Relatório do Projeto

Programação de Sistemas

2018/2019

Inês Bernardino - 83689
Rodrigo Oliveira - 83728

Mestrado Integrado em Engenharia Aeroespacial

Docentes:
João Nuno De Oliveira e Silva
Ricardo Miguel Ferreira Martins

Índice

1 Arquitetura

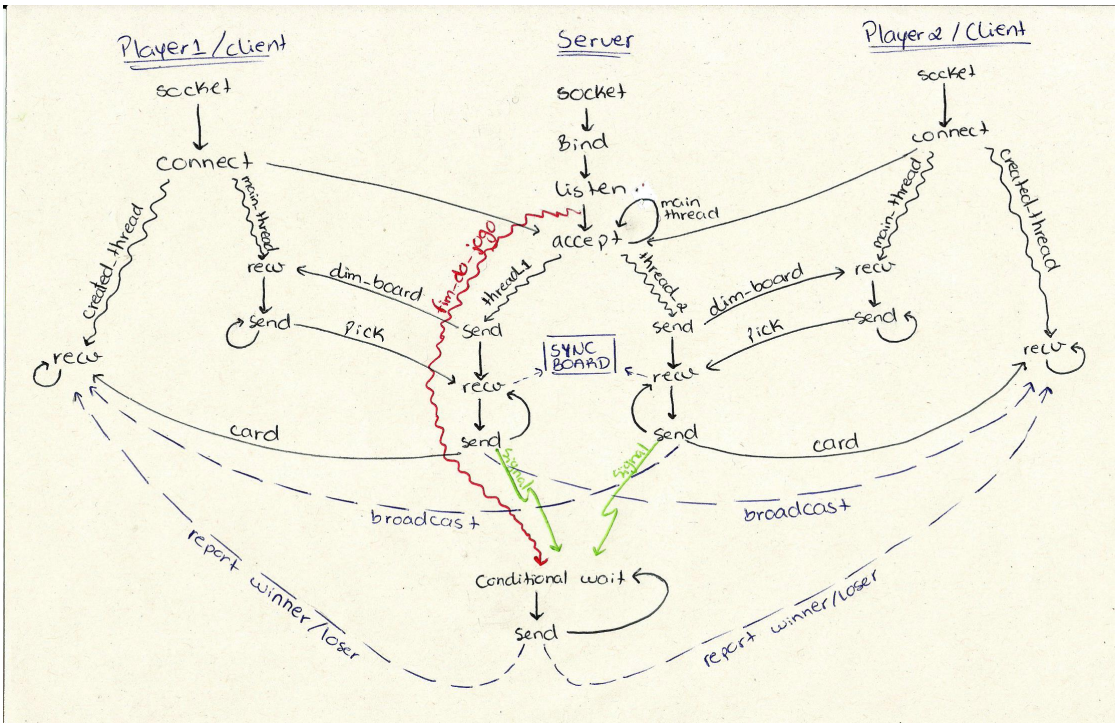

Figura 1: Arquitetura do Programa

Na figura 1 podemos observar um esquema gráfico do decurso da lógica do programa quando estão conectados 2 jogadores/clientes.

Existem 5 tipos de threads no programa, são eles:

- main_thread (Server)
- thread_1 / thread_2 (Server)
- fim_do_jogo (Server)
- main_thread (Client)
- created_thread (Client)

main_thread (Server)

Thread existente na função *main* do programa do Servidor, colocada em ciclo infinito de aceitação de novos jogadores. Quando um novo cliente se conecta, incrementa a variável *numero_jogadores* que contem o número de jogadores ativos no jogo em cada momento, cria uma nova cor para lhe atribuir, adiciona o novo cliente à lista de clientes

e, por fim, cria uma thread (*thread_1 ou thread_2*, por exemplo) para o tratamento das jogadas deste jogador.

thread_1 / thread_2 (Server)

Thread criada no ciclo infinito da função *main* do programa do Servidor, após o acept de um cliente, para o tratamento das jogadas deste jogador.

É criada 1 thread destas por cada cliente ativo e é destruída quando o cliente se desconecta.

Começa por enviar para o seu cliente a dimensão do tabuleiro. Depois fica presa num ciclo infinito de receção de jogadas do cliente, tratamento das mesmas e posterior envio para todos os clientes das alterações necessárias ao tabuleiro. Quando é atingido o fim do jogo, lança um sinal, de variáveis condicionais, para desbloquear a thread *fim_do_jogo*.

fim_do_jogo (Server)

Thread criada antes do ciclo infinito da função *main* do programa do Servidor, para o tratamento dos procedimentos aquando do fim do jogo.

A thread é constituída por um ciclo infinito. Esta é lançada e fica imediatamente bloqueada no início, sendo desbloqueada por um sinal de uma variável condicional, presente na thread do jogador que efetue a última jogada.

Esta thread começa por ativar a variável *ignora* a 1, para todas as jogadas efetuadas nos 10 segundos seguintes serem efetuadas e não tratadas, verifica que jogador (ou jogadores) é o vencedor, envia para cada jogador a informação de que é vencedor ou perdedor, remove da lista de jogadores os clientes inativos, faz um novo *init_board* onde gera um novo tabuleiro e, por fim, após a espera de 10 segundos, pinta o tabuleiro do servidor a branco e volta a desativar a variável *ignora=0* para que daí em diante as jogadas possam ser tratadas pela thread dos jogadores.

A *condicional wait* foi implementada para que não fosse preciso criar uma nova thread de *fim_do_jogo* por cada fim do jogo registado, assim irá ser criada apenas 1 em todo o programa.

main_thread (Client)

Thread existente na função *main* do programa do Cliente, colocada em ciclo infinito de recolha de novas jogadas e posterior envio para o servidor.

created_thread (Client)

Thread criada antes do ciclo infinito da função *main* do programa do Client, para o tratamento das mensagens recebidas pelo servidor após uma jogada.

Esta thread efetua 1 de 3 tarefas, sendo elas, pintar a carta escolhida, escrever as letras na carta escolhida ou anunciar que o jogador é vencedor ou perdedor.

2 Organização do Código

O nosso código está distribuído por 10 ficheiros, dos quais, 6 são ficheiros .c e 4 são ficheiros .h.

Os ficheiros .c são:

- server.c
- client.c
- simple_bot.c
- lista.c
- board_library.c
- UI_library.c

Os ficheiros .h são:

- connection.h
- lista.h
- board_library.h
- UI_library.h

Os ficheiros .h contêm os cabeçalhos funções existentes no respetivo .c, sempre que estas funções forem executadas por outra função fora do .c. Contêm também algumas estruturas de dados relevantes ao respetivo .c. Estas estruturas de dados serão explicadas em detalhe na secção 3.

Vamos agora explorar todas as funções de cada ficheiro .c.

