`. Note that this scenario is not always realisable under Lubuntu/Ubuntu, however we have managed to consistently replicate it in Mac OS X.

See the Limitations and Future Improvements section for more details on the attempts to mitigate the issues, further discussions, etc. These are also discussed and highlighted in the video demonstration.

Test 9 On successful completion of a multi–player game, the P2P network is gracefully closed and clients can join another game session.

Expected Any sockets opened to receive P2P connections and subsequent sockets for connected clients on the P2P network are correctly closed. Furthermore, the `game_idx` is set to `-1` for all clients still connected to the server after game termination, indicating that they can join a new game session.

Pass/Fail **Pass**

Actual As expected.

3 Communication Protocols

In this section we investigate whether the server and clients can successfully send and receive messages correctly, encoding and decoding as explained in previous sections, whether all clients in a P2P network successfully connect with each other, etc...

Test 10 Messages encoded and decoded correctly by sender and recipient respectively.

Expected Given that the sender and recipient are connected and remain so while a message is being sent, the recipient should receive first a correctly encoded header, from which they can in particular decode the message type and how much bytes will follow for the data part.

Pass/Fail **Pass**

Actual As expected. Also see the video demonstration and the sample output of messages received by the server, showing the encoding.

Test 11 P2P network is always a full mesh network.

Expected Given the list of n clients in the `NEW_GAME` message, every client should be connected to all the other $n - 1$ clients.

Pass/Fail *(Partial)* **Pass**

Actual On rare occasions, it appears that two clients do not successfully connect with each other on the P2P network, even if they know each others IPv4 address, allocated port number by the server, and have both sent a `P2P_READY` message to the server (indicating that they are indeed accepting P2P connections). We suspect that this might be related either to the port block offset, or since we primarily tested on a single system. Note that both clients, albeit not connected with each other, still successfully remain connected with the server (and possibly other clients). We evaluate further on this in the Limitations and Future Improvements section. It is however worth noting that when we (briefly) tested across two machines on the same network, the issue did not occur.

Test 12 Stress testing message sending and receiving.

Expected With two clients connected, each sending 100 messages per second across the network via the server, no issues should arise.

Pass/Fail **Pass**

Actual As expected. Indeed, the test passed for even more messages per second, however near the 1000 messages per second (i.e. sends with a 1ms sleep in between) the front–end was struggling to refresh. In the original implementation of the front–end, where we attempted to

update the `ncurses` `WINDOW` instances from separate threads, this stress test resulted in garbled data being written to the console, such that the terminal session could not be restored after exiting. It was due to this stress test that we decided to integrate all the updating of the screen in the main thread.

Test 13 Communication between separate networked machines.

Expected The behaviour described in the prior tests should also hold between separate networked machines, at a minimum on the same local network.

Pass/Fail **Pass**

Actual As expected. Note that we tested with two machines on the same network: One machine ran Ubuntu[2] and hosted the server along with a single client, while the other ran Lubuntu and hosted 3 clients.

4 Commands, Maintaining Correct State, etc...

In this section we investigate whether the commands and their options are interpreted correctly, whether correct state is maintained in a multiplayer environment (hinting at correct use of thread–safety principles) whenever data modification or access occurs as a result of an operation, etc...

Throughout these tests, which focus on data modification and access, we especially look out for any deadlock situations resulting from mutex locks being held indefinitely.

Test 14 All clients in a multiplayer game session have the same sequence of tetrominoes.

Expected The generated sequence of tetrominoes for each client is the same, depending on the given seed by the server.

Pass/Fail **Pass**

Actual As expected. See video demonstration as well.

Test 15 The score updates continuously sent to the server by a client during game play are correct.

Expected The front–end of the client sends a score update to the server every second, while the client may update this score at any move during game play. The thread–safe getter and setter provided by the client library should ensure that this value is not corrupted.

Pass/Fail **Pass**

Actual As expected. See video demonstration as well.

Test 16 The `!gamestats` command displays correct and current information on (at most) the top–three (non–zero) scoring players for all or a specified active game session.

Expected Multiple clients may be continuously sending score updates to the server, resulting in constant modification of each game session's respective top–three non–zero[3] rankings. Correct multi–threading principles should ensure that no data is corrupted and hence the scores displayed by the `!gamestats` command are correct and current.

Pass/Fail **Pass**

Actual As expected.

Test 17 The `!leaderboard` command displays correct and current information on (at most) the top–three (non–zero) scoring players for each game mode.

