Instituto Superior Técnico - UL

TÉCNICO LISBOA

Projecto de Programação de Sistemas

Memory Game

Authors:	*Numbers:*
Gonçalo Mestre	87005
Luís Gonçalves	87058

Professor João Silva

---

10 de Julho 2019

Conteúdo

Capítulo 1: Arquitectura

1.1 Nós

A arquitectura geral do projecto segue a distribuição dos processos principais esquematizada na Figura 1.1.

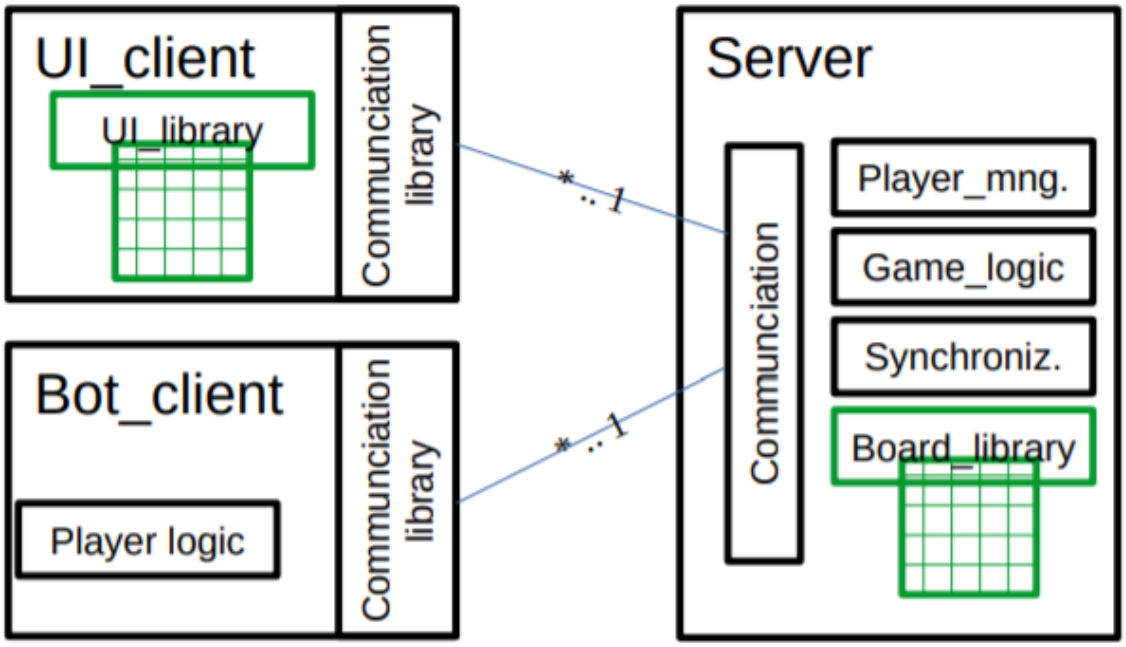

Figura 1.1: Principais processos do Sistema

Este sistema é formado por 3 tipos de componentes, sendo necessário o servidor para que o mesmo funcione e pelo menos 2 UI_clients/Bot_clients:

- Server - Servidor desenvolvido para controlar o jogo, processando todas as jogadas recebidas por parte de cada jogador e enviando as alterações aos jogadores em jogo. Também contém um tabuleiro gráfico igual ao dos utilizadores para monitorização do jogo
- UI_client - Interface gráfica que permite a um utilizador jogar selecionando as peças no tabuleiro. Esta interface vai sendo actualizada com as mudanças no tabuleiro pelo Server.
- Bot_client - Programa que está constantemente a enviar jogadas ao servidor, funcionando como qualquer utilizador, mas que joga automaticamente. Também contém uma interface gráfica caso se pretenda visualizar o tabuleiro.

1.2 Módulos

Este sistema tem também vários módulos, que são partes de um sistema responsáveis por uma tarefa bem definida, e que podem ser acopolados ao mesmo para permitir que este execute a tarefa representada pelo módulo. Alguns destes módulos descritos de seguida, são inclusivé comuns a vários nós.

- Interface Gráfica - Módulo que implementa a criação do tabuleiro e a impressão de alterações no mesmo. Foi incluído em todos os nós, mas podia apenas ser incluído no UI_client caso fosse escolhido não se ter tabuleiro gráfico nos outros 2 nós.
- Verificações - Módulo que implementa a verificação dos argumentos de entrada de cada ficheiro e a verificação das alocações de memória para cada execução.
- Inicialização da Comunicação - Módulo que inicializa a comunicação utilizada nos vários nós verificando que os sockets do tipo SOCK_STREAM estão em condições dee estabelecer as comunicações entre estes.
- Lógica do Jogador - Lógica utilizada no Bot_client para que o mesmo pudesse ir enviando jogadas continuamente e que estas fossem também sendo geradas.
- Gestão do Jogador - Módulo utilizado no Server para ir gerindo ao longo do jogo as jogadas feitas pelos clients.
- Gestão da Resposta - Módulo utilizado no Server para gerir a informação a enviar e a imprimir no tabuleiro para uma determinada resposta a uma jogada efectuada.
- Gestão do Tabuleiro - Módulo que implementa as funções que permitem as alterações ao tabuleiro e verificar as jogadas efectuadas pelos jogadores. Apenas foi utilizado no Server.
- Gestão de Utilizadores - Conjunto de funções do Server que permitem implementar a gestão dos utilizadores em jogo e tanto dos utilizadores que entram como dos que saem do jogo.
- Lógica do jogo - Módulo que implementa a gestão do jogo no Server, verificando várias questões como os 5 segundos da primeira jogada ou os 2 segundos da jogada errada.

