Relatório de Projeto

Jogo da memória

Diogo Alves nº 86980 (Grupo 32)

20/06/2019

Índice

1 Arquitetura **1**

2 Organização do código **1**
2.1 Board Library . . . 1
2.2 UI Library . . . 2
2.3 UI Client . . . 2
2.4 Bot Client . . . 3
2.5 Memory Server . . . 3

3 Estruturas de Dados **4**
3.1 Player . . . 4
3.2 Player List . . . 5
3.3 Play . . . 6
3.4 Board Place . . . 6
3.5 Play Response . . . 7
3.6 Endgame Info . . . 8

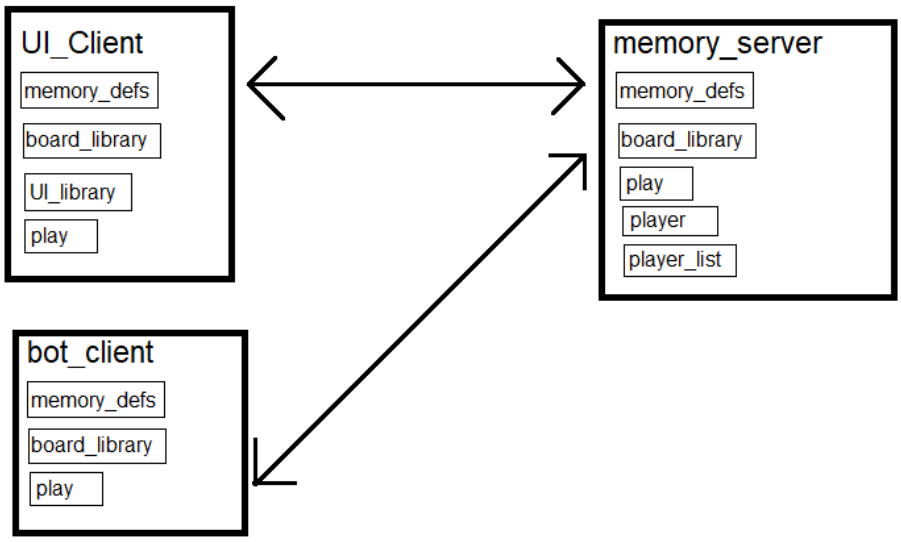
4 Protocolos de Comunicação **8**
4.1 Início do jogo . . . 8
4.2 Jogadas . . . 8
4.3 Atualização do tabuleiro . . . 8
4.4 Jogo na pausa/fora da pausa . . . 9
4.5 Fim do jogo . . . 9
4.6 Saída de um jogador . . . 10

5 Validação de dados tranferidos entre funções e processos **10**

6 Regiões criticas/Sincronização **10**

7 Descrição das funcionalidades implementadas **10**
7.1 Número mínimo de jogadores . . . 10
7.2 Distinção entre primeira e segunda jogada . . . 11
7.3 5 segundos entre jogadas . . . 11
7.4 Delay de 2 segundos após segunda jogada errada . . . 13
7.5 Fim de um jogo . . . 14
7.6 Limpeza após a morte/desconexão de um cliente . . . 16
7.7 Bot . . . 17

1 Arquitetura



2 Organização do código

De seguida apresentam-se os módulos desenvolvidos para este projeto:

2.1 Board Library

Os ficheiros *board_library.c* e *board_library.h* permitem a gestão de um único jogo num tabuleiro. Este tabuleiro é declarado como variável global e é manipulado pelas seguintes funções:

play_response boardPlay(play play, player* player)

Esta função recebe a jogada e o jogador que a efetuou e decide, conforme as regras do jogo, qual deverá ser o resultado da jogada e implementa-o no tabuleiro.

void initBoard(int dim)

Inicializa o tabuleiro com todas as cartas viradas para baixo ou, caso este já tenha sido inicializado, reinicia-o para as condições originais.

board_place* getBoardCardsFaceUp()

Retorna um array de *board_place*, que corresponde a uma cópia do tabuleiro, mas apenas com as *strings* correspondentes a cartas *UP* ou *LOCKED* preenchidas: não há informação acerca das cartas viradas para baixo (*DOWN*).

void closeBoard()

Liberta a memória associada ao tabuleiro.

2.2 UI Library

Este módulo permite ao programador desenhar um tabuleiro com determinado número de pixeis e número de cartas. A janela que irá mostrar o tabuleiro é declarada como variável global e manipulada pelas funções disponibilizadas. Existe um alarme cuja função é atualizar a janela a cada 30 ms.

int createBoardWindow(int width, int height, int dim)

Cria uma janela com *width*×*height* pixeis e nela um tabuleiro com *dim*×*dim* cartas viradas para baixo (*DOWN*).

void closeBoardWindow()

Destrói a janela que estava a ser usada para mostrar o tabuleiro.

void getBoardCard(int mouse_x, int mouse_y, int* board_x, int* board_y)

Recebe como argumentos as coordenadas de um clique do rato (*mouse_x* e *mouse_y*) e retorna a correspondente posição no tabuleiro (*board_x* e *board_y*).

void writeCard(int board_x, int board_y, char* text, int r, int g, int b)

Recebe as coordenadas da carta (*board_x* e *board_y*) e escreve nessa posição o texto *text* com a cor (*r,g,b*).

void paintCard(int board_x, int board_y, int r, int g, int b)

Recebe as coordenadas (*board_x* e *board_y*) e pinta esta carta com a cor (*r,g,b*).

void clearCard(int board_x, int board_y)

Pinta a carta na posição (*board_x, board_y*) de branco.

