Instituto Superior Técnico

Engenharia Eletrotécnica e de Computadores

Relatório do Projeto

Docente:
João Silva

Grupo:
Francisco Melo Nº 86998
Inês Moreira Nº 88050

Índice

1 Arquitetura do Sistema

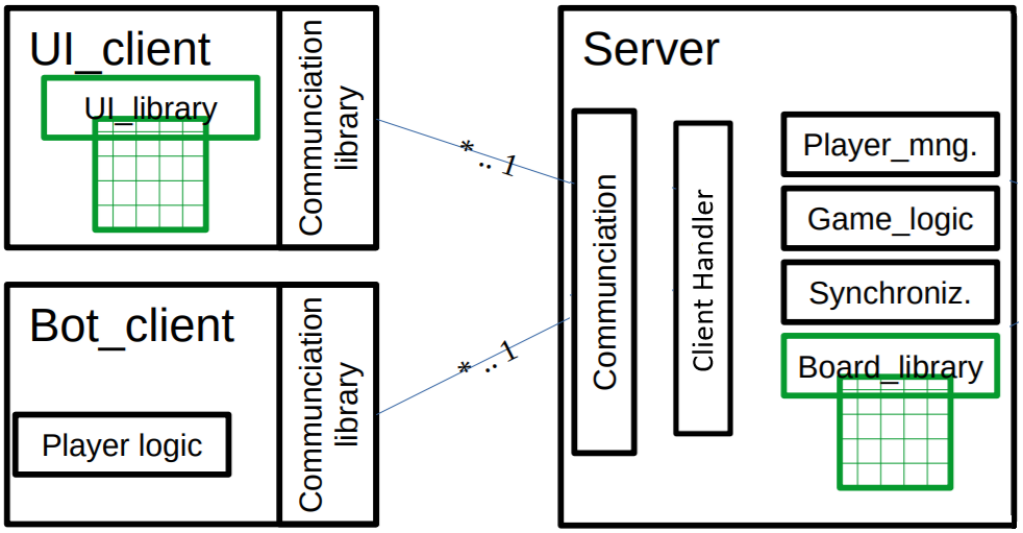

Figura 1: Arquitetura do projeto com os diferentes módulos e nós representados

Este trabalho tem como objetivo implementar uma versão *distributed multiplayer* do jogo da memória. Para tal, começamos por dividir o projeto nos seguintes nós:

- Servidor;
- Cliente UI;
- Cliente Bot.

Para garantir o funcionamento com robustez do jogo multijogador, a gestão e lógica do jogo encontra-se centralizada no servidor.

Este servidor tem a função de gerir a lógica do jogo, sincronizar todos os jogadores em todos os momentos em relação às alterações feitas no tabuleiro de jogo. O cliente terá apenas a função de receber as alterações realizados no tabuleiro (*clicks* do jogador) identificando assim a posição do *board* que foi alterada, enviando-a de seguida para o servidor, onde este irá efetuar o processamento da informação.

A presença de múltiplos jogadores tornou obrigatória a implementação de um servidor que recebesse as alterações no tabuleiro de diversos jogadores simultaneamente. Então estes jogadores (através da execução do cliente), irão conectar-se ao servidor e enviar as suas escolhas, recebendo o estado do tabuleiro (através de cartas visíveis ou não) que será comum a todos os clientes que estiverem a jogar. Quando uma carta é virada por um cliente, todos os outros serão capazes de observar tal acontecimento, e o tabuleiro deverá assim ser atualizado.

Tendo em conta esta atualização constante do tabuleiro através de múltiplos acontecimentos simultâneos, tornou necessário a utilização de Sincronização, que irá ser analisada em detalhe ao longo deste relatório.

É de referir que dada a natureza do jogo (fazer *match* de pares), a dimensão do board tem que ser sempre par.

1.1 Servidor

A main thread tem o papel de aceitar novas conexões de novos jogadores e ainda de fazer o *shutdown* deste. Quando um novo jogador se conecta, a main thread cria uma nova thread responsável (que por sua vez também irá criar outra thread de auxílio para os *timers*) dedicada ao novo cliente.

De modo a promover *Data Abstraction* e *Least Knowledge*, dividimos o projeto nos seguintes módulos lógicos:

- Player Management;
- Game Logic;
- Synchronization;
- Board Library;
- Communication.

Os módulos lógicos anteriores são partilhados pelas diferentes N threads que gerem N jogadores. Sendo o módulo *Client Handler* responsável pela gestão de cada cliente.

1.1.1 Módulos lógicos

Módulo	Descrição
Game Logic	Código que garante o funcionamento correto do jogo de acordo com as regras do enunciado
Synchronization	Módulo responsável por evitar *race conditions* em estruturas de dados partilhadas por várias threads
Board Library	Todo o código necessário para manipular a **board**
Communication	Layer que faz a interface de comunicação entre o servidor e cliente

Tabela 1: Descrição dos módulos do servidor

1.2 Cliente

A arquitetura no cliente UI e no bot são semelhantes, estas partilham o módulo de comunicação e apenas diferem na maneira com que interpretam a data recebida pelo servidor e como processam os dados a serem enviados. Enquanto que o cliente UI envia a jogada que utilizador fez no tabuleiro gráfico e posteriormente também manifesta a resposta do servidor neste tabuleiro, o bot envia jogadas random sem qualquer tipo de suporte gráfico e quando recebe a resposta do servidor simplesmente imprime-as no terminal. Reutilizámos algum código utilizado no servidor nomeadamente do módulo de comunicação.

Para ambos os tipos de clientes, temos uma thread que trata do envio das jogadas e outra que recebe e interpreta as respostas do server.

Módulo	Descrição
Communication	Layer que faz a interface de comunicação entre o servidor e cliente
UI_Library (UI Client)	Módulo responsável pela interação gráfica entre jogador e cliente
Player Logic (Bot)	Tem o papel de interpretar as respostas do servidor às jogadas enviadas

Tabela 2: Descrição dos módulos do cliente

2 Organização do Código

De modo a organizar o projeto, dividimo-lo em 3 grandes secções: servidor, cliente UI e bot. Estas secções estão então dividas nos seguintes módulos:

- **Board_library** (Servidor);
- **UI_library** (Cliente UI);
- **Biblioteca de Comunicação** (servidor e clientes);
- **Gestão de Jogadores** (servidor);
- **Lógica do jogo** (servidor);
- **Sincronização** (servidor);
- **Lógica do Jogador** (bot).

O módulo **Lógica do jogo** está implementado no servidor através do ficheiro board_library.c estando centrado na função *boardPlay*. Este módulo é o responsável por todo o processamento do jogo, analisando a jogada recebida pelo cliente que corresponda a umas coordenadas (x,y) do board e processando posteriormente uma resposta a tal jogada.

