# **Relatório Final – Agar.io – Programação Concorrente e Distribuída**

O primeiro passo de implementação do atual projeto consistiu em alterarmos a classe *Direction* para o tipo “*enum*” e definir possibilidades de direção para os jogadores automáticos através de um vetor. No segundo passo alterámos a classe *Coordinate* para ter um método que traduzisse as informações do vetor para um ponto (x,y). A classe abstrata *Player* tornou-se numa *thread* e todos os jogadores (automáticos e humanos) são *threads*. Criámos uma classe *AiPlayer* para os jogadores automáticos que corre uma *thread* segundo todas as condições descritas no enunciado e estende a classe *Player*.

As funções principais para a construção e execução do jogo estão na classe *Game* e na classe *Cell*. A sincronização está o mais localizada possível na classe Cell.

**O método utilizado para resolver a colocação inicial dos jogadores é diferente do processo utilizado para resolver a imobilização dos jogadores. Em ambos casos, estamos a falar de processos ligeiros mas o método que resolve a imobilização do movimento (classe *Cell* - *PlayerNotifier*) diferencia-se porque utiliza um *sleep* e um *interrupt* para acordar o jogador imobilizado.**

**A *thread* responsável pela colocação inicial dos jogadores (*PlayerPlacer*) apenas utiliza o *AddPlayerToGame* para adicionar o jogador ao tabuleiro e faz start() do jogador sendo invocada e despoletada no procedimento *InitializePlayers* responsável por definir e inicializar a posição aleatória de cada um dos jogadores que pretende entrar no jogo.**

Utilizámos um processo de sincronização para o movimento (*tryMovement*) na classe *Cell* que evita que a thread *PlayerPlacer* continue a colocar jogadores em jogo caso um jogador já esteja a ocupar a posição. Invocámos o *wait* para caso a posição esteja ocupada, a *thread* *PlayerPlacer* poder ser acordada de novo por processo ligeiro para poder voltar a colocar os jogadores no tabuleiro na posição posteriormente desocupada. A colisão é realizada nesta classe com recurso a procedimento próprio.

A estrutura de sincronização temporal que utilizámos para terminar o jogo foi o *CountDownLatch* clássico regressivo a começar no 3. Quando três threads (jogadores automáticos ou humanos) chegassem ao valor de 10 (máximo), o contador baixa um valor e para no 0. Colocámos uma *thread* (*EndGame*) na classe *Game* que estendia a classe referente a esta estrutura de sincronização temporal que faz o *await* e um *interrupt* de todos os jogadores. Para não haver incoerências de dados e conseguirmos interromper o movimento de todos os jogadores com exatidão, colocámos todos os jogadores numa lista (*Player\_List*) para garantir a integridade do tabuleiro.

Para implementar a possibilidade de jogar com jogadores reais, criámos uma classe *HumanPlayer* com os requisitos definidos no enunciado e utilizámos a estrutura cliente-servidor clássica com um *handler*, cliente e servidor. A comunicação entre o servidor e o cliente realiza-se no sentido servidor-cliente com recurso a canais de objetos (*ObjectOutputStream* e *ObjectInputStream*), e, no sentido oposto, utilizando canais simples de texto (*BufferedReader* e *BufferedWriter*).

Criámos ainda uma classe (*PlayerSummary*) com dados para serem enviados através da rede pela classe *ClientBoardComponent* que contém a parte do tabuleiro que o cliente pode ter acesso.

**Identificámos duas possíveis situações de conflito ou de melhoria deste projeto. Durante os nossos testes, não verificámos que existisse *starvation* porque todas as *threads* implementadas conseguem ter acesso aos recursos partilhadose seções críticas.**

***Deadlock* poderá verificar-se na colocação inicial dos jogadores se dois ou mais jogadores (*threads*) estiverem à espera um do outro para serem colocados na mesma posição “aleatória” e fiquem em espera até que a posição seja libertada.**

**Durante a realização do projeto, verificámos após várias execuções que o movimento por ciclos dos jogadores tinha um atraso normal e poderia acontecer que o propósito do *CountDownLatch* para parar o movimento dos jogadores apenas parasse efetivamente com 4 ou 5 jogadores a terem chegado à pontuação máxima. Isto é claramente uma situação de conflito porque o jogo termina se apenas 3 chegarem à pontuação máxima. O jogo apenas executa corretamente algumas vezes segundo esses critérios. Não encontrámos medidas para mitigar este problema.**

**A segunda situação de conflito centra-se na lista de *handlers* que não é sincronizada e pode gerar problemas de coerência na execução do jogo. Uma possível medida para mitigar esta inconsistência baseava-se em colocar o *handler* numa lista de *handlers* (*ArrayList* ou *LinkedList*) à semelhança do que fizemos com as listas de jogadores para a implementação do *CountDownLatch* (*Player\_List*).**

***Para colocar o nosso jogo “Agar.io” a correr apenas com jogadores automáticos basta executar a classe “Server”. Para controlar um jogador, deve-se executar ainda a classe “ClientMain” após execução da classe “Server”.***

**Trabalho realizado por:** Guilherme Raimundo nº88730 e Ana Rita Ramos nº81973