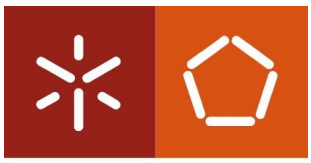

Programação Concorrente
(3º ano de LCC)
Trabalho Prático
Relatório
Grupo 7

Beatriz Fernandes Marques	(A100068)
Eduardo Manuel Sousa Pereira	(A70619)
João Manuel Franqueira da Silva	(A91638)
Matilde Galhetas Domingues	(A98982)

1 de junho de 2024

Resumo

Neste relatório descreve-se o desenvolvimento de um jogo chamado Gravidade, no âmbito da Unidade Curricular de Programação Concorrente, do 3º ano do Curso de Licenciatura em Ciências da Computação, na Universidade do Minho.
Este jogo consiste na comunicação entre um servidor e um, ou mais, utilizadores, de forma a permitir que o jogo corra de forma fluída e concorrente.

O desenvolvimento deste projeto foi dividido em 3 fases:

- Identificação dos pontos centrais do jogo
- Implementação do servidor e da interface utilizador
- Revisão de concorrência e das regras de jogo

O estado final deste relatório conta com todas as etapas definidas no início do desenvolvimento do mesmo.

Índice

1 Introdução **1**

2 Funcionalidades **2**
2.1 Registo e informações de Utilizadores 2
2.2 Regras de Jogo 2

3 Estrutura de Implementação **4**
3.1 Cliente 4
3.2 Servidor 5

4 Conclusão **7**

Capítulo 1

Introdução

O objetivo deste jogo é permitir que os utilizadores interajam com o servidor, através de uma interface gráfica e obtenham respostas do mesmo, de forma que todos os jogadores tenham as mesmas informações sobre o jogo em que estão inseridos. Também se pretende que seja possível que cada utilizador registar-se, fazer login e logout neste jogo.

O jogo foi implementado baseado em duas vertentes, servidor e cliente. Dada a natureza do jogo, estes foram desenvolvidos em duas linguagens de programação diferentes. A implementação do servidor foi feita na linguagem Erlang, que trouxe grandes benefícios nesta vertente por ser uma linguagem funcional que permite elevada concorrência de processos. A interface cliente foi desenvolvida em Java, com recurso à ferramenta *Processing*.

Cada jogo irá decorrer num espaço 2D, espaço esse que terá um Sol no ponto central do espaço e alguns planetas, entre 2 e 4, a movimentarem-se a velocidades pré-definidas aleatoriamente. No decorrer do jogo, cada jogador possui um avatar, que será diferente para cada jogador e que se irá movimentar no espaço designado. Estes avatares podem colidir entre si, alterando a sua direção de movimento, mas uma colisão entre um avatar e um corpo de maior dimensão, seja ele o Sol ou qualquer planeta, resulta na derrota imediata do respetivo jogador.

A comunicação entre Servidor e Cliente foi feita fazendo-se uso de *Sockets TCP* de forma a manter uma comunicação eficiente e coerente entre as duas partes.

Ao longo deste relatório, detalhamos as decisões de implementação tomadas, quais os desafios encontrados durante o desenvolvimento do projeto e as soluções encontradas.

Capítulo 2

Funcionalidades

2.1 Registo e informações de Utilizadores

- **Registo de Utilizador:**
  - Realizado a partir da interface Cliente, sendo que as informações relevantes ao registo são enviadas para o Servidor, que trata de as guardar;
  - O cliente escolhe o seu *Username* e a sua *Password*, que serão usados para o *Login*, *Logout* e para outras operações relacionadas com a conta de cada jogador.
- **Listagem do top 10**
  - É disponibilizado um *Ranking* do top 10 de jogadores;
  - Este *Ranking* é ordenado por nível e, em caso de empate, pela última série de vitórias consecutivas.