Relativamente ao módulo **Sincronização**, encontra-se presente em todos os módulos e ficheiros somente do servidor, visto que não é necessário sincronismo no cliente, de modo a garantir o correto funcionamento do modo multijogador (que é implementado em multi-threading).

O módulo **Biblioteca de Comunicação** é o que estabelece a comunicação entre o cliente e o servidor através da utilização de *socket stream* (TCP), estando presente no ficheiro com.c.

O módulo **Board_library** é o módulo que contém as funções de manipulação do board, este está presente no ficheiro board_library.c

O módulo **UI_library** contém todas as funções relacionadas com SDL, estando assim presente apenas no cliente UI no ficheiro UI_library.c.

O módulo **Gestão de Jogadores** presente no servidor é o que faz a gestão dos jogadores ao longo de todo o jogo e está presente essencialmente no ficheiro player_manage.c. Neste ficheiro estão presentes as funções que geram a cor de cada jogador, função que permite criar um novo jogador no jogo quando uma nova conexão chega e ainda as funções essenciais que permitem que no fim do jogo seja identificado um vencedor.

O módulo **Lógica de Jogador** no bot interpreta os dados recebidos pelo servidor.

No ficheiro utils.c estão funções de auxílio/compactação de código de diversos módulos.

3 Estrutura de Dados

Ao longo do projeto, optámos por utilizar bastantes variáveis do tipo *char*, pois estas variáveis têm apenas 1 byte, e em muitos casos (por exemplo em flags) isso é suficiente. Portanto este tipo de dados não é usado apenas para representar caracteres. Posto isto, nas seguintes secções vamos analisar como concretizámos certas estruturas de dados que são fulcrais para o armazenamento de dados durante a execução do sistema.

3.1 Servidor

3.1.1 Lista de jogadores

```
typedef struct Node_Client_Struct{
    int sock_fd;
    sem_t* sem_pointer;
    struct Node_Client_Struct* next;
    struct Node_Client_Struct* prev;
}Node_Client;
```

Figura 2: Estrutura para cada nó da lista de jogadores

A primeira decisão foi escolher a estrutura de dados para guardar a informação sobre cada jogador, havia várias opções como:

- Pilha simplesmente ligada;
- Pilha duplamente ligada;
- Vetor estático;
- Vetor dinâmico.

Primeiramente, o facto de não haver necessidade de manter algum tipo de ordem específica na lista fez-nos escolher uma lista do tipo *stack* (ou pilha), pois esta estrutura de

dados permite ter apenas uma variável para guardar a head, que é utilizada em simultâneo para inserções na lista e para leituras da mesma. O mesmo não acontecia com os outros tipos de lista pois é necessário guardar também a tail.

Com o fim de permitir a remoção de um nó sem necessidade de procura (a thread que quer remover o nó tem um ponteiro para este) decidiu-se fazer uma pilha duplamente ligada. Esta estrutura de dados pertence ao módulo **Player Management**.

Membro	Descrição
int sock_fd	File descriptor do socket do jogador correspondente
sem_t* sem_pointer	Pointer para o semáforo do jogador correspondente
Node_Client* next	Pointer para o próximo nó da pilha
Node_Client* prev	Pointer para o nó anterior da pilha

Tabela 3: Descrição dos membros da estrutura Node_Cliente

3.1.2 Board

```
typedef struct Board_Place_Struct{
    char str[3];
    char is_up;     //0 - se a carta estiver virada para baixo (carta branca)
                    //1 se a carta estiver UP ou LOCKED (carta não branca)
    char owner_color[3];
    int code;
    pthread_mutex_t mutex_board;
}Board_Place;
```

Figura 3: Estrutura para cada nó da lista de jogadores

O board é uma matriz NxN jogadores (quadrada). Foi concretizada através de um *array* de 2 dimensões da estrutura Board_Place. Optámos por esta concretização pois deixar de haver a necessidade de ter funções de auxílio à indexação do vetor e também para tornar o código mais intuitivo. Esta estrutura de dados pertence ao módulo **Board Library**.

Membro	Descrição
char str[3]	String de 2 caracteres (+ \0)
char is_up	Flag que indica que se a carta está UP ou LOCKED
char owner_color[3]	Vetor que contém guardada a cor do jogador dono da casa
int code	Código (resp.code) proveniente do processamento da jogada por parte da lógica do jogo
pthread_mutex_t mutex_board	Mutex do elemento do board

Tabela 4: Descrição dos membros da estrutura Board_Place

3.1.3 Lista de Jogadores Vencedores

```
typedef struct Score_List_Struct{
    Score_Node* head;
    int top_score;
    int count;
    sem_t* sem_pointer;
}Score_List;
```

(a) Estrutura da lista do score

```
typedef struct Score_Node_Struct{
    int sock_fd;
    struct Score_Node_Struct* next;
}Score_Node;
```

(b) Estrutura para cada nó da lista de score

Figura 4: Estruturas para a lista de vencedores

Apesar de ser rara a vez em que mais do que 1 jogador é vencedor, o programa tem que estar preparado para tal. Por isso, implementámos uma lista (também um *stack*) de modo a conseguirmos guardar múltiplos vencedores. Esta lista pertence ao módulo **Player Management**.

Membro	Descrição
Score_Node* head	Pointer para a cabeça da lista de vencedores
int top_score	Inteiro que durante o jogo vai sendo atualizado como número de jogadas e também serve para durante o procedimento de escolher o(s) jogador(es) vencedor(es) se saber qual é o top score
int count	Contador do número de jogadores que já foram considerados para vencedor durante o processo de eleição de melhor jogador
sem_t* sem_pointer	Pointer para o semáforo da *Master Thread* que é responsável pelo reset

Tabela 5: Descrição dos membros da estrutura Score_List

Membro	Descrição
int sock_fd	File descriptor da socket do vencedor (ou de um dos vencedores)
Score_Node* next	Pointer para o próximo nó

Tabela 6: Descrição dos membros da estrutura Score_Node

3.1.4 Cmn_Thr_Data

```
typedef struct Cmn_Thr_Data_Struct{
    Play_Response resp;         //Estrutura que é enviada ao cliente
    char* buff_send;        //Buffer auxiliar para enviar data ao cliente
    int sock_fd;
    int n_corrects;         //Score do jogador
    sem_t sem;
    pthread_mutex_t mutex_timer;
}Cmn_Thr_Data;
```

Figura 5: Estrutura para cada nó da lista de jogadores

Esta é uma estrutura que contém os dados partilhados entre as 2 threads dedicadas a cada cliente. Esta estrutura pertence ao módulo **Client Handler**