2.1 server.c

handle_disc_client

Função simples que apenas faz um printf quando o servidor tentou escrever num socket já fechado pelo cliente.

ctrl_c_callback_handler_server

Função executada quando é detetado CTRL_C no terminal do Servidor. Fecha os sockets de todos os jogadores ativos, apaga a lista de jogadores, fecha o socket em si (para que nenhum novo cliente se possa conectar), destrói todos os *mutex* existentes e liberta toda a memória alocada.

preenche_buffer

Função de preenchimento dos campos da estrutura de dados, *layer_com*, de apoio à comunicação do servidor para o cliente.

Recebe como input os dados que queremos preencher na estrutura e retorna a estrutura preenchida.

board_client_meio

Função que envia para o cliente que se conectou já a meio do decurso do programa, a informação com o estado de todas as cartas do tabuleiro, de modo a que este jogador possa ver no seu tabuleiro, uma versão atualizada do mesmo, em vez de um novo tabuleiro em branco.

Recebe como input o file descriptor do jogador que se conectou a meio.

out_client

Quando um jogador se desconecta, as suas cartas ativas no tabuleiro (estado = 1 ou 2) são reiniciadas (estado = -2 e cor = -1) e voltadas para baixo (Pintadas de branco). Os pares previamente feitos pelo jogador (estado=0) permanecem assim.

atualiza_board

Atualiza a versão local das cartas, presente na estrutura *board_place*, atribuindo a cada carta, a cor com a qual ela está pintada, isto é, branco quando o estado a colocar for -2 ou a cor do jogador quando o estado a colocar for um dos outros.

A função recebe como input o estado a que se pretende colocar a carta e as coordenadas x e y da carta em questão.

fim_do_jogo

Função de apoio à thread do fim do jogo, previamente explicada na secção 1.

thread_func

Função de apoio à thread de cada jogador, que efetua o tratamento das jogadas deste jogador, previamente explicada na secção 1.

Começa por enviar para o cliente a dimensão do tabuleiro. Caso o cliente se conecte a meio do jogo, atualiza o board para este cliente.

Entra num ciclo infinito de tratamento de jogadas que se inicia com a leitura das coordenadas da carta escolhida, enviadas pelo cliente. Caso o cliente se desconecte, a função *read*, retorna 0, e portanto decrementamos o número atual de jogadores, atualizamos a board caso se verifique necessário e atualizamos o estado do jogador para inativo. Segue-se a verificação das variáveis *ignorar* e *numero_jogadores*. Caso *ignorar=1* (10 segundos após fim de jogo ou 2 segundos após 2º jogada incorreta) ou *numero_jogadores* < 2, as jogadas serão ignoradas.

Caso estas condições acima mencionadas não se verifiquem, é então feito o tratamento das jogadas, o qual se divide em 6 casos:

- *case=1*, 1º pick, atualizar o tabuleiro, pintar a carta da cor do jogador e as letras de cinzento. Estas informações são enviadas para todos os jogadores ativos. É também inicializado o timer de 5 segundos.

- *case=0*, o timer chegou a 0 sem que fosse efetuada a 2º pick. O tabuleiro deve ser atualizado e a carta correspondente à 1º pick deve ser pintada de branco no servido e em todos os clientes.
- *case=-1*, a 2º pick do jogador foi a 1º pick ou a uma outra carta do tabuleiro já voltada para cima. O tabuleiro deve ser atualizado e a carta correspondente à 1º pick deve ser pintada de branco no servido e em todos os clientes.
- *case=3*, é atingido o fim do jogo. Incrementa-se o score do jogador que completou o último par, atualiza-se o tabuleiro e pinta-se no servidor e em todos os clientes as 2 cartas da cor do jogador e as letras de preto. É lançado um sinal da *conditional variable* para desbloquear a thread de fim de jogo.
- *case=2*, a 2º pick do jogador é o par da 1º pick. Incrementa-se o score do jogador, atualiza-se o tabuleiro e pinta-se no servidor e em todos os clientes as 2 cartas da cor do jogador e as letras de preto.
- *case=-2*, a 2º pick não é o par da 1º pick. Atualiza-se o tabuleiro e pinta-se no servidor e em todos os clientes as 2 cartas da cor do jogador e as letras de vermelho. É lançado um timer para a contagem de 2 segundos de espera e quando este chega ao fim, volta atualizar o tabuleiro e a pintar de branco as 2 cartas no servidor e nos clientes.

De certo modo, podemos dizer que é a função principal do programa.

main

Recebe através da linha de comandos a dimensão do tabuleiro. Confirma se esta dimensão é par e inferior ou igual a 26.

Cria o socket, cria uma janela gráfica para o servidor, cria a thread de apoio ao fim do jogo, inicia a lista de jogadores e por fim, entra num ciclo infinito de aceitação de clientes e criação de uma thread por cada cliente novo, tal como foi explicado em maior detalhe na secção 1.

2.2 client.c

ctrl_c_callback_handler

Função executada quando é detetado CTRL_C no terminal do Cliente.

Fecha o socket de comunicação com o servidor e liberta a memória previamente alocada.

thread_func

Função de apoio à thread criada para a receção de informações do servidor. Efetua 1 de 3 tarefas, mediante a flag recebida no buffer de comunicação enviado pelo servidor, sendo elas, pintar a carta escolhida, escrever as letras na carta escolhida ou anunciar que o jogador é vencedor ou perdedor, tal como foi previamente explicado na secção 1.

main

Começa por se conectar ao socket, recebe a dimensão do tabuleiro, cria uma janela gráfica, lança a thread de receção de informação do servidor e entra num ciclo infinito de recolha de jogadas através do rato e posterior envio das mesmas para o servidor, tal como referido na secção 1.

2.3 simple_bot.c

ctrl_c_callback_handler

Função igual à da secção 2.2.

main

Função semelhante à da secção 2.2.

Começa por se conectar ao socket, recebe a dimensão do tabuleiro, contudo, como é um cliente virtual, não necessita de janela gráfica nem terá recolha de jogadas pelo rato. Em vez disto, entra num ciclo infinito de geração aleatória de jogadas, dentro dos limites do tabuleiro e posterior envio das mesmas para o servidor.

2.4 lista.c

IniciaLista

Inicia a lista de jogadores.

Nesta lista, o primeiro elemento não tem informação útil, desta forma o topo da lista é sempre constante e nunca tem de ser atualizado.

AdicionaElemento

Adiciona um novo elemento à lista de jogadores.

Adiciona sempre ao fim da lista uma vez que no nosso caso, a ordem dos jogadores não é relevante.

Cria um novo elemento, alocando-lhe memória, inicia os campos da estrutura de dados e por fim, insere-o na lista.

Recebe como input um apontador para o início da lista de jogadores, o file descriptor do jogador a adicionar, a sua cor e o apontador para o mutex da lista.

Retorna um apontador para a estrutura de dados do novo jogador.