2.3 UI Client

Este módulo implementa um cliente para o jogo da memória, que disponibiliza uma interface gráfica onde o jogador pode selecionar as suas jogadas e visualizar o progresso do jogo.

void quitClient()

Função que envia para o servidor a intenção de sair do jogo. É executado quando *CTRL + c* é premido ou quando a janela da interface gráfica é fechada.

void* UIReadRoutine(void* args)

Rotina que corre numa thread e constantemente lê jogadas da interface gráfica e as envia para o servidor, até que o cliente saia do jogo.

void updateWindow(play_response resp)

Atualiza a interface gráfica, com base na resposta recebida pela servidor (*resp*).

void* receiveResponsesRoutine(void* args)

Rotina que corre numa thread e constantemente recebe respostas do servidor, até o cliente sair do jogo ou o jogo terminar.

void fillKnownCards(board_place* board_cards_face_up)

Preenche na interface gráfica as cartas conhecidas, presentes no vetor *board_cards_face_up*.

2.4 Bot Client

Este módulo implementa um cliente para o jogo da memória que faz jogadas de forma automática com base em respostas às jogadas anteriores de todos os jogadores, através de um tabuleiro guardado internamente que memoriza o conteúdo das cartas que vão sendo conhecidas.

void updateInternalBoard(play_response resp)

Atualiza o tabuleiro guardado internamente com a resposta *resp* recebida.

int findPair(play* play1, play* play2)

Função cujo objetivo é encontrar um par de cartas que sejam uma correspondência e que estejam viradas para baixo (*DOWN*), com base na informação do tabuleiro guardado internamente. Caso encontre, retorna 1 e as jogadas correspondentes em *play1* e *play2*. Caso contrário retorna 0.

int firstUnknown(play* play)

O objetivo desta função é encontrar a primeira carta virada para baixo (*DOWN*) e que não seja conhecida no tabuleiro guardado internamente. É retornada a jogada (em *play*) com essa carta e 1 em caso de sucesso; caso contrário é retornado 0.

void* sendPlaysRoutine(void* args)

Rotina que corre numa thread e constantemente envia jogadas para o servidor, até o jogador sair. A lógica para determinar as jogadas a fazer é a seguinte: quando existir um par de cartas que sejam uma correspondência e estejam viradas para baixo (*DOWN*), jogar essas duas cartas, caso contrário jogar a primeira carta que esteja virada para baixo e não seja conhecida e de seguida uma carta aleatória.

void* receiveResponsesRoutine(void* args)

Rotina que corre numa thread e constantemente recebe respostas do servidor, até o cliente sair do jogo ou o jogo terminar.

2.5 Memory Server

Este módulo implementa o servidor do jogo da memória, ao qual se ligam os clientes. O servidor gere as jogadas, o tabuleiro e os jogadores.

int numberActivePlayers()

Determina e retorna o número atual de jogadores ativos (aqueles que ainda não sairam do jogo).

void setGameState(int _game_state)

Atualiza o estado do jogo para *_game_state* (*GAME_ON*, *GAME_OVER* ou *GAME_WAITING*).

void quitPlayer(player* player)

Faz um cliente sair do jogo.

int getGameState()

Retorna o estado atual do jogo (*GAME_ON*, *GAME_OVER* ou *GAME_WAITING*).

int quitHandler(int dummy)

Termina a execução do servidor, expulsando todos os jogadores e fechando o tabuleiro. É executada quando é premido *CTRL + c*.

int maxPoints()

Retorna o número máximo de pontos de entre todos os jogadores.

int sendGameInfo()

Envia a cada jogador a informação de fim de jogo: o número de pontos e se perdeu ou ganhou.

void checkAndUpdateGameOn()

Verifica se o estado do jogo deveria ser alterado para *GAME_ON* e atualiza o que for necessário.

void resetPlayers()

Faz reset de todos os jogadores na lista que ainda estão em jogo (estado, primeira jogada e pontuação).

void* endGameRoutine(void* args

Rotina que é executada quando o jogo termina: faz uma pausa de 10 segundos e retorna o jogo às suas configurações iniciais.

void* hideFirstPickRoutine(void* args)

Rotina usada pela *thread* que "vira para baixo" a primeira carta escolhida por um jogador passados 5 segundos, caso o jogador não tenha feito jogadas depois.

void hideSecondPickRoutine(void* args)

Rotina usada pela *thread* que vira para baixo as duas cartas do jogador, 2 segundos após a sua segunda jogada, caso esta não seja uma correspondência com a sua primeira.

void* handlePlayerRoutine(void* args)

Rotina que corre na *thread* específica de cada jogador. Corre enquanto o jogador não sair do jogo. Recebe do cliente as jogadas e processa-as no tabuleiro. Envia também a resposta de cada jogada para todos os jogadores em jogo.

