TÉCNICO LISBOA

Concentration (jogo de cartas) Multi-jogador Distribuído

Programação de Sistemas, 2º Semestre 2018/2019

Engenharia Eletrotécnica e de Computadores, Instituto Superior Técnico

José Silva, 84109
Miguel D'Ajuda, 84144

5 de Junho, 2019

Resumo

Neste relatório é realizada um análise detalhada do programa realizado em C que implementa um jogo de memória, multi-jogador distribuído. Este jogo implementa conceitos fundamentais da cadeira de Programação de Sistemas, desde sincronização entre clientes e servidor até ao uso de *threads*.

Serão explicados todos os tópicos referidos na sugestão de relatório, incluindo também criticas ás maiores decisões tomadas ao longo do projeto. O funcionamento individual de cada função está descriminado nos comentários do código realizado.

Conteúdo

1 Arquitetura

A arquitetura do programa encontra-se na Figura 1.

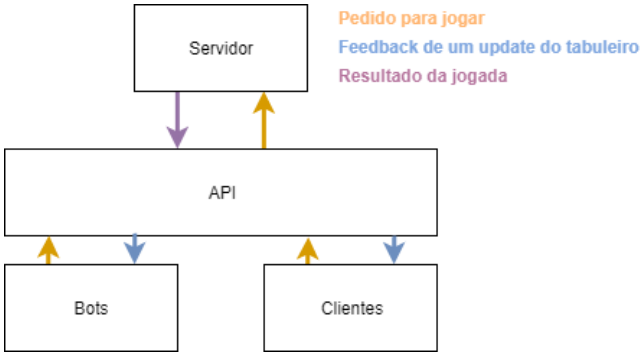

Figura 1: Arquitetura do sistema

Como é evidente pela Figura 1, foi adotada uma arquitetura de comunicação do género Cliente-Servidor, com a adição da API de comunicação fazendo assim uma mistura com a arquitetura N-Tier.

A estrutura da solução apresenta-se, com um grau significativo de abstração na Figura 2.

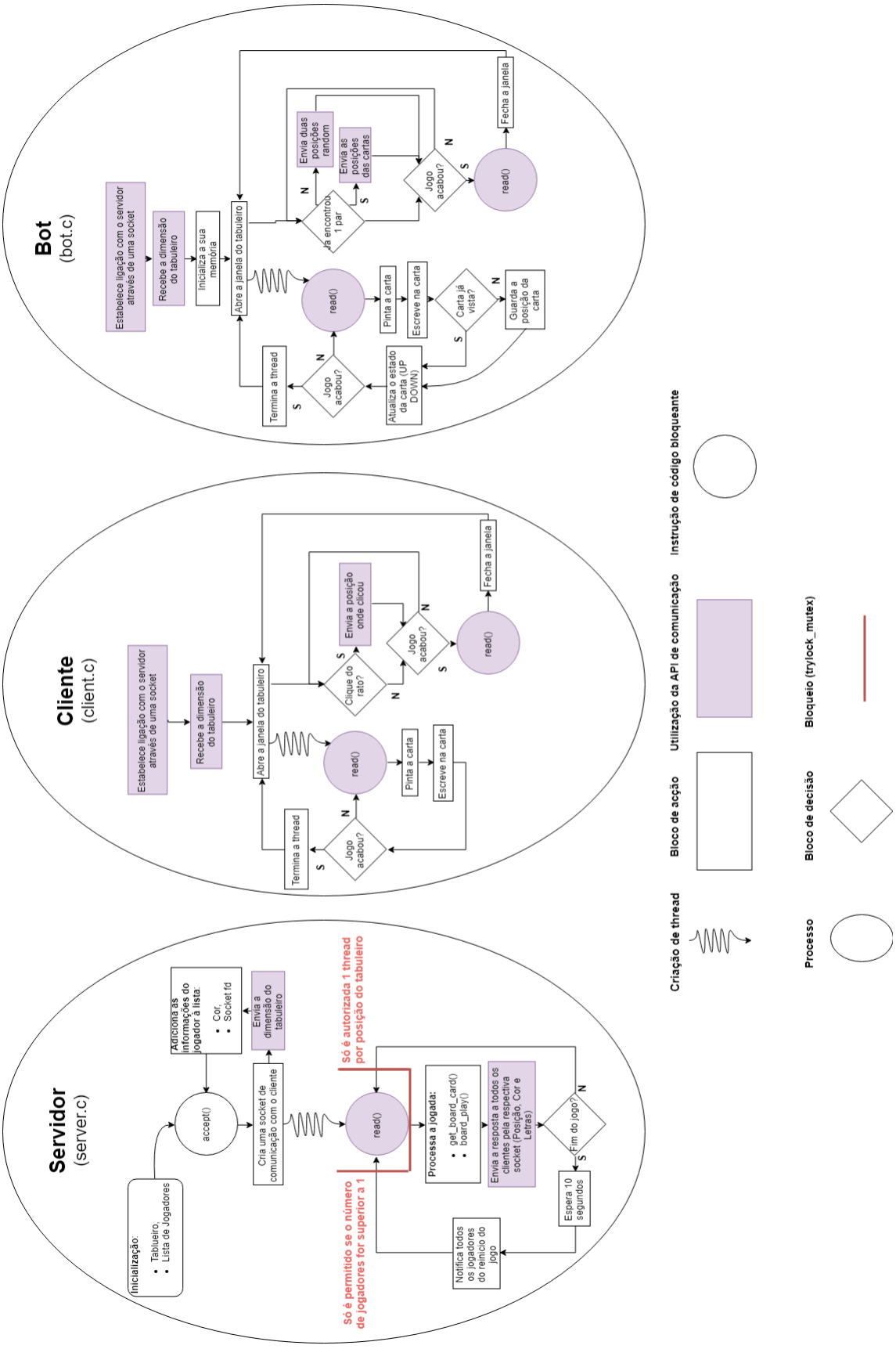

Figura 2: Esquema da solução.

2 Organização do código

O código consiste em sete ficheiros .c, seis .h e um ficheiro *Makefile* que compila este conjunto criando os executáveis.

Começando pelos ficheiros fornecidos pelo corpo docente:

- **board_library.c:**

  Neste ficheiro foram realizadas algumas alterações nas funções init_board(1) e board_play(4) que permitiram guardar mais informações na estrutura do tabuleiro e lidar com alguns tipos de jogadas que não estavam definidos, por exemplo, carregar na própria carta. Deste modo, a estrutura retornada pela segunda função referida, embora esteja igual, pode ter mais valores com diferentes significados no retorno (explicado em maior detalhe da secção seguinte).

  Para além disso, foram acrescentadas duas funções, free_board(0) e get_piece(1), que libertam a memória do vetor do tabuleiro e retornam a estrutura com uma posição especificada no tabuleiro, respetivamente. Estas funções serão chamadas dentro do ficheiro do servidor.

- **board_library.h:**

  Este ficheiro permaneceu igual, com a exceção das mudanças necessárias para que as alterações efetuadas no .c correspondente a este *header* fossem possíveis. Deste modo, foi também incluída uma nova biblioteca que permite definições do tipo *color*, criado por nós.

- **UI_library.c:**

  Neste ficheiro foram acrescentados pequenos atrasos de 10 milisegundos por cada render (três no total). Devido a alguns erros de falha de memória na biblioteca SDL, foram também criados *mutexes* que só permitem a escrita de uma carta de cada vez.

- **UI_library.h:**

  Não foi realizada qualquer alteração neste ficheiro.

Do ficheiro *memory-single.c* fornecido, foi reutilizado grande parte do código na construção das funções que se seguem. Assim, os próximos ficheiros são relativos aos servidor.

- **sock_dg.h:**

  À luz de um dos trabalhos de laboratório, foi criado este simples ficheiro que define a localização, na máquina, da *socket* e qual o número da porta a ser usada nas comunicações entre o servidor e cliente/*bot*.

- **our_list.c:**

  Este ficheiro serve de auxilio ao servidor e neste estão contidas todas as funções referentes à manipulação da lista de clientes (do servidor). Os nomes de cada função tornam explicito o que fazem. Os comentários no código descrevem isto mesmo e também explicitam os seus argumentos e retornos.

- **our_list.h:**

  Neste módulo *header* estão os declarados os cabeçalhos das funções definidas em *our_list.c*. Este também inclui a definição da estrutura dos jogadores usada no servidor.

- **server.c:**

  Este é o maior módulo do projeto e inclui seis variáveis globais que são responsáveis pelos *metexes*, pela lista e quantidade de jogadores, pelo tamanho do tabuleiro (recebido como argumento do programa) e pelo estado de fim do jogo. Este ficheiro contém duas funções ("client_handler"e "main") que constituem parte da camada superior do código, sendo responsáveis por gerir grande parte das funções dos módulos descritos anteriormente.

  Uma descrição individual de cada função encontra-se por tópicos abaixo.

  – color CorJogada(int fd):

    O único argumento desta função foi usado somente para depurar os resultados da mesma. A sua funcionalidade é gerar uma nova cor aleatória para cada novo jogador assim que este entra na sua *thread*.

    De modo a não existirem cores de cartas que tornam as letras incompreensíveis, criaram-se três condições que filtram estes casos e tentam uma cor nova. Assim, as cores criticas seriam o cinzento, o vermelho e o preto, sendo que a nova cor teria de estar a uma distância (calculada pela norma Euclidiana) maior que 75 unidades rgb de cada uma desdas. Esta lógica pode ser melhor interpretada pensando num cubo com 255 de lado (red, green e blue) onde a nova cor tem de estar a um raio superior a 75 de cada um dos três pontos correspondentes ás três cores criticas, tal como representado na figura 3.

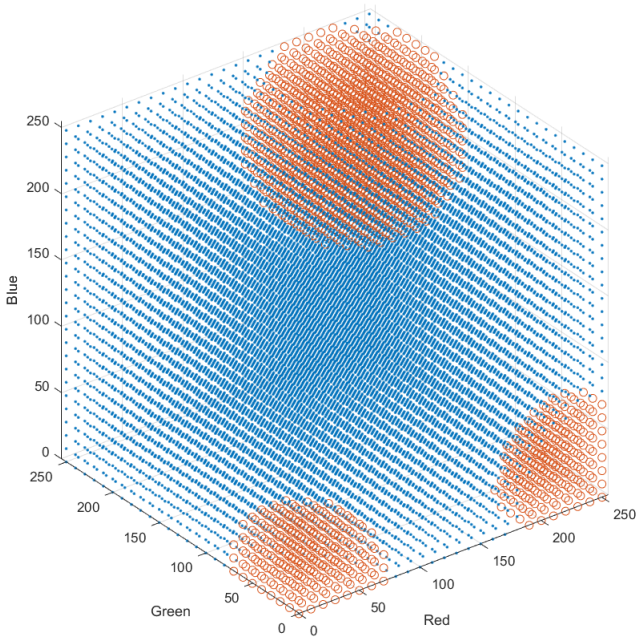

Figura 3: Zonas a vermelho - interditas; zonas a azul - disponíveis

  – void jogada(int play[2], char str_play[3], color c_letra, color c_carta):

    Esta função preenche a estrutura com as informações gráficas da jogada e percorre a lista dos jogadores enviando a jogada para cada um. O desenho no servidor também é realizado nesta função.

  – void timeout(struct pollfd *fds, int x, int y, int play1[2], int *locked, color cor_xpto):

    Esta função é responsável por fazer o *timeout* caso o jogador demore mais do que 5 segundos a fazer a segunda jogada. Para implementar esta funcionalidade usou-se a função *poll(3)* e um *mutex* que desbloqueia a carta depois da espera.

- void *garbage_collector(void *sock_fd):

  Esta função faz uma leitura não bloqueante das jogadas dos clientes e é chamada após a segunda jogada, caso esta esteja errada. Assim, todo o lixo que iria encher o *buffer* das leituras do servidor durante os dois segundos de espera é absorvido aqui e ignorado. Note-se que esta função é uma *thread* separada que é destruída após os dois segundos de espera.

- void *wait10(void *sock_fd):

  Aplicando a mesma lógica que na função anterior, criou-se uma *thread* que absorve os possíveis *clicks* dos clientes durante os 10 segundos de espera até ao inicio do próximo jogo. Para além disso, caso seja detetada uma mensagem inválida (com erro), é assumido que o jogador desconectou e é atualizada a informação da lista dos jogadores. Esta *thread* é destruída no final dos 10. segundos.

- void novo_cliente(int fd):

  Aqui é feita a lógica da parte gráfica de acrescentar um novo jogador. Esta função foi construida para os casos em que um jogador entra a meio do jogo.

- void *client_handler(void *sock_fd):

  Esta função rege o decorrer do jogo, do ponto de vista de um cliente/*bot*, no que toca ao servidor. Assim, esta é lançada numa *thread* quando a ligação a um novo jogador é aceite.

  De modo a lidar com a espera de 10 segundos entre jogos, foi criada uma máquina de estados controlada pela variável global "done", que avisa o final do jogo. Caso "done=1": é mostrada a pontuação final (chamando a função "leaderboard"); lançada a *thread* da função "wait10"; feita a preparação do tabuleiro para o próximo jogo; as pontuações são postas a zero; é verificado se o jogador que fez a ultima jogada saiu (jogador da *thread* em causa - os outros são verificados no read em que estão bloqueados); a variável "done"para a ser zero novamente e o próximo jogo recomeça.

  Para "done=0": é recebida a jogada e verificado se o jogador desconectou, eliminando-o da lista nesse caso; uma vez obtida a posição das coordenadas da jogada, é verificado se existem jogadores suficientes para continuar e se a carta está bloqueada por algum *mutex*, caso contrário e se a jogada for na própria carta, o *mutex* é desbloqueado e a jogada processada; sendo validada a jogada, esta é processada pela função "board_play"; consoante o retorno dessa função, foi criado um *switch case* baseado no ficheiro memory-single.c que realiza cada hipótese e os *mutexes* correspondentes.

  Grande parte da lógica do sincronismo está nesta função mas será apenas explicado em detalhe na secção 6.

- void check_args(int argc, char *argv[]):

  Esta função valida os argumentos de entrada e inicializa a variável global com o tamanho do tabuleiro. Os teste de validação serão descritos na secção r5.

- int main(int argc, char *argv[]):

  Inicialmente, é chamada a função "check_args". De seguida, a memória para os *mutexes* do tabuleiro é alocada e estes são inicializados. É também inicializada a

parte gráfica da biblioteca SDL. De seguida, é criado o *socket* de comunicação com os clientes/*bots* através das funções "socket", "bind"e "listen".

Por fim, dentro de um ciclo, é feita a aceitação de novos clientes através da função "accept"e a inserção dos mesmo na lista de clientes. É ainda lançada a **thread** do novo cliente (função "client_handler") e enviado tamanho do tabuleiro para o mesmo.

- **messages.c:**

  Este ficheiro contém as quatro funções comuns ao cliente e ao *bot*, sendo estas responsáveis pela comunicação. A descrição destas funções encontra-se em comentário por cima da sua declaração.

  Note-se que, embora semelhantes, não se incluíram as funções responsáveis por validar os argumentos de entrada do cliente e do *bot*, pois estas são propicias a serem alteradas no futuro. Por exemplo, se se quiser acrescentar um argumento ao*bot* que define a dificuldade do mesmo.

- **messages.h:**

  Neste módulo, pode-se encontrar o cabeçalho das funções definidas em messages.c, bem como a definição da estrutura "message"que permite a comunicação no sentido de envio do servidor. É também feito o *#define* da constante "TIMEOUT"que designa o limite de tempo em segundos da duração de uma jogada.

  De seguida, é também definida a estrutura "color"que contém os valores rgb de uma determinada cor. Assim, as cores são organizadas mais claramente. Por fim, são criadas constantes estáticas que definem 4 cores base, usadas ao longo do decorrer do programa com regularidade. Para prevenir a redefinição destas constantes, foi necessário usar diretivas de compilação (com "#ifndef"e "#endif").

- **client.c:**

  Este modulo consistem em três funções que são executadas em duas *threads* diferentes. Assim, a função "check_args"valida os argumentos do programa e é chamada no "main", seguida pelas inicializações da biblioteca SDL, criação da *socket* e leitura do tamanha do tabuleiro enviado pelo servidor.

  De forma semelhante ao modulo server.c, foi construida uma máquina de estados na função "main"que depende duma variável global "done". Caso esta seja igual a 1, a parte gráfica é reiniciada e começa de novo a enviar mensagens para o servidor. Caso seja igual a 0, é criada uma *thread* para leitura da *socket*. De seguida, é lida a entrada do rato e enviada para o servidor, ciclicamente.

  A função "receiver"é chamada como a nova thread no "main"e fica constante a ler a informação enviada pelo servidor e a desenhar as cartas consoante esses dados. Quando o jogo acaba, esta *thread* é morta.

- **bot.c:**

  Este ficheiro tem por base o client.c, sendo constituído pelas mesmas três funções. No entanto, aqui são implementadas funcionalidades que guardam as cartas já mostradas (*bot* com memória), jogando com base nessa informação.

  O funcionamento do *bot* é explicado na secção 7.7, mas adianta-se a informação que o código implementado está preparado para funcionar com dificuldades diferentes definidas pela variável "diff", compreendidas entre 0 e 10, definindo o tempo entre jogadas do mesmo.

- **bot.h:**

  Este módulo inclui apenas a declaração da estrutura "bot_memuit"que será usada para guardar as informações do tabuleiro num vetor criado no ficheiro anterior.

Como se pode observar, sempre que possível implementou-se uma *interface* através dos ficheiros com extensão .h que permitiu a melhor integração de todo o código.

Por fim, também foi criado um ficheiro Makefile que cria os executáveis do servidor, do cliente e do *bot*. Caso se deseje, foram acrescentados comandos neste ficheiro que correm o programa com o *valgrind*, especificamente com as *flags* da *threads* (helgrind).

3 Estruturas de Dados

Nos módulos server.c e message.c foi usada uma estrutura já definida na biblioteca <netinet/in.h> do tipo sockaddr_in, onde se guardaram os dados referentes ao estabelecimento de uma conceção entre servidor e client/*bot*. Esta estrutura foi usada devido ao tipo dos argumentos das funções "connect"do lado do cliente/*bot* e "bind"e "accept"do lado do servidor.

Foi usada outra estrutura já definida numa biblioteca do C, pollfd da <sys/poll.h>, onde foi guardada a informação necessária para a chamada da função "poll", responsável pela espera de 5 segundos após a primeira jogada.

Quanto ás duas estruturas fornecidas pelo corpo docente, foram realizadas algumas alterações, como se pode ver abaixo:

- **board_place:** Acrescentaram-se dois campo a esta estrutura, de modo a satisfazer as novas regras para multi-jogador. No ficheiro board_library.c é usado um vetor com esta estrutura, onde está guardado o estado do tabuleiro.
  - char v[3]: continua a guardar as duas letras duma carta
  - color owner: identifica o dono da carta pelo código das cores atribuido ao mesmo
  - int available: identifica se a carta está disponível
- **play_response:** Esta estrutura manteve-se igual, mas os valores tomados pelo campo "code"foram desenvolvidos. É usada na passagem do resultado duma jogada no servidor.
  - int code: 0 - carta preenchida; 1 - primeira jogada; 2 - segunda jogada certa; -2 - segunda jogada errada; -1 - segunda jogada na mesma carta; 3 - final do jogo
  - int play1[2]: posição da primeira jogada
  - int play2[2]: posiçao da segunda jogada
  - char str_play1[3]: letras da primeira jogada
  - char str_play2[3]: letras da segunda jogada

Quanto ás estruturas criadas por nós, estão presentes nos ficheiros com extensão .h e encontram-se listadas abaixo:

- **Node:** Esta estrutura é usada como lista ligada, para guardar as informações dos jogadores.

  - int fd: descritor do ficheiro da conexão com o jogador
  - int score: pontuação do jogador
  - struct _Node *next: apontador para o próximo item da lista

- **color:** Esta estrutura guarda as três componentes rgb (*red green blue*) em três inteiros, r, g e b. Foi criada para simplificar o uso de várias cores ao longo do projeto. Também ajudou na definição das 4 cores constantes e estáticas.

- **message:** Esta estrutura é usada para enviar as informações do servidor para o cliente/*bot* referentes à parte gráfica, de cada jogada.

  - int fim: assinala o final do jogo
  - int pos[2]: posição da jogada
  - char letras[3]: letras da jogada
  - color color_letra: cor das letras
  - color color_carta: cor da carta

- **bot_memunit:** Esta estrutura é usada como vetor no *bot* e é usada para guardar as informações do tabuleiro

  - int play[2]: posição no tabuleiro
  - char str_play[3]: letras dessa posição
  - int available: disponibilidade da carta

4 Protocolos de Comunicação

Nesta secção serão descriminadas todas as trocas de informação ao longo do decorrer do jogo. Embora se usem diferentes funções, a comunicação é realizada com*socket streams*.

4.1 Inicio do jogo

No inicio do jogo, assim que é lançada a *thread* de um novo jogador no servidor, é também enviado o tamanho do tabuleiro para esse jogador.

Do lado do servidor é usada função "write"e do lado do cliente/*bot* chamada a função "read_size"que usa o "read"para receber essa informação.

4.2 Seleção da carta

Na seleção de carta é usada a função "write_pos"(que trabalha com o write) do lado do cliente/*bot*. Do lado do servidor, a leitura é realizada ciclicamente no inicio da *thread* do respetivo utilizador e usando a função "read".

4.3 Atualização do tabuleiro

No lado do servidor, a atualização do tabuleiro é feita dentro do *switch case* do utilizador que efetuou a jogada. Assim, é percorrida a lista dos jogadores e enviada esta informação para cada um, através de um "write".

Do outro lado da comunicação, existe uma *thread* que está sempre a ler as mensagens do servidor com a função "read_msg"que usa um "read".

4.4 Final do jogo

No final do jogo, é feita a mesma lógica que se usa na atualização do tabuleiro, com a diferença de se usar um "write"adicional que desbloqueia os "reads"do utilizador para o próximo jogo.

É também usada a função "recv"com uma *flag* não bloqueante nos 10 segundos de espera entre jogos. Esta função absorve todo o lixo enviado pelos utilizadores e verifica se estes desconectaram.

4.5 Outros

A lógica do final do jogo também é usada na função "garbage_collector", usando a função "recv"para obsorver qualquer *input* do utilizador durante os 2 segundos depois de uma jogada errada.

5 Validação

5.1 Entre processos

Se modo a depurar o código, acrescentaram-se estrategicamente vários "printfs"em zonas críticas, que nos permitiram avaliar possíveis erros e a sua localização.

Desprezando os *inputs* do utilizador, o código terá um comportamento relativamente previsível. No entanto, os argumentos do programa podem causar situações imprevisíveis, no caso de terem um formato inválido. Deste modo, no que toca à validação, dirigiu-se o foco para o utilizador, criando as funções "check_args"que terminam o programa com *exit(0)* e com uma mensagem explicativa para cada caso. Os testes realizados encontram-se listados abaixo:
Comuns a todos os ficheiros:

- argc $>$ 2: demasiados argumentos
- argc $<$ 2: argumentos a menos

Apenas no servidor:

- "sscanf == 0": falha no *cast* da conversão do argumento para inteiro
- board_size $<$ 2: tabuleiro muito pequeno
- board_size $>$ 26: tabuleiro demasiado grande - não há letras suficientes para os pares não se repetirem
- board_size%2 == 1: número ímpar final de cartas

Apenas no cliente/*bot*:

- argv[1] $>$ 15: endereço IP demasiado pequeno
- argv[1] $<$ 7: endereço IP demasiado grande

5.2 Retorno de funções

Os valores de retorno das funções que se consideraram relevantes, foram testados antes de retornarem nos próprios métodos. Assim, o único retorno criado que apresenta um valor correspondente a um erro é no "main", caso os *mutexes* falhem na inicialização.

Em todas as funções de criação de *sockets* e estabelecimento de ligações é verificado o retorno para encontrar na ligação. Caso tal aconteça, o programa é encerrado, mostrando uma mensagem de erro com a função "perror"da biblioteca <errno.h>. Este tipo de validação também é usada em todas as funções de escrita e leitura entre o servidor e os utilizadores. Inclusivamente, quando se usa a função "recv"na espera de 10 segundos pelo próximo jogo, faz-se uma comparação com uma *flag* (errno != EAGAIN) para verificar se o jogador desconectou.

No módulo da parte gráfica, é verificado se a criação da janela foi bem sucedida, sendo que, caso não seja, o programa encerra com *exit(-1)* e com um "printf"adequado.

6 Regiões Criticas e Sincronização

6.1 Regiões Críticas

Cada posição do tabuleiro representa uma região crítica visto que só se pretende que 1 jogador tenha a posse da carta em qualquer instante de tempo.

Para tal serão necessários D^2 *mutexes* onde D corresponde às dimensão do tabuleiro de jogo. Para além disso estes *mutexes* não poderão ser bloqueantes visto que se assim o fossem, um jogador a tentar aceder a uma carta a ser utilizada por um outro iria ficar bloqueado até o jogador com posse da carta terminar a sua jogada.

O servidor, assim que recebe o pedido de jogada válido, bloqueia o *mutex* que corresponde à posição que se pretende ser jogada, estando este guardado num vetor de D^2 dimensões. Caso seja a primeira jogada, é guardada a posição bloqueada para posterior desbloqueio. Assim que o servidor receber a segunda jogada, se esta completar o par de jogadas (o jogador virou 2 cartas), após a jogada ser processada, ambas os *mutexes* são desbloqueados permitindo novamente o acesso às cartas.

6.2 Sincronização

No programa existem no fundo 2 eventos necessários de serem sincronizados, sendo um deles o reinicio do jogo e a pausa do jogo por jogadores insuficientes.

O primeiro, é feito enviando uma mensagem indicando o reinicio do jogo. Enquanto esta não for enviada, todos os clientes encontram-se bloqueados a ler a socket que os liga ao servidor.

O segundo encontra-se explicitado no ponto 7.1.

7 Descrição das funcionalidades implementadas

7.1 Número mínimo de jogadores

Sempre que um jogador é ligado ao servidor, o valor de uma variável global no servidor é incrementada. Caso um dos jogadores se desconecte, esta é decrementada. Na Figura 2 é evidente onde é feito o bloqueio caso o valor desta variável global no processo do servidor se encontre abaixo de 2.

Deste modo, esta testada é testada antes de se realizar uma jogada e resolve a questão do número mínimo de jogadores tanto no inicio como durante o jogo.

7.2 Diferenciação entre primeira e segunda jogada

Para diferenciar entre a primeira e a segunda jogada, aplicou-se a lógica já existente no código fornecido pelo corpo docente, em que se usa um vetor que guarda a posição da primeira jogada, sendo que este fica a -1"caso seja a primeira jogada. Este vetor passa por referencia do módulo do servidor para o módulo da lógica do tabuleiro. O *bot* aplica uma lógica semelhante, mas isoladamente, onde guarda cada jogada (primeira e segunda) em dois vetores separados. Estes vetores também ficam com o valor -1"caso não exista jogada.

7.3 Espera de cinco segundos entre jogadas

Nesta secção voltará a ser visitado o tema do processamento e distinção entre jogadas.

7.3.1 Na primeira jogada

A espera de 5 segundos é feita com o usa da função "poll", chamada dentro do servidor, sempre que é feita uma primeira jogada. Após os 5 segundos, é simulada uma jogada na mesma carta (caso com code=-1) para voltar a virar a carta para baixo.

7.3.2 Na segunda jogada

O "poll"permite que o código continue a execução normal, sem que se acrescente código adicional para cancelar a espera (esse cancelamento é feito interiormente pela própria função).

7.4 Espera de dois segundos após segunda jogada errada

Caso o jogador tenha errado a sua segunda jogada, este terá de esperar 2 segundos enquanto as cartas selecionadas são destacadas a vermelho. Para isso, caso se verifique a segunda jogada errada, o servidor lança uma *thread*, $garbage_collector(\ void * strut)$ a qual recebe um vetor com duas casa sendo a primeira o descritor da socket de comunicação do cliente que efetuou a jogada e a segunda um *token*. Este mecanismo de ignorar as mensagens recebidas do jogador enquanto este se encontra a espera para jogar encontra-se mais à frente explicado no ponto 7.5.2.

7.5 Final do jogo

No programa existem 2 processos distintos em os ciclos de funcionamento dos mesmos são regidos por uma variável global dentro de cada um deles denominada de *done*. Esta encontra-se a 1 caso se verifique o final do jogo, 0 caso o jogo esteja a decorrer. Com isto, o final do jogo é assinalado como a passagem do valor da variável de 1 para 0 no servidor, que é posteriormente comunicada pela resposta da jogada ao processo do cliente via socket.

7.5.1 Aviso com o vencedor

Antes de reiniciar o jogo, a lista de jogadores +e percorrida e imprime no terminal do servidor a pontuação de cada jogador (10 pontos por cada par encontrado) juntamente com o seu descritor (fd da socket). Por fim determina o jogador com a maior pontuação e anuncia-o, imprimindo o descritor do(s) jogador(s) vencedor(s).

7.5.2 Espera de dez segundos entre jogos

No final de cada jogo, a espera de 10 segundos é assegurada pela thread *wait10(void * strut)*. Esta, recebe um vetor cuja primeira posição indica o descritor da socket e a segunda posição corresponde a um *token*. Enquanto o valor do *token* estiver a 1, são lidas da socket todas as mensagens de forma não bloqueante, sendo estas descartadas. Caso se verifique a perde de conexão com a socket, significa que um dos cliente se desconectou. Como tal, será necessário remover-lo do jogo.

Assim que que o jogo acaba, o token é inicializado a 1 no vetor e a thread é iniciada. A seguir ao inicio da *thread*, o código permanece bloqueado por 10 segundos através da função *wait()*. Passados 10 segundos, a primeira posição do vetor, o *token*, é atualizado para 0, terminando o *thread*.

Por fim, o servidor envia a todos os clientes em jogo, uma mensagem a notificar do re-inicio do jogo.

7.5.3 Recomeço do jogo no cliente

Assim que é recebida a indicação que o jogo termina, o cliente termina a *thread* que lida com as mensagens do servidor e interrompe o ciclo que verifica a existência de eventos do rato. De seguida espera pela mensagem do servidor a notificar do re-inicio do jogo.

Assim que a recebe, fecha a janela do tabuleiro e volta a a abrir, estando agora vazio. Por fim, volta a iniciar a *thread* que lida com as mensagens do servidor e retoma o ciclo do envio de jogadas.

7.6 Limpeza após saída de clientes/*bots*

As informações de cada jogador (cor, fd e pontuação) são guardadas numa lista de estruturas no servidor. O número de jogadores ativos encontra-se também no servidor sob a forma de variável global. Sempre que um jogador é adicionado ao jogo, é criada a estrutura com os seus dados e esta é adicionada à lista, incrementando o número de jogadores. Ao sair, é procurado o jogador na lista através do seu descritor e é removido o respetivo nó da lista, decrementando o número de jogadores.

7.7 *Bot*

O *bot* possui um funcionamento muito semelhante ao de um cliente normal. Este comunica com o servidor da mesma forma que o cliente e também mostra o tabuleiro na sua máquina. Adicionalmente, este possui uma memória com capacidade de guardar D^2 cartas, sendo D a dimensão do tabuleiro.

Assim, sempre que receber uma resposta do servidor, este percorre a memória verificando se a posição recebida é já conhecida, se assim for atualiza o estado da carta. Se tiver recebido uma carta com letras pretas, atualiza a posição como indisponível, caso contrário atualiza o estado da posição para disponível.

A jogada de um *bot* é feita com base no conteúdo da memória. Se estiverem guardadas duas posições com a mesma *string*, são adicionadas as duas posição a uma fila sendo estas as próximas jogadas a serem efetuadas. Caso contrário são adicionadas à fila duas jogadas aleatórias.

8 Erros conhecidos

No decorrer do primeiro jogo o programa segue as especificações desejadas sem qualquer indicativo de existirem problemas. No entanto existem alguns bugs na parte gráfica quando o programa é corrido nos computadores do laboratório, nomeadamente a pintura de certas cartas a preto sem qualquer razão aparente. A razão por detrás destes erros é desconhecida.

No decorrer dos jogos após o final do primeiro, foram encontrados alguns erros.

8.1 Cartas bloqueadas

No jogo reiniciado por vezes é verificada a existência de algumas cartas às quais é impossível aceder. Este problema provém do bloqueamento das cartas quando estão na posse de um determinado jogador. No final do jogo algumas não devem estar a ser desbloqueadas o que resulta no erro verificado.

8.2 Libertação de memória

Embora no código estejam presentes os elementos necessários para libertar toda a memória alocada, o programa nunca chega a executar-las. Inicialmente, a variável *done* nºao indicava propriamente o final do jogo mas sim o final do programa. Com isto as instruções para libertar a memória eram executadas, até se ter implementado o mecanismo de espera de 10 segundos e reinicio do jogo.

Referências

[1] IBM. *C language examples*. URL: `https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/fg18800_.htm`.

[2] Michael Kerrisk. *The Linux Programming Interface*. URL: `http://man7.org/linux/man-pages/index.html`.

[3] João Nuno Silva e Ricardo Martins. «Slides das aulas teóricas e código dos trabalhos de laboratório».

[4] *Stack Overflow*. URL: `https://stackoverflow.com`.