MostraElementosLista

Função criada para *debugging*. Imprime, quando invocada, o estado da lista, isto é a cor de cada jogador, o seu file descriptor, o seu score e o seu estado e no fim indica o número total de jogadores na lista (isto é jogadores ativos e inativos que já jogaram o jogo atual).

Recebe o apontador da lista de jogadores e o apontador do mutex desta lista.

vizinhanca

Função criada para que as cores geradas aleatoriamente para cada jogador, não sejam muito próximas de outras já existentes ou das cores proibidas.

Recebe 3 parâmetros da cor gerada e 3 parâmetros da cor com a qual queremos comparar.

Retorna 1 se a cor gerada estiver ok e portanto fora da vizinhança da outra cor, retorna 0 se a cor gerada for próxima ou até igual da cor com a qual a estamos a comparar.

cria_cor

Gera, ciclicamente, 3 parâmetros de cor enquanto a cor não for declarada como *ok*. Após geração dos parâmetros de cor estes são comparados, em 1º lugar, com as 4 cores proibidas (cinzento, branco, preto e vermelho) e depois com as cores dos outros jogadores.

Se a cor gerada não estiver na vizinhança de nenhuma das cores testadas, a função sai do ciclo e retorna a cor gerada.

Recebe o apontador da lista de jogadores e o apontador do mutex desta lista.

retira_lista

Esta função não remove efetivamente o jogador da lista, apenas o desativa, colocando o bit *ativo=0* e fecha o socket de comunicação do cliente que se removeu.

Recebe o apontador da lista de jogadores, a cor do jogador a remover e o apontador do mutex desta lista.

apaga_lista

Apaga a lista e faz free da memória para ela alocada.

Recebe o apontador da lista de jogadores e o apontador do mutex desta lista.

atualiza_lista_end_game

Percorre a lista e procura elementos inativos e caso encontre, tira-os da lista e faz free da sua memória.

Recebe o apontador da lista de jogadores e o apontador do mutex desta lista.

close_sockets

No caso do servidor se fechar, apaga as linhas de comunicação ativas.

Recebe o apontador da lista de jogadores e o apontador do mutex desta lista.

broadcast

Preenhe a estrutura de dados *layer_com* para comunicação do servidor com os clientes, com a informação desejada, percorre tod a a lista e envia para todos os jogadores ativos a informação.

Recebe o apontador da lista de jogadores, as coordenadas x e y da carta, 3 parâmetros de cor, a flag que ditará no cliente o que fazer com a informação, a string da carta e o apontador do mutex desta lista.

ve_vencedor

Percorre a lista de jogadores em busca do score mais elevado registado.

Percorre novamente a lista para atribuir a esse (ou esses) jogador o estado de vencedor, pondo o campo *vencedor = 1*.

Recebe o apontador da lista de jogadores e o apontador do mutex desta lista.

2.5 board_library.c

linear_conv

Recebe as coordenada x e y de uma carta e converte-as para um único número inteiro através de $y * dim\_board + x$.

Retorna esta coordenada linear.

get_board_place_str

Recebe as coordenada x e y de uma carta e retorna a string associada à carta desta posição.

get_board_place_card

Recebe as coordenada x e y de uma carta e retorna um apontador para a estrutura de dados da carta desejada.

init_board

Aloca memória para *dim_board * dim_board* estroturas de dados de cartas e inicia os campos de todas elas, deixando as cartas inativas.

Gera de forma pseudo aleatória as strings de cada carta e garante que existem sempre pares possíveis para cada carta.

Imprime no terminal do Servidor a string presente em cada posição.

Recebe a dimensão do tabuleiro e retorna um apotador para o vetor de cartas.

board_play

Analisa as jogadas e atribui um *case* dos já vistos anteriormante na secção 2.1 consoante o jogador esteja na 1º ou 2º pick, e dentro da 2º pick, consoante a relação desta com as restantes cartas.

A cada jogada atualiza o estado da carta.

Recebe as coordenadas x e y, um apontador para a 1º pick, um apontadpr para a cor do jogador, um mutex do score total do jogo e mutex para as cartas.

2.6 UI_library.c

write_card

Pinta a string da cor desejada.
Recebe as coordenada x e y de uma carta, a string que queremos pintar e 3 parâmetros de cor com a qual pretendemos pintar a string.

paint_card

Pinta a carta da cor desejada.
Recebe as coordenada x e y de uma carta e 3 parâmetros de cor com a qual pretendemos pintar a string.

clear_card

Pinta a carta de branco.
Recebe as coordenada x e y de uma carta.

get_board_card

Transforma os valores recebidos pelo rato quando uma carta é selecionada e transforma-os em coordenadas perceptíveis.
Recebe os valores x e y do rato e apontadores para as novas coordenadas x e y.

create_board_window

Cria/desenha a janela gráfica.
Recebe o número de pixeis desejados de largura e altura e a dimensão do tabuleiro.

reinicia_board_branco

Volta a pintar a branco todas as cartas a janela gráfica.
Recebe o número de pixeis desejados de largura e altura e a dimensão do tabuleiro.

close_board_windows

Fecha uma janela gráfica.

3 Estruturas de Dados

Ao todo, no nosso projeto, existem 5 estruturas de dados. São elas:

- layer_com
- coordinates

- jogador
- board_place
- play_response

layer_com

```
//connection.h
typedef struct layer_com{
    int x;
    int y;
    int r;
    int g;
    int b;
    short flag; //0 para print_card, 1 para write_card, 2 end game
    char string[3];
}layer_com;
```

Esta estrutura serve de apoio à comunicação entre o servidor e os vários clientes. É através dela que são enviadas informações para o cliente pintar o seu tabuleiro.

x e y indicam a posição da carta que deve ser pintada.

r, g e b codificam a cor com a qual a pintura vai ser efetuada.

$flag$ vai indicar a ação a efetuar do lado do cliente. Quando $flag = 0$, a carta é pintada invocando a função *paint_card*, quando $flag = 1$ pintam-se as letras através da função *write_card* e finalmente quando $flag = 2$ ocorreu o fim do jogo, e são feitos os prints de vencedor ou perdedor (em caso de fim de jogo, x irá conter o score do jogador e $y = 1$ indicará que o jogador é vencedor e $y \neq 1$ indicará que o jogador é perdedor).