3 Estruturas de Dados

3.1 Player

Nos ficheiros *player.c* e *player.h* está definida a estrutura de dados *player*. Esta estrutura representa um jogador e contém:

int color[3]

Cor *(r,g,b)* atribuida ao jogador.

int state

Estado atual em que se encontra o jogador:
- *NO_PICK*: o jogador ainda não efetuou a primeira jogada;
- *FIRST_PICK*: o jogador efetuou a primeira jogada;
- *SECOND_PICK*: o jogador fez a segunda jogada e está à espera para poder voltar a jogar;
- *QUIT*: o jogador saiu do jogo;
- *WAITING_TO_JOIN*: o jogador está à espera para poder entrar no jogo.

int sock_fd

Descriptor do socket utilizado pelo servidor para comunicar com o cliente associado ao jogador.

play play1

Primeira jogada do jogador (*first pick*).

int n_corrects

Pontuação do jogador.

As funções que permitem manipular esta estrutura são:

player* createNewPlayer(int sock_fd)

Cria um novo jogador, com os parâmetros *default* e atribui a cada um uma nova cor (única).

int getSockFd(player* player)

Retorna o descriptor do socket associado ao jogador.

void resetPlayer(player* player)

Atribui aos campos *state*, *play1* e *n_corrects* o seu valor *default*.

3.2 Player List

Nos ficheiros *player_list.h* e *player_list.c* está definida a estrutura de dados *player_list* que representa um nó de uma lista de jogadores. Esta estrutura contém:

player* this_player

Ponteiro para o jogador correspondente ao nó da lista.

struct _player_list *next

Ponteiro para o próximo elemento da lista.

As funções que permitem manipular esta lista são:

player_list* initPlayerList(void)

Inicializa a lista.

player_list* createNewNodePlayerList(player_list* lp, player* new_player)

Cria e retorna um novo nó que será posteriormente adicionado à lista.

player* getPlayerFromList(player_list* lp)

Retorna o jogador correspondente ao nó *lp*.

player_list* getNextElementPlayerList(player_list* lp)

Retorna um ponteiro para o elemento seguinte da lista.

void freePlayerList(player_list* lp)

Liberta a memória da lista, incluindo a memória dos jogadores.

int isEmptyPlayerList(player_list* lp)

Retorna 1 caso a lista esteja vazia e 0 caso contrário.

3.3 Play

Nos ficheiros *play.c* e *play.h* encontra-se definida a estrutura *play*, que representa uma jogada feita por um jogador. Esta estrutura contém:

int id

identificador da jogada (único).

int coordinates[2]

Coordenadas da carta escolhida no tabuleiro.

As funções que permitem manipular esta estrutura de dados são:

play newPlay(int x, int y)

Função que cria e retorna uma nova variável *play* cuja jogada é a carta na posição *(x,y)*.

int getXPlay(play play)

Retorna a coordenada x da jogada.

int getXPlay(play play)

Retorna a coordenada y da jogada.

int playsAreSame(play play_a, play play_b)

Retorna 1 se as duas jogadas recebidas forem a mesma (igual *id*).

3.4 Board Place

Nos ficheiros *board_library.h* e *board_library.c* está definida a estrutura *board_place*, que representa uma carta do tabuleiro, O tabuleiro é na realidade um vetor de tamanho *dim*dim* de *board_place*. Esta estrutura contém:

char v[3]

Texto escrito na carta.

int state

Estado da carta:
- *STATE_UP*: carta virada para cima;
- *STATE_DOWN*: carta virada para baixo;
- *STATE_LOCKED*: carta já acertada.

As funções que manipulam esta estrutura são:

char* getBoardPlaceStr(int i, int j)

Esta função retorna um ponteiro para a *string* da carta na posição *(i,j)*.

void setBoardPlaceState(int x, int y, int state)

Atribui o estado *state* à posição do tabuleiro *(x,y)*.

void setBoardPlaceColor(int x, int y, int* color)

Atribui a cor *color* (vetor 1*3) à posição do tabuleiro *(x,y)*.

int getBoardPlaceState(int x, int y)

Retorna o estado da carta na posição *(x,y)*: *STATE_DOWN*, *STATE_UP* ou *STATE_LOCKED*.

3.5 Play Response

Nos ficheiros *board_library.h* e *board_library.c* está definida a estrutura *play_response*, que representa a resposta a uma jogada, após esta ter sido processada no tabuleiro. Esta estrutura contém:

int code

Código da jogada:
- *CODE_FILLED*: lugar escolhido não corresponde a uma carta *DOWN*;
- *CODE_FIRST_PLAY*: jogada corresponde a uma *first pick*;
- *CODE_SECOND_PLAY_RIGHT*: jogada corresponde a uma *second pick* com sucesso;
- *CODE_GAME_OVER*: jogada resultou no término do jogo;
- *CODE_SECOND_PLAY_WRONG*: jogada corresponde a uma *second pick* sem sucesso;
- *CODE_HIDE_FIRST_PLAY*: resposta em como a primeira jogada passou a estar virada para baixo;
- *CODE_HIDE_BOTH_PLAYS*: resposta em como ambas as jogadas passam a estar viradas para baixo;
- *CODE_WAITING_WRONG_PLAY_TIME_UP*: resposta em como o jogador não está ainda em condições de jogar, após fazer um par de jogadas sem sucesso;
- *CODE_PERMISSION_TO_LEAVE*: resposta em como o jogador tem a permissão do servidor para sair do jogo;
- *CODE_GAME_FROZEN*: resposta em como o jogo passou a estar em pausa;
- *CODE_GAME_UNFROZEN*: resposta em como o jogo deixou de estar na pausa,

play play1

Primeira jogada do jogador (*first pick*),

play play2

Segunda jogada do jogador (*second pick*),

char str_play1[2]