Membro	Descrição
Play_Response resp	Resposta da jogada processada por parte do módulo da lógica do jogo
char* buff_send	Buffer utilizado para enviar o resp ao jogador, partilha-se para evitar alocações desnecessárias
int sock_fd	File descriptor da socket do cliente respetivo
int n_corrects	Pontuação atual do jogador
sem_t sem	Semáforo para controlo da timer thread
pthread_mutex_t mutex_timer	Mutex para garantir sincronismo desta estrutura entre as 2 threads

Tabela 7: Descrição dos membros da estrutura Cmn_Thr_Data

4 Protocolos de Comunicação

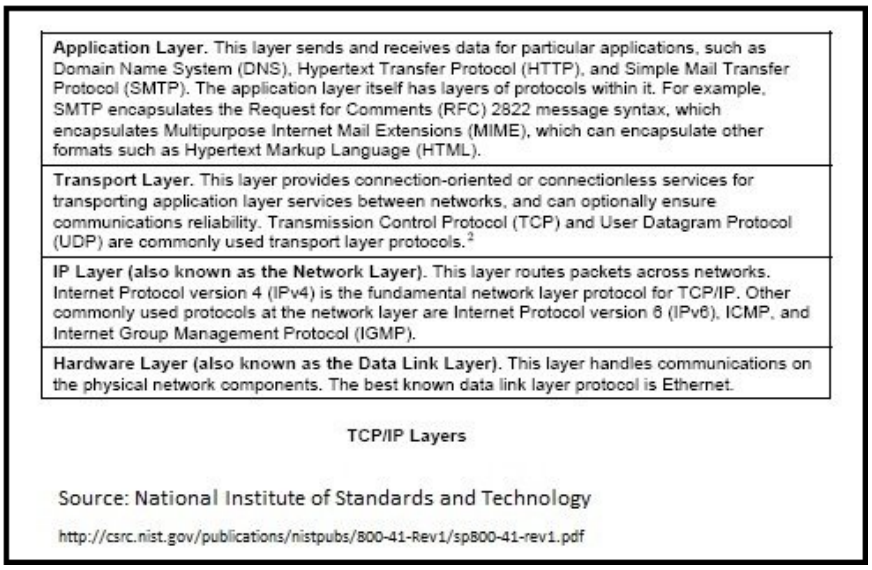

Figura 6: TCP/IP stack layers

Relativamente ao modo como é feita a comunicação, foi tido como referência em todo o projeto, a existência de um único servidor que efetua todo o processamento de jogadas, ou seja, recebe a conexão de clientes, recebe a escolha de uma dada peça do tabuleiro, efetua o processamento dessa informação, e envia-a de seguida a todos os clientes conectados.

Os clientes não irão comunicar então diretamente entre si, mas sim através de um servidor comum, constatando assim que não estamos perante um sistema que se rege segundo padrões *peer-to-peer*, mas sim segundo uma topologia *Client - Server* (apesar de não ser a "clássica"). Nesta topologia, existe um servidor comum que servirá de intermediário a todos os clientes, sendo que toda a parte lógica do código se encontra aqui focalizada e centralizada.

Para concretizar a comunicação entre servidor e clientes, decidimos utilizar sockets *stream* no domínio AF_INET - *TCP* (protocolo orientado para conexões retratado na figura 6). Apesar de ser também possível utilizar UDP (pois a ordem de chegada de dados não é importante e a possível perda de pacotes também não é crítico) saber se um cliente se desconectou é difícil neste tipo de protocolo pois não é *connection based* e por isso seria necessário um aviso por parte do cliente de que este iria se desconectar. Este "adeus"pode falhar, por exemplo, se o cliente for terminado de uma maneira não "graciosa"(por exemplo com um SIGKILL). Esta vulnerabilidade pode fazer com que haja ocupação perpétua e desnecessária de recursos e isso interfere com o nosso objetivo de otimizar a utilização de recursos de modo a oferecer 100% de longevidade por parte do servidor, enquanto que com *TCP* o servidor consegue facilmente detetar a perda de conexão com o cliente.

Decidimos definir bem o tamanho dos *streams* enviados do servidor para o cliente e do cliente para o servidor (apesar de por vezes o servidor apenas precisar de enviar o resp.code (1 inteiro)). De modo a manter um protocolo consistente:

- Servidor → Cliente: Um stream do tamanho da estrutura *Play_Response*;
- Cliente → Servidor: Um stream do tamanho de 2 inteiros.

A estrutura *Play_Response* tem um membro **code** (inteiro). É através desta variável que foi desenvolvida um sistema de código para o servidor avisar o cliente de diversos eventos bem definidos.

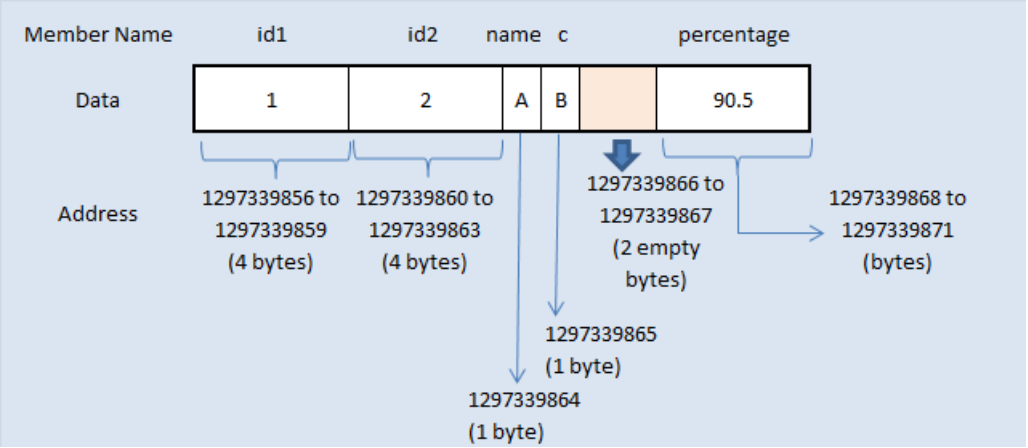

Figura 7: *Structure Padding*

Para enviar uma estrutura entre diferentes máquinas é necessário ter em atenção a *structure padding*. Isto está relacionado com a forma com que as estruturas são guardadas na memória, pois dependendo de diversos fatores como o nº de bits de trabalho da máquina (64bits/32bits...) ,vai haver um padding diferente no armazenamento de estruturas na memória, portanto, se enviarmos diretamente uma estrutura os endereços em branco utilizados para fazer os arranjos de *packets* também irão ser enviados. Se a máquina destinatária tiver um sistema de padding diferente, isto irá resultar uma corrupção de dados. Uma maneira de resolver isto é usando a função *memcpy* para copiar o conteúdo de uma estrutura para um buffer. Outra forma seria serializar membro a membro da estrutura, por exemplo, gravar byte a byte de um inteiro num buffer através de *shift rights*.

