CPS2008 Assignment

Super Battle Tetris

Tony Valentine

Contents

Project Overview

Server Backend

Super Battle Tetris required the implementation of an IRC style server, where users interact with the server through a text based interface. The main functionality required was the implementation of the chat relay, and for the server to execute custom commands denoted by the `!` symbol at the beginning of a line.

Server Architecture

To start the server the `run_server` command is used, where the only argument given is the port number to bind to.

`run_server` creates a new TCP socket before attempting to bind the socket to the requested port number. If this is successful then the server calls `listen` and will attempt to `accept` new connections until shutdown.

When the `accept` call succeeds the server logs the clients IP address, port number and socket file descriptor before creating a new thread to run the `handle_connection` function. An important note here is the use of `setsockopt` to apply timeouts when sending to the new socket, this prevents slow clients from disrupting the execution of the server as `send` is typically a blocking call.

```
New Client with IP: 127.0.0.1 On port: 61095 with socket number 4
```

Figure 1: Client Connection Message

Connection Handling

As mentioned previously connections are handled in a new thread running the `handle_connection` function.

Initially the function generates a new player name and creates a `player_data` class to store all the information needed for a client, this data is stored in a map (`player_list`) accessible to all threads. The thread will then enter an infinite loop trying to process data from the client. To reduce server overhead the socket file descriptor is monitored using the Linux `poll` system call, this allows the `recv` call to only be used when there is data to be received.

Upon sucessful return of the `poll` call, `recv` is used to get the data from the socket and the total number of bytes read are recorded. If zero bytes are read from the socket this is indicative of the client on the other end closing the socket, the thread will then enter a cleanup phase to remove all references to the client it's serving before terminating. Otherwise the server will decode the message from network byte ordering (`ntohl`) before calling `handle_message`.

```
Client with IP: 127.0.0.1:61095 connected to socket 4 disconnected
```

Figure 2: Client Disconnection Message

Message Handling

To reduce load on the server a strict message protocol was adopted, this protocol can be found in the `tprotocol.hh` header file.

```
struct tmessage {
  tmessage_t message_type; /**< Enumeration denoting the type of the message*/
  int32_t arg1, arg2, arg3, arg4, arg5,
      arg6;                     /**< Optional argument sfor each message type*/
  char buffer[MESSAGE_LENGTH]; /**< Char buffer for chat messages or information
                                  to decode */

} typedef tmessage;
```

`tmessage_t` is especially important to the implementation as it allows the server to switch over handlers for the different message types and used in the implementation of `handle_message`.

Chat Messages

Chat messages are handled by copying the contents of the message `buffer` and prepending the player name to the buffer. To send messages to all the clients a `relay` function was implemented. The function works by creating and encoding a valid `tmessage` with the requested string before sending the message to all the keys from `player_list` (map from sockets to player data).

```
case CHAT: {
  string str;
  auto iter = player_list.find(sock);
  if (iter != player_list.end()) {
    str = iter->second.name;
  }
  str.append("> ");
  str.append(string(msg->buffer));
  relay(str);
  break;
 }
```

Nickname

Nickname changing is implemented through changing the `name` field in the `player_data` class corresponding to the current user. Error messages are returned to the user in the form of chat messages if the new nickname is used by another player on the server or exceeds the maximum nickname length.

To solve the situation where two clients request the same nickname at the same time, the `player_list` has a corresponding mutex `player_list_mutex` which is used to lock the map during manipulation.

```
case NICKNAME: {

  string str;
  str = "Updated Nickname to: " + string(msg->buffer);
  // If an error is encountred str will be updated to the error message

  player_list_mutex.lock();
  // Critical Section setting the player nickname
  player_list_mutex.unlock();

  send_chat(sock, str);
  break;
}
```

Leaderboard(s)

The win, loss and scores for games are stored in the `player_data` entry for each client. To reduce the amount of time in the critical section the `player_list` is locked and a local copy is created. Since none of the entities in the map are dynamically allocated the default copy constructor in c++ will implement the behaviour required.

```
player_list_mutex.lock();
auto local_plist(player_list);
player_list_mutex.unlock();
```

Another added benefit is that the leader-board shown will be at the time of the message request, not including any new games ending during processing.