Capítulo 2: Organização do Código

Para a organização do código dividiu-se o mesmo em vários ficheiros contidos na pasta entregue, sendo estes os indicados de seguida.

- board_library.c
- board_library.h
- General.c
- General.h
- UI_library.c
- UI_library.h
- multiplayer.h
- memserver.c
- memserver.h
- memclient.c
- bot2.c

De seguida ir-se-á para cada ficheiro ou par de ficheiros indicar os módulos que estes implementam, as estruturas que contém e que funções estão presentes na implementação dos módulos. Não se irá proceder à explicação individual de cada função, pois considera-se que os comentários deixados no código entregue são claros e suficientes em relação ao que cada função implementa. Também não será explicada cada estrutura de dados, pois as mesmas são explicadas no capítulo seguinte.

2.1 board_library.c e board_library.h

Através destes ficheiros foi implementado o módulo de "Gestão do Tabuleiro"descrito no capítulo 1. Este ficheiro contém as estruturas "board_place", "play_response"e "play_node". Para a implementação do módulo referido utilizaram-se as funções "get_board_place_str", "init_board", "getbackfirst", "board_play", "freethepiece", "savethecolor", "checkboardstate", "getboardcolor", "checkboardnull"e "activateboardlock".

2.2 General.c e General.h

Estes 2 ficheiros implementam o módulo "Verificações"recorrendo às funções "socketserver"e "socketclient". Também implementam o módulo "Inicialização da Comunicação". O ficheiro contém também as funções "serverinputs", , "clientinputs", e "verifyalloc".

2.3 UI_library.c e UI_library.h

Estes dois ficheiros implementam o módulo da "Interface Gráfica", recorrendo às funções "write_card", "paint_card", "clear_card", "get_board_card", "create_board_window", "close_board_windows"e "StartingSDL". Também contém a definição das estruturas "piece"e "boardpos".

2.4 multiplayer.h

Este ficheiro não implementa nenhum módulo, contendo apenas a definição das estruturas "player"e "player_node".

2.5 memserver.c e memserver.h

Nestes ficheiros está contida a declaração da estrutura "respplayer", sendo também implementado o módulo de "Gestão do Jogador"utilizando a função "playerfunc"e o módulo "Lógica do jogo"com as funções "timerfplay"e "stopignore". Este conjunto de ficheiros implementa também o módulo de "Gestão da Resposta"recorrendo às funções "dealwithresp", "sendpiecetoclient", "producepiece", "print_piece", "print_response_server"e "cleanpiece". Por fim, o último módulo que este par de ficheiros implementa é o módulo de "Gestão de Utilizadores", utilizando as funções "newplayers", "get_pnode", "newplayer", "serverkill", "prepare_client_exit"e "invasionentry".

2.6 memclient.c

Este ficheiro não implementa nenhum dos módulos descritos, analisando apenas as respostas envidadas pelo servidor e imprimindo as alterações no tabuleiro de acordo com as mesmas.

2.7 bot2.c

Este ficheiro implementa o módulo de "Lógica do Jogador"recorrendo às funções "send_plays", "getcoordinates", "filled"e "reactivate".

Capítulo 3: Estrutura de Dados

Começa-se por descrever as estruturas dadas pelo professor na Figura 3.1 às quais se acrescentaram algumas variavéis.