2.2 Regras de Jogo

- **Progressão**
  - É possível evoluir no jogo, subindo ou descendo de nível;
  - Um jogador que esteja num determinado nível *N*, sobe de nível se ganhar *N+1* partidas consecutivas e desce de nível caso perca *N/2* partidas consecutivas.
- **Partidas**
  - As partidas têm um número variável de jogadores, entre 2 e 4, sendo que estes apenas podem diferir, no máximo, em um nível;
  - Após a entrada de cada jogador para uma determinada sala de jogo, o servidor espera 5 segundos antes de começar o jogo;
  - No caso de uma sala ter 4 jogadores, o servidor não permite a entrada de mais jogadores nessa sala e começa o jogo 5 segundos depois.
- **Espaço e Elementos de Jogo**
  - O espaço é 2D, retangular e sem limites nas bordas;

- Existe um Sol, fixo no centro do espaço. Este possui uma força gravitacional que modifica o movimento dos restantes corpos. Também existem planetas de diferentes tamanhos, a movimentarem-se a diferentes velocidades em torno do Sol;
- Cada jogador é identificado por um avatar circular, distinto para cada jogador dentro da mesma sala. Este avatar tem um indicador da direção para o qual está virado;
- Também o Sol e os planetas existentes são representados por avatares circulares distintos;

- ***Game Over* e Pontuação**
  - Um jogador perde quando sai da área visível de jogo. Qualquer colisão entre um jogador e um planeta ou o Sol também resulta na sua derrota;
  - Um jogador é nomeado vencedor se, quando todos os outros perdem, este consegue sobreviver durante mais 5 segundos;
  - Caso este não consiga, não há nenhum vencedor daquela partida, mas os outros jogadores são considerados perdedores;
  - Após o final de um jogo, a pontuação dos jogadores em questão é atualizada conforme necessário.

Capítulo 3

Estrutura de Implementação

3.1 Cliente

O **Cliente** é composto por várias Classes

- ***Menu*** - trata da renderização das imagens e captura cliques em teclas para vários efeitos como movimentação dos jogadores. Contêm várias funções de desenho, uma para cada estado possível em que o utilizador se encontre.
- ***Player*** - classe que representa os jogadores que se encontram num dado jogo, também usada para descrição dos planetas e do sol.
- ***TCP*** - utilizado para comunicação TCP com o servidor, enviando e recebendo informações importantes.
- ***Auxiliar*** - Intermediário entre o Menu e o servidor. Aguarda respostas do servidor, comunicando com o Menu a informação recebida.
- ***Informacao*** - guarda informações importantes sobre o estado do jogo e variáveis referentes ao utilizador e ao jogo. Contem dois Locks, um utilizado para o Menu esperar por dados, sendo o outro usado para acordar um objeto da classe Auxiliar, quando o Menu necessitar de comunicar com o servidor.

Iremos agora explicar de forma mais técnica como é realizada a comunicação entre o cliente e o servidor.
Inicializado o programa, o menu renderiza o menu inicial, dando a possibilidade ao utilizador de criar uma conta nova ou efetuar login.

Consoante a tecla premida, o menu passará a manter registo dos botões carregados, para username e password. Eventualmente, o menu irá invocar a função *handleTCPState*, que começara por acordar um objeto da classe *Auxiliar*, objeto este, que partilha uma instância da classe *Informacao* com o Menu, esperando então o Menu por alguma resposta. Dependendo do estado em que nos encontramos, este objeto irá efetuar um pedido de login ou criação de conta ao servidor, através da classe *TCP*.

Depois de efetuado o pedido, uma mensagem será recebida. Caso esta mensagem indique que a operação se efetuou com sucesso, o objeto *Auxiliar* irá sinalizar o *Menu*, indicando que a operação foi efetuada com sucesso, e que este pode prosseguir para o próximo estado.
Neste caso, irá ser inicializado o menu do estado **LOGGED**, apresentando agora a possibilidade ao utilizador de se juntar a um jogo, ver o ranking dos utilizadores com as melhores pontuações, efetuar logout ou remover a sua conta.