[2]It is worth noting that on this machine, pressing the backspace key was populating the input buffer with garbled data. We believe that this might be some issue in our use of `ncurses'` `wgetch` function. However, it is worth mentioning that we did add handling for all the backspace keys – indeed in Lubuntu, this is a non–issue. Possibly OS or terminal dependent?

[3]We only display those players who have scored at least 1 point, as initially when all the players have 0 points, there is no fair ranking.

Expected Multiple sessions may terminate during access to the leaderboard struct, at any given moment. Correct multi–threading principles should ensure that no data is corrupted and hence the top–three scoring players for each game mode displayed by the `!leaderboard` command are correct and current.

Pass/Fail **Pass**

Actual As expected.

Test 18 The `!playerstats` command displays correct and current information on the high score, number of wins and number of losses, for each currently connected client.

Expected Correct multi–threading principles should ensure that no data is corrupted and hence the statistic displayed should be correct and current.

Pass/Fail **Pass**

Actual As expected.

Test 19 On game completion, the player statistics and leaderboards are updated correctly.

Expected The number of wins and losses for each player are updated accordingly, along with the leaderboard if any players beat the top three scoring players in the leaderboard. Single player sessions should not modify the number of wins and losses (since one could inflate their win/loss ratio).

Pass/Fail **Pass**

Actual As expected.

Test 20 On receiving multiple game invites, a client can accept at most one, and after the invite expires for the other game sessions, the client is considered as having declined to join. Hence a client is only attributed to a single game session.

Expected In particular, any further `!go` commands sent by the client should display an appropriate error. Moreover, the other game sessions should not expect a `FINISHED` message from the client for them to terminate, and no client in the declined game sessions attempts to connect to the client in question via P2P.

Pass/Fail **Pass**

Actual As expected.

Test 21 Usernames are always unique, including on the use of the `!nickname` command.

Expected In a thread–safe manner, auto–generated and user specified (via `!nickname`) usernames are checked against all existing usernames for uniqueness. In the case of `!nickname`, an appropriate error message should be sent to the calling client if the nickname is not unique.

Pass/Fail **Pass**

Actual As expected.

Test 22 When a username is changed via the `!nickname` command, all clients are notified of the change and the nickname is updated in any active game sessions the client is in, leaderboard, etc...

Expected As above.

Pass/Fail **Pass**

Actual As expected. See video demonstration as well.

Test 23 The `!players` command prints the current list of active players that are not in a game session.

Expected At any point in time, a game session may terminate, changing the `game_idx` of all the clients in the session to `-1`, indicating that they are no longer in a game session. Access to the `game_idx` during a call to `!players` should be in a thread–safe manner to ensure correctness.

Pass/Fail **Pass**

Actual As expected.

Test 24 The `!battle` and `!quick` commands ensure that only active clients not in a game session are invited to a game.

Expected Access to the `game_idx` variable in a `client` struct during a call to `!battle` or `!quick` should be in a thread–safe manner to ensure correctness. Indeed, so must be any subsequent modification of `game_idx` as a result of the invite being sent. More so, if a client specified via the `!battle` command is already in a game session, an appropriate error message should be sent to the caller.

Pass/Fail **Pass**

Actual As expected.

Test 25 The `!battle` and `!quick` commands carry out exhaustive checks on the options passed, reporting appropriate errors back to the caller.

Expected Any options such as the number of baselines, game mode specification, etc... are all checked for validity, to ensure that correct state is maintained throughout.

Pass/Fail **Pass**

Actual As expected.

5 Limitations and Future Improvements

1 P2P Communication

- Presently, if a client finishes a game and immediately joins another, if the server allocates the same port offset to the client as in the previous game session, it may be the case that the operating system may not be able to allocate the port immediately, as shown in the testing section. We attempted to mitigate this issue by setting various socket options, however the problem persisted. One future solution would be to devise a mechanism with which the same port offset is not allocated successively to the same client. The same could be said for the server – if it crashes and we immediately restart it, the port `8080` might not be made immediately available by the OS.
- The primary reason for setting up a P2P network between the clients in the game session was so that game state is not shared via the server, hence off–loading some work. This could be further improved, by having a single client sending score updates on behalf of all the clients, in a single message, rather than all clients sending separate score updates to the server. One would however have to devise a mechanism by which if the designated client disconnects or finishes from the game session, another client in the game session would be selected to gather and send the score updates.
- Referring back to the flow–chart given for the setting up of a new game, including the P2P setup, we observe that a form of synchronisation mechanism was implemented between the server and the clients via the exchange of `P2P_READY` and `START_GAME` messages. In order to mitigate the issue outlined in Test 11, we tried to introduce a *handshake*[1] between clients during P2P setup, where a game session would start if every client received an acknowledgment from every other client. Once this is achieved, the client would send an acknowledgement to the server. Hence the `START_GAME` message is sent by the server if it has:

[1]For those interested, this initial approach to setting up a P2P network can be found in commit e5b072a of the client library repository, illustrating the exchange of `CLIENTS_CONNECTED` messages.

