

Programação Concorrente (3º ano de LCC)  
Projeto Prático  
**Duelo**

Diogo Coelho  
(A100092)

João Barbosa  
(A100054)

Pedro Oliveira  
(A97686)

16 de maio de 2025

## **Resumo**

Este relatório descreve o desenvolvimento de um projeto no âmbito da Unidade Curricular de Programação Concorrente, do 3<sup>o</sup> ano da Licenciatura em Ciências da Computação, na Universidade do Minho.

O projeto consiste na implementação de um mini-jogo chamado "Duelo", onde vários utilizadores podem interagir usando uma aplicação cliente com interface gráfica, escrita em Java (Processing), intermediados por um servidor escrito em Erlang. Os jogadores movimentam-se num espaço 2D, interagindo entre si e com o ambiente que os rodeia, segundo uma simulação efetuada pelo servidor.

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Análise e Especificação</b>	<b>3</b>
2.1	Descrição do Problema . . . . .	3
2.2	Arquitetura do Sistema . . . . .	3
<b>3</b>	<b>Implementação do Cliente</b>	<b>5</b>
3.1	Estrutura e Estados . . . . .	5
3.2	Concorrência e Exclusão Mútua . . . . .	5
3.3	Monitores e Variáveis de Condição . . . . .	6
3.4	Threads Explícitas . . . . .	6
3.5	Interface Gráfica . . . . .	7
3.6	Sistema de Movimento e Física . . . . .	7
3.7	Sistema de Projéteis e Modificadores . . . . .	7
3.8	Comunicação com o Servidor . . . . .	8
<b>4</b>	<b>Implementação do Servidor</b>	<b>9</b>
4.1	Estrutura e Gestão de Dados . . . . .	9
4.2	Concorrência em Erlang . . . . .	9
4.3	Autenticação e Gestão de Utilizadores . . . . .	10
4.4	Sistema de Níveis e Matchmaking . . . . .	10
4.5	Gestão de Partidas e Simulação . . . . .	10
4.6	Processamento de Colisões . . . . .	11
<b>5</b>	<b>Protocolos de Comunicação</b>	<b>12</b>
5.1	Principais Mensagens . . . . .	12
<b>6</b>	<b>Conclusão</b>	<b>13</b>

# Capítulo 1

## Introdução

No âmbito da Unidade Curricular de Programação Concorrente, foi-nos proposto o desenvolvimento de um mini-jogo chamado "Duelo". Este jogo implementa um sistema cliente-servidor onde os clientes são desenvolvidos em Java (utilizando a biblioteca Processing) e o servidor em Erlang.

O "Duelo" é um jogo *multiplayer* onde os jogadores controlam avatares num espaço 2D, podendo disparar projéteis contra os adversários e colecionar modificadores que alteram temporariamente as suas capacidades de jogo. O sistema implementa funcionalidades como registo e autenticação de utilizadores, sistema de níveis baseado no desempenho, **matchmaking** entre jogadores de níveis semelhantes, gestão de múltiplas partidas simultâneas, e uma tabela de classificação (**leaderboard**).

# Capítulo 2

## Análise e Especificação

### 2.1 Descrição do Problema

O jogo "Duelo" implementa as seguintes especificações principais:

- **Registo e Autenticação:** Sistema para registo de novos utilizadores e autenticação de utilizadores existentes.
- **Sistema de Níveis:** Jogadores começam no nível 1, progridem após  $n$  vitórias consecutivas e descem após  $\lceil n/2 \rceil$  derrotas consecutivas, nunca abaixo do nível 1.
- **Partidas:** Cada partida envolve 2 jogadores com diferença máxima de um nível, permitindo múltiplas partidas simultâneas.
- **Ambiente de Jogo:** Espaço 2D retangular com paredes nos quatro lados, onde os jogadores (círculos) aparecem inicialmente em lados opostos.
- **Física do Movimento:** Os avatares possuem inércia, com aceleração controlada por teclas de direção.
- **Projéteis:** Jogadores podem disparar projéteis em intervalos regulares na direção do cursor.
- **Modificadores:** Quatro tipos (verde, laranja, azul e vermelho) que afetam velocidade de projéteis ou intervalo entre disparos.
- **Colisões:** Entre jogador-projétil (+1 ponto para o atirador), jogador-parede (+2 pontos para o adversário), e jogador-modificador (aplicação do efeito).
- **Pontuação:** Vence o jogador com maior pontuação após 2 minutos. Empates são ignorados para efeitos de nível.
- **Leaderboard:** Top 10 jogadores ordenados por série de vitórias/derrotas.

### 2.2 Arquitetura do Sistema

O sistema segue uma arquitetura cliente-servidor onde:

- **Cliente:** Desenvolvido em **Java/Processing**, responsável pela interface gráfica, captura de **inputs** do utilizador e comunicação com o servidor.
- **Servidor:** Implementado em **Erlang**, gerindo utilizadores, processando a lógica do jogo e coordenando múltiplas partidas simultâneas.