Generating the leaderboard required sorting the `player_data` in descending order according to the specified gamemode, the implementation of which used the `<algorithm>` library is used in combination with a custom comparison function. The top 3 entries for the gamemode are formatted into a string and placed in a `vector` which is returned at the end of the function. All of this functionality is wrapped inside the `get_leaderboard` function to increase re-usability.

```
auto sort_order = [game](player_data &p1, player_data &p2) -> bool {
    switch (game) {
    case RISING_TIDE:
      return get<0>(p1.rising_games) > get<0>(p2.rising_games);
    case BOOMER:
      return get<0>(p1.boomer_games) > get<0>(p2.boomer_games);
    case FAST_TRACK:
      return get<0>(p1.fasttrack_games) > get<0>(p2.fasttrack_games);
    case CHILLER:
      return p1.chill > p2.chill;
    default:
      throw logic_error("Invalid Gamemode");
    }
};

sort(plist.begin(), plist.end(), sort_order);
```

To ensure that the multiple messages which construct the leaderboard arrive uninterrupted the socket file descriptor for the client is locked using the `lockf` system call. Following this all messages can be sent to the client before unlocking the file descriptor. This locking and unlocking behaviour is implemented in all the sending commands from the server to ensure correctness.

Leaderboards The implementation for `!leaderboards` involved calling the `get_leaderboard` function for each gamemode and sending the messages to the client.

Players

Listing the current players required traversing the `player_list` and extracting the `name` field for each active user's `player_data`. These names are then appended to a string before sending the message to the user.

Playerstats

The playerstats command is required to list the current win/loss statistics for all the active players. As with the leaderboards command a local copy of the map is created. The local copy is iterated over where the stats are obtained from the `player_data`, the information is formatted and placed into a vector before sending all the information to the client. A small delay is added in between the stats for each player to allow the client a chance to read the incoming stats.

```
for (auto &[k, v] : local_plist) { // Iterating over local player list
      string str = "Player: " + string(v.name);
      scores.push_back(str);
      formatted_out("Mode", "Score", "Wins", "Losses"); // lambda for string formatting
 ↪  and pushing to scores
      scores_out("Rising", v.rising, v.rising_games); // lambda for string formatting and
 ↪  pushing to scores
      scores_out("Boomer", v.boomer, v.boomer_games);
      scores_out("FastTrack", v.fasttrack, v.fasttrack_games);
```

```
        scores.push_back(
            formatted_out("Chiller", to_string(v.chill), "N/A", "N/A"));
        send_multiple(sock, scores);
        scores.clear();
        this_thread::sleep_for(
            chrono::milliseconds(250)); // As to allow the user to read a bit
}
```

Battle

Currently super battle tetris supports 4 gamemodes, of those 3 are supported in the battle command. Upon recieving a battle request the server must first decode the player names into the corresponding socket fds.

If all or some of the players are found then a `game` object can be created, otherwise an error message is sent to the user. The `game` object is used to store all the information required to host a game, including the gamemode, gamemode arguments, and playerlist. The playerlist is a mapping from socket fd to a boolean, where the boolean flags if a player has accepted or declined a match.

The new game is then added to the `game_list` map where the key is generated from an `atomic_uint32_t`, this guarantees that each pending game has a unique number with an upper limit of no more than 4×10^9 games being pending at one time. Invited players are then messaged, and the game id is added to the `games` vector inside of their `player_data`. The need to store all invited games is to remove the users from games if they disconnect.

Finally a new thread is created running the `handle_game` function which used to handle game creation and initiation logic.

Game Launching The `handle_game` function first goes into a 30 second timeout to give players a chance to accept the game request. After 30 seconds the game is removed from the `game_list` to prevent late accept or decline messages. Following this the function collects the list of accepting players, if less then 2 an error message is sent to the user.

```
void handle_game(int game_id) {
  // Start a 30 second timeout
  this_thread::sleep_for(chrono::seconds(LOBBY_TIME));
  // Timeout ended
  game_list_mutex.lock(); // locking the game_list so no one can edit
  auto match = game_list.at(game_id);
  game_list.erase(game_id);
  game_list_mutex.unlock();
```

Typically multiplayer games use a client-server architecture, this allows all clients to send and receive updates on a single port number. Unfortunately one of the requirements of Super Battle Tetris was for game updates to be implemented in a peer-to-peer fashion, as well as allowing multiple clients to exist on the same computer. This means that the server needs to assign unique port numbers to each of the clients so that data can be transmitted and received properly.