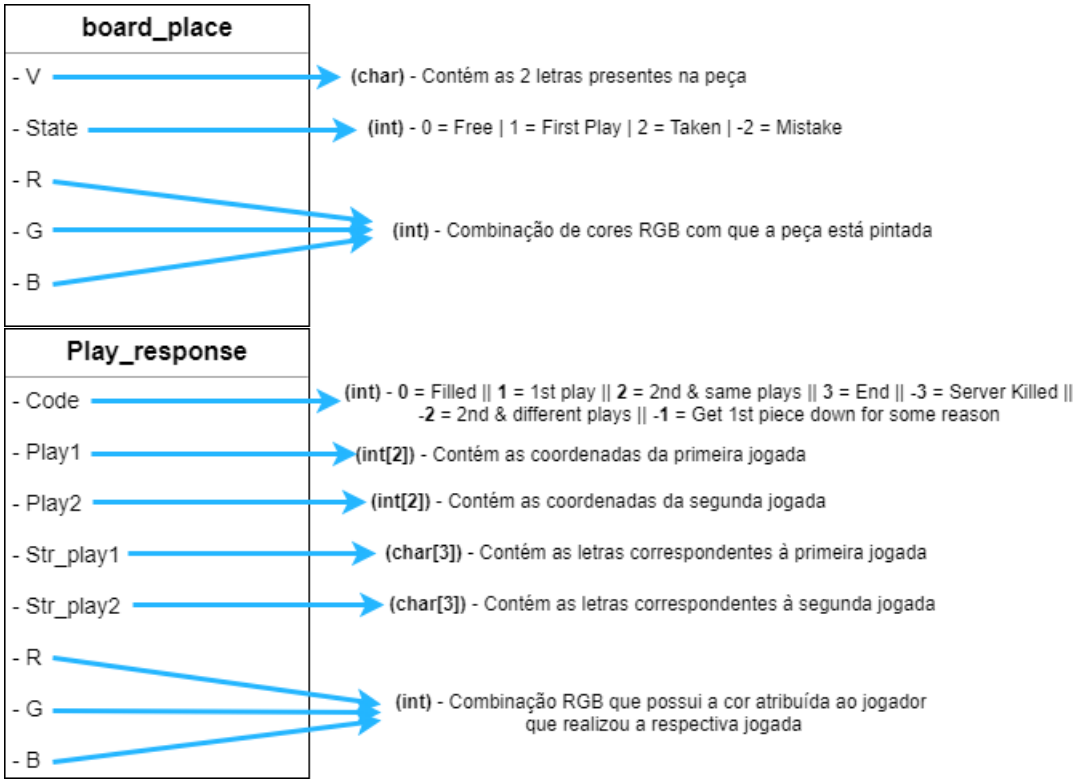

Figura 3.1: Composição das Estruturas fornecidas pelo professor

Para estabelecer a comunicação client-server, é necessário que exista uma estrutura que contenha a linha e coluna do tabuleiro que o utilizador pressionou, utilizando-se a estrutura **boardpos** descrita na Figura 3.2.

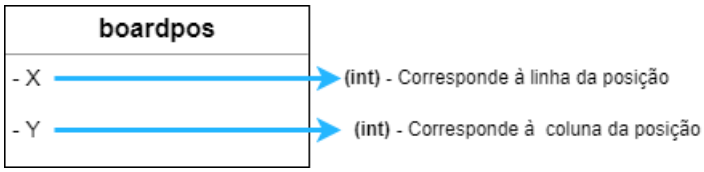

Figura 3.2: Composição da Estrutura boardpos

Ao implementar o jogo, foi criada uma estrutura para cada jogador, sendo depois acrescentadas algumas variáveis a esta estrutura na altura da implementação da funcionalidade multiplayer. Esta estrutura é a estrutura **Player**, representada na Figura 3.3.

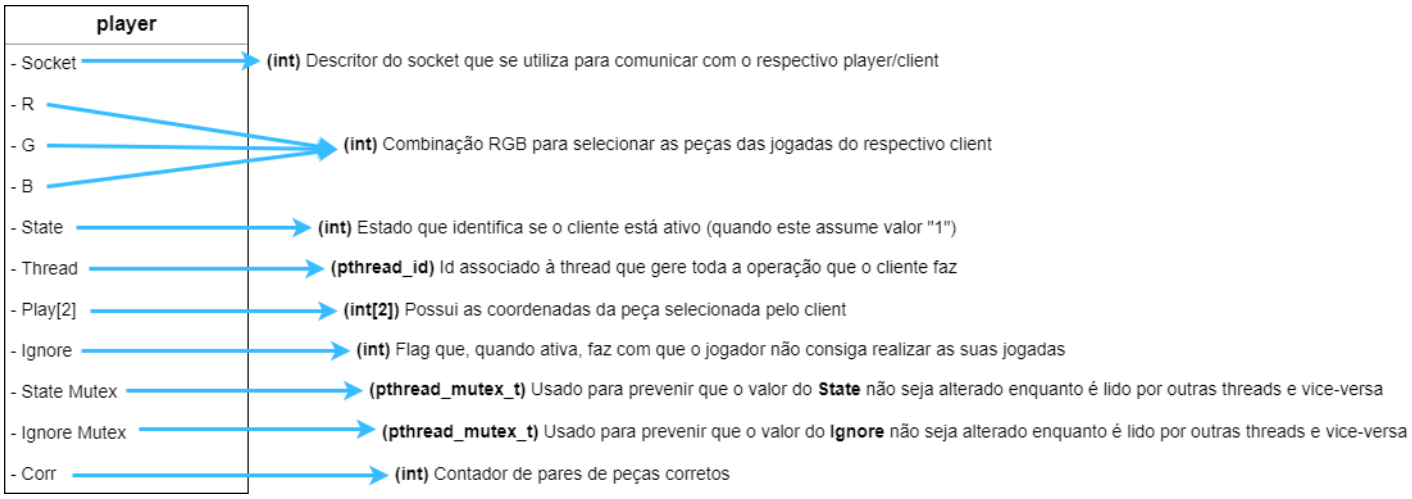

Figura 3.3: Composição da Estrutura player

Com esta nova estrutura, é possível fazer com que o servidor tenha toda a informação necessária para enviar as respostas às jogadas feitas pelos jogadores a todos os clientes em jogo. Para se agrupar todos os jogadores criou-se uma lista destes representada na Figura 3.4.

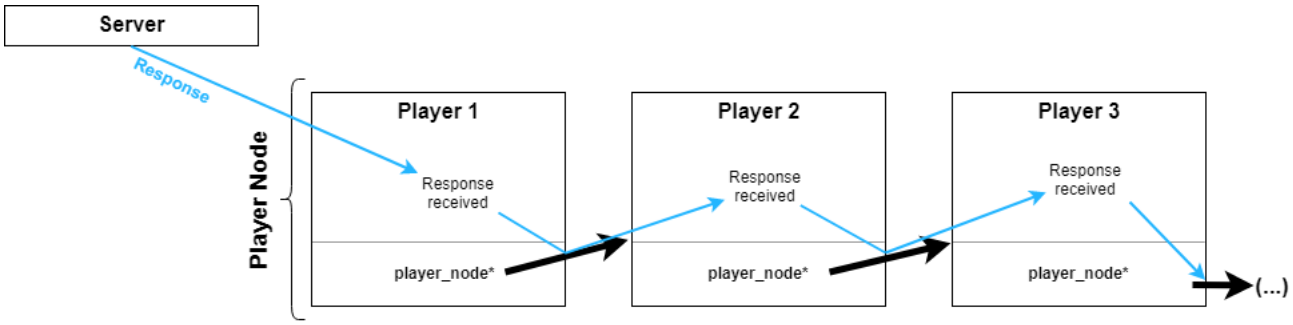

Figura 3.4: Lista de Jogadas

Para efeitos de análise da jogada efectuada por um jogador, foi necessário criar a estrutura **respplayer** para conter a informação sobre a jogada efectuada e o jogador que a efectuou na Figura 3.5.