Texto da carta da primeira jogada.

char str_play2[2]

Texto da carta da segunda jogada.

int player_color[3]

Cor do jogador,

3.6 Endgame Info

Nos ficheiros *board_library.h* e *board_library.c* está definida a estrutura *endgame_info* que representa a informação de fim de jogo que é enviada a cada jogador quando termina um jogo. Esta estrutura contém:

int result

Resultado de um jogo para cada jogador:
- *RESULT_WIN*: vitória;
- *RESULT_LOST*: derrota.

int points

Pontuação do jogador,

As funções que manipulam esta estrutura são:

endgame_info newEndgameInfo(int result, int points)

Retorna uma estrutura do tipo *endgame_info* contendo as informações relativas aos argumentos recebidos (*result* e *points*).

int getEndgameInfoPoints(endgame_info info)

Retorna o campo *points* (número de pontos) na estrutura *endgame_info* recebida.

int getEndGameInfoResult(endgame_info info)

Retorna o campo *result* da estrutura *endgame_info* recebida (*RESULT_WIN* ou *RESULT_LOST*).

4 Protocolos de Comunicação

4.1 Início do jogo

Quando um cliente se liga ao servidor, o servidor envia-lhe imediatamente uma mensagem com a dimensão do tabuleiro (*int borad_dim*), Depois, o cliente fica à espera de uma mensagem de tamanho *sizeof(board_place)*dim*dim*, que corresponde ao tabuleiro, com a informação relativa apenas às cartas viradas para cima (*STATE_UP*) ou que já foram acertadas (*STATE_LOCKED*). O servidor irá enviar esta mensagem imediatamente caso o jogo esteja a decorrer ou, caso o jogo estiver em pausa (*GAME_FROZEN*) devido à falta de o mínimo de 2 jogadores ou esteja no intervalo de 10 segundos entre dois jogos (*GAME_OVER*), esta mensagem só será enviada quando o jogo iniciar (*GAME_ON*),

4.2 Jogadas

Para fazerem uma jogada, os clientes deverão enviar ao servidor uma mensagem do tipo de dados *play*, em que no campo *coordinates* deverão estar as coordenadas no tabuleiro da carta que se pretende selecionar (coordenadas positivas). Todas as cartas são permitidas selecionar pelo cliente: cabe ao servidor validar a jogada.

4.3 Atualização do tabuleiro

Após processar no tabuleiro a jogada de um jogador, o servidor irá enviar a todos os clientes uma mensagem com o resultado dessa jogada. Esta mensagem será do tipo de dados *play_response*. Consoante o campo *code* desta mensagem, os restantes campos, além de *int player_color[3]* que terá sempre a cor do jogador, deverão ter a seguinte informação:

code = CODE_FILLED

A carta selecionada já está selecionada ou já foi acertada. Não é necessário preencher os restantes campos.

code = CODE_FIRST_PLAY

A jogada é válida e corresponde a uma *first pick*, O campo *play1* deverá corresponder à jogada feita e *str_play1* ao conteúdo da carta.

code = CODE_SECOND_PLAY_RIGHT

A jogada é válida, corresponde a uma *second pick* e é uma correspondência com a *first pick* do mesmo jogador. Os campos *play1* e *play2* devem estar preenchidos com as jogadas correspondentes à *first pick* e *second pick* e os campos *str_play1* e *str_play2* com os conteúdos das respetivas cartas.

code = CODE_GAME_OVER

O mesmo que a anterior, com a diferença que esta jogada resultou no fim do jogo (todas as cartas acertadas),

code = CODE_SECOND_PLAY_WRONG

A jogada é válida, corresponde a uma *second pick*, mas não é uma correspondência com a *first pick*. Os campos devem estar preenchidos da mesma forma que para o caso em que *code = CODE_SECOND_PLAY_RIGHT*.

code = CODE_WAITING_WRONG_PLAY_TIME_UP

A jogada não é permitida, pois o jogador ainda está à espera de poder jogar após fazer duas jogadas que não são uma correspondência. Os restantes campos não precisam de estar preenchidos.

code = CODE_HIDE_FIRST_PLAY

O jogador fez a segunda jogada numa carta que já estava virada para cima (*STATE_UP* ou *STATE_LOCKED*), por isso a primeira carta que selecionou é virada para baixo (*STATE_DOWN*). Este cógido é também utilizado quando o jogador seleciona uma carta e não volta a fazer uma jogada nos seguintes 5 segundos, o que leva a que a sua primeira carta seja virada para baixo (*STATE_DOWN*). No campo *play1* deve estar a primeira jogada feita pelo jogador.

code = CODE_HIDE_BOTH_PLAYS

Já passaram 2 segundos desde que o jogador fez uma jogada que não correspondia com a sua primeira e essas duas cartas devem ser viradas para baixo. Nos campos *play1* e *play2* devem estar a primeira e segunda jogada desse jogador, respetivamente.

4.4 Jogo na pausa/fora da pausa

Quando um jogador sai do jogo e o número de jogadores fica menor que 2, o jogo entra em pausa (*GAME_FROZEN*) e neste estado os jogadores não podem fazer jogadas. Os clientes são informados quando um jogo entra e sai deste modo de pausa através de uma mensagem do servidor do tipo de dados *play_response*, em que o campo *code* é igual a *CODE_GAME_FROZEN* ou *CODE_GAME_UNFROZEN* caso o jogo entre ou saia deste modo, respetivamente.