Um inteiro pode corresponder a diferentes tamanhos em diferentes compiladores e máquinas (por exemplo o tamanho de um inteiro no servidor ser 8bytes e no cliente ser 4bytes) e nesses casos este protocolo falhará. No entanto, a resolução deste *bug* não é do âmbito desta cadeira e por isso não a resolvemos.

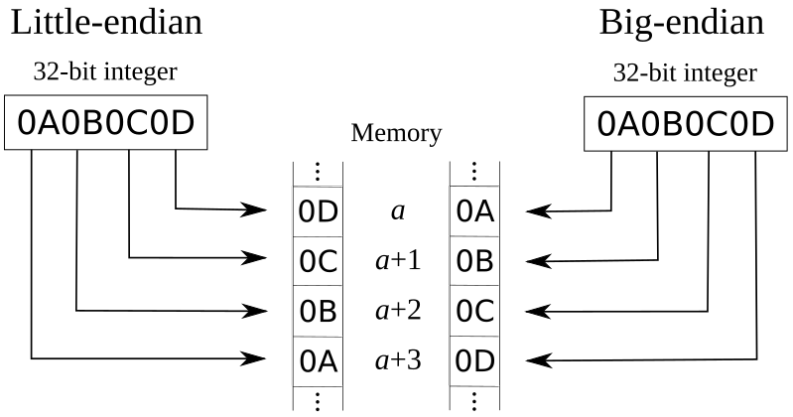

Figura 8: Problema de *Endiness*

No entanto, quando se envia variáveis com tamanho superior a 1byte (ex: inteiros), temos de nos certificar que a ordem dos bytes que constituem uma só palavra não é alterada na chegada, pois se a ordem mudar irá haver corrupção de dados.

Isto acontece quando a máquina do remetente guarda palavras num dos *endian* e a máquina do destinatário guarda palavras no *endian* inverso designando-se este problema por *Endiness* e está retratado na figura 8. Para tal utilizamos a função htonl() para converter de *host byte order* para *network byte order* no envio e a função ntohl() para converter de *network byte order* para *host byte order* na receção. Este processo apenas é necessário para o envio de inteiros pois os caracteres são palavras de apenas 1 byte, não correndo assim o risco de verem a ordem dos seus bytes desfeita como já acima mencionado.

Em suma, quando enviamos dados usando sockets no domínio *AF_INET*, o nosso protocolo de comunicação tem que estar adaptado para:

- Diferentes *endianness*;
- Diferentes *paddings*;
- Diferenças no tamanho de tipos de dados intrínsecos (e.g int, char..).

4.0.1 Início do jogo

Quando é iniciado o servidor, sendo introduzida a dimensão do tabuleiro, é gerado o board e são inicializados todos os parâmetros da socket que irá fazer a conexão servidor - cliente. Sendo assim, na main thread do servidor, estamos apenas a aceitar conexões de novos jogadores (isto enquanto o número de jogadores for inferior ao número máximo permitido, o que será explicado mais à frente).

4.0.2 Estado atual do Board

Sempre que um novo cliente pretende entrar no jogo, caso ainda não tenha sido atingido o número máximo de jogadores, este poderá conectar-se. Quando este se conecta, irá receber

a dimensão do tabuleiro pelo servidor, bem como o estado atual do tabuleiro (através da função *sendActualBoard*) onde são percorridos todos os elementos do tabuleiro e apenas são enviadas as cartas que estão viradas para cima. Sendo que no fim da transmissão é utilizada uma flag para notificar o cliente que terminou a atualização inicial do tabuleiro.

4.0.3 Escolha de uma carta

Sempre que um cliente seleciona uma carta, esta carta será traduzida em coordenadas (x,y) do tabuleiro, sendo estas depois enviadas através de uma socket definida aquando da entrada deste cliente no jogo, para o servidor, onde esta jogada será processada na thread correspondente. Ou seja, tal como foi referido anteriormente, o cliente apenas irá enviar para o servidor 2 inteiros. É de salientar que todas as jogadas são primeiramente analisadas para ver se são válidas a partir da função *checkPlay*, pertencente ao módulo de lógica do jogo, e se uma não for válida, é descartada.

4.0.4 Atualização do board

Assim que uma jogada for recebida e posteriormente processada pelo módulo de lógica jogo, irá ocorrer a atualização do board a todos os jogadores. Esta é feita pela thread responsável do jogador, que faz um *broadcast* a todos os clientes percorrendo a lista de jogadores com as novas alterações feitas pelo jogador à qual ela é responsável.

4.0.5 Fim do Jogo

O jogo irá terminar quando todas as cartas do tabuleiro forem preenchidas. Uma vez que é conhecido o tamanho do tabuleiro, através do contador de jogadas **score.top_score** conseguimos ir atualizando o número de peças preenchidas no board, sendo que quando este número atinge a dimensão do tabuleiro, sabemos que o jogo terminou.

Quando um jogador terminar o jogo, a thread deste jogador repara em tal informação e atualiza a flag **flag_reset** dar início ao processo de restart do jogo. O servidor notifica os clientes de que o jogo terminou (a partir do membro **code** da estrutura *Play_Response* como acima mencionado) e adicionalmente também é enviada ao(s) jogador(es) vencedor(es) informação da vitória.

5 Validação e Gestão de Erros

5.1 Sincronização

Para evitar erros que pudessem afetar o funcionamento de todo o projeto, começámos por verificar o retorno das system calls relacionadas com a iniciação dos métodos de sincronização (*mutex*, *rw_lock* e *semaphores*), bem como o retorno de todos os locks/unlocks (e post/wait para semáforos).

Foi também tida em atenção a possibilidade de existência de *deadlocks*. De modo a evitar tal efeito, certificámo-nos que todos os locks efetuados teriam sempre um unlock

correspondente. Na figura 9 está retratada um exemplo de uma possível situação de *deadlock*. Estas situações impedem o seguimento e evolução do programa e as suas prevenções são cruciais para o bom funcionamento do software.

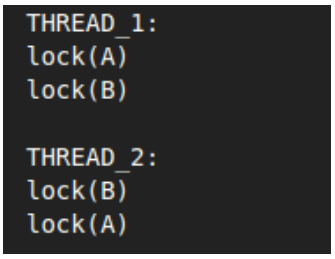

Figura 9: Exemplo de uma possível situação de deadlock

5.2 Comunicação

Utilizamos o retorno da system call *read* para saber quando o cliente/servidor se desconectou de modo a fazermos o respectivo clean up necessário. Ao longo dos jogos, vamos também verificando o envio de dados para saber se a socket ainda está ativa.

No raro caso em que uma thread 1 está a tentar enviar uma atualização de board ao jogador 2, que acabou de se desconectar, correspondente da thread 2 e esta thread 2 ainda não teve tempo de eliminar o nó correspondente da lista de jogadores, a system call *write* irá produzir o sinal *SIGPIPE* (correspondente ao erro de escrita de um PIPE, pois utilizamos sockets stream). Este sinal interromperia o funcionamento do nosso servidor e por isso decidimos simplesmente ignorar o *SIGPIPE*.

5.3 Memória

Não são apenas as system calls relacionadas com a sincronização que têm analisadas o seu valor de retorno. Todas as alocações de memória são também verificadas, tendo para isso sido desenvolvida uma função denominada *verifyErr* que verifica a correta alocação de memória.

5.4 Argumentos de inicialização

5.4.1 Servidor: Dimensão

Quando é iniciado o jogo pelo servidor, é introduzida a dimensão do tabuleiro de jogo pretendido. Essa dimensão terá de ser par (uma vez que uma dimensão ímpar irá corresponder a um número de elementos ímpar no tabuleiro, o que implica que uma peça não tenha par). Esta verificação também é feita juntamente com a dimensão ser superior a 2, e inferior à dimensão máxima permitida, este limite máximo é imposto pela própria natureza das combinações de 2 letras:

$$\sqrt{26^2 \cdot 2} \approx 36.77 \rightarrow 36 \tag{1}$$

Portanto, a dimensão também tem que ser menor ou igual a 36. Caso estes critérios não sejam respeitados, não será iniciado o servidor.

5.4.2 Cliente: IP

O IP do servidor é inserido como argumento quando se executa o cliente. Verifica-se se o número de argumentos é 1 e só 1. Se tal não acontecer, o cliente não é executado.

5.5 Jogadas recebidas pelo servidor

De modo a garantir a validade das jogadas recebidas, o servidor verifica sempre se todas as coordenadas recebidas das jogadas estão dentro dos limites da dimensão do board (menores ou igual à dimensão e as coordenadas têm que ser número positivos).

6 Sincronização

A arquitetura do sistema ter sido desenvolvimento com multi-threading traz muitos benefícios em termos de eficiência e rapidez devido ao paralelismo de tratamento de jogadas, porém levanta alguns problemas a nível de sincronização. Ou seja, ao ter várias threads a partilharem a mesma memória pode haver a possibilidade de diferentes threads tentem aceder e modificar ao mesmo tempo a memória partilhada, criando situações de *race condition*.

Nesta secção iremos aprofundar este tema, identificando as regiões críticas e analisando como o programa lida com esses conflitos de sincronização. É de notar que seria possível reutilizar variáveis de sincronização para diferentes regiões críticas, no entanto, isto iria torná-la maior do que o necessário. Optámos assim por dedicar variáveis de sincronização para cada região crítica de modo a otimizar a sincronização.

Apesar de para a maior parte dos processadores instruções de escrita num inteiro como *inteiro = 10* serem atómicas, para processadores mais fracos esse pode não ser o caso. E como queremos que este programa tenha a maior portabilidade possível, inserimos também sincronização neste tipo de ocasiões.

6.1 Implementação de *rw_lock* na Lista de Jogadores

```
pthread_rwlock_t rwlock_stack_head;
pthread_rwlock_t rwlock_stack;
```

Figura 10: Variáveis de sincronização utilizadas para a lista de jogadores

A lista, que na verdade é um *stack*, está dividia em 2 regiões críticas diferentes, havendo assim 2 *rw_lock* (1 para cada região crítica):

- Cabeça da pilha;
- Resto da pilha (pilha toda excepto a cabeça).

Existe assim 1 *rw_lock* exclusivo para a head e outro para o resto do *stack*, pois como a main thread adiciona nós somente pela cabeça e como a inserção só interage com 1 único nó (a head), não há necessidade de bloquear a lista inteira no momento da inserção.

Como a lista é percorrida muito frequentemente (sempre que uma jogada é processada e enviada a todos os clientes), há assim grande necessidade de leitura desta estrutura de dados e, em contra partida, como as escritas na lista serão "raras"(apenas nas conexões e desconexões) o método de sincronização escolhido para esta estrutura de dados foi *rw_lock* (**rwlock_stack** e **rwlock_stack_head**).

É de salientar que existem outras maneiras de implementar a sincronização na remoção de nós da lista que seriam mais adequadas, dependendo do ambiente em que o programa iria ser executado (dependendo do número de jogadores, se havia frequentemente muitos jogadores não ativos). Decidimos conjeturar que o servidor tem um número médio/alto de jogadores e que as conexões e disconexões não são frequentes. No entanto, nas seguintes subsecções iremos explorar outras alternativas mais aptas para outras cargas de utilizadores.

O contador **n_players** indica em tempo real o número de jogadores ativo. Visto que esta é sempre atualizada quando se elimina um nó da lista de jogadores, decidiu-se inserir esta variável global dentro desta região crítica.

6.1.1 Alternativas

6.1.1.1 Não remover elementos

Se o número de jogadores for relativamente pequeno, não há grande necessidade de remoção de elementos (e o armazenamento da informação dos jogadores até pode ser conseguido através de um vetor sem grande custo adicional de memória). Não havendo remoção de elementos, deixa de haver necessidade de haver região crítica nesta estrutura de dados e assim são poupadas idas ao *kernel*, apesar de continuar a ser sempre preciso sincronização na head. No entanto, como a lista será continuamente grande, também o *sweep* da lista no momento do *broadcast* a todos os jogadores será continuamente mais longo, tornando este varrimento ineficiente.

Uma outra grande desvantagem desta estratégia é o facto de não oferecer 100% longevidade do serviço aos clientes, visto que possuir uma lista que aumenta perpetuamente irá, eventualmente, esgotar a memória.

6.1.1.2 Uma lock exclusiva para cada nó da lista

Se existirem muitos jogadores e grande frequência de conexões/desconexões, tanto a escrita como a leitura da lista serão bastante frequentes logo, uma sincronização eficiente para este cenário, seria dedicar um *mutex* para cada nó de modo a que quando houvesse uma escrita na lista, esta não bloquearia a lista inteira. Para conseguir isto, no ato de remoção de um nó, a thread responsável iria bloquear 3 *mutex*, o do nó a remover e ainda os mutex dos nós adjacentes, de modo a garantir que nenhuma outra thread pudesse ler e/ou escrever durante o período de remoção.