Finalmente, *string* é um vetor de 3 posições, as 2 primeiras para cada uma das letras presente nesta carta escolhida pelo jogador e a última para o \0 a sinalizar o fim da string.

coordinates

```
//connection.h
typedef struct coordinates{
  int x;
  int y;
}coordinates;
```

Esta estrutura serve de apoio à comunicação entre um cliente e o servidor. É através dela que são enviadas para o servidor, as coordenadas da carta escolhida pelo jogador.

x e y indicam a posição da carta escolhida pelo jogador e que deve ser processada pelo servidor.

jogador

```
//lista.h
typedef struct jogador jogador;
typedef struct jogador{
```

```
4   int player_fd;
5   int cor[3];
6   int score;
7   jogador* next;
8   int ativo; //1 jogador ativo, 0jogador saiu
9   int vencedor;
10 } jogador;
```

Esta estrutura serve de apoio à lista. Existe uma variável deste tipo por cada jogador existente.

player_fd contem o *file descriptor* do jogador.

cor é um vetor de 3 posições de inteiros, uma para cada um dos 3 números que codificam a sua cor. A posição 1 codifica r, a 2 codifica g e a 3 codifica b.

score indica o resultado de cada jogador, ou seja, o número de pares por ele feitos.

next é um apontador para o jogador que depois dele se conectou, de modo a manter todos os jogadores numa lista, sendo assim possível aceder a todos eles quando necessário.

ativo é uma variável que indica se o jogador ainda se encontra ativo ($ativo = 1$) no jogo ou se já se desconectou ($ativo = 0$). Esta variável foi introduzida uma vez que optámos por não remover o jogador da lista aquando da sua desconexão, fazendo-o apenas quando ocorre o fim do jogo. Assim esta variável inicia-se a 1 quando o jogador se conecta e passa a 0 quando se desconect, permite-nos continuar o jogo e efetuar *broadcast* apenas para os jogadores ativos.

vencedor é uma variável que é ativada a 1 no fim do jogo para o jogador vencedor (ou os jogadores vencedores).

board_place

```
1 //board_library.h
2 typedef struct board_place{
3   char v[3];
4   int cor[3]; //(-1,-1,-1)
5   int estado; //0 para par feito(letras a preto), 1 para 1st pick(cinzento)
6               //2 para jogada errada (vermelho)  -2 inicializacao
7 } board_place;
```

Esta estrutura serve de cópia em tempo real de cada posição do board. Toma um papel particularmente importante quando um cliente se conecta a meio de um jogo e precisa de receber o estado de cada posição do tabuleiro, bem como a cor com que a deve pintar.

v é um vetor de 3 posições, as 2 primeiras para cada uma das letras presente nesta carta e a última para o \0 a sinalizar o fim da string.

cor é um vetor de 3 posições de inteiros, uma para cada um dos 3 números que codificam a cor da carta. A posição 1 codifica r, a 2 codifica g e a 3 codifica b.

estado indica o estado atual de cada carta. É importante saber o estado para podermos pintar as letras da cor certa. Assim, o estado será 0 se a carta estiver emparelhada, sendo a string pintada de preto, $estado = 1$ se só tiver sido efetuada a 1º pick, sendo a string pintada de cinzento, $estado = 2$ se tiver sido feita uma 2º pick incorreta, pintando

assim as strings de vermelho e finalmente *estado* = −2 indica que a carta está fora de jogo e virada para baixo.

play_response

```
//board_library.h
typedef struct play_response{
  int code;
            // 10 - na inicializacao
            // -1 - virar 1 carta para baixo
            // 0 - filled / fora de jogo
            // 1 - 1st play
            // 2 2nd - same plays
            // 3 END - fim de jogo
            // -2 2nd - diffrent
  int play1[2];
  int play2[2];
  char str_play1[3], str_play2[3];
} play_response;
```

Finalmente temos a estrutura de apoio e tratamento das jogadas de cada jogador.

play1 é um vetor de inteiros de 2 posições contendo as coordenadas x e y, respetivamente, da 1º pick.

play2 é um vetor de inteiros de 2 posições contendo as coordenadas x e y, respetivamente, da 2º pick.

str_play1 é um vetor de 3 posições, as 2 primeiras para cada uma das letras presente na carta da 1º pick e a última para o \0 a sinalizar o fim da string.

str_play2 é um vetor de 3 posições, as 2 primeiras para cada uma das letras presente na carta da 2º pick e a última para o \0 a sinalizar o fim da string.

Finalmente, *code* indica como proceder no servidor e atualizar os dados e enviá-los para os clientes. *code* = 1 faz o tratamento da 1º jogada. *code* = 0 faz com que a 1º carta seja voltada para baixo porque passaram os 5 segundos. *code* = −1 faz com que a 1º carta seja voltada para baixo porque o cliente voltou a carregar nela na 2º jogada ou porque o cliente carregou numa carta ja escolhida na 2º jogada. *code* = 2 acontece quando a 2º pick do cliente é igual à 1º pick. *code* = −2 acontece quando a 2º pick do cliente é diferente da 1º pick. *code* = 3 ocorre quando é atingido o fim do jogo.

4 Protocolos de Comunicação

Entre os vários nodos (processos) do sistema são enviadas vários fluxos de mensagens ao longo da execução do programa. Note-se que a comunicação é bidirecional e dá-se sempre entre o servidor e cada um dos clientes, não havendo comunicação direta entre dois clientes. Em particular, são enviadas mensagens nas seguintes fases de execução:

- Início do jogo
- Envio de uma jogada (*pick*)

- Atualização do tabuleiro (*board*)
- Fim / reinício do jogo

Início do jogo

Para que cada cliente consiga criar a sua janela gráfica, necessita de conhecer a dimensão do tabuleiro. Esta é passada por argumento (*argv*) ao servidor através da linha de comandos no momento da sua execução. Assim, após a conexão de um cliente, o servidor envia-lhe a dimensão do tabuleiro.

```
//server.c
write(this_player->player_fd, &dim_board, sizeof(dim_board));
```

Envio de uma jogada

Quando um cliente carrega numa posição do tabuleiro, a informação relativa a esta posição (coordenadas x e y) é enviada para o servidor para ser processada. Para tal é utilizada a estrutura de dados do tipo *coordinates*. Esta é enviada para o servidor da seguinte forma:

```
//client.c
int board_x, board_y;
get_board_card(event.button.x, event.button.y, &board_x, &board_y);
coord.x = board_x;
coord.y = board_y;
memcpy(coord_aux, &coord, sizeof(coordinates));
write(sock_fd, coord_aux, sizeof(coordinates));
```

Note-se que a estrutura é previamente serializada para poder ser enviada corretamente através do *socket*.