4.5 Fim do jogo

Como foi dito anteriormente, quando existe uma jogada que resulte no término do jogo, o servidor envia para os clientes uma mensagem do tipo de dados *play_response* em que o campo *code* é igual a *CODE_GAME_OVER*. A partir daqui, os clientes estão prontos para receber uma estrutura do tipo *endgame_info*, que é enviada pelo servidor a cada cliente. com o seu resultado no jogo (*RESULT_WIN* ou *RESULT_LOST*) no campo *result* e a sua pontuação no campo *points*.

4.6 Saída de um jogador

Se um jogador quiser sair do jogo, deverá enviar para o servidor uma estrutura do tipo *play*, em que o campo *coordinates* é (-1, -1). Depois, o servidor enviar-lhe-à uma mensagem do tipo *play_response* em que o campo *code* é *CODE_PREMISSION_TO_LEAVE*. Se for o servidor a sair, este enviará a todos os clientes ativos uma mensagem deste ultimo tipo, de forma a que todos os processos dos clientes possam terminar.

5 Validação de dados tranferidos entre funções e processos

Para transferir dados entre o servidor e o cliente e vice-versa, são utilizados *sockets* e as funções *write* e *read*. Ao fazer *write* no servidor, verifica-se o retorno desta função e, se for igual a -1, sempre que seja possível o jogador ao qual se estava a tentar enviar a mensagem é expulso do jogo. No cliente acontece o mesmo: se o retorno do *write* for -1, o cliente termina o seu processo. Ao efetuar um *read*, tanto no cliente como no servidor, é verificado se o tamanho da mensagem recebida corresponde ao tamanho da mensagem esperada e o código seguinte só é executado se tal se verificar. É também verificado se o retorno da função é igual a -1 e se tal se confirmar, acontece o mesmo que o descrito para a função *write*. Para as funções externas (*socket(...)*,*bind(...)*, *accept(...)*, *connect(...)*) é também verificado se o retorno é igual a -1 e se tal acontecer, os processos são terminados,

6 Regiões criticas/Sincronização

Para resolver os problemas de sincronização no servidor são utilizados *mutexes*. São utilizados dois tipos de *mutexes*: uma matriz de *board_dim*board_dim*, que irão servir para resolver problemas de sincronia no tabuleiro e um outro *mutex* (*game_state_mutex*) que servirá para resolver problemas relacionados com a mudança do estado do jogo.

Cada carta no tabuleiro corresponde a uma região critica. Assim, há que impedir duas *threads* de executarem a função *boardPlay*, para a mesma carta, ao mesmo tempo. Tem então que se bloquear esta função com o *mutex* da respetiva carta. Para além disso, caso o jogador esteja na sua *second pick*, esta função também poderá alterar o estado da carta correspondente à *first pick* do jogador, pelo que nesta situação também é necessário bloquear o acesso simultâneo de duas *threads* a essa posição, recorrendo ao *mutex*.

Outra região critica que existe é o envio do tabuleiro com as cartas viradas para cima quando um jogador se liga ao servidor: pretende-se que durante o processo de envio do tabuleiro, os restantes clientes não interfiram, pelo que antes de enviar se faz *lock* de todos os *mutexes* relativos a cartas no tabuleiro.

Por fim, temos as regiões criticas relacionadas com a mudança e leitura do estado do jogo: a primeira encontra-se na função *setGameState*: pretende-se que esta função seja executada por não mais de uma *thread* em simultâneo. Como a função *setGameState* atualiza tudo o que implica a mudança de estado, poderia dar-se o caso de, por exemplo, o estado do jogo ser alterado enquanto ainda se está a atualizar os jogadores devido a uma mudança de estado anterior.

A função *getGameState* também é bloqueada por este *mutex* para evitar lermos o estado do jogo num momento em que ele está a ser atualizado.

São também bloqueados por este *mutex* o envio das respostas às jogadas para todos os clientes. Isto porque durante estas operações, se o estado do jogo mudar, elas poderão não ser executadas até ao fim.

7 Descrição das funcionalidades implementadas

7.1 Número mínimo de jogadores

A verificação de que o jogo deve começar/sair da pausa por existirem já dois jogadores é feita no módulo *memory_server* pela função *checkAndUpdateGameOn*, que é executada sempre que um novo cliente se liga:

```
194  /*checks wether the game state should change to GAME_ON and updates everything accondingly*/
195  void checkAndUpdateGameOn()
196  {
197    int i;
198    //if the state is WAITING and there are 2 or more players
199    if(getGameState() == GAME_WAITING && numberActivePlayers() >= 2 )
200    {
201      setGameState(GAME_ON);
202      printf("GAME ON. Players: %d\n", numberActivePlayers());
```