6.2 Implementação de exclusão mútua para cada elemento do *board*

A sincronização do board, sendo esta a estrutura de dados mais acedida no servidor, tem que ser bastante eficiente. Por isso, decidimos dedicar um *mutex* a cada elemento do board, de modo a ser possível várias threads poderem manipular o board ao mesmo tempo. Garantindo assim que a região crítica seja o mais baixo nível possível.

6.3 *Mutex* para contador de cores

O contador de cores é composto pela variável **prox_RGB** e pelo vetor **last_color**. Este contador serve para o algoritmo de geração de cores saber qual a última cor gerada. Como esta variável é manipulada por várias threads, quando existem novas conexões existe a situação de *race condition* quando 2 jogadores se conectam ao mesmo tempo.

Selecionou-se o método de exclusão mútua para esta região crítica pelo facto de esta ser 4 caracteres, não havendo nenhuma exigência especial em termos de sincronização.

6.4 Implementação de exclusão mútua para a flag *reset_flag*

Apesar de apenas uma thread poder modificar esta flag (somente 1 thread pode receber a última jogada devido à exclusão mútua de cada elemento do *board*) pode acontecer estar outra thread a ler ao mesmo tempo que está a existir esta modificação. Assim, temos uma situação de *race condition* que tem de ser resolvida.

Decidimos então utilizar um *mutex* também pelo facto de esta flag ser apenas um char e raramente ser lida ao mesmo tempo.

6.5 Implementação de *rw_lock* na estrutura global *score*

Esta estrutura global tem 2 papeis:

- Ao longo de um jogo, o número de jogadas globais é guardado em **score.top_score** (ver figura 4a para relembrar os membros desta estrutura **score**);
- No final do jogo, no processo de eleição dos jogadores vencedores, é guardado o top score ao longo da eleição também no membro **score.top_score** e é adicionado os jogadores vencedores à lista cuja address da head está guardada no ponteiro **score.head**.

Assim, para ambos os papeis desta estrutura global, é necessária a resolução de potenciais *race conditions*. E dado que durante o processo de eleição o **score.top_score** é lido bastantes vezes, decidiu-se utilizar read & write lock para permitir vários readers ao mesmo tempo.

6.6 Implementação de *rw_lock* na flag *end_flag*

O valor desta variável somente é modificado quando o processo do servidor recebe um *SIGINT*. Dada a raridade da escrita e a alta frequência de leitura, decidiu-se usar novamente *rw_lock*.

6.7 Sincronização entre as duas threads de cada cliente

Como já mencionado anteriormente, existem 2 threads dedicadas a cada cliente que partilham uma estrutura **common_data** (relembrar membros desta estrutura na figura 5).

Apesar de existir um mecanismo de ativação de threads utilizando *semaphores*, este não garante a resolução de *race_conditions* no acesso/escrita na estrutura **common_data** pois há casos em que as 2 threads estão simultaneamente ativas, por isso é necessário sincronização de exclusão mútua para proteger esta região crítica utilizando-se um *mutex* para alcançar tal requisito.

7 Funcionalidades

7.1 Determinação do número máximo de jogadores permitido

Sempre que um novo jogador se conecta no jogo, as jogadas deste terão de ser distinguidas das jogadas de todos os outros jogadores. Para tal, realizámos um algoritmo de geração de cores que permite realizar tal distinção.

Para escolhermos a cor, fizemos de uma maneira já predefinida para não ter a necessidade de criar uma cor random e depois ir verificar se essa cor já foi escolhida. Este algoritmo permite gerar 67 cores diferentes sem interferirem com as cores de controlo do jogo (cinzento, vermelho, branco e preto), sendo assim, apenas podem existir, no máximo, 67 jogadores em jogo.

7.2 Conexão de um novo jogador

Quando um novo jogador se conecta, caso o número de jogadores atuais no jogo seja menor do que o número máximo de jogadores permitidos (67), a conexão será válida. O número de jogadores em jogo é guardado sempre numa variável global denominada como n_players que é incrementada quando um novo jogador se conecta e decrementada quando um jogador se desconecta.

Caso este novo jogador não seja permitido no jogo, a função close é chamada, terminando a conexão deste jogador, impedindo assim este jogador de jogar.

Caso a conexão do jogador seja permitida, é inicializado todo o processo de criação de um jogador, isto é:

- Criação da thread principal que faz gestão do jogador;
- Jogador é adicionado à lista de jogadores;

- Geração de uma cor para este jogador;
- Envio da dimensão do board ao cliente;
- Criação do tabuleiro de jogo;
- Envio do estado atual do tabuleiro;
- Criação da thread responsável pelos timers.

7.3 Número mínimo de jogadores

Uma das funcionalidades a implementar diz respeito ao número mínimo de jogadores necessários para haver jogo. Tendo sido estipulado que este número teria de ser 2, se apenas um jogador estiver conectado, as suas jogadas terão de ser desprezadas.

Para controlar o número de jogadores presentes em jogo, começámos por utilizar uma variável global com tal informação. Quando o número de jogadores for igual a 1, o cliente irá continuar a enviar as coordenadas do tabuleiro selecionadas para o servidor.

A thread deste jogador no servidor, terá um grande ciclo que irá receber as informações enviadas pelo cliente. Quando uma coordenada for recebida, a primeira coisa a verificar é se essa jogada é válida, ou seja, se as coordenadas estão dentro do board, e se o número de jogadores é superior a 1. Tais verificações são efetuadas através de uma função denominada como *checkPlay* onde é analisado o seu retorno (0 em sucesso e 1 em insucesso).

Caso o número de jogadores seja 1, *checkPlay* irá retornar 1, fazendo assim *continue* e obrigando a thread deste jogador a aguardar uma nova jogada, uma vez que a recebida anteriormente não poderá ser processada, pois não cumpre as regras do jogo, ignorando assim a jogada previamente recebida (Figura 11).

```
//Verifica se a jogada é válida
if(checkPlay(play_recv[0], play_recv[1])){
    continue;
}
```

Figura 11: Função cujo retorno é determinante para não ser possível jogar com 1 jogador

O processo repete-se até que o número de jogadores seja superior a 1, o que, quando tal acontecer, fará com que a função *checkPlay* retorne 0, validando assim a jogada como válida e podendo assim esta ser processada. Esta verificação é utilizada tanto no início do jogo como durante o jogo.

