1 Introduction

O projeto em análise teve como objetivo a implementação de um pequeno jogo de perseguição, que contém clientes, representados como bolas, bots, e prémios espalhados pelo campo de jogo, sendo o objetivo de um jogador eliminar os outros, colidindo com os mesmos, enquanto dispõe de prémios, como opção de cura.

O projeto em si tem como objetivo o desenvolvimento e aplicação prática dos fundamentos apreendidos, nomeadamente os conceitos inerentes às conexões entre processos, e, posteriormente, a manipulação de threads, como alternativa à implementação distribuída do controlador do jogo, entre vários processos, bem como a segurança e os mecanismos de sincronização intrínsecos às primeiras.

2 Implemented functionalities

2.1 Part 1

Table 1: Implemented functionalities (PART A)

	Not implemented	With faults	Totally Correct
ChaseThem Server			
New client on server			X
Assignment of letter			X
Array of clients on server			X
Ball movement			X
ChaseThem Client			
Ball Movement			X
Update of client			X
Bots			
Array of bots on server			X
Bots movement			X
Prizes			
5 prizes on the field			X
Creation of prize every 5 seconds			X
Rules			
Ball rams ball			X
Bot rams ball			X
Ball rams prize			X
Update ball health			X

2.2 Part 2

Table 2: Implemented functionalities (PART B)

	Not implemented	With faults	Totally Correct
BetterChaseThem Server			
Connection of new client to server			X
Assignment of letter			X
Array/list of clients on server			X
Limit to the number of clients			X
Ball movement			X
ChaseThem Client			
Ball Movement			X
Update of clients			X
Bots			
Array of bots on server			X
Bots movement			X
Prizes			
5 prizes on the field			X
Creation of prize every 5 seconds			X
Rules			
Ball rams ball			X
Bot rams ball			X
Ball rams prize			X
Update ball health			X
Ball death (+ 10 seconds wait)			X
Synchronization			
Movement of balls			X
Movement of bots			X
Updates of health			X

2.3 Description of faulty functionalities

3 PART A Description of code

3.1 Data types

Estrutura do jogo:

Game: Contem o estado atual do jogo, informações sobre os players, bots e prizes, e número total de players e prizes.

Lista de estruturas que contem dados sobre os elementos do jogo:

Client: Representa um player, guarda as suas coordenadas, a vida, o id, o seu unix address e ponteiro do próximo cliente.

Bot: Representa um Bot, guarda número total de bots, as coordenadas, o id, e ponteiro para o próximo Bot.

Prize: Representa um Prize guarda o seu valor, as coordenadas, o id, e ponteiro para o próximo Prize.

Estrutura no Cliente:

Player: Contem as informações do cliente.

Estruturas para comunicação:

***Message_Types*:** Define todos os tipos de mensagem disponíveis.

***Ball_Info*:** Define uma mensagem do tipo ball info. Contém o id atribuído ao cliente.

***Ball_Movement*:** Define uma mensagem do tipo ball movement. Contém o movimento feito pelo cliente.

***Field_Status*:** Define uma mensagem do tipo field status. Contém uma representação matricial do campo atual. Permite ao client ver o jogo.

***Message*:** Estrutura genérica utilizada para a troca de mensagens. Contém todas as quatro estruturas anteriores, permitindo a troca de qualquer tipo de mensagem.

3.2 Functions List

Server:

bool position_available(Game game, int x, int y): verifica se a coordenada dada está disponível.

bool client_id_available(Game game, int id): verifica se o dado id está disponivel.