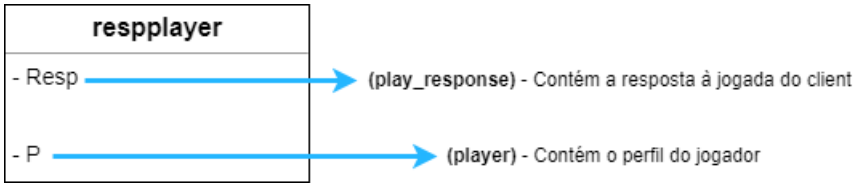

Figura 3.5: Composição da Estrutura respplayer

Para estabelecer a comunicação Server-Clients é necessária a estrutura **Piece** na qual se coloca a informação necessária para estabelecer esta comunicação.

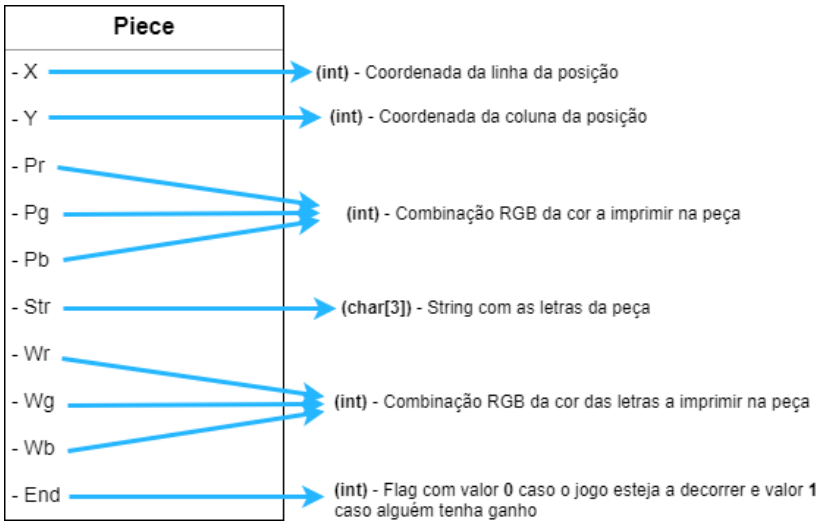

Figura 3.6: Composição da Estrutura Piece

Capítulo 4: Protocolos de Comunicação

A comunicação entre o Server e os Clients é fundamental para o projecto, visto que o Clients não processam qualquer parte do jogo, sendo todo o processamento de informação realizado no Server. A análise dos protocolos de comunicação é feita por secções em que se analisam os vários momentos em que é enviada informação. Em todos os Clients e no Server procede-se à serilização de dados, convertendo-se a informação toda da estrutura a enviar(seja a estrutura piece, boardpos ou um inteiro) para um vector do tipo char. Na recepção é convertido o vector de chars para a estrutura do tipo pretendido. 0

4.1 Inicialização da Comunicação

Para que exista comunicação foi necessário criar um socket para cada Client que estabeleça a comunicação entre esse Client e o Server. Para que isto seja possível recorreu-se à função **socketserver**, do ficheiro **General.h**, para inicializar o socket do lado do Server e à função **socketclient**, do ficheiro **General.h**. É necessário ter em conta que na função **newplayers** do ficheiro **memserver.c** é utilizada a função **accept** para aceitar que vá sendo criado o descritor de socket para cada Client que se liga ao Server.

4.2 Dimensão

Para que os Clients consigam imprimir o tabuleiro e jogar, é enviado na função **newplayer** do ficheiro **memserver.c** a dimensão do tabuleiro para o Client que acabou de se conectar.

4.3 Seleção de Posição

Quando um Client seleciona uma posição do tabuleiro através do clique do rato é enviada a posição para o Server recorrendo-se à estrutura **boardpos** e a função **sendpos** do ficheiro **memclient.c** ou **bot2.c** que executa os passos necessários para processar o envio. Do lado do Server é recebida a posição, sendo depois analisada.

4.4 Saída do Client

Quando um Client fecha a janela, o modo de lidar com a informação é igual ao da Seleção de Posição com a pequena diferença de a posição enviada não pertencer ao tabuleiro, tendo as coordenadas x=-1 e y=-1.

4.5 Actualização do Tabuleiro

Após a realização de uma jogada por algum Client é analisada a jogada e na função **dealwithresp** é chamada a função **sendpiecetoclient** com os parâmetros de entrada adequados para que seja enviada a peça que se pretende imprimir no tabuleiro para todos os Clients.