7.4 Distinção entre primeira escolha e segunda escolha

No jogo da memória, cada jogador efetua 2 combinações. Uma vez que possuímos uma thread para cada jogador que efetua todo o processamento de jogadas (com o auxílio do

módulo de lógica de jogo), definimos uma variável local a cada thread (portanto guardada no *stack* de cada thread), denominada como n_play. Com esta variável a 1, sabemos que terminámos de processar a primeira jogada, entrando agora na segunda, podendo ativar o timer de 5 segundos, e enviando o novo estado do board a todos os outros jogadores, com a nova carta virada para cima.

```
if(n_play == 1){        //Se estivermos na primeira jogada
```

Figura 12: Detetar que estamos no fim do processo da primeira jogada

Com o n_play = 1, podemos atualizá-lo para 0 (ao jogador saber que está de novo na primeira jogada) de duas formas formas distintas:

- **Fim do timer de 5s**: O timer de 5 segundos chega ao fim, não tendo o jogador selecionado uma segunda carta, sendo a primeira carta virada para baixo, o novo estado do board enviado a todos os jogadores e o n_play restituído para 0;
- **Fim do processamento da segunda jogada**: A segunda jogada é processada corretamente na função *boardPlay* sendo que no fim a variável n_play é atualizada para 0.

7.5 Timer de 2 segundos

Uma das regras do jogo é existir um período de bloqueio de 2s em que o jogador não pode fazer jogadas, quando este errou uma combinação.

Uma maneira de fazer isto, seria com um *sleep(2)*. No entanto, as jogadas do jogador que errou a combinação seriam guardadas durante este período e após os 2s acabarem estas jogadas guardadas seriam imediatamente processadas, ou seja, não iria ocorrer um real desprezo das jogadas durante o período de proibição de jogadas.

Para resolver isso, decidiu-se dedicar uma thread adicional a cada jogador de modo a auxiliar este timer, havendo assim um paralelismo que permite as jogadas recebidas durante o 2s serem totalmente descartadas.

Temos um resp.code específico para o caso da combinação ser errada (resp.case = -2) desbloqueando a thread dos timers através do uso de um semáforo (Figura 13). Apesar de também ser possível o uso de *conditional variables*.

```
}else if(common_data.resp.code == -2){          //Se o jogador errou na combinação, ativar timer 2s
    semaphore(POST, &common_data.sem);          //Ativar thread de timer
    continue;
}
```

Figura 13: Desbloqueio da thread dos timers para ativar os 2 segundos

A thread que trata da gestão de todo o jogador, com o **resp.code** = -2 irá descartar as jogadas recebidas, não fazendo o processamento da jogada (Figura 14).

```
if(common_data.resp.code != -2 && common_data.resp.code != 4)
```

Figura 14: Não processar as coordenadas recebidas

Depois da thread dos timers estar desbloqueada, é analisado qual o resp.code que a thread possui. No caso de ser resp.code = -2, sabemos que é necessário ativar o timer de 2s, entrando no if que faz tal, fazendo sleep(2). No fim dos dois segundos, as duas cartas são viradas para baixo e é enviado a todos os jogadores o novo estado do tabuleiro.

Depois de todo este processo ter terminado, é atualizado o resp.code para 0, de modo à thread de gestão do jogador saber que já pode processar jogadas.

É de salientar que, como já mencionado anteriormente, existem mecanismos de sincronização para a comunicação entre estas duas threads.

7.6 Timer de 5 segundos

```
if(n_play == 1){                                    //Se tivermos na primeira jogada
    if(!poll(&poll_sock_fd, 1, 5000)){              //Ativar o timer 5s
```

Figura 15: Implementação do *poll*

O timer de 5s tem de ser começado assim que a primeira jogada é feita, de modo a que, se não houver receção da segunda jogada nesse período, a carta selecionada tem que ser voltada para baixo.

Esta funcionalidade é implementada através da função *poll*. Esta função permite ter um *timeout* que "acorda"a thread se não houver atividade até ao *timeout* especificado. Portanto se forem atingidos estes 5 segundos sem ser recebido nada na socket que faz a conexão cliente-servidor, a carta correspondente à primeira jogada será voltada para baixo atualizando assim o board para todos os jogadores e redefinindo a flag n_play = 0 de modo a sabermos que a próxima coordenada recebida diz respeito à primeira jogada.

7.7 Bot

Para a implementação do Bot servimos o protocolo de comunicação utilizado no caso de possuirmos um jogador real. No entanto, em vez de existirem eventos SDL do rato de modo a detetar a coordenada (x,y) do board, esta coordenada foi gerada através da função rand() e enviada de seguida para o servidor para ser processada (Figura 16).

```c
while(1){
    play[0] = rand()%dim;
    sleep(1);
    play[1] = rand()%dim;
    if(enviar(sock_fd, play) < 0){
        break;
    }
}
```

Figura 16: Envio de coordenadas random para o servidor

De modo análogo ao anterior, o bot irá receber do servidor a estrutura *Play_Response* depois da jogada enviada ter sido processada identificando assim o resultado deste processamento. Esta estrutura recebida será então interpretada pelo módulo de lógica de jogador que imprimirá no terminal o significado da resposta recebida.

7.8 Saída de um jogador do jogo

```c
//Cliente disconectou-se ou o servidor está a terminar
//Destruição da thread timer
mutex(LOCK, &common_data.mutex_timer);  //Ter a certeza que o thread de timer não fica com nenhuma lock antes de morrer -> esperar que ela faça tudo o que tem a fazer
common_data.n_corrects = CANCEL_TIMER_THREAD;
mutex(UNLOCK, &common_data.mutex_timer);        //Não é preciso pois a lock vai ser agora destruída, é meramente uma formalidade
semaphore(POST, &common_data.sem);                          //Ativar a thread timer para ela se auto destruir
pthread_join(thread_id, NULL);

rwLock(R_LOCK, &rwlock_end);
if(!end_flag){                                  //Só vale a pena fazer estas operações se o servidor não estiver a terminar
    rwLock(UNLOCK, &rwlock_end);
    deleteNode(client_data);                    //Apaga nó correspondente ao player da lista de players
    //Se o jogador tiver saido durante o timer de 5s, temos de virar a carta para baixo
    if(n_play == 1){
        mutex(LOCK, &board[common_data.resp.play1[0]][common_data.resp.play1[1]].mutex_board);
        fillCard(common_data.resp, 0, common_data.resp.play1[0], common_data.resp.play1[1]);
        mutex(UNLOCK, &board[common_data.resp.play1[0]][common_data.resp.play1[1]].mutex_board);
        common_data.resp.code = -1;
        broadcastBoard(common_data.resp, common_data.buff_send);              //Mandar alterações do board a todos os jogadores
    }
}else{
    rwLock(UNLOCK, &rwlock_end);
}

//Libertar recursos
sem_destroy(&common_data.sem);
pthread_mutex_destroy(&common_data.mutex_timer);
free(common_data.buff_send);
printf("Client disconnected\n");
return NULL;
}
```

Figura 17: *Clean up* de um cliente

Há duas situações que acionam o *clean up* de um jogador:

- Quando o cliente se desconecta;
- Quando o servidor está a terminar.