Void draw_player(WINDOW* win, void* object, char field[WINDOW_SIZE][WINDOW_SIZE], int delete, char* type): desenha o id do objecto no win, conforme se o type igual a Client, Bot ou Prize, caso delete igual a false irá apagar o id na coordenada do objecto dado.

void add_prizes(WINDOW* win, int quantity, int field[4][30], Game* game): adiciona certo número de prizes contidos no field[4][30] ao estado do jogo (game), faz display no win do estado atual.

void create_bot(WINDOW* win, Game* game, Message message): adiciona certo número de bots contidos na message enviada pelo bot ao estado do jogo (game), faz display no win do estado atual.

void create_client(WINDOW* win, Game* game): cria e adiciona o cliente no estado do jogo (game), faz display no win do estado atual.

void moove_player(WINDOW* win, void* object, char* type, int direction, Game* game): conforme o type (Client, Bot ou Prize) do object será verificado se a sua proxima coordenada provocada pela direction será válida e as consequência para si e outros elementos presentes no estado atual do jogo(game).

int HP_GAIN(int Prize, int ActualHP): retorna o resultado da HP em consequência de colidir com um prize.

void init_field(Game* game): inicializa a matriz field_status.

void copy_field(Game game, int field[4][30], char* type): insere o estado atual no jogo(game) para a matriz field.

int get_direction(int new_coord_x, int new_coord_y, int coord_x, int coord_y): retorna a direção que corresponde a mudança de coordenada.

void print_MessageWindow(WINDOW* win, Game game): apresenta no message window, o id e HP de todos os jogadores no estado atual do jogo (game) .

void Client_updade(Client* cliente, int id, int HP, int x, int y, int Mov, struct Client* next): actualiza os dados do cliente dado.

Client:

void moove_player(Player* player, int direction): move o player para a dada direção .

void updateWindow(WINDOW* win, WINDOW* message_win, int field[4][30], Player* player): apresenta no ecrã o estado atual o jogo no win.

Player updadePlayerInfo(int x, int y, int HP, int id): actualiza os dados do cliente.

Bot:

void move_bot(Bot* bot, direction_t direction): move o bot para a dada direção.

void init_bot_field(char f[WINDOW_SIZE][WINDOW_SIZE]): inicializa a matrix dado.

void copy_field(Bot* head, int field[4][30]): transcreve os dados de todos os bots para a matriz field.

Prize_generator:

void generate_prize(int quantity, int memory[4][30]): gera uma dada quantidade de prizes e é guardada na matriz memory.

Manipulação de Listas:

void* init_node(char* node_type): o node_type pode ser do tipo Cliente, Bot ou Prize, conforme o dado tipo o nó é criado.

void* append_list(void* head, void* node, char* type): insere o node dado no final da lista.

void* Remove_node(void* head, int id_out, char* node_type): remove o node com um determinado id_out da lista.

void* find_node(void* head, int id, char* node_type): encontra o node com um determinado id, e é retornado.

3.3 Implementation details

Tanto os movimentos bots como a colocação de novos prizes foram feitas recorrendo a temporizadores, sendo que são enviados movimentos aleatórios ao server, no caso dos bots, e uma simples notificação com o número de prizes a adicionar, no caso dos prizes, ficando o server encarrege de atribuir cada movimento, se válido, ao respetivo bot, e de colocar cada prize de forma aleatória em campo.

Inicialmente, o server aguarda uma mensagem inicial vinda dos prizes, e de seguida outra vinda dos bots, de forma a garantir ligações fiáveis e o mais privadas possíveis com os mesmos. Após o estabelecimento destas conexões iniciais, o server irá então começar o jogo e estar aberto a novos clientes. Contudo, utiliza os endereços previamente recebidos para os prizes e bots, de forma a validar a comunicação com os mesmos, impedindo que terceiros tentem desempenhar a função dos processos controladores dos bot/prizes.

Foi também pré-definido um número máximo de players em jogo em simultâneo, 10, pelo que não serão aceites novos clientes caso este limite seja atingido.

4 PART B Description of code

4.1 Data types

Estruturas para comunicação com o cliente:

Message_Types: Define todos os tipos de mensagem disponíveis.

Ball_Info: Define uma mensagem do tipo ball info. Contém o id atribuído ao cliente.