Therefore the `handle_game` function needs to generate an unique port number for each client when sending the list of participating ip addresses. Apart from this quirk the rest of the function involves creating the message with the game information to send to the participating clients.

```
random_device rd;                // Used to generate random numbers
tmessage msg;                    // message to send to the user
msg.message_type = INIT_GAME;
msg.arg2 = match.gamemode;       // Gamemode
msg.arg3 = game_id;
```

```
msg.arg4 = match.arg1;          // Gamemode Arguments [Time or Baselines]
msg.arg5 = match.arg2;          // Gamemode Arguments [Winlines]
msg.arg6 = rd();                // Game Seed
```

The participating IPs and port numbers are stored in the `buffer` member variable of `tmessage` where the final ip and port belong to the respective client.

Quickplay

The quickplay implementation is almost identical to that of Battle. The primary difference is that game arguments and players are generated by the server rather than being included in the player arguments.

```

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> game_distribution(0, 2); // Valid Gamemodes
game new_game;
int gamemode = game_distribution(gen);
new_game.gamemode = gamemode;
switch (gamemode) {
    case BOOMER: {
        std::uniform_int_distribution<> time_distribution(30,300); // 30 seconds to 5 minutes
        new_game.arg1 = time_distribution(gen);

    } break;
    case FAST_TRACK: {
        std::uniform_int_distribution<> line_distribution(0,10); // 0-10 lines
        new_game.arg1 = line_distribution(gen); // Baselines
        new_game.arg2 = line_distribution(gen); // Winlines
    } break;
```

Parameter randomisation was rather implemented by switching over the current gamemode, and generating random numbers in a range deemed reasonable for the respective parameter. In the code snippet above the time duration for a boomer game is being set in the range of 30 seconds to 5 minutes.

Given an iterator number n the `advance` function can be used to get the $n + k$ iterator, and since c++ maps implement iterators this can be used to generate random players. The implementation locks the `player_list` and then generates a random offset from the start of the list, if the player is not the game creator or an already selected player they are added to the invite list.

```
vector<int> offsets;
player_list_mutex.lock();
std::uniform_int_distribution<> invite_distribution(0, player_list.size() - 1); // Used to generate offsets from the starting iterator
while (new_game.players.size() < msg->arg1) {
    auto iter = player_list.begin();
    advance(iter, invite_distribution(gen)); // Get a random player
    // Preventing the game creator being added, and preventing a player from
    // having multiple invites
    if (iter->first != sock && new_game.players.count(iter->first) == 0) {
    new_game.players.insert({iter->first, false});
    }
}
player_list_mutex.unlock();
```

At this point the `game` object is configured and the implementation is identical to that of Battle.

Chill

The `!chill` command is used by clients to initiate a single player game. The server creates a `game_data` struct to store the game information, and places it in the `chill_games` map. Players are sent an `INIT_GAME` message with the chill gamemode, game id, and random seed.

Gamestats

The `!gamestats` command is indented to output the individual player scores and lines cleared during a game. Since the list of active games is stored in the `ongoing_games` map, the selected game can be pulled from there before formatting the scores and sending a message back to the user. If the selected gamemode does not exist then the server will send an error message to the client.

Go

The `!go` command is intended to allow users to accept a specific game invitation. To implement this functionality the game number is parsed from the command, where the server can then check the `game_list` for this game id. If the game is found and the client was invited to the game, then the `players` map inside the game is updated to indicate the clients choice. Otherwise the client is sent an error message informing them that either the game does not exist or they were not invited.

```cpp
string str;
game_list_mutex.lock();
auto g = game_list.find(msg->arg1);
if (g != game_list.end()) {
    auto entry =
        g->second.players.find(sock); // Checking if we are part of the game
    if (entry != g->second.players.end()) {
        entry->second = true; // Setting acceptance boolean to true
        str = "You sucessfully accepted to join the game";
    } else {
        str = "You were not invited to the game in question";
    }
} else {
    str = "The game you requested to join does not exist!";
}
game_list_mutex.unlock();
send_chat(sock, str); // Sending outside of critical section
```

Ignore

The `!ignore` command allows users to decline invitations to games. The processing of the command is identical to that of the `!go` command, except the boolean for accepting is set to false.

Score Updates

Score updates are commands issued to the server by clients during battle games. The messages consists of the game id, current score and current lines cleared. The server can then update the game in the `ongoing_games` map using the game id.

```
case SCORE_UPDATE: {
    int game_number = msg->arg1;
    int score = msg->arg2;
    int lines = msg->arg3;
    if (ongoing_games.find(game_number) != ongoing_games.end()) {
        auto game = ongoing_games.find(game_number);
        game->second.update_player(sock, score, lines);
    } else if (chill_games.find(game_number) != chill_games.end()) {
        auto game = chill_games.find(game_number);
        game->second.score = score;
        game->second.lines = lines;
    }
    break;
}
```

Game End

The game end command is issued to the server by clients to update the server with the final scores of a game. The message consists of the game id, final score and lines. The server then updates the game with the players information and checks if all players have ended the match. When updating the player information the time is recorded for later use.