4.6 Reinicio do Jogo

Quando um jogo termina, é enviado junto com a última peça a informação de quem ganhou para os Clients. De seguida é enviada para todos os Clients uma estrutura **piece** com a cor de escrita da peça a azul, com os parâmetros "r=0", "g=0", "b=255", para que o client possa saber que tem que colocar o jogo como inactivo. Passados os 10 segundos em que o Server volta a iniciar um novo jogo, é enviada para todos os Clients ligados uma estrutura **piece** com a cor de escrita da peça a verde, com os parâmetros "r=0", "g=255", "b=0", para que os Clients saibam que vai recomeçar o jogo e o coloquem como activo.

4.7 Fecho do Server

Quando o Server é fechado, é enviado para todos os Clients uma estrutura **piece** que contém a cor de escrita da peça a azul claro, com os parâmetros "r=0", "g=255"e "b=255", sendo de seguida fechado o Server. Do lado dos Clients, ao receberem esta peça, os Clients verificam a cor de escrita e fecham depois de obter esta combinação.

Capítulo 5: Validação

Ao longo do projecto, foi necessário realizar validações de alguns dados, de modo a que o programa conseguisse seguir o plano pretendido.

5.1 Verificação de alocações de memória

De modo a assegurar que as alocações de memória foram bem executadas, foi necessário criar uma função terminasse o programa em caso de algum erro associado à alocação. Sendo assim, foi criada a função **verifyalloc**, presente na Figura 5.1.

```
/** verifyalloc: Function that verifies if the memory was correctly allocated
 * \param arg - Variable for which the memory was allocated
*/
void verifyalloc(void *arg){
    if(arg == NULL){
        printf("Memory allocation failed!\n");
        exit(-1);
    }
}
```

Figura 5.1: General: verifyalloc

5.2 Serialização de dados

Surgiu a necessidade de realizar a serialização de dados de modo a que o envio de estruturas seja efectivamente bem feito, sem perda de dados.

5.2.1 Envio de estruturas

Para o envio de uma estrutura é criado uma string do tamanho da estrutura a ser enviada, de seguida é feito um **memcpy** em função da variável a ser enviada e finalmente é enviada a string através do socket, tal como no exemplo da Figura 5.2.

```
void *sendpos(void *arg){
  boardpos bp = *(boardpos*) arg;
  char *data = malloc(sizeof(boardpos));
  verifyalloc((void *)data);
  memcpy(data, &bp, sizeof(boardpos));
  send(sock_fd, data, sizeof(boardpos),0);
  free(data);
  return 0;
}
```

Figura 5.2: memclient: sendpos

5.2.2 Receção de estruturas

Para a receção de estruturas é necessário criar também uma string do tamanho da estrutura a receber, de seguida é feito o **recv** da estrutura e finalmente é feito o **memcpy** da string para a variável que irá conter a estrutura enviada.

```
void *recv_from_server() {
  piece *p = (piece*)malloc(sizeof(piece));
  char *recvBuff = malloc(sizeof(piece));

  verifyalloc((void *)p);
  verifyalloc((void *)recvBuff);
  while(1){
    recv(sock_fd, recvBuff, sizeof(piece), 0);
    memcpy(p, recvBuff, sizeof(piece));
```

Figura 5.3: memclient: recv_from

5.3 Validação da dimensão do tabuleiro

Foi necessário criar certas condições para validar os argumentos de entrada do servidor. Perante isto, o servidor garante que a dimensão introduzida é par e que esta tem um valor menor ou igual a 26. Caso estas condições não sejas respeitadas, o programa encerra, informando ao utilizador que tais condições não foram satisfeitas.

```
/** serverinputs: Function that verifies if the arguments introduced by the user are valid
 * \param argc - Number of arguments
 * \param argv - Arguments
*/
int serverinputs(int argc, char *argv[]){
    int dim;

    if(argc < 2){
        printf("Invalid number of arguments, specify the board size!\n");
        exit(-1);
    }
    dim = atoi(argv[1]);
    if(dim%2 != 0  || dim < 2 || dim >26){
        printf("Introduce a valid board size, being pair, greater then 1, and under 26!\n");
        exit(-1);
    }
    return dim;
}
```

Figura 5.4: General: serverinputs

5.4 Validação da morte/disconexão do cliente por parte do servidor

De modo a que o servidor receba a informação de que um cliente foi encerrado, este recebe uma jogada específica desse cliente (estrutura boarpos), nomeadamente a posição x = **-1** e y = **-1**. O

servidor reconhece esta posição específica e realiza as operações dedicadas à disconexão do cliente por parte do servidor. Este processo é explicado mais à frente no capítulo 7.

5.5 Validação da morte do servidor por parte dos clientes

Quando o utilizador manda fechar o servidor, este envia uma peça específica para o cliente, com as condições apresentadas na figura 5.5.

```
void serverkill(){
    player_node *aux = phead;

    sendpiecetoclient(NULL, NULL, NULL, 0, 0, 255, 255);
    activeplayers = 0;
    done = 1;
    close_board_windows();
```

Figura 5.5: memserver: serverkill

Estando o cliente preparado para receber esta mensagem específica e interpretando-a como uma mensagem de serverkill, este executa todas as operações dedicadas ao fecho do servidor por parte do cliente.

Capítulo 6: Regiões Críticas e Sincronização

A protecção das regiões críticas do programa também foi um caso importante para este projecto, pois ao utilizar programação em várias threads, podiam existir threads a aceder a algumas variáveis quase simultaneamente, podendo estas originar erros de sincronização. Para que isto não acontecsse, utilizaram-se mutexes para bloquear algumas das variáveis utilizadas no servidor.