Ball_Movement: Define uma mensagem do tipo ball movement. Contém o movimento feito pelo cliente.

Field_Status: Define uma mensagem do tipo field status. Contém uma representação matricial do campo atual. Permite ao client ver o jogo.

Message: Estrutura genérica utilizada para a troca de mensagens. Contém todas as quatro estruturas anteriores, permitindo a troca de qualquer tipo de mensagem.

Connection**:** Descreve uma ligação TCP. Contém o socket atribuído à ligação, bem como o endereço do destinatário ou o próprio endereço, caso se seja o listener do server.

Estruturas auxiliares para implementação da dinâmica do jogo:

Coord: Descreve uma coordenada no ecrã.

List: Descreve uma lista. Cada element da lista contém uma coordenada (***Coord***) e um ponteiro para dados (***void* data***).

Estruturas para implementação da dinâmica do jogo:

Game: Gere o estado global e todas as variáveis do jogo. Contém listas para os clientes, bots, prizes, e threads utilizadas. Contém também contadores, que registam o número total de clientes, bots e prizes.

Client: Representa um player, dentro da semântica do jogo. Contém o id, vida e coordenada do player.

Client_Con: Estrutura especializada para a gestão das interações com cliente. Particularmente útil para lidar com a morte de um player. Contém a conexão ao cliente (***Connection***), o seu respetivo id, a thread que o gere, e uma variável bolleana, indicadora do estado atual do cliente, isto é, em jogo ou não.

Estrutura para visualização do jogo:

Display**:** Estrutura genérica que serve de interface para a utilização da biblioteca ncurses. Permite a exibição do jogo em ecrã tendo um ***Field_Status***, e a construção e envio do mesmo para todos os clientes, tendo o ***Game***.

4.2 Functions List

Funções para comunicação com o cliente:

Funções acessíveis ao cliente:

Connection TCP_Connect(char* IP , int port): Inicia um ligação tcp com o servidor recebido.

void TCP_Delete(Connection* con): Apaga estrutura ***Connection***.

bool TCP_Send(Connection con, Message message): Envia uma messagem.

Message TCP_Read(Connection con): Recebe uma messagem.

bool TCP_Still_Connected(Connection con): Verifica se ainda existe ligação.

Funções acessíveis ao server:

Connection TCP_Listen(char* IP , int port): Cria uma ***Connection*** com um socket listener, com o ip e porto recebidos.

Connection TCP_Accept(Connection listener): Aceita um cliente.

Funções para manipulação de coordenadas:

Coord random_coord(Game game): Retorna uma coordenada livre aleatória.

Coord random_move(Coord old): Devolve uma coordenada que resulta de um movimento aleatório.

Funções para manipulação de listas:

List List_init(): Inicia lista.

List List_append(List head, Coord coord, void* data): Adiciona um elemento.

List List_get(List head, Coord coord): Devolve elemento com a coordenada recebida.

List List_remove(List head, Coord coord , void (*delete_data) (void*)): Apaga elemento com a coordenada recebida, de acordo com a função recebida.

void List_delete(List head , void (*delete_data) (void*)): Apaga a lista, de acordo com a função recebida.

Funções para implementação da dinâmica do jogo:

Funções da dinâmica dos bots e prizes:

void Bots_Move(Game game , List Bots): Move todo os bots, e aplica as consequências desses movimentos.

void Prizes_Add(Game* game , int quantity): Adiciona “quantity” prizes ao jogo.

int Prize_Remove(Game* game, Coord coord): Apaga prize na coordenada dada.

Funções da dinâmica dos clientes:

Client Client_Add(Game* game): Adiciona um novo cliente ao jogo.

void Client_Move(Game* game , Client client , int move): Move o cliente em causa, e aplica as consequências desse movimento.

void Client_Remove(Game* game, Client* player): Remove cliente do jogo.