Posteriormente, o servidor processa a jogada e envia a todos os clientes as alterações a realizar ao tabuleiro. Para tal utiliza-se a estrutura de dados *layer_com*. Este fluxo é processado da seguinte maneira:

```
//lista.c
void broadcast(jogador* lista, int x, int y, int r, int g, int b, int flag,
     char*str,pthread_rwlock_t *mux_lista){
    void* aux_comm = malloc(sizeof(layer_com));
    memset(aux_comm, '0', sizeof(layer_com));

    layer_com buffer;
    buffer = preenche_buffer(x,y,r,g,b,flag,str);
    memcpy(aux_comm, &buffer, sizeof(buffer));

    pthread_rwlock_rdlock(mux_lista);
        for (lista = lista->next; lista!= NULL; lista= lista->next)
            if (lista->ativo==1)
                write(lista->player_fd, aux_comm, sizeof(layer_com));
    pthread_rwlock_unlock(mux_lista);
    free(aux_comm);
}
```

Atualização do tabuleiro

Quando um cliente se conecta a meio de um jogo, é necessário enviar-lhe o estado atual do tabuleiro (pares feitos e jogadas que outros clientes estejam a realizar). Para este fluxo de informação pode ser aproveitada a estrutura de dados utilizada no ponto anterior. No entanto, neste caso a informação é apenas enviada para este cliente (não de trata de um *broadcast* geral). Assim, aquando da conexão de um cliente, o seguinte troço de código é executado:

```
//server.c
//board_client_meio function
  for (i=0; i< dim_board; i++){
     for (j=0; j<dim_board; j++){
        card = get_board_place_card(i,j);
        if (card->cor[0] != -1){ //active card
            buffer = preenche_buffer(i, j, card->cor[0],
   card->cor[1], card->cor[2], 0, "\0");
            memcpy(aux_comm, &buffer, sizeof(buffer));
            write(fd, aux_comm, sizeof(buffer));
            if (card->estado == 0)
                    buffer = preenche_buffer(i, j, 0, 0, 0, 1, card->v);
            else if(card->estado == 1)
                    buffer = preenche_buffer(i, j, 200, 200, 200, 1, card->
   v);
            else if(card->estado == 2)
                    buffer = preenche_buffer(i, j, 255, 0, 0, 1, card->v);
            memcpy(aux_comm, &buffer, sizeof(buffer));
            write(fd, aux_comm, sizeof(buffer));
        }
      }
}
```

Fim / reinício do jogo

O fim do jogo dá-se quando a variável que conta o número de pares feitos (*n_corrects*) atinge o valor máximo ($n\_corrects = dim\_board \times dim\_board$). Quando esta condição se verifica, o servidor diz a cada um dos clientes se estes ganharam ou perderam o jogo, complementando esta informação com o seu *score*.

Para enviar esta informação, é aproveitada a estrutura de dados *layer_com*, que já servia a comunicação servidor → cliente. Assim, utilizou-se o seguinte protocolo:

- No campo x da estrutura é enviado o *score* do jogador em questão;
- No campo y é enviada uma *flag* lógica de vencedor (0 se o cliente em questão perdeu o jogo, 1 se ganhou);
- O campo *flag* é preenchido com o valor 2 para avisar o cliente que o jogo terminou;
- Todos os outros campos são colocados a 0.

```c
//server.c
//fim_do_jogo thread function
for (lista = lista->next; lista!= NULL; lista= lista->next){
    if (lista->ativo == 1){
        layer_com buffer = preenche_buffer(lista->score,lista->vencedor
    ,0,0,0,2,"");
        lista->score =0;
        lista->vencedor=0;
        memcpy(aux_comm, &buffer, sizeof(buffer));
        write(lista->player_fd, aux_comm, sizeof(layer_com));
    }
}
```

5 Validação

5.1 Valores de retorno das funções

No que diz respeito aos valores de retorno das funções, estes são utilizados para percebermos quando é que um dos processos fechou o seu lado da linha de comunicação. Quando isso acontece, o outro processo consegue identificar essa desconexão.

Quando o cliente deteta que o servidor encerrou a linha de comunicação, este liberta a memória alocada e termina o seu processo.

```c
//client.c
ret = read(sock_fd, aux_comm, sizeof(layer_com));
    if (ret == 0){
        printf("O servidor fechou esta linha de comunicacao (servidor
    abortou)\n");
        printf("A fechar cliente\n");
        close(sock_fd);
        free (aux_comm);
        free(coord_aux);
        exit(-1);
    }
```

De forma semelhante, quando o servidor deteta que o cliente se desconectou (também olhando para o valor de retorno do *read*), desativa o cliente (alterando o campo *ativo* da sua estrutura na lista de jogadores) e apaga as suas jogadas não concluídas do tabuleiro.

5.2 Validação da informação transferida entre processos

Não é garantido que a informação enviada pelo *socket* por um processo chegue corretamente ao outro. Uma forma simples de validar a informação é verificar se a função *read* retornou o número de *bytes* esperados.

Assim, tanto no cliente como no servidor, faz-se esta verificação. Exemplifica-se a implementação do lado do cliente (no servidor a implementação é análoga):

```c
//client.c
//thread_func

```

```
ret = read(sock_fd, aux_comm, sizeof(layer_com));
    if (ret == 0){
            //... server disconnected
        }
        else if(ret == sizeof(layer_com)){
            //... process the info read (paint or write card)
    }

//...

//main function

if (read(sock_fd, &dim_board, sizeof(int)) != sizeof(int)){
    printf("board dimension not read successfully\n Exiting process\n");
    exit(-1);
}
```

6 Regiões Críticas / Sincronização

Durante o desenvolvimento dos diferentes processos, identificámos a existência de quatro regiões críticas que, se não forem protegidas, poderiam levar ao aparecimento de *race conditions*, que afetariam o correto funcionamento do programa. São estas:

- A lista de jogadores (*lista*)
- O tabuleiro (*board*)
- A variável que conta o número de pares feitos (*n_corrects*)
- A variável que conta o número de jogadores ativos (*numero_jogadores*)

Lista de jogadores

Para proteger a lista foi criada uma variável de sincronismo do tipo *pthread_rwlock_t*. Considerámos que todos os troços de código que têm ciclos que percorrem a lista são regiões críticas associadas a esta variável. No entanto as funções que apenas acedem aos elementos (por exemplo a função de *broadcast*) podem ser realizadas em simultâneo, pelo que nestas funções a região crítica é protegida, em modo *read*, com o seguinte código:

```
pthread_rwlock_rdlock(mux_lista);
  for (lista = lista->next; lista!= NULL; lista= lista->next){
            //Acesso aos elementos da lista em modo leitura
      }
pthread_rwlock_unlock(mux_lista);
```

Por outro lado, as funções que adicionam e retiram elementos da lista (*AdicionaElemento*, *Apaga_Lista* e *atualiza_lista_end_game*) não podem ser realizadas em paralelo com outras funções que também acedam à lista. Assim, a região crítica neste caso é definida em modo *write*, da seguinte forma:

```
pthread_rwlock_wrlock(mux_lista);
  for (lista = lista->next; lista!= NULL; lista= lista->next){
            //Inserir ou retirar elementos da lista
      }
pthread_rwlock_unlock(mux_lista);
```

Tabuleiro

Para proteger o tabuleiro foram utilizadas variáveis de sincronismo do tipo *pthread_mutex_t*. Optámos por criar uma variável *mutex* para cada posição do tabuleiro. Desta forma torna-se possível o processamento de diferentes jogadas (em diferentes posições) em simultâneo. Dado que o tamanho do tabuleiro não é conhecido à partida, é necessário criar, no servidor, um vetor dinâmico de *mutexes*.

```
//server.c

//global variable
pthread_mutex_t * mux_board;

//main function
mux_board = malloc(sizeof(pthread_mutex_t)* dim_board *dim_board);
for (i=0; i< (dim_board*dim_board); i++){
      pthread_mutex_init(&(mux_board[i]), NULL);
  }
```

Os *mutexes* foram implementados na função *board_play*, o que nos permitiu criar regiões críticas de dimensões bastante reduzidas. Sempre que um cliente escolhe uma carta (primeira ou segunda jogada) é feita a tentativa de *lock* do *mutex* associado a essa posição. Caso seja possível continuar e a carta esteja disponível (*estado = -2*) é alterado o estado da carta e faz-se o *unlock* do *mutex*. Caso esta não esteja disponível ($estado \neq -2$) então faz-se o *unlock* do *mutex* sem alterar o estado da carta.