Quando um jogador sai do jogo, é verificado pela função *quitPlayer* se o número de jogadores é inferior a 2 e neste caso o jogo deve entrar em modo de pausa:

```
89  /*function to make a player quit the game*/
90  void quitPlayer(player* player)
91  {
92    if(player->state != QUIT)
93    {
94      //send to player permission to leave
95      play_response resp;
96      resp.code = CODE_PERMISSION_TO_LEAVE;
97      write(getSockFd(player), &resp, sizeof(play_response));
98    }
99
100   player->state = QUIT; //update the player state
101   close(player->sock_fd);
102   printf("Player Quit. Current number of players: %d\n", numberActivePlayers());
103
104   if(numberActivePlayers() < 2)
105   {
106     //if there are less than 2 players, freeze the game;
107     setGameState(GAME_WAITING);
108     printf("GAME FROZEN, WAITING FOR PLAYERS\n");
109   }
110 }
```

7.2 Distinção entre primeira e segunda jogada

A distinção entre primeira e segunda jogada é feita ao nível do *player*, através do seu campo *state*. Se este campo for igual a *NO_PICK*, o jogador ainda não fez jogadas e portanto a sua próxima jogada será uma *first pick*. Caso este campo seja igual a *FIRST_PICK*, o jogador já fez a primeira jogada e portanto a sua próxima será uma *second pick*. Este campo é atualizado pela função *boardPlay* do módulo *board_library* sempre que o jogador faz uma jogada.

```
7   //possible player states
8   #define NO_PICK 0 //player hasn't made his first pick
9   #define FIRST_PICK 1 //player has made his first pick
10  #define SECOND_PICK 2 //player has made his second pick and is waiting to play
11  #define QUIT 4 //player has quit
12  #define WAITING_TO_JOIN 5 //player is waiting to join the game
```

7.3 5 segundos entre jogadas

Viragem da primeira carta se não for escolhida uma segunda

A viragem da primeira carta é feita por uma *thread* que é lançada no servidor após se processar a jogada no tabuleiro:

```
385       if(resp.code == CODE_FIRST_PLAY)
386       {
387         //set a thread to hide this card in 5 secs
388         struct _aux* thread_args = (struct _aux*) malloc(sizeof(struct _aux));
389         thread_args->player = player;
390         thread_args->play = play;
391         pthread_t delayed_thread;
392         pthread_create(&delayed_thread, NULL, hideFistPickRoutine, (void*)thread_args);
393       }
```

Esta *thread* espera 5 segundos e depois, caso o jogador não tenha feita jogadas a seguir, faz com que a carta correspondente à sua primeira jogada seja virada para baixo:

```c
void* hideFistPickRoutine(void* args)
{
    struct _aux* arguments = args;
    player* player = arguments->player;
    play play = arguments->play;
    free(arguments);

    sleep(5);

    if(getGameState() == GAME_OVER)
      return NULL;

    //hide card only if current player first pick is the same as the pick received and the player state is still FIRST_PICK (or the player has left and that card is still up)
    if(playsAreSame(player->play1, play) && (player->state == FIRST_PICK || (player->state == QUIT && getBoardPlaceState(getXPlay(play), getYPlay(play)) == STATE_UP)))
    {
      play_response delayed_response;

      setBoardPlaceState(getXPlay(play), getYPlay(play), STATE_DOWN); //update card state in board
      delayed_response.code = CODE_HIDE_FIRST_PLAY;
      delayed_response.play1 = play;

      //send to all players info to hide the card
      player_list* cursor = list_of_players;
      while(!isEmptyPlayerList(cursor))
      {
        if(getPlayerFromList(cursor)->state != QUIT && getPlayerFromList(cursor)->state != WAITING_TO_JOIN)
        {
          int ret = write(getSockFd(getPlayerFromList(cursor)), &delayed_response, sizeof(play_response));
          if(ret == -1)
            quitPlayer(getPlayerFromList(cursor));
        }
        cursor = getNextElementPlayerList(cursor);
      }

      if(player->state == FIRST_PICK)
        player->state = NO_PICK; //update the player state
    }
```

Processamento da segunda jogada

O processamento da segunda jogada, como qualquer outra jogada, é feito pela função *boardPlay* do módulo *board_library*. Esta função recebe o jogador que efetuou a jogada e, se o seu estado for *FIRST_PICK*, a nova jogada será processada como segunda jogada:

```c
    else if(player->state == FIRST_PICK)
    {
      //player makes his second pick
      char * first_str = getBoardPlaceStr(getXPlay(player->play1), getYPlay(player->play1));
      char * secnd_str = getBoardPlaceStr(getXPlay(play), getYPlay(play));

      resp.play1 = player->play1;
      strcpy(resp.str_play1, first_str);
      resp.play2 = play;
      strcpy(resp.str_play2, secnd_str);

      if (strcmp(first_str, secnd_str) == 0)
      {
        //the pair second pick and first pick is correct
        player->n_corrects += 2;
        n_corrects += 2;
        if (n_corrects == dim_board * dim_board)
          resp.code = CODE_GAME_OVER;
        else
          resp.code = CODE_SECOND_PLAY_RIGHT;

        setBoardPlaceState(getXPlay(resp.play1), getYPlay(resp.play1), STATE_LOCKED);
        setBoardPlaceState(getXPlay(resp.play2), getYPlay(resp.play2), STATE_LOCKED);
        setBoardPlaceColor(getXPlay(resp.play1), getYPlay(resp.play1), player->color);
        setBoardPlaceColor(getXPlay(resp.play2), getYPlay(resp.play2), player->color);
        player->state = NO_PICK;
      }
      else
      {
        //the pair second pick and first pick is incorrect
        resp.code = CODE_SECOND_PLAY_WRONG;
        setBoardPlaceState(getXPlay(resp.play2), getYPlay(resp.play2), STATE_UP);
        setBoardPlaceColor(getXPlay(resp.play2), getYPlay(resp.play2), player->color);
        player->state = SECOND_PICK;
      }

    }
```