6.1 Inicialização dos mutexes

Para que isto fosse possível para cada mutex utilizado foi necessário proceder à inicialização do mesmo, utilizando-se em todos os casos menos num, código semelhante ao exemplo da Figura 6.1.

```
553     if(pthread_mutex_init(&(lockergraphic), NULL) != 0){
554         printf("\n mutex init failed \n");
555         exit(-1);
556     }
```

Figura 6.1: Exemplo de inicialização do mutex para bloqueio da impressão no tabuleiro

6.2 Bloqueio da Gráfica

Do lado do servidor que contém um tabuleiro gráfico em SDL2, foi necessário bloquear a impressão de jogadas no mesmo, pois descobriu-se um "bug"da SDL2 quando se metia o jogo a correr com pelo menos 2 Bots. O bug consistia em que estando o jogo a correr com dois bots e estes enviado jogadas em tempos muito próximos, as mesmas eram frequentemente impressas em momentos quase simultâneos.

Verificava-se que quando isto acontecia, por vezes no primeiro jogo, por vezes noutros, mas nalgum momento, o servidor crashava devido à parte gráfica. Daí decidiu-se colocar um bloqueio na impressão no tabuleiro do Server, em que antes de se imprimir uma peça no Server se bloqueava o código que fazia essa impressão e depois de se imprimir essa peça se desbloqueava. Como mostra a função **printpiece**, do ficheiro **memserver.c** na Figura 6.2.

```
553     if(pthread_mutex_init(&(lockergraphic), NULL) != 0){
554         printf("\n mutex init failed \n");
555         exit(-1);
556     }
```

Figura 6.2: Função printpiece que executa a impressão de alterações no tabuleiro do lado do Server

6.3 Bloqueio do Tabuleiro

Existindo vários jogadores a jogar em simultâneo e tendo cada um a sua própria thread, verificou-se que se podia ter várias threads a tentar aceder às mesmas peças, o que poderia gerar problemas de sincronização. Este foi o mutex que se inicializou de modo diferente, pois aqui em vez de se utilizar um único mutex recorreu-se a um vector destes.

A inicialização deste vector pode ser observada na Figura 6.3.

```
544     locker = (pthread_mutex_t *)malloc(dim * sizeof(pthread_mutex_t));
545     verifyalloc((void *)locker);
546     for(int i=0; i < dim; i = i + 1){
547         if(pthread_mutex_init(&(locker[i]), NULL) != 0){
548             printf("\n mutex init %d failed \n", i);
549             exit(-1);
550         }
551     }
```

Figura 6.3: Inicialização do vector de mutexes para bloqueio do tabuleiro

Utilizou-se um vector, pois recorrendo a um único mutex estaria-se a bloquear sempre o tabuleiro todo, o que tornaria o jogo bastante lento e pouco ineficiente. Recorrendo a um vector de mutexes do tamanho dim, poder-se-ia bloquear apenas linha a linha a análise do tabuleiro.

Não se mostra o troço de código, pois considera-se a função demasiado grande para colocar aqui, mas para executar este bloqueio, na função **board_play**, do ficheiro **board_library.c** bloqueia-se o troço de código consoante a linha **x**. Por fim, após já não ser mais necessário, consoante as condições da posição a analisar, procede-se ao desbloqueio do troço de código utilizando a função **pthread_mutex_unlock**.

6.4 Bloqueio da Flag de Ignore

Em cada jogador existe uma flag de ignore, para quando as jogadas destes precisam de ser ignoradas devido a questões como ter errado uma jogada nos últimos 2 segundos. Como podem existir várias threads aceder a esta flag em simultâneo, incluiu-se na estrutura do jogador um mutex para bloquear esta flag. Por fim, para qualquer troço de código que tentasse aceder a esta flag para cada jogador, era bloqueado antes da leitura ou escrita do valor da flag e desbloqueado depois da leitura da flag. Tudo isto recorrendo ao mutex individual desta flag para cada jogador.

6.5 Bloqueio da Flag de State

Para cada jogador existia também uma flag que indicava o estado do mesmo, se activo ou inactivo, inactivo em situações como o abandono do jogo. Isto para que em certos casos não se tentasse enviar informação para o jogador ou aceder a informação do jogador. Esta flag também podia estar a ser escrita ou lida em várias threads em simultâneo, logo para que não existissem erros de sincronização, acrescentou-se um segundo mutex à estrutura player e utilizou-se esse mutex do state para bloquear os troços de código que acediam a esta flag, antes da leitura/escrita e desbloquear depois da leitura/escrita.

Capítulo 7: Descrição das várias funcionalidades implementadas

7.1 Número mínimo de jogadores

Foi necessário criar a condição de se iniciar o jogo com pelo menos dois jogadores. Perante isto, foi criada a variável **activeplayers**, que possui o número de jogadores ativos, sendo que esta variável decrementa quando existe a morte/disconexão de um cliente que já esteve ligado ao servidor.

7.1.1 Início do jogo

```
printf("Waiting for at least 2 players to login!\n");
pthread_create(&logins, NULL, newplayers, NULL);
StartingSDL();

/* Main Loop */
while(!done){
  init_board(dim);
    while(activeplayers <  2){
        sleep(1);
    }
    create_board_window(300, 300,  dim, window_title);
```

Figura 7.1: Condição até ao início do jogo com um mínimo de 2 clientes

Como se pode observar através do código da Figura 7.1, é criada a thread que fica encarregue de preparar a ligação e a atribuição dos parâmetros ao cliente. De seguida, é feita a condição de o jogo não iniciar até existirem pelo menos dois jogadores ativos. Após a condição, as boards do servidor e dos clientes são criadas.