Caso a mensagem do servidor indique que a operação não foi realizada, o objeto *Auxiliar* irá sinalizar o menu, acordando-o com a indicação do próximo estado para que este prossiga. Neste caso concreto, novamente o Menu inicial.

Para o jogo em si, o Menu irá constantemente captar se as teclas *w,a,d* são premidas e, se são, verificar quando deixam de o ser. O objeto da classe *Auxiliar* irá constantemente escutar as mensagens do servidor, que contêm todas as variáveis do jogo, com as posições dos jogadores e planetas, guardando-as e permitindo que o Menu as obtenha para desenhar no Menu do jogo.

É de notar que, através da função *handleTCPState*, os objetos Menu e Auxiliar encontram-se em sintonia, através dos *locks* e *waits* nas variáveis de condição. Nunca acontece que um deles prive o outro de efetuar as suas operações.

3.2 Servidor

No módulo **Server** encontra-se a implementação de um servidor que controla a interação entre os utilizadores e com os elementos do jogo. Este inicia o servidor na porta "1"e regista dois processos: *startLogin* e *serverStart*.

O processo **startLogin** é responsável por inicializar o processo *loopContas* e registar o *Process ID* do mesmo como sendo o do utilizador atual.

- **Processo *loopContas***
  - Inicializa o mapa de todas as contas de utilizadores registados;
  - Este mapa é inicializado a partir do ficheiro "users.dat"através da função *loadUsers*;
  - Este ficheiro é atualizado quando um utilizador faz alguma ação relativa à sua conta, por exemplo, registo, *Login/Logout* através da função *saveUsers()*.
  - Também são registadas alterações no mapa aos utilizadores de um jogo terminado, podendo ser incrementado ou decrementado o seu nível e as suas vitórias.
  - É também responsável pela gestão geral de todas as contas.

O processo **serverStart** é responsável por inicializar o processo *waitingRoom*, aceitar a ligação da *Socket* do utilizador, criar um processo *acceptor* e enviar-lhe o *PID* da *waitingRoom* e a identificação da *Socket*.

- **Processo *acceptor***
  - Aceita a ligação da socket recebida do *serverStart* e começa outro processo *acceptor* de forma a ser possível criar várias salas;
  - Invoca a função *clientParser* com a *socket* e o *Pid* de uma sala, de forma a esta escalonar os pedidos que chegam do Cliente.
- **Processo *clientParser***
  - Responsável por comunicar diretamente com um cliente.
  - Recebe diversos tipos de mensagens, sejam estas de login, logout, criar ou remover contas, pedidos para jogar, entre outros.

- Aquando o começo de um jogo, para cada jogador que participará no mesmo, o seu processo *clientParser* irá receber uma mensagem, contendo o pid do jogo, que irá ser guardado como argumento na chamada recursiva da função. Sendo assim, quando é recebida uma mensagem de movimentação no jogo, esta é redirecionada para o pid em questão.

- **Processo *waitingRoom***
  - O processo *waitingRoom* lida com vários tipos de mensagens:
  - **userConnected:** quando um utilizador se conecta, a função adiciona o jogador ao mapa *Users* (associando os utilizadores aos seus sockets) e envia uma mensagem indicando a sua conexão.
  - **userDisconnected:** quando um utilizador se desconecta, a função remove o jogador do mapa *Users* e indica a sua desconexão.
  - **clearRoom:** remove a sala da lista *Warmups* do processo *waitingRoom*.
  - **play:** quando um utilizador pretende jogar, chama-se a função *matchmaking* para encontrar uma *warmup room* adequada e envia-se uma mensagem ao processo para inserir o novo jogador na respetiva sala. Caso não exista uma sala adequada, cria-se uma nova com esse jogador.