A comunicação entre cliente e servidor ocorre através de **sockets** TCP, utilizando um protocolo de mensagens baseado em texto com campos separados por ponto e vírgula (;).

# Capítulo 3

## Implementação do Cliente

### 3.1 Estrutura e Estados

O cliente foi implementado seguindo um padrão baseado em estados, onde cada estado representa uma fase distinta de interação. Os estados principais são: `Login`, `Registo`, `Menu Principal`, `Leaderboard`, `Sala de Espera` e `Jogo`.

Para cada estado, existe uma função específica de tratamento que gerencia a renderização e a lógica associada. Esta abordagem facilita a transição entre diferentes ecrãs e funcionalidades, mantendo o código organizado e modular.

### 3.2 Concorrência e Exclusão Mútua

Uma parte fundamental da implementação do cliente é a gestão da concorrência. Para evitar condições de corrida e garantir que as operações simultâneas não comprometam a integridade dos dados, implementámos diversos mecanismos de sincronização:

- **Monitores nativos:** Utilizámos a palavra-chave `synchronized` de Java para criar blocos de código que só podem ser executados por uma `thread` de cada vez.
- **Objetos de bloqueio (locks):** Criámos objetos específicos para controlar o acesso a diferentes recursos partilhados:
  - `p1Lock` e `p2Lock`: Protegem o acesso aos objetos dos jogadores
  - `bulletsLock`: Controla o acesso à lista de projéteis
  - `modifiersLock`: Garante acesso seguro à lista de modificadores
  - `userInfoLock`: Protege as informações do utilizador atual
  - `leaderboardLock`: Controla o acesso à tabela de classificação
- **Coleções thread-safe:** Utilizámos `Collections.synchronizedList` para criar listas seguras para acesso concorrente:

```
– private final List<Bullet> bullets = Collections.synchronizedList(new ArrayList<Bullet>());
– private final List<Modifier> modifiers = Collections.synchronizedList(new ArrayList<Modifier>());
```

### 3.3 Monitores e Variáveis de Condição

Implementámos dois monitores principais para gerir o estado do jogo e as conexões:

- **GameStateMonitor:** Utiliza `ReentrantLock` e variáveis de condição (`Condition`) do pacote `java.util.concurrent.locks` para gerir o estado do jogo:
    - `gameStarted`: Condição sinalizada quando o jogo começa
    - `gameEnded`: Condição sinalizada quando o jogo termina
- Este monitor permite que certas `threads` aguardem determinadas condições de jogo, como o início ou fim de uma partida, através dos métodos `waitForGameStart()` e `waitForGameEnd()`.
- **ConnectionMonitor:** Implementa um monitor para gerir a conexão com o servidor, utilizando o paradigma de monitores nativos de Java (com `synchronized`, `wait()` e `notifyAll()`):
    - Controla o estado da conexão e as respostas do servidor
    - Permite que `threads` aguardem por eventos como respostas de autenticação

### 3.4 Threads Explícitas

Para melhorar a responsividade da aplicação e evitar bloqueios na interface do utilizador, implementámos `threads` dedicadas:

- **Thread de leitura do servidor:** Uma `thread` permanente que lê continuamente mensagens do servidor, processando-as sem bloquear a `thread` principal:

```
serverListener = new Thread(new Runnable() {
    public void run() {
        while (running) {
            try {
                readServerMessages();
                Thread.sleep(10); // Pausa para evitar uso excessivo de CPU
            } catch (Exception e) {
                // Tratamento de erro e tentativa de reconexão
            }
        }
    }
});
serverListener.start();
```

- **Threads para operações de autenticação:** Threads temporárias que executam operações de login e registo de forma assíncrona:



```

new Thread(new Runnable() {
    public void run() {
        try {
            output.println("LOGIN;" + inputUsername + ";" + inputPassword);
            connectionMon.waitForResponse(5000); // 5 segundos de timeout
        } catch (Exception e) {
            // Tratamento de erro
        }
    }
}).start();

```

## 3.5 Interface Gráfica

A interface gráfica foi projetada para ser intuitiva e informativa, incluindo:

- **Ecrãs de Login/Registo:** Formulários simples para introdução de credenciais, com validação e feedback de erros (ex.: palavra-passe incorreta).
- **Menu Principal:** Apresenta opções como jogar, ver leaderboard, e sair do jogo.
- **Sala de Espera:** Exibe status de **matchmaking** em tempo real.
- **Ecrã de Jogo:** Mostra o espaço de jogo, avatares, projéteis, modificadores e informações como pontuação, tempo restante e status dos modificadores ativos.

Utilizamos a biblioteca **Processing** para simplificar o desenho dos elementos gráficos, aproveitando suas funções para criar círculos (jogadores, projéteis), textos e outros elementos visuais.