7.2 Distinção da primeira e segunda jogada

A distinção entre a primeira e segunda jogada é feita, essencialmente, pela variável **code** presente na estrutura (play_response) **resp** e pela função **board_play**.

Ao receber as coordenadas da peça selecionada pelo cliente, é verificado se a **play1[0]** está vazia, em que tal acontece caso assuma valor **-1**. Sendo assim, é possível diferenciar facilmente a primeira da segunda jogada do cliente e com base nisso fazer, mais tarde, as respetivas validações da peça selecionada pelo jogador.

```
if(play1[0]== -1){
    printf("FIRST\n");
    resp.code =1;
    play1[0]=x;
    play1[1]=y;
    resp.play1[0]= play1[0];
    resp.play1[1]= play1[1];
    strcpy(resp.str_play1, get_board_place_str(x, y));
    board[linear_conv(x, y)].state = 1;
    pthread_mutex_unlock(&(locker[x]));
```

Figura 7.2: Verificação se a peça selecionada corresponde à primeira ou à segunda jogada

7.3 Implementação dos 5 segundos após a primeira jogada

Surgiu a necessidade de, caso após a primeira jogada não seja feita uma segunda, voltar a virar a carta selecionada na primeira. Perante isto, foi criada a thread **timerfplay**, representada na Figura 7.3.

```
void*timerfplay(void *arg){
    respplayer *aux = (respplayer *)arg;
    play_response prev;

    prev.play1[0] = aux->resp->play1[0];
    prev.play1[1] = aux->resp->play1[1];
    sleep(5);
    if(aux->resp->code == 1 && prev.play1[0] == aux->resp->play1[0] &&
    prev.play1[1] == aux->resp->play1[1]){
        aux->resp->code = -1;
        dealwithresp(aux->resp, aux->p);
        getbackfirst(aux->p->play);
    }
    return 0;
}
```

Figura 7.3: Thread responsável por orientar os 5 segundos após a primeira jogada

Através do código apresentado, percebemos que é feito o sleep de 5 segundos e que, após esse, é verificado se apenas foi feito a primeira jogada (code == **1**) e se as posições selecionadas mantêm-se as mesmas.

7.3.1 Turn da carta caso não haja 2ª jogada

O turn é feito através da função **dealwithresp** (agora com o code do resp == **-1**) que volta a colocar a peça a branco, tal como mostra o código da figura 7.4

```
case -1: /* Free one piece */
    freethepiece(resp->play1[0], resp->play1[1]);
    peca = sendpiecetoclient(p, resp->play1, resp->str_play1, 0, 255, 255, 255);
    print_piece(peca);
    free(peca);
    break;
```

Figura 7.4: Case -1 da função dealwithresp

7.4 Fim do Jogo

O fim do jogo acontece quando não existem mais peças por ser preenchidas. Para tal, foi usado a variável **n_corrects** que corresponde ao número de peças já corretamente preenchidas. Sendo assim, quando o número de **n_corrects** assume valor dim*dim (número de peças total), podemos considerar que o jogo chegou ao fim.

```
if (strcmp(first_str, secnd_str) == 0){
  printf("CORRECT!!!\n");
  board[linear_conv(play1[0], play1[1])].state = 2;
  board[linear_conv(x, y)].state = 2;
  corr +=2;
  n_corrects +=2;
  if (n_corrects == dim_board* dim_board){
      resp.code =3;
      pthread_mutex_unlock(&(locker[x]));
  }
  else{
    resp.code =2;
    pthread_mutex_unlock(&(locker[x]));
  }
```

```
case 3: /* End Game */
    savethecolor(p->r, p->g, p->b, resp->play1[0], resp->play1[1]);
    savethecolor(p->r, p->g, p->b, resp->play2[0], resp->play2[1]);
    peca = sendpiecetoclient(p, resp->play1, resp->str_play1, 0, 0, 0, 0);
    print_piece(peca);
    free(peca);
    peca = sendpiecetoclient(p, resp->play2, resp->str_play2, 1, 0, 0, 0);
    print_piece(peca);
    free(peca);
    endgame = 1;
    break;
```

Figura 7.5: Código board_play || Código dealwithresp

Foi então acrescentado um tipo de code (==**3**), que corresponde precisamente ao fim do jogo, de modo a proceder à transmissão do vencedor e às outras tarefas para realizar o reinício do jogo.

7.4.1 Transmissão do vencedor

De modo a enviar o vencedor ao respectivo cliente e o perdedor aos restantes, foi utilizado a variável **corr** da estrutura player, possuindo o número de peças acertadas pelo respectivo jogador.

Após a declaração do fim do jogo, a lista de jogadores é percorrida de modo a procurar o jogador com mais peças acertadas, tal como mostra figura 7.6. Após encontrar o vencedor, o servidor envia a peça com a variável **end** (da estrutura piece) de valor **-1**.