  Este processo é essencial para garantir que todos os jogadores recebam informações consistentes sobre o estado das partidas e das *Warmup Rooms*, proporcionando uma experiência de jogo sincronizada e organizada.

- **Processo *initGame***
  - Prepara o estado inicial do jogo, criando um mapa com os atributos de cada jogador (posição, ângulo, velocidade angular, velocidade, combustível) e um mapa de planetas, também com informações relativas a estes (posição, distância, raio, velocidade).
  - Cria um processo *match*, guardando o seu pid e envia-o para os *clientParser* de todos os jogadores, de forma a estes poderem comunicar com o jogo
  - Sinaliza os clientes do começo do jogo e, por último, inicia o timer para os updates constantes durante o jogo, bem como um timer extra para terminar o jogo, passados alguns segundos.

- **Processo *match***
  - Responsável pela simulação do jogo, contém dois mapas, um com informações relativas aos jogadores, outro com informações dos planetas.
  - Recebe constantemente informações das funções *clientParser* dos seus jogadores, referentes aos movimentos, alterando então o mapa de jogadores.
  - Recebe mensagens de update a cada 21 milissegundos (resulta em cerca de 45 ticks por segundo), alterando o mapa dos jogadores consoante os seus atributos, o mapa dos planetas de forma gradual e também detetando colisões entre os objetos do jogo, adicionando eventuais jogadores derrotados a uma lista especial, de forma a estes não alterarem o seu estado no jogo.
  - Quando apenas um jogador resta no jogo, é enviada uma única mensagem passados 5 segundos. Depois desse tempo, se o jogador em questão ainda prevalecer no jogo, é considerado o vencedor.
  - De forma a evitar partidas infinitas (por alguma razão), 2 minutos depois da iniciação do jogo, é recebida uma mensagem indicando o fim do mesmo. Casa apenas um jogador estiver ativo, então esse é declarado como vencedor e, em caso contrário, não existe nenhum vencedor.

Capítulo 4

Conclusão

Este relatório descreve o processo de implementação de um Cliente e um Servidor que, em conjunto, possibilitam a execução de um jogo.

Foram implementadas todas as funcionalidades propostas pelo docente no enunciado deste projeto, nomeadamente a facilidade de criação de várias salas que, por sua vez, tornam possível que vários jogos decorram simultaneamente e de forma independente.

Sentimos que a possibilidade de criação de vários jogos foi um ponto fulcral no desenvolvimento deste projeto, dado que o objetivo desta UC é executar processos concorrentemente e, através deste jogo, conseguimos identificar os problemas de desempenho que podem surgir se um servidor não for bem implementado.

Foram encontradas certas dificuldades, como encontrar forma de restringir cada sala de jogo a um máximo de 4 jogadores, assim como permitir o início do jogo com um número variável de jogadores. Mas pensamos que a parte mais desafiante foi manter uma comunicação coerente entre Cliente e Servidor, pois sem esse pormenor, não era possível fazer avanços significativos no desenvolvimento.

Este projeto foi um bom meio de aprendizagem, pois ajudou a consolidar os conhecimentos adquiridos ao longo da cadeira. Em particular, expandimos a nossa experiência com implementações com concorrência e percebemos a importância de proteger certas partes das nossas implementações de forma a evitar erros e inconsistências. Além disso, aprendemos a trabalhar com a linguagem Erlang e expandimos o nosso conhecimento da linguagem Java.

Percebemos que Erlang é uma boa forma de implementar este tipo de sistemas, por ser altamente concorrente e permitir a troca de mensagens entre processos. Também graças à existência de threads, locks e métodos bloqueantes em Java, foi possível manter segurança e integridade no jogo desenvolvido.

Uma boa evolução deste projeto no futuro, seria a melhoria da interface gráfica, por exemplo, para disponibilizar funcionalidades de consulta de estatísticas pós-jogo, como vencedor e níveis atualizados dos intervenientes da partida.