Once all players have ended the match a winner needs to be decided.

- Boomer: Select the player with the highest score
- Rising Tide: Select the player with the longest game duration
- Fast Track: If both players completed the required number of lines then select the player with the shortest game duration. Otherwise select the player with the larger number of lines with duration as a fallback in the case of a tie.

This functionality is wrapped inside a lambda function so that the player can be selected using `std::max_element` from the `<algorithm>` library. All players will then have their individual stats updated for win/loss numbers as well as high-scores.

Client Connectivity

Client-Server

Client-Client

Peer to peer connectivity is implemented in the `src/p2p` directory.

The main component of the library is the `communicate_state` function. The function is designed to take a `gamestate` struct which contains the information used during a game. This struct is passed by reference to the function by the tetris games to ensure that the information can be read properly. The function is designed to establish the necessary sockets to send or recieve state from peers during the game. Once the sockets have been established then `receive_state` and `broadcast_state` are called.

`recieve_state` is used to retrieve game updates from peers, and update the `gamestate` with the new information. Since peer information is only updated by the `recieve_state` function and tetris games will only be reading the information contained, mutual exclusion was not required during reads or writes.

```
update_msg *msg = (update_msg *)buffer;
decode_state(msg); // Network to host
if (state.players.find(msg->player_no) != state.players.end()) {
    auto player = state.players.find(msg->player_no);
    player->second.score = msg->score;
    player->second.lines = msg->lines;
} else {
    // First update, add player to list
    state.players.insert({msg->player_no, (playstate){.score = 0, .lines = 0}});
  }
}
```

`broadcast_state` is used to send the current `gamestate` information to all peers every 25 milliseconds, and to the server every one second. The information is sent in the form of an `update_msg` struct to peers and a `tmessage` to the server. To increase the frequency of updates a `TICKRATE` macro is defined in the headerfile, this alters the number of updates to peers per second.

```
while (1) {
    if (*termination_flag) {
      break;
    }
    counter = (counter + 1) % TICKRATE;
    struct update_msg msg;
    msg.score = state.local.score;
    msg.lines = state.local.lines;
    msg.player_no = state.player_no;
    encode_state(msg);
    for (auto &[sock, other] : peers) {
      int num_bytes;
      if (sendto(sock, (char *)&msg, sizeof(update_msg), 0,
                 (struct sockaddr *)&other, sizeof(other)) == -1) {
        close(sock);
        perror("sendto()");
        return;
      }
    }
    if (counter == 0) {
      tmessage server_msg;
      server_msg.message_type = (tmessage_t)htonl((int32_t)SCORE_UPDATE);
      server_msg.arg1 = htonl(state.game_no);
      server_msg.arg2 = htonl(state.local.score);
      server_msg.arg3 = htonl(state.local.lines);
```

```
      send(sock_fd, (char *)&server_msg, sizeof(server_msg), 0);
    }

    this_thread::sleep_for(
        TICKDURATION);
}
```

Terminating threads safely in c++ is slightly more cumbersome than with native POSIX threads. To work around this problem a `termination_flag` is used to tell the threads if they need to terminate, the flag is set by tetris games when the player loses or wins their match. This ensures that all threads are cleaned up properly and the cleanup code is self contained within the function.

Client Front-end

The client front-end consists of two distinct components, the chat client used to send and display messages, and the individual games implemented with being implemented using `ncurses`.

On startup the client uses the `establish_connection` command to open a TCP socket with the server. The arguments to the command are the servers hostname, as well as the port, both of which should be passed by the command line.

The client will then enter a loop where the socket file descriptor and standard input are polled using the `poll` system call. If standard input is ready then the `send_message` function is used to take the input, wrap it in a `tmessage` struct and send it to the server. If data is ready to be received from the server then `recieve_message` is used to get the message from the server and handle it. The possible messages from the server are either chat messages, or start game messages.

```c
 struct pollfd pfds[2];
pfds[0].fd = 0; // STDIN
pfds[0].events = POLLIN;

pfds[1].fd = sockfd;
pfds[1].events = POLLIN;

int xMax, yMax;
getmaxyx(stdscr, yMax, xMax);

for (;;) {
  move(yMax - 4, 1);
  int poll_count = poll(pfds, 2, -1); // Wait till we have input

  if (poll_count == -1) {
    exit(1);
  } else {
    for (int i = 0; i < 2; i++) {
      if (pfds[i].revents & POLLIN) {
        if (pfds[i].fd == 0) {
          // Standard Input is ready
          send_message(sockfd);
        } else if (pfds[i].fd == sockfd) {
          // Data to receive from the client
          recieve_message(sockfd);
        }
      }
    }
  }
}
```

Handling Messages

As mentioned previously `recieve_message` is used to handle incoming messages from the client. The received messages are handled according to their `message_type` parameter.