Posto isto, é feito o envio do aviso de fim de jogo aos restantes clientes, tal como mostra a figura 7.7.

```
piece *sendpiecetoclient(player *p, int play[2], char *str, int e, int wr, int wg, int wb){
    player_node *auxplayer;
    piece *peca;
    player *winner;
    char *data = malloc(sizeof(piece));

    if(e == 1){
        auxplayer = phead;
        winner = phead->p;
        while(auxplayer != NULL){
            if(auxplayer->p->corr >= winner->corr)
                winner = auxplayer->p;
            auxplayer = auxplayer->next;
        }
        peca = producepiece(p, play, str, -1, wr, wg, wb);
        memcpy(data, peca, sizeof(piece));
        pthread_mutex_lock(&(winner->statelock));
            if(winner->state == 1)
                send(winner->socket, data, sizeof(piece), 0);
        pthread_mutex_unlock(&(winner->statelock));
    }
```

Figura 7.6: Código sendpiecetoclient - fim do jogo

7.4.2 Implementação dos 10 segundos de delay até ao próximo jogo

A implementação dos 10 segundos é feita através do sleep presente na figura 7.7, onde a seguir é feito, mais uma vez, a criação de uma nova board, juntamente com a espera do mínimo de dois jogadores, onde se volta a repetir a mesma lógica que o primeiro jogo.

7.4.3 Reinício do jogo no cliente

Após os 10 segundos, é realizado o clean up da board utilizada durante o jogo já acabado, de modo a não haver informação relativa ao jogo anterior.

Observando a figura 7.8, percebe-se que a tal board do jogo anterior é destruída, sendo posteriormente criado uma nova para o novo jogo.

```
/* Main Loop */
while(!done){
  init_board(dim);
    while(activeplayers <  2){
        sleep(1);
    }
    create_board_window(300, 300,  dim, window_title);
    endgame = 0;
    sendpiecetoclient(NULL, NULL, NULL, 0, 0, 255, 0);
    while(endgame == 0){
        while(SDL_PollEvent(&event)){
            switch(event.type){
                case SDL_QUIT: {
                    serverkill();
                    exit(-1);
                }
            }
        }
    }
    sendpiecetoclient(NULL, NULL, NULL, 0, 0, 0, 255);
    close_board_windows();
    sleep(10);
}
mutexdestroy();
}
```

```
if(p->end != 0  &&  endignore == 0){
  printf("The game ended, in ten seconds a new window will appear!\n");
  activegame = 0;
  endignore = 1;
  if(p->end == -1)
    printf("You are the winner!\n");
  sleep(2);
  endgame = 1;
}
```

Figura 7.7: Código memserver: main || memclient: recv_from_server

7.5 Clean up após a morte/disconexão de um cliente

Quando um jogador realiza a desconexão/morte, este envia uma posição nula (x = **-1** e y = **-1**) que quando recebida pelo servidor ativa a função **prepare_client_exit**.

Esta função limita-se a meter o **state** deste jogador a **0**, de modo a que o servidor não envie as posteriores alterações da board para este cliente, pois o socket associado a este é fechado também nesta função. Posteriormente e, ainda nesta função, é feito de imediato o turn da 1ª jogada caso o jogador não tenha selecionado a 2ª carta.

7.6 Bot

O bot implementado corresponde basicamente a um cliente, em que este envia jogadas por si só ao invés de esperar por um clique de um utilizador. Para a geração da coordenada jogada foram usadas as funções **getcoordinates** e **filled**, representadas na figura 7.9.

Como é possível ver, é usada a variável **pieces** que corresponde a uma matriz com as mesmas dimensões que a board. O seu objetivo é possuir valor **1** nas respetivas coordenadas da board já corretamente preenchidas e possuir valor **0** nas restantes coordenadas. Sendo assim, o bot apenas gera jogadas em posições não corretamente preenchidas!

Com isto o bot consegue ser mais eficaz, visto que previne o facto de selecionar uma 1ª jogada e posteriormente uma 2ª jogada numa peça já correcta/selecionada, que faria com que a 1ª carta

```
while (!done){
  while(activegame == 0)
    sleep(1);
  endgame = 0;
  endignore = 0;
  while(endgame == 0){
    while (SDL_PollEvent(&event)) {
      switch (event.type) {
        case SDL_QUIT: {
          done = SDL_TRUE;
          pthread_create(&exitt, NULL, exitthread, NULL);
          endgame = 1;
          break;
        }
        case SDL_MOUSEBUTTONDOWN:{
          if(activegame == 1){
            get_board_card(300/dim, 300/dim, event.button.x, event.button.y, &(bp->x), &(bp->y));
            //printf("click (%d %d) -> (%d %d)\n", event.button.x, event.button.y, bp->x, bp->y);
            pthread_create(&thread_send, NULL, sendpos, (void*) bp);
          }
        }
      }
    }
  }
  close_board_windows();
      if(!done){
      sleep(8);
      create_board_window(300, 300,  dim, window_title);
      }
```

Figura 7.8: Código memclient: Main loop

```
void getcoordinates(boardpos *bp){
    do{
        bp->x = random() %dim;
        bp->y = random() %dim;
    }while(filled(bp->x, bp->y));
}
```

```
/** filled: Function that verifies if a piece is empty or ffilled
 * \param x - x coordinate of the piece on the board
 * \param y - y coordinate of the piece on the board
 * \return - Returns 0 if the piece is empty, 1 if it is filled
 */
int filled(int x, int y){
    if(pieces[x][y] == 0)
        return 0;
    return 1;
}
```

Figura 7.9: Código bot2: getcoordinates || filled

voltasse a dar o turn.