Funções para manipulação do ecrã do jogo:

Funções acessíveis ao cliente:

Display Display_init(): Inicia um display.

void Display_delete(Display disp): Apaga um display.

int Display_get_char(Display disp): Lê um caracter selecionado no teclado.

void Field_Status_Display(Display disp , Field_Status field): Apresenta o jogo.

Funções acessíveis apenas ao server:

void Display_Game(Display disp , Game game , List connections): Constrói um ***Field_Status*** com base no **Game** recebido, e envia-o a todos os clientes em ***connections***.

Funções para gestão de cada interveniente no jogo (funções thread):

void Client_Controller(void* client_con_): Gere a comunicação e movimentos dos clientes, bem como as suas consequências.

void Bot_Controller(void* nothing): Gere a criação e movimentos dos bots, bem como as suas consequências.

void Prize_Controller(void* nothing): Gere a criação dos prizes.

4.3 Implementation details

Em ambas as fases do projeto, tanto os movimentos bots como a colocação de novos prizes foram feitas recorrendo a temporizadores, sendo que, na parte A, eram enviados movimentos aleatórios ao server, no caso dos bots, e uma simples notificação

com o número de prizes a adicionar, no caso dos prizes, ficando o server encarregue de colocar cada um de forma aleatória em campo.

Na parte B, estando os bots e os prizes incluídos no server, estes modificam diretamente o campo, e, portanto, o ***Bot_Controller*** move diretamente cada bot e aplica as suas consequências, enquanto o ***Prize_Controller*** define onde colocar o prize que criou.

Foi também pré-definido um espaço mínimo livre no campo de jogo, um oitavo de todo o campo, pelo que não serão aceites novos clientes caso este limite seja ultrapassado.

4.4 Threads

No server, existe uma thread para os bots, outra para os prizes, e ainda uma terceira para a comunicação particular com um cliente. É ainda criada uma thread sempre que, após um movimento de um cliente ou de um bot, esse movimento resulta na morte de um outro cliente. O processo de envio da mensagem de morte, bem como o compasso de espera e internal shutdown do cliente em caso de inoperação, são geridos por esta thread. Por fim, existe ainda uma primeira thread, que de facto desempenha o papel de host, tornando possível o término correto do server, por leitura de um caracter. Em baixo apresenta-se as funções que definem cada thread mencionada, pela ordem apresentada:

- ***Bot_Controller***
- ***Prize_Controller***
- ***Client_Controller***
- ***No_Time_To_Die***
- ***Actual_server***

No cliente, existe, à parte da thread inicial, cuja função se resume a ler caracteres do teclado, uma outra thread, que recebe mensagens do server e mantém o ecrã atualizado, e ainda uma terceira, que lida com o caso particular da morte do cliente, exibindo uma contagem decrescente, e terminando o cliente em caso de inoperação. Seguem-se as funções correspondentes a cada thread, pela ordem abordada:

- ***Online_Screen***
- ***Waiting_ten_s***

4.5 Shared variables

Na solução apresentada, existe a variável global game, que sintetiza toda a dinâmica de jogo, guardando todos os players, bots e prizes, e portanto é acedida por todas as threads. A variável display, não incluída na estrutura game, é também uma variável global, acedida em todas as threads do server. Por fim, existe ainda uma lista, responsável por guardar a ligação tcp de cada cliente, que tamém é acedida em diversas ocasiões.

No cliente, as variáveis partilhadas entre threads resumem-se à ligação tcp com o server, o id do cliente, atribuído automaticamente pelo server, e o seu display, todas variáveis globais.

4.6 Synchronization

Sincronização entre as operações das threads foi alcançada com recurso a mutexes, resolvendo as racing conditions envolvendo, no caso do server, a variável game e a lista de connections, e no caso do cliente apenas o seu display.