Chat Messages

Chat messages are handled by extracting the `buffer` field from the received message, and appending it to a limited buffer. The limited buffer is used to store the n most recent messages to emulate text scrolling on the display.

```
case CHAT: {
    chat_messages.push_back(string(msg->buffer));

    wclear(chat_window);
    box(chat_window, 0, 0);

    if (chat_messages.size() > (yMax - 12)) { // yMax is the maximum buffer size based on the screen
      chat_messages.erase(chat_messages.begin(), chat_messages.begin() + (chat_messages.size() - (yMax - 12))); // Remove the old messages
    }
    // Display the messages
    for (int i = 0; i < chat_messages.size(); i++) {
      mvwprintw(chat_window, i + 1, 1, chat_messages[i].c_str());
      wrefresh(chat_window);
    }
    break;
  }
```

Game Messages

Games starts are triggered by recieving `INIT_GAME` messages from the server. The message structure contains all the information to start the game;

1. Player number
2. Gamemode
3. Game id
4. Gamemode Argument 1
5. Gamemode Argument 2
6. Game seed

The arguments are then passed to the constructors for the tetris games, before allowing the game to run and getting the final score.

```
switch (msg->arg2) {
  case BOOMER: {
    BoomerGame game(msg->arg6, ips, msg->arg1, msg->arg3, sockfd, msg->arg4);
    game.run();
    score = game.get_final_score();
  } break;
```

The score is needed to send a final `GAME_END` message to the server before returning to the chat client.

Games

The tetris game implementation code is located in the `src/tetris` directory. The base `TetrisGame` class is used to implement a basic tetris game with the code for piece generation, rotation, collision, scoring, line clearing etc.

This gamemode also acts as the implementation for the singleplayer CHILL gamemode.

Boomer

The Boomer gamemode was intended to end the game after a specified duration. To implement this functionality the current time elapsed is checked after each gametick, meaning that the game will terminate with a maximum delay of 25 milliseconds.

```
auto start = chrono::high_resolution_clock::now();
auto current = chrono::high_resolution_clock::now();
auto time_elapsed =
    chrono::duration_cast<chrono::milliseconds>(current - start);
while (do_gametick(new_piece, piece_flag, counter) == 0 &&
        time_elapsed < game_duration) {
  current = chrono::high_resolution_clock::now();
  time_elapsed = chrono::duration_cast<chrono::milliseconds>(current - start);
}
```

Rising Tide

Rising tide requires that all lines completed by other players are transferred to your own board.

This functionality was implemented by keeping track of the total number of lines completed at each game tick, if on the new gametick more lines have been completed, then the lines are added to the players board. The `insert_lines` function was also implemented to handle the logic of inserting lines and managing the pieces.

```
while (do_gametick(new_piece, piece_flag, counter) == 0) {
    new_lines = 0;
    if (state.players.size()) { // Only checking lines if there are lines to check
      old_lines = total_lines; // Noting the previous number of lines completed
      total_lines = 0;
      for (auto &[k, v] : state.players) {
        total_lines += v.lines; // Calculating the total number of lines completed
      }
      new_lines = total_lines - old_lines; // Calculating the new lines completed
      if (new_lines > 0) {
        if (insert_lines() != 0) { // Inserting the lines
        /*
        * Cleanup current piece
        */
```

Fast Track

The fast track gamemode the playing field is initially populated with a number of rows and the game ends when the player completed a stipulated number of lines or loses.

Filling the board was a trivial implementation and involved setting all the columns in the specified rows to an unclearable number.

```
for (int i = playing_field.size() - 1; i >= playing_field.size() - init_lines; i--) {
    playing_field[i] = vector<char>(playing_field[i].size() + 1, STATIC_ROW);
}
```

Ensuring that the game ends after the stipulated number of lines involved checking the current number of completed lines against the needed amount.

```
while (do_gametick(new_piece, piece_flag, counter) == 0 &&
         lines_cleared < win_lines) {
}
```