No caso de dois clientes carregarem na mesma carta "ao mesmo tempo"apenas um deles entra na região crítica, alterando o *estado* da carta. Desta forma, quando este sai, o outro cliente já não irá conseguir ficar com a carta, uma vez que esta já tem o *estado* indisponível.

O código simplificado (com ênfase na implementação das regiões críticas) é mostrado de seguida:

```
//board_library.c

//board_play function
pthread_mutex_t mutex_card = vetor_mux[y*dim_board+x];
board_place* card = get_board_place_card(x,y);

if(play1[0]== -1){ //1st PICK
    pthread_mutex_lock(&mutex_card);
    if (card->estado != -2){
        resp.code = 10; //ignore play (FILLED card)
        pthread_mutex_unlock(&mutex_card);
        }
```

```
13
14      else{ //available card
15          card->estado = 1;
16          pthread_mutex_unlock(&mutex_card);
17          //... filling the resp structure
18
19  }else{ //2nd PICK
20      board_place* card1 = get_board_place_card(play1[0], play1[1]);//1st
        play card
21
22      pthread_mutex_lock(&mutex_card);
23      if(card->estado != -2){ //caso a carta escolhida nao seja valida
24          resp.code = -1; //virar a primeira carta para baixo
25          card1->estado = -2;
26          pthread_mutex_unlock(&mutex_card);
27          }
28
29      else if (strcmp(card->v, card1->v) == 0){ //correct pair
30          card1->estado = 0;
31          card->estado = 0;
32          pthread_mutex_unlock(&mutex_card);
33          //filling  the resp structure and other logic
34
35
36      }else{ //incorrect pair
37          card1->estado = 2;
38          card->estado = 2;
39          pthread_mutex_unlock(&mutex_card);
```

Número de pares feitos

A variável contadora de número de pares feitos (*n_corrects*) representa também uma região crítica, dado que a sua incrementação, sempre que um par é feito, não é uma instrução atómica. Esta *race condition* é também resolvida com uma variável do tipo *pthread_mutex_t*.

A sua implementação é também feita na função *board_play*, aquando da contabilização de um novo par.

```
1  //board_library.c
2
3  //board_play function
4  //...
5  else if (strcmp(card->v, card1->v) == 0){ //correct pair
6      pthread_mutex_lock(mux_score);
7      *n_corrects +=2;
8          if (*n_corrects == dim_board* dim_board)
9                  resp.code =3;
10         else{
11             resp.code =2;
12             if (*n_corrects == (dim_board*dim_board) - 2) //ultimo case 2 (
       o proximo sera case 3)
13                 sem_wait(sem);//decremento semaforo
```

```
14          }
15          pthread_mutex_unlock(mux_score); //
16          // para impedir que hajam duas threads a retornar code = 3 ao mesmo
        tempo o que provocaria o reinicio do jogo duas vezes
17      }
```

Note-se que com esta implementação se os dois últimos pares do jogo forem realizados "ao mesmo tempo"apenas o último a entrar na região crítica sairá com *resp.code = 3*, impedindo assim que a *thread* de fim de jogo seja executada duas vezes. Repare-se que neste caso hipotético é ainda necessário garantir que, no servidor, o *case 3* seja realizado só depois do *case 2*. Se tal não for garantido, pode acontecer que o jogo reinicie sem que tenham sido efetivamente pintadas todas as cartas. Assim criou-se um semáforo *sem* (inicializado com o contador a 1). A dinâmica foi implementada nesta função e no servidor como se mostra:

```
1  //server.c
2  //player thread
3
4  //...
5  case 2: //segunda jogada certa
6      //play
7      if (n_corrects == (dim_board*dim_board) - 2)
8          sem_post(& sem);
9  case 3: //END - fim de jogo
10     sem_wait(& sem); //tentativa de decrementar o semaforo
11     //play
12     //reset game
13     sem_post(& sem);
```

Contador do número de jogadores ativos

Para garantir que o jogo só se pode desenrolar com, pelo menos, dois jogadores, criou-se uma variável global (*numero_jogadores*) no servidor que contabiliza, em cada instante, o número de jogadores ativos: sempre que um cliente se conecta a variável é incrementada; quando um cliente se desconecta é decrementada.

Tratando-se de uma variável que pode ser acedida "simultaneamente"por várias *threads* no servidor, é necessário garantir a sua sincronização, através da criação de uma variável do tipo *pthread_mutex_t*.

A sua implementação no código é a seguinte:

```
1  //server.c
2  //main function
3  while(1){
4      players_fd= accept(sock_fd , NULL, NULL);
5      printf("client connected\n");
6
7      pthread_mutex_lock(&mux_n_jogadores);
8      numero_jogadores++;
9      pthread_mutex_unlock(&mux_n_jogadores);
10     //...
11   }
```

```c
//server.c
//player thread function
fim = read(this_player->player_fd, coord_aux, sizeof(coordinates));
    if (fim == 0){ //player disconnected
        pthread_mutex_lock(&mux_n_jogadores);
        numero_jogadores--;
        pthread_mutex_unlock(&mux_n_jogadores);
        //...
    }
```

7 Descrição das várias funcionalidades implementadas

7.1 Número mínimo de jogadores

Início do jogo

Temos uma variável denominada *numero_jogadores*. Esta variável é incrementada sempre que um novo cliente se conecta e decrementada quando algum cliente se desconecta.

No início da thread criada no servidor para cada jogador, é verificado o número de jogadores assim, se *numero_jogadores* < 2 em qualquer momento do jogo (seja no início ou durante) as jogadas serão ignoradas se só tivermos 1 cliente conectado.

```c
//server.c
    if ((ignorar == 1)||(numero_jogadores<2)){
        printf("a ignorar\n");
    }
```

Como podemos observar no excerto de código, a verificação do número de jogadores é feita em paralelo com o estado de ignora, usado na espera de 10 segundos no fim do jogo, a qual veremos em detalhe mais à frente.

Durante o jogo

Como explicado no ponto anterior, a implementação do número mínimo de jogadores durante o jogo é exatamente a mesma da verificação durante o decurso do programa.