No server, tornou-se essencial utilizar mutexes para acesso às variáveis game e connections, uma vez que, como foi dito, estas são utilizadas em inúmeras ocasiões, por todas as threads.

No cliente por sua vez, houve apenas a necessidade de garantir o sincronismo entre uma mensagem enviada pela thread de leitura e as mensagens recebidas do server, processadas pela thread ***Online_Screen***.

5 PART A Communication

5.1 Transferred data

Entre o servidor e clientes a troca de informação é efetuada através da estrutura definido como variável do tipo **Messages,** com o tipo da mensagem definido dentro da própria estrutura no qual pode ser:

-CONNECT, cliente avisa que quer conectar;
-BALL_MOVEMENT, cliente envia o movimento da bola ao servidor;
-BALL_INFORMATION, servidor envia a informação da bola criada ao servidor;
-FIELD_STATUS, servidor envia o estado atual do jogo ao cliente;
-SERVER_FULL, servidor avisa o cliente que a sua capacidade está no máximo;
-HEALTH_0, servidor avisa ao cliente que está morto, ou seja, HP = 0;
-DISCONNECT, cliente deu quit e avisa o servidor da sua saída;

5.2 Error treatment

No server se algum dos clientes conectados enviar um tipo de mensagem invalida, tal mensagem é ignorada. No read e write é verificado se o número de bytes returnado é um valor valido, ou seja, maior que zero.

6 PART B Communication

6.1 Transferred data

Como já foi abordado, as seguintes estruturas refletem os tipos de mensagem e dados transferidos entre um cliente e o servidor:

Message_Types: Define todos os tipos de mensagem disponíveis.

Ball_Info: Define uma mensagem do tipo ball info. Contém o id atribuído ao cliente.

Ball_Movement: Define uma mensagem do tipo ball movement. Contém o movimento feito pelo cliente.

Field_Status: Define uma mensagem do tipo field status. Contém uma representação matricial do campo atual. Permite ao client ver o jogo.

Message: Estrutura genérica utilizada para a troca de mensagens. Contém todas as quatro estruturas anteriores, permitindo a troca de qualquer tipo de mensagem.

***Connection*:** Descreve uma ligação TCP. Contém o socket atribuído à ligação, bem como o endereço do destinatário ou o próprio endereço, caso se seja o listener do server.

O estabelecimento de uma comunicação bem como a troca efetiva de mensagens são garantidas pelas seguintes funções:

Funções acessíveis ao cliente:

Connection TCP_Connect(char* IP , int port)

void TCP_Delete(Connection* con)

bool TCP_Send(Connection con, Message message)

Message TCP_Read(Connection con)

bool TCP_Still_Connected(Connection con)

Funções acessíveis ao server:

Connection TCP_Listen(char* IP , int port)

Connection TCP_Accept(Connection listener)

6.2 Error treatment

De forma a validar e aumentar a robustez das mensagens trocadas, as respetivas funções de envio/recessão verificam se ainda existe uma ligação fiável bem como, sabendo que se irá ler uma estrutura ***Message***, garantem que todos os dados são lidos/enviados, caso não sejam todos enviados de uma só vez, ou que a troca não foi bem-sucedida, caso não se recebam os dados todos.

Do lado do servidor, uma vez que as únicas mensagens recebidas que de facto acarretam a leitura e processamento de dados são as mensagens ***Ball_Movement***, é verificado se o movimento recebido corresponde a um dos movimentos possíveis, de acordo com as constantes definidas pela biblioteca ncurses.

7 Conclusion

Os conceitos aprendidos ao longo do período foram bem interiorizados, nomeadamente os mecanismos de comunicação entre processos, seja por UDP o TCP, bem como a implementação de multi-threading, que permitiu a execução em concorrência de diversas tarefas, que por sua vez resulta numa melhoria da velocidade de resposta aos clientes. No geral, a solução desenvolvida não possui falhas significativas, e consideram-se então alcançados os objetivos propostos.