7.4 Delay de 2 segundos após segunda jogada errada

Caso o jogador faça um par de jogadas erradas, devem ser esperados 2 segundos e depois ambas as cartas devem ser viradas para baixo. A viragem é feita por uma *thread* que é lançada no servidor após se processar a segunda jogada no tabuleiro e se determinar que está errada:

```
394             if(resp.code == CODE_SECOND_PLAY_WRONG)
395             {
396               //set a thread to hide both cards in 2 secs
397               struct _aux* thread_args = (struct _aux*) malloc(sizeof(struct _aux));
398               thread_args->player = player;
399               thread_args->play = play;
400               pthread_t delayed_thread;
401               pthread_create(&delayed_thread, NULL, hideSecondPickRoutine, (void*)thread_args);
402             }
```

Esta *thread* espera 2 segundos e depois faz com que ambas as cartas sejam viradas para baixo:

```
309   void* hideSecondPickRoutine(void* args)
310   {
311       struct _aux* arguments = args;
312       player* player = arguments->player;
313       play play2 = arguments->play;
314       free(arguments);
315
316       sleep(2);
317
318       if(getGameState() == GAME_OVER)
319         return NULL;
320
321       play_response delayed_response;
322
323       //fist play is player->play1 and second play is the one from the arguments
324       setBoardPlaceState(getXPlay(play2), getYPlay(play2), STATE_DOWN);//update card state in board
325       setBoardPlaceState(getXPlay(player->play1), getYPlay(player->play1), STATE_DOWN);//update card state in board
326       delayed_response.code = CODE_HIDE_BOTH_PLAYS;
327       delayed_response.play1 = player->play1;
328       delayed_response.play2 = play2;
329
330       //send to all players info to hide both cards
331       player_list* cursor = list_of_players;
332       while(!isEmptyPlayerList(cursor))
333       {
334         if(getPlayerFromList(cursor)->state != QUIT && getPlayerFromList(cursor)->state != WAITING_TO_JOIN)
335         {
336           int ret = write(getSockFd(getPlayerFromList(cursor)), &delayed_response, sizeof(play_response));
337           if(ret == -1)
338             quitPlayer(getPlayerFromList(cursor));
339         }
340         cursor = getNextElementPlayerList(cursor);
341
342         if(player->state == SECOND_PICK)
343           player->state = NO_PICK; //update player state
344       }
345
```

7.5 Fim de um jogo

O jogo termina quando algum dos jogadores faz uma jogada que resulta em que todas as posições do tabuleiro fiquem acertadas. Quando isto acontece o estado do jogo é alterado para *GAME_OVER*:

```
421             if(resp.code == CODE_GAME_OVER)
422             {
423               //if the game is over, send scores to the players
424               pthread_mutex_lock(&game_state_mutex);
425               printf("GAME OVER\n");
426               game_state = GAME_OVER;
427               sendEndGameInfo();
428               pthread_mutex_unlock(&game_state_mutex);
429               //thread to start new game in 10 secs
430               pthread_t endgame_thread;
431               pthread_create(&endgame_thread, NULL, endGameRoutine, NULL);
432             }
```

Envio do resultado

É executada a função *sendEndgameInfo* no servidor, que envia a cada cliente uma estrutura de dados do tipo *endgame_info* com a informação se o jogador ganhou ou perdeu e a sua pontuação no jogo:

```
169   /*sends to the players the information of game over*/
170   void sendEndGameInfo()
171   {
172     endgame_info info; //contains player score and wether player won or lost
173     player_list* cursor = list_of_players;
174     int max = maxPoints();
175     while(!isEmptyPlayerList(cursor))
176     {
177       if(getPlayerFromList(cursor)->state != WAITING_TO_JOIN && getPlayerFromList(cursor)->state != QUIT)
178       {
179         if(getPlayerFromList(cursor)->n_corrects == max)
180           info = newEndGameInfo(RESULT_WIN, getPlayerFromList(cursor)->n_corrects); //player won
181         else
182           info = newEndGameInfo(RESULT_LOST, getPlayerFromList(cursor)->n_corrects); //player lost
183
184         int ret = write(getSockFd(getPlayerFromList(cursor)), &info, sizeof(endgame_info));
185         if(ret == -1)
186           quitPlayer(getPlayerFromList(cursor));
187       }
188       cursor = getNextElementPlayerList(cursor);
189     }
190     printf("End Game Info sent\n");
191   }
```

Intervalo de 10 segundos entre jogos

É também lançada uma *thread* que corre a função *endGameRoutine*. O que esta função faz é uma pausa de 10 segundos antes de fazer *reset* aos campos dos jogadores ativos, bem como ao tabuleiro e atualizar o estado do jogo:

```
255   /*routine that is executed when game over*/
256   void* endGameRoutine(void* args)
257   {
258     printf("NEW GAME STARTING IN 10 SECS...\n");
259     sleep(10); //wait 10 secs for next game to start
260     resetPlayers(); //reset the players
261     initBoard(board_dim); //generate a different board
262     setGameState(GAME_WAITING);
263     checkAndUpdateGameOn();
264   }
```

Depois, o servidor volta a enviar aos clientes a dimensão do tabuleiro, como se fosse o primeiro jogo.