## 3.6 Sistema de Movimento e Física

Implementamos um sistema de movimento que simula inércia e fricção. Quando um jogador pressiona uma tecla de direção, o avatar não muda de posição instantaneamente, mas recebe uma força de aceleração naquela direção. A velocidade é incrementada gradualmente até atingir um valor máximo predefinido.

A fricção é aplicada a cada atualização, desacelerando o avatar quando nenhuma tecla é pressionada. Este sistema cria uma sensação de movimento mais natural e fluida, exigindo mais habilidade dos jogadores para controlar seus avatares com precisão.

## 3.7 Sistema de Projéteis e Modificadores

Os projéteis são criados quando o jogador clica com o rato, respeitando um intervalo mínimo entre disparos. Cada projétil movimenta-se na direção do cursor no momento do disparo, com velocidade constante.

Os modificadores aparecem aleatoriamente no mapa e afetam temporariamente as capacidades do jogador quando coletados:

- **Verde:** Aumenta a velocidade dos projéteis (+50%)
- **Laranja:** Diminui a velocidade dos projéteis (-30%)
- **Azul:** Diminui o tempo de espera entre disparos (-30%)
- **Vermelho:** Aumenta o tempo de espera entre disparos (+50%)

Estes efeitos são temporários e vão desaparecendo gradualmente, retornando aos valores originais após alguns segundos.

## 3.8 Comunicação com o Servidor

A comunicação com o servidor é estabelecida via **sockets** TCP no início da execução. O cliente mantém duas **threads**: uma para a interface gráfica e processamento de inputs, e outra para ler continuamente mensagens do servidor.

A função `readServerMessages()` processa as mensagens recebidas, interpretando comandos como **START** (início de partida), **BULLET** (projétil disparado por outro jogador), **HIT** (colisão com projétil), **MODIFIER** (novo modificador gerado) e outros.

Quando ocorrem eventos relevantes (como disparos ou colisões), o cliente envia mensagens ao servidor, que processa a lógica do jogo e notifica os clientes afetados.

# Capítulo 4

## Implementação do Servidor

### 4.1 Estrutura e Gestão de Dados

O servidor foi implementado em **Erlang**, aproveitando as capacidades nativas de concorrência. Para armazenar dados, utilizamos tabelas **ETS** (**Erlang Term Storage**), que proporcionam acesso concorrente eficiente:

- **users**: Armazena informações de utilizadores registados (nome, password, nível, **streak**)
- **players**: Mantém jogadores atualmente ligados e suas conexões
- **waiting\_players**: Lista de jogadores à espera de partida
- **active\_games**: Partidas em curso
- **modifiers**: Modificadores ativos em cada partida

### 4.2 Concorrência em Erlang

A escolha de Erlang para o servidor deveu-se principalmente ao seu modelo de concorrência nativa, baseado em processos leves e troca de mensagens. Este modelo proporciona várias vantagens:

- **Tolerância a falhas**: Os processos Erlang são isolados uns dos outros, permitindo que o servidor continue a funcionar mesmo se um processo específico falhar.
- **Escalabilidade**: Erlang pode gerir milhares de processos concorrentes de forma eficiente, o que é ideal para um servidor de jogo com múltiplas partidas simultâneas.
- **Concorrência sem locks**: O modelo de troca de mensagens de Erlang elimina a necessidade de locks explícitos, reduzindo a possibilidade de deadlocks.

No nosso servidor, cada conexão de cliente é gerida por um processo Erlang dedicado, e processos adicionais são criados para gerir temporizadores de jogo e geração de modificadores. Esta arquitetura permite lidar facilmente com múltiplas partidas simultâneas.

## 4.3 Autenticação e Gestão de Utilizadores

O sistema de autenticação verifica as credenciais fornecidas pelos utilizadores contra os registos armazenados. Ao autenticar-se com sucesso, o servidor envia informações sobre o nível atual e **streak** do jogador.

Para novos utilizadores, o sistema de registo verifica se o nome de utilizador está disponível e se a **password** cumpre requisitos mínimos (pelo menos 3 caracteres). Após o registo, o utilizador começa no nível 1 com **streak** 0.

Todo o histórico de utilizadores é mantido em memória durante a execução do servidor, podendo ser persistido em arquivo para uso futuro.

## 4.4 Sistema de Níveis e Matchmaking

O sistema de níveis foi implementado conforme os requisitos, atualizando o nível e **streak** dos jogadores após cada partida:

- Vitória: Incrementa o **streak** positivo ou reseta streak negativo para 1
- Derrota: Decrementa o **streak** negativo ou reseta streak positivo para -1
- Subida de nível: Ocorre quando o **streak** positivo atinge ou ultrapassa o nível atual
- Descida de nível: Ocorre quando o **streak** negativo atinge ou ultrