Universidade do Minho

Nova Arena

Programação Concorrente

Grupo nº 2

Carlos Machado (A96936) — Hugo Rocha (A96463) — André Lucena (A94956) — Gonçalo Sousa (A91693)

27 de maio de 2023

Conteúdo

1 Introdução

Na unidade curricular de Programação Concorrente foi-nos proposto o desenvolvimento de um pequeno jogo onde os clientes podem interagir com um interface gráfica, desenvolvida em Java, comunicando com um servidor, escrito em Erlang, através de sockets TCP. Os vários avatares interagem entre si e com o ambiente que os rodeia, segundo uma simulação efetuada pelo servidor.
Ao longo deste documento, apresentaremos, de forma breve, as diferentes decisões tomadas, tanto no cliente como no servidor, para o desenvolvimento deste mini-jogo.

2 Servidor

2.1 File-manager

Neste arquivo, são criados dois processos, **level_manager** e **login_manager**. No **login_manager** são guardados os Usernames de cada jogador e as suas Passwords num mapa que deve ser atualizado conforme o pedido solicitado. Um jogador pode criar conta, fechar conta e efetuar log-in. Quanto ao **level_manager** são guardados os níveis de cada jogador também num mapa que tem como chave o Username do jogador e como valor um tuplo que guarda o seu nível e o número de vitórias. Como pedido no enunciado, um jogador recém-criado começa com nível 1, através da função **set-level**. Além disso, é possível obter o nível de um jogador através da função **check-level** e também é possível receber uma mensagem **end_game** onde é recebido o vencedor e o derrotado de um jogo, incrementando o número de vitórias do vencedor e, se for o caso, o seu nível (caso o seu nível seja igual ao dobro do número de vitórias). Foi também definida uma função **stop** que interrompe a execução destes dois processos. Por conveniência, a informção relativa aos Usernames, Passwords e níveis de cada jogador foram guardadas em ficheiro, aquando do término do servidor. Assim, ao iniciar os processos, os mapa são iniciados com a informação desses ficheiros, caso exista. Caso contrário, os mapas iniciam-se vazios.

2.2 Simulation

É neste ficheiro que é feita a simulação do jogo. O processo "**game**"é responsável por manter todas as variáveis de estado para ambos os jogadores, posição, ângulo, velocidade, aceleração, velocidade angular, pontos e o conjunto de teclas que se encontram premidas, para além de variáveis globais de jogo como a lista de **powerups**. Além disso, é gerado um processo "**ticker**"e um processo "**timer**"que controlam o tempo entre ticks e o tempo de jogo respetivamente, por envio de mensagens ao processo **game**. O **timer** é processado de forma a que, quando esse tempo passar, a simulação termine e seja determinado um vencedor, ou, em caso de empate, o jogo entra num estado de "**golden point**"no qual o primeiro jogador a marcar ponto é o vencedor.
O simulador pode alterar as teclas premidas entre **tick**'s, todas as outras mudanças de estado são feitas no momento em que é recebida a mensagem **tick**. Para tal são calculadas as novas posições, processadas as teclas para alterar a velocidade e o ângulo e a nova posição é comunicada ao cliente. Além disso, é verificado se os jogadores estão dentro do espaço de jogo, e caso contrário o jogo é terminado. Caso ambos os jogadores estejam no espaço de jogo são verificadas as colisões com os **powerups**, atualizadas a aceleração e a velocidade angular(o cálculo destes é feito tendo em conta a diferença entre o valor atual e o valor base para que os **powerups** confiram um ganho diminuído), e, caso necessário, adicionado um novo **powerup**, além disso, está implementado um sistema de **decay**, para que os efeitos dos **powerups**'s vão a diminuir até que o jogador volte aos valores-base de aceleração e velocidade angular. As adições e remoções de **powerup**'s são ambas comunicadas ao cliente.
Em último lugar é verificada a colisão entre jogadores, que só é válida se a diferença entre os respetivos ângulos seja menor do que $\frac{\pi}{2}$, se uma colisão for registada, o vencedor do ponto é calculado pelo sinal do produto interno entre o vetor velocidade e o vetor que une as posições dos jogadores. Quando um ponto é marcado é calculada uma nova posição para o jogador que perdeu e ambos os jogadores voltam à aceleração e velocidade angular base, são adicionalmente comunicadas ao cliente as novas posições e os pontos atualizados.

2.3 Space-server

Neste ficheiro encontra-se o início do servidor, na função **start**, onde é criado o servidor numa porta específica de forma semelhante à vista nas aulas práticas, isto é, utilizando uma função **acceptor** que permite aos clientes conectaram-se ao **Socket** para comunicação com o servidor, através de um **Listening Socket**. Além disso, inicia-se os processo do **login_manager** e do **file_manager** e são criados três processos, **lobby**, **win_manager** e **game_manager**, responsáveis por controlar os estados de cada cliente, as vitórias de cada utilizador, e por controlar os jogos em andamento, respetivamente. Após isso, podem ser enviadas várias mensagens do cliente para o servidor tais como: registar uma conta ou efetuar log-in. Nesse momento, são invocadas as funções do **file-manager** que tratam da gestão de clientes. O servidor indica de forma clara, quando as operações correram com sucesso ou se ocorreu algum problema. Após um cliente estar logged-in, por predefinição encontra-se **unready** para o início de uma partida.

2.3.1 lobby

A função **lobby**, que corre recursivamente no processo lobby, tem: um mapa, que associa o PID de um jogador ao estado em que ele se encontra (ready, unready, game), tal como ao seu Username, e que representa apenas os *online*; e uma lista que guarda todos os processos do tipo **acceptor**. Nesta função é possível receber vários tipos de mensagens, tais como: obter um certo número de jogadores obter os jogadores online e saída de um jogador do lobby. Também é responsável por terminar os processos dos utilizadores quando o servidor termina, através do átomo **stop**.

2.3.2 win_manager

O processo **win_manager** regista as vitórias de cada Username, desde o início do servidor. Pode receber mensagens dos utilizadores tanto para consultar estas vitórias como para adicionar novas. Estas vitórias não têm em conta os níveis dos utilizadores.

2.3.3 game_manager

O processo **game_manager** integra um mapa que, a cada nível, associa aos jogadores que estão prontos (no estado ready) a começar um jogo o processo do respetivo jogo. Ao receber a mensagem **unready** relativa a um jogador, a chave com o nível desse jogador é removida do mapa. Se for recebida a mensagem **ready** e o nível desse jogador ainda não existir no mapa, o respetivo par é adicionado ao mapa e o jogador fica em espera, gerando um processo que eventualmente se tornará num jogo. Por fim, ao receber esta última mensagem, se já estiver lá um jogador ready, significa que dois jogadores do mesmo nível estão prontos a começar um jogo e procede-se para o iniciar do jogo, avisando o jogo dessa possibilidade.

2.3.4 Processo do Jogo

Cada um dos jogos será representado por um único processo. Ele atravessará 3 estados diferentes, cada um representado por uma função: **ready**, que representa um jogo com apenas um jogador; **sync_up**, que testa se ambos os jogadores ainda estão prontos; **game** (presente no módulo *simulation*).

A primeira justifica-se pela necessidade de ter em conta o caso em que um segundo jogador se coloca pronto e, logo de seguida, o primeiro termina de estar pronto para jogar, depois do **game_manager** considerar o jogo começado. O **ready** não pode receber *abort* diretamente do primeiro jogador já que o segundo podia ser associado ao **ready** e ficar preso depois deste terminar, já que o **game_manager** já o esqueceu como jogo em espera.

A função **sync_up** serve para avisar os jogadores de que o jogo está prestes a começar, dando uma última oportunidade para desconexões sem penalização. Se ambos se mantiverem conectados, o jogo começa, a partir do simulador, invocando a função **simulation:start_game**.

2.3.5 Processo Utilizador

Cada um dos Utilizadores do servidor, depois de conectados ao **Socket**, seguem um ciclo de vida por diferentes estados: **main_menu**, onde podem conectar-se ou criar contas; **user**, onde podem começar jogos, ver o *top* ou apagar conta; **user_ready**, onde se encontram entre o jogo e o estado natural; **loading**, que apenas acontece quando um dos jogadores está a demorar na sua conexão; **player**, que representa um jogador no decorrer de uma partida, comunicando pelo **Socket** com o utilizador e com a simulação por mensagens *erlang*.

Sempre que uma conexão tcp for fechada ou der erro, o processo utilizador respetivo avisa o **lobby** do seu término. Dependendo da situação, passos extra devem ser tomados, como quando o utilizador está em modo **user_ready**. Esta dessincronização que permite jogadores **ready** de se desconectarem é corrigida nos momentos **sync_up** e **loading**.

A função **player** representa um jogador de uma partida, sincronizado. Esta função recebe comunicações tanto do utilizador como do jogo em curso, do servidor. A simulação não recebe os dados diretamente do **Socket** para maior facilidade de distinção e para não se perder a conexão tcp quando o jogo terminar, A função utiliza funções de API do ficheiro **simulation** tal como disponibiliza funções próprias do seu módulo, para comunicação com a simulação. A comunicação com o cliente depende dos tipos definidos para a distinção das mensagens, enviadas e recebidos pelo **Socket** tcp.

3 Cliente

3.1 GameState

Neste ficheiro são definidas duas classes: **Triple** e **GameState**. Esta última é responsável por guardar o estado de cada jogo, isto é, as posições, ângulos e pontuações dos dois jogadores. É também guardada uma variável do estado atual do jogo. As esferas coloridas relativas aos *powerups* são guardados num conjunto composto por instâncias da classe **Triple**. Esta classe é usada com dois propósitos distintos, podendo armazenar dois **floats**, que indicam a posição da caixa na arena, e um **char** que indica a sua cor ou, noutro caso, para armazenar três **floats** que representam a cor em formato **RGB**.

3.2 ConnectionManager

Este ficheiro é responsável pela conexão entre cliente e servidor e pelo envio/receção de mensagens entre os mesmos via socket-TCP. Para armazenamento das mensagens, é usado um mapa que mapeia o tipo de mensagem numa **Queue** do conteúdo de cada mensagem desse tipo. Numa fase inicial, todas as **Queues** são iniciadas vazias através da função **fillTypeMap**. Foi implementada nesta classe uma **Thread reader** que é responsável por ler do socket as mensagens vindas do servidor e por adicionar ao mapa essas mesmas mensagens. O processo de adicionar mensagens ao mapa é feito dentro dum bloco **synchronized** de forma a que seja adicionada ao mapa uma mensagem de cada vez. As mensagens têm o formato **type:conteúdo**. Foram definidas duas funções **send** e **receive** onde esta última deve bloquear, usando o método **wait**, sempre que não existirem mensagens de um dado tipo no mapa. Por essa lógica, a **thread reader** sempre que lê uma mensagem do Socket, acorda os processos que estão bloqueados na função **receive**, da **Queue** em que escreveu, através do método **notify**. Por fim, foi definida uma função **close** que fecha a conexão do cliente com o socket.

3.3 Communicator

Os **communicators** funcionam como intermediários entre a **thread** de desenho e a **thread** que lê do **socket**. Todas as **threads** possuem o **ConnectionManager** como instância e portanto, conseguem usar o método bloqueante **receive** para obter a string do seu tipo, portanto, o seu comportamento será ficar em **loop**, fazendo **receive** a cada iteração, deste modo evitamos **polling**, pois caso não haja nada na **queue**, a mesma vai adormecer. Quando notificada, vai tratar da sua mensagem, ao converter e escreve-la no objeto **GameState**, objeto que é partilhado com a **thread** gráfica. Para poder escrever no **GameState**, é necessário obter o **lock** de leitura, como cada uma escreve na sua parte, não é preciso fazer concorrência ao nível das instâncias, assim podemos usar um **ReadWriteLock** em que o **lock** de escrita é usado pelo processo gráfico para apenas copiar o estado do jogo, bloqueando o objeto inteiro. Temos como **comunicators**: posição do jogador, inimigo, caixas, pontuação e condição do jogo.

3.4 Processing

Esta é a classe responsável por ter as funções de desenho, menus e o **main**. Temos diferentes menus: inicio, **login**, registo, conta **logged-in**, **top**, eliminar conta, **loading** e, finalmente, o jogo. Toda a lógica de seleção é feita com recurso à função **KeyPressed** do **Processing**, desse modo podemos trocar de menus premindo no botão certo. Além disso, há uma classe para a construção das caixas de texto que permite simular uma caixa de texto. Interações fora de jogo entre cliente e servidor são feitas diretas na thread do processing, ou seja, o **receive** e **send** é feito diretamente pela mesma. Uma vez que estamos a fazer operações sequenciais, quando entramos no jogo, cada **thread** responsável por cada elemento do jogo que mencionamos anteriormente adquire o **lock** de leitura e escreve no **GameState**. Para poder desenhar, a **thread** de desenho faz o **lock** de escrita sobre o **GameState**, copia para um **gameState** local e liberta o **lock**, em vez de ficar mais tempo desenhando diretamente para a cena. Para enviar as teclas premidas, com o uso de uma classe privada que contém um **array** em que cada elemento corresponde a uma tecla, o elemento passa a ser "t"e quando largamos volta a ser "f". Depois testamos sempre se as teclas foram premidas ou largadas, caso afirmativo enviamos para o servidor uma mensagem com o estado das teclas. Assim, só avisamos o servidor quando premimos ou largamos as teclas.

4 Conclusão

Em suma, este trabalho permitiu-nos uma melhor compreensão da comunicação **cliente-servidor** via TCP e também do mecanismo de controlo de concorrência através de locks explícitos, em Java. Na realização do servidor, ficamos a entender melhor como funciona o sistema de receção e envio de mensagens em **Erlang** e adaptámo-nos bem a esta nova linguagem.

Entendemos que o trabalho cumpre as expetativas do enunciado em questão e vai-nos permitir, no futuro, um bom uso de todos os mecanismos de concorrência estudados na disciplina.