Quando um cliente é desconectado, a *system call read* deteta este acontecimento e por sua vez quebra o loop de processamento de jogadas. Depois é inicializada o processo de desconexão do cliente que consiste em:

- Acordar a thread de auxílio para timers para esta se auto-destruir e de seguida fazer *pthread_join* para libertar por completo os recursos dessa thread;
- Apagar o nó correspondente da lista de jogadores, ao fazer *wr_lock* na lista;
- Se o jogador estivesse para fazer a segunda jogada, voltar a primeira jogada para baixo;
- Libertar recursos: Destruir *mutex e semaphore* e mandar free no **buff_send**.

Quando o servidor estiver a terminar, o processo é idêntico com exceção da remoção do nó da lista de jogadores, pois esta lista é totalmente apagada pela main thread.

7.9 Fim do jogo

```
if(common_data.resp.code == 3){
    reset_flag = 1;
    mutex(UNLOCK, &mutex_reset);
    semaphore(POST, &common_data.sem);
```

Figura 18: Ativar semáforo para a thread dos timers desbloquear

Tal como referido anteriormente, uma vez que sabemos a dimensão do tabuleiro, conseguimos saber quantas peças este contém. Através do contador de jogadas **score.top_score** consegue-se saber se acabou o jogo comparando este valor com a dimensão do tabuleiro.

Todos os jogadores possuem 1 thread responsável por processar as jogadas. Quando um dado jogador efetua a ultima combinação possível no tabuleiro, o número de jogadas é atualizado, possuindo assim este jogador algo de especial em relação a todos os outros jogadores: a estrutura *Play_Response* tem o **resp.code** = 3. Dado isto, a thread de processamento de jogadas deste jogador irá ativar a flag reset_flag a 1, e ainda vai ativar a sua thread de timers.

A partir de agora, esta thread de timers ativada será a reset master, que tratará de fazer o reset do jogo. Primeiramente esta thread irá acordar todas as outras threads timers para notificá-las do fim do jogo (com a **flag_reset**). De seguida, todas as estas threads timers tornar-se-ão slaves.

7.9.1 Transmissão do vencedor

O primeiro passo de reiniciar o jogo é a eleição dos vencedores. O contador de jogadas, **score.top_score** é posto a 0 e passa agora a ser uma variável de referência para se saber a melhor pontuação do jogo. Todas as threads timer vão agora invocar a função *tryUpdateScore* e irão eleger os vencedores do jogo, ficando a reset master à espera que todos os slaves acabem de eleger o jogador vencedor, caso exista um empate, os jogadores vencedores são inseridos numa lista cuja cabeça é guardada em *score.head*. Quando o(s) jogador(es) vencedor(es) é eleito, a thread master é desbloqueada e todas as threads timers indicam os seus clientes de que o jogo acabou, adicionalmente, a reset master irá notificar os jogadores vencedores de que eles ganharam.

Nota: O processo de acordar uma thread acordar outra(s) é feito através de *semaphores*. No entanto, como já foi explicado anteriormente, este mecanismo também podia ser alcançado utilizando *conditional variables*.

7.9.2 10 segundos de intervalo entre jogos

Depois de todas as threads serem notificadas do fim do jogo, a thread master será responsável por ativar o timer de 10 segundos, Para tal, começa por redefinir o resp.code = 4, de modo a que a thread responsável pelo processamento de jogados ignore os pedidos vindos do cliente, dormindo depois durante 10 segundos. Também os restantes jogadores, com o resp.code = 4 irão ignorar a informação recebida do cliente, não as processando.

7.9.3 Restart do jogo no cliente

```
case 4:     //Começar novo jogo, reset do board
    resetBoard();
    break;
```

Figura 19: Reset do board no cliente

A thread reset master, tal como já vimos anteriormente desempenha um papel muito importante no reinício do jogo. Depois do timer dos 10 segundos ter terminado, esta thread irá ficar responsável por preencher de novo um tabuleiro com as posições das strings no tabuleiro diferentes das posições anteriores, de modo a gerar um novo jogo. Serão então notificados todos os clientes para limparem os seus tabuleiros ao fazer um *broadcastBoard* de um **resp** com um **resp.code** = 4, que faz com que os clientes limpem os seus boards (figura 19.

7.9.4 Restart do jogo no servidor

De seguida, a thread master irá acordar todas as outras thread slaves avisando-as de que o timer de 10s terminou. Agora todas as threads fazem reset de variáveis de lógica de jogo como o contador de pontos **n_corrects**, a reset master adicionalmente mete a *reset_flag* a 0 e o jogo é recomeçado.

8 Conclusão

Todo o projeto foi efetuado tendo em conta a eficiência do código a realizar, bem como quais os métodos de sincronização mais adequados para o objetivo desejado. Implementámos vários tipos de mecanismo de sincronização: *rw_lock*, *mutex* e utilizámos *semaphores* como mecanismo de "ativação"de threads, mostrando assim a diversidade em todo o projeto.

De modo a diminuir o tamanho das regiões críticas, optámos por dividir estas de modo a conseguir maximizar a eficiência da sincronização *multi thread*.

Conseguimos concretizar todos os tópicos exigidos pelo enunciado e achámos que a maior dificuldade foi corrigir bugs muito específicos associados às funcionalidades. Estes foram encontrados à medida que fomos testando todos os módulos independentemente e quando efectuámos testes de integração incrementais, ou seja, quando integrámos módulos juntando-os 1 a 1 ao sistema num geral.

Com o objetivo de poupar CPU, em todo o projeto não existe nenhum caso de espera ativa, apesar de existirem locais que compensaria a presença de espera ativa (*spin locks*), por exemplo quando uma thread quer bloquear a lista de jogadores para escrita. Neste caso a lista de jogador é lida muitas vezes e por isso, uma thread que queira adquirir a *wr_lock* provavelmente iria ter que esperar um pouco, mas com espera ativa poderíamos acelerar esse bloqueio de escrita na lista. No entanto, o pior que pode acontecer é uma thread tentar escrever numa socket já não existente, o que não faria mal visto que já se ignora o *SIGPIPE*. Assim, decidimos não implementar nem neste caso espera ativa.