Reinicio do jogo no cliente

Quando o cliente recebe a informação de que o jogo terminou, a *thread* responsável por receber as respostas do servidor é terminada:

```
124  /*routine that runs in a thread and constantly receives responses from the server, until the client quits or the game is over*/
125  void* receiveResponsesRoutine(void* args)
126  {
127    play_response resp;
128    while(game_state == GAME_ON)
129    {
130      int ret = read(sock_fd, &resp, sizeof(play_response));
131      if(ret == sizeof(play_response))
132      {
133        //when a valid reply is received, update the window
134        updateWindow(resp);
135        if(resp.code == CODE_GAME_OVER)
136          game_state = GAME_OVER; //game over
```

O cliente prepara-se então para receber as informações de fim de jogo:

```
250        //wait for this thread to finish
251        pthread_join(receive_responses_thread, NULL);
252
253        if(game_state == GAME_OVER)
254        {
255          //if the thread left from the game being over
256          endgame_info info;
257          //read the endgame info from the server
258          int ret = read(sock_fd, &info, sizeof(endgame_info));
259          if(ret == sizeof(endgame_info))
260          {
261            //display endgame info
262            printf("GAME OVER: Points: %d. ", getEndGameInfoPoints(info));
263            switch (getEndGameInfoResult(info)) {
264              case RESULT_WIN:
265                printf("YOU WON!\n");
266                break;
267              case RESULT_LOST:
268                printf("YOU LOST!\n");
269                break;
270            }
271          }
272          printf("NEXT GAME STARTING IN 10 SEC...\n");
273        }
```

Após isto, o cliente volta ao estado inicial: espera por receber o tamanho do tabuleiro, indicando o inicio de um novo jogo.

7.6 Limpeza após a morte/desconexão de um cliente

Quando um cliente pede permissão ao servidor para sair ou o servidor deteta que ele foi desconectado, é executada a função *quitPlayer*. Esta função envia ao cliente uma mensagem, dando-lhe permissão para terminar o seu processo (caso tenha sido o cliente a pedir para sair), atualiza o seu estado para *QUIT*, fecha o *socket* a ele associado e verifica se passaram a estar menos de 2 jogadores em jogo:

```
89  /*function to make a player quit the game*/
90  void quitPlayer(player* player)
91  {
92    if(player->state != QUIT)
93    {
94      //send to player permission to leave
95      play_response resp;
96      resp.code = CODE_PERMISSION_TO_LEAVE;
97      write(getSockFd(player), &resp, sizeof(play_response));
98    }
99
100   player->state = QUIT; //update the player state
101   close(player->sock_fd);
102   printf("Player Quit. Current number of players: %d\n", numberActivePlayers());
103
104   if(numberActivePlayers() < 2)
105   {
106     //if there are less than 2 players, freeze the game;
107     setGameState(GAME_WAITING);
108     printf("GAME FROZEN, WAITING FOR PLAYERS\n");
109   }
110 }
```

O *player* permanece na lista até o processo do servidor morrer, mas com o seu estado a *QUIT*, o que indica que será ignorado daí em diante.

7.7 Bot

O Bot é um cliente igual ao cliente UI, com a diferença de que não tem interface gráfica e as jogadas são feitas de modo automático. Isto é feito à custa de um tabuleiro que é guardado internamente e que memoriza o conteúdo das cartas que vão sendo viradas por todos os jogadores. A lógica para determinar as jogadas a fazer é a seguinte: quando existir um par de cartas que sejam uma correspondência e estejam viradas para baixo (*STATE_DOWN*), jogar essas duas cartas, caso contrário jogar a primeira carta que esteja virada para baixo e não seja conhecida e de seguida uma carta aleatória:

```
126  /*routine that runs in a thread and constantly sends plays to the server, until the client quits*/
127  void* sendPlaysRoutine(void* args)
128  {
129    play play1, play2;
130      while (game_state != GAME_QUIT)
131    {
132      if(game_state == GAME_ON)
133      {
134        //try to find a matching pair
135        if(findPair(&play1, &play2) == 1)
136        {
137          //send first pick
138            int ret = write(sock_fd, &play1, sizeof(play));
139            if(ret == -1)
140            {
141              quitClient();
142              continue;
143            }
144            usleep(250000); //wait 0.25 sec between each pick
145            //send second pick
146            ret = write(sock_fd, &play2, sizeof(play));
147            if(ret == -1)
148            {
149              quitClient();
150              continue;
151            }
152        }
153        else if(firstUnknown(&play1) == 1)
154        {
155            //if no pair is found, the fist pick will be an unknown card and the second pick will be a random card
156            int ret = write(sock_fd, &play1, sizeof(play));
157            if(ret == -1)
158            {
159              quitClient();
160              continue;
161            }
162            usleep(250000); //wait 0.25 sec between each pick
163            //second pick is random
164            play2 = newPlay(rand()%board_dim, rand()%board_dim);
165            ret = write(sock_fd, &play2, sizeof(play));
166            if(ret == -1)
167            {
168              quitClient();
169              continue;
170            }
171        }
172        usleep(250000); //wait 0.25 sec between each pick
173      }
174
175    }
176
177
178  }
```