1. Received a `P2P_READY` message from all the clients.
2. Received a further acknowledgement (via the now defunct `CLIENTS_CONNECTED` message type) from all the clients that have successfully communicated with all the other clients.

The issue with this simplistic approach was that if two clients could not communicate with each other, then an indefinite wait was occurring (the server would not receive the second acknowledgement from all the clients, hence never sending a `START_GAME` message). A more robust protocol would have to be designed, namely one that is capable of avoiding this indefinite wait by eg. removing unconnected pairs of clients from the game session or cancelling the game session entirely.

- Similarly, as outlined in Test 8 and the video demonstration, in the final implementation sometimes the server might receive all the `P2P_READY` messages, yet not send a `START_GAME` message. We believe that this is due to our use of `pthread_cond_wait` and `pthread_cond_broadcast` to keep track of the number of `P2P_READY` messages we received.

  Indeed, while the issue rarely occurs on a server instance running on Lubuntu or Ubuntu, we were able to consistently replicate the issue with the server running on Mac OS X. Initially we thought that this issue was arising due to the aforementioned port availability issue, which turned out to be a red herring. With this realisation being made at a late stage (19/06/2021) and time constraints with in–person examinations, we were sadly not able to properly rectify the issue.

Possible Multi–Threaded Handshake Solution

Recall that during game setup, the server maintains a count of the number of handshakes[2] received. This count is maintained in the `game_session` struct for the game session in question, where the mutex lock for this struct must be held by the `service_game_request` thread throughout setup, to ensure consistent state.

Hence, we modify this count by *temporarily* releasing this mutex lock and making it available only for a client thread that is about to increment the handshake count (on receipt of a `P2P_READY` message in this case). We achieve this using `pthread_cond_wait` and `pthread_cond_broadcast`, along with a `pthread_cond_t` conditional variable associated with the `game_session` struct. There are however a number of downsides:

- Firstly, as discussed, this solution sometimes fails in fact, due to what we believe is a deadlock arising from the game session mutex being held indefinitely at some point.
- Secondly, the handshaking mechanism is somewhat convoluted and difficult to extend – recall that we stated earlier that we wish to introduce further handshakes in future implementation, to ensure that a *full, bi–directional* P2P network is setup.

We then propose the following solution:

- Each client instance maintains flags indicating whether they have sent/received an acknowledgment. We maintain a flag for each handshake type (in this case for `P2P_READY` only, but we may add further handshakes in the future). This replaces in particular the `pthread_cond_t` conditional variable mentioned earlier.
- Modifying and accessing these flags only entails acquiring and then releasing the mutex locks associated with the clients, as we have done a countless number of times. This eliminates the need for using `pthread_cond_wait` and `pthread_cond_broadcast` – i.e. the mutex associated with the game session is always held by the `service_game_request` thread during setup.

[2]We are assuming that social distancing is maintained or that both the client and server are fully vaccinated.

- To check the count of acknowledgements received during setup then, we simply check whether the clients in the game session have set their own respective flag, by iterating through them and obtaining/releasing their respective mutex locks for thread–safety. We repeat the check until the count is the expected number.
- To eliminate any possible indefinite waiting arising from the count never achieving the required value, we add a timeout or a limit to the number of attempts. Once these have been achieved, the game session terminates erroneously.

2 Other Areas of Improvement

- Separate the live chat and game play `WINDOW` instances to be updated by separate threads, to acheive overall better responsiveness. As alluded to before, since `ncurses` is not inherently multi–threaded, this would require a number of wrappers around `ncurses` functions in order to modify and access any data accessed by `ncurses` in a thread–safe manner.
- It would be worth exploring in future implementations thread–safe persistence of user data and leader–boards onto disk. This would also allow for the addition of a log–in system (implemented accordingly in both the server and the front–end).
- The leaderboard works exclusively on a score-based ranking system, where the score is determined by the number of lines cleared across all game modes. However in some game modes, such as *Rising Tide*, a player takes first place if they are the last player to finish – technically, they could have the *lowest score amongst all the other players*. To this extent, future versions should introduce a score based system specific to each game mode, instead of a universal one for all modes (the ranking of which might not be reflective of the actual winner).