7.2 Distinção entre a 1º pick e a 2º pick

Esta distinção é feita na função *board_play* do ficheiro *board_library.c*.

```c
//board\_library.c
//board_play

//inicializacoes
if(play1[0]== -1){
    printf("FIRST pick\n");
    //verificar se a primeira pick e uma carta ativa de outro jogador
    pthread_mutex_lock(&mutex_card);
    if (card->estado != -2){
        resp.code = 10;
        pthread_mutex_unlock(&mutex_card);
```

```
12          printf("FILLED (carta nao pode ser selecionada)\n");
13      }
14      else{ //caso a pick nao seja uma carta ativa de outro jogador
15          card->estado = 1;
16          pthread_mutex_unlock(&mutex_card);
17          resp.code =1;
18          play1[0]=x;
19          play1[1]=y;
20          resp.play1[0]= play1[0];
21          resp.play1[1]= play1[1];
22          strcpy(resp.str_play1, get_board_place_str(x, y));
23      }
24  }
25  else{
26
27      //logica de tratamento da 2 pick
28
29      play1[0]= -1;//de volta a primeira pick
30  }
```

Como podemos observar no código acima, o que caracteriza a 1º jogada é o facto de a 1º posição do vetor *play1 = -1*.

É visível na linha 18, que após uma 1º pick válida (isto é, uma carta que não esteja ativa por outro jogador), a 1º posição do vetor *play1*, toma o valor da coordenada x da carta escolhida pelo jogador, assim, quando o jogador efetuar a 2º pick, já não irá entrar neste *if*, entendo no *else* final que se vê na linha 25.

A partir daí são processados os diversos tipos de 2º pick que podem existir, sendo elas, carregar na msm carta que a 1º pick, carregar numa carta não válida (isto é, ativa por outro jogador), escolher o par da 1º pick ou escolher uma carta válida que não é o par da 1º pick.

No fim da 2º pick, *play1* volra a ser inicializada a -1 para que a jogada seguinte volte a ser tratada como 1º pick.

7.3 5 segundos entre picks

Quando um jogador efetua uma primeira jogada tem cinco segundos para realizar a sua segunda jogada. Caso não a faça, a primeira carta escolhida deve ser pintada de branco e volta a ficar disponível para ser escolhida. Nesse caso o jogador deve retornar a fazer uma primeira jogada. Para solucionar estes requisitos foi utilizada a função de *poll* com um evento de *POLLIN* associado ao *socket* do cliente em questão.

A implementação encontra-se em baixo:

```
1 //server.c
2
3 //player thread function
4
5 //Poll initialization
6 struct pollfd fd;
7 fd.fd = this_player->player_fd;
8 fd.events = POLLIN;
```

```
9
10 //...
11
12 play_response resp = board_play(coord.x, coord.y, play1, this_player->cor,
       &mux_score,mux_board);
13     switch (resp.code) {
14         case 1: //1st play
15             //...broadcasting the player's color to paint the card and the
       string
16
17             printf("setting timer\n");
18             ret = poll(&fd, 1, 5000);
19                 switch (ret) {
20                     case -1:
21                         // Error
22                         break;
23                     case 0:
24                         // Timeout  paint white
25                         atualiza_board(-2, resp.play1[0], resp.play1[1],
       this_player,resp.str_play1);
26                         play1[0] = -1;
27                         printf("end of timer\n");
28                         broadcast(lista,resp.play1[0],resp.play1
       [1],255,255,255,0,"\0",&mux_lista); //painting the card in white (
       turned down)
29                         break;
30
31                     default: //client made his 2nd play before the timeout
32                         printf("abrupt end of timer\n");
33             }
34         break;
35
36         //the other cases
```

Note-se que quando um cliente faz uma primeira jogada tem-se que *resp.code = 1* e portanto é feito o *broadcast* para todos os clientes com a informação da carta escolhida (coordenadas), cor do cliente que escolheu a carta e a *string* da mesma. Após este envio, o *"timer"* inicia-se, ou seja durante 5 segundos é feito um *poll* ao *socket* deste cliente e caso haja informação a chegar (sinónimo de que o cliente efetuou uma segunda jogada), então o servidor irá processá-la. Caso contrário, se os cinco segundos chegarem ao fim, a função *poll* (linha 18) retorna com o valor 0 e é executado o código associado (a partir da linha 24), onde se pinta a carta de branco e se dá indicação de que o jogador voltará a fazer uma primeira jogada (*play1[0] = -1*).

7.4 Espera de 2 segundos caso o par escolhido esteja errado

Quando um jogador faz um par e este está errado, deve ficar impedido de jogar durante dois segundos. Se tentar jogar durante esse tempo, as suas jogadas devem ser ignoradas. Para resolver este problema também recorremos à função de *poll*. A implementação desenvolvida foi a seguinte:

```
1 //server.c
```

```
play_response resp = board_play(coord.x, coord.y, play1, this_player->cor,
    &mux_score,mux_board);
    switch (resp.code) {
        //... the other cases


        case -2:  //2nd play, wrong pair
            //broadcasting the player's color to paint the cards and their
    string with the red color
            time_t begin = time(NULL);
            while((time(NULL) - begin) < 2){
                ret = poll(&fd, 1, 2000-(time(NULL)-begin)*1000);
                switch (ret) {
                    case -1:
                        // Error
                        break;
                    case 0:
                        break;
                    default:
                        read(this_player->player_fd, coord_aux, sizeof(
    coordinates));
                    }

            //...paint both cards in white (broadcasting) and update their
    state
    }
```

Como se pode ver, no início do *"timer"* obtém-se o tempo no instante inicial (através da função *time*). Quando o jogador carrega numa carta, o *poll* retorna e a função *read* lê a jogada mas não a processa (ou seja, é ignorada). Posteriormente, o *timer* do *poll* é atualizado com o tempo que ainda falta cumprir para satisfazer os dois segundos de pausa.

7.5 Fim do jogo

Quando o tabuleiro fica todo preenchido, é necessário reinicializá-lo. Para isso desenvolvemos uma *thread* de fim de jogo, que faz as seguintes operações:

- Quando esta *thread* é executada, a variável *ignorar* é colocada a 1, permitindo ignorar todas as jogadas dos jogadores nesta fase;
- Esta *thread* percorre a lista de jogadores e vê qual ou quais os jogadores que têm o *score* mais alto. Posteriormente atualiza este resultado no campo *vencedor* da estrutura de cada jogador;
- Depois o servidor envia a todos os jogadores esta informação, ou seja diz-lhes se estes venceram ou perderam o jogo e envia-lhes o seu *score*;
- Posteriormente são retirados da lista todos os jogadores que se desconectaram durante este jogo;

- É feito o *free* do tabuleiro e inicializa-se um novo;
- Faz-se uma espera de 10 segundos, através de um *sleep(10)*;
- Por fim, a variável *ignorar* volta a ser colocada a 0 o que permite ao servidor voltar a poder processar as jogadas.

De seguida representa-se a função descrita implementada (simplificada):

```
//server.c

void* fim_do_jogo (void * arg){
    while(1){
        //... initializations
        pthread_mutex_lock(& mux_end );
        pthread_cond_wait(&cond_end, &mux_end);
        ignorar = 1;
        ve_vencedor(lista, &mux_lista);

        pthread_rwlock_rdlock(&mux_lista);
        for (aux = aux->next; aux!= NULL; aux= aux->next){
            if (aux->ativo == 1){
                layer_com buffer = preenche_buffer(aux->score,aux->vencedor
    ,0,0,0,2,"");
                aux->score =0;
                aux->vencedor=0;
                memcpy(aux_comm, &buffer, sizeof(buffer));
                write(aux->player_fd, aux_comm, sizeof(layer_com));
            }
        }

        pthread_rwlock_unlock(&mux_lista);
        atualiza_lista_end_game(lista, &mux_lista);
        free(board);
        board = init_board(dim_board);
        sleep(10);
        ignorar = 0;
        pthread_mutex_unlock(& mux_end);
    }
}
```

O cliente deteta o fim do jogo através do campo *flag* da estrutura de dados que recebe do servidor. Quando isso acontece, este reinicializa o tabuleiro e faz também uma espera de 10 segundos (apenas para o *board* não ficar imediatamente a branco). O código desenvolvido é o seguinte:

```
//client.c

//thread_func
ret = read(sock_fd, aux_comm, sizeof(layer_com));
memcpy(&buffer, aux_comm, sizeof(layer_com));
if (buffer.flag == 0)
    paint_card(buffer.x, buffer.y , buffer.r, buffer.g, buffer.b);

```

```
else if (buffer.flag == 1)
    write_card(buffer.x, buffer.y, buffer.string, buffer.r, buffer.g,
    buffer.b);

else if(buffer.flag == 2){ //fim do jogo
        if(buffer.y == 1)
            printf("VENCEDOR (score = %d)\n", buffer.x);
        else
            printf("PERDEDOR (score = %d)\n", buffer.x);
      sleep(10);
      printf("A reiniciar a board\n");
      reinicia_board_branco(300, 300, dim_board);
}
```

7.6 Limpeza após morte/desconexão de um cliente

Na nossa implementação optámos por não remover um cliente da lista imediatamente quando este se desconecta.

```
//lista.c
void retira_lista(jogador* lista, int *cor, pthread_rwlock_t *mux_lista){
    if ((lista==NULL)||(lista->next == NULL))
        return;

    pthread_rwlock_rdlock(mux_lista);
    for (lista = lista->next; lista!=NULL; lista = lista->next){
        if ((lista->cor[0] == cor[0])&& (lista->cor[1] == cor[1]) && (lista
    ->cor[2] == cor[2])){
            lista->ativo = 0;
            close(lista->player_fd);
            break;
        }
    }
    pthread_rwlock_unlock(mux_lista);
}
```

Assim, como podemos observar no código acima, atualizamos a variável *ativo=0*, da estrutura de dados da lista de jogadores. Todas as suas cartas que estejam voltadas para cima em 1º pick ou 1º e 2º pick incorretas, são atualizadas para o estado inativo *estado=-2* e voltadas para baixo (pintadas de branco), tal como podemos observar na seguinte função.

```
//server.c
void out_client (jogador* this_player){
  int i, j,k;

  for (i=0; i< dim_board; i++){
    for (j=0; j<dim_board; j++){
      board_place* card = get_board_place_card(i,j);
      if ((card->cor[0]==this_player->cor[0]) && (card->cor[1]==this_player
    ->cor[1]) && (card->cor[2]==this_player->cor[2])){
        if ((card->estado == 1) || (card->estado == 2)){
          for (k=0; k<3; k++)
```

```
11              card->cor[k]=-1;
12            card->estado = -2;
13            printf("apaguei a carta %d %d, estado %d\n", i,j,card->estado);
14            broadcast(lista, i,j,255,255,255,0,"",&mux_lista);
15            clear_card(i,j);
16          }
17        }
18      }
19    }
20  }
```

A remoção dos jogadores inativos é feita apenas no fim do jogo, uma vez que pode dar-se o caso do vencedor ser um dos jogadores inativos, foi com vista a este pormenor que optámos por manter a sua estrutura na lista até ao fim. Assim, todos os jogadores ativos receberão informação de que são perdedores e nenhum vencedor será anunciado. O código da remoção dos clientes inativos no fim do jogo pode ser vista a seguir.

```
1  //lista.c
2  void atualiza_lista_end_game (jogador*lista,pthread_rwlock_t *mux_lista){
3      if ((lista==NULL)||(lista->next == NULL))
4          return;
5      jogador* aux = lista;
6      jogador* eliminar;
7      jogador* lista_local = lista;
8      int change = 0;
9      pthread_rwlock_wrlock(mux_lista);
10     while(1){
11         change = 0;
12         //percorre a lista e procura um elemento inativo e se encontrar
       tira da lista
13         for (lista = lista->next; lista!=NULL; lista = lista->next){
14             if (lista->ativo == 0){
15                 eliminar = aux->next;
16                 (aux->next) = eliminar->next;
17                 free(eliminar);
18                 change = 1;
19                 break;
20             }
21             aux = aux->next;
22         }
23         if (change == 0)
24             break;
25         lista = lista_local;
26         aux = lista;
27     }
28     pthread_rwlock_unlock(mux_lista);
29 }
```

7.7 Bot

O bot e um cliente semelhante aos jogadores reais, com excepção de que não tem uma janela gráfica nem precisa de recolher jogogadas do rato, uma vez que as gera de forma

aleatória.

Começa por se conectar ao socket, recebe a dimensão do tabuleiro e aloca memória para uma versão completa de um tabuleiro local. Entra num ciclo infinito de geração aleatória de jogadas, dentro dos limites do tabuleiro e posteriormente envia-as para o servidor. Podemos ver esta parte da implementação no seguinte excerto de código.

```
1  //simple\_bot.c
2  int main (int argc, char * argv[]) {
3
4      //inicializacoes
5
6      if( -1 == connect(sock_fd, (const struct sockaddr *) & server_addr,
       sizeof(server_addr))){
7        printf("Error connecting\n");
8      exit(-1);
9    }
10     printf("client connected\n");
11
12   read(sock_fd, &dim_board, sizeof(int));
13     board = malloc(sizeof(board_place)* dim_board *dim_board);
14
15     while(1){
16         x = rand()%dim_board;
17         y = rand()%dim_board;
18         coord.x = x;
19         coord.y = y;
20         memcpy(coord_aux, &coord, sizeof(coordinates));
21         printf("click (%d %d) \n", x,y);
22         write(sock_fd, coord_aux, sizeof(coordinates));
23         sleep(1);
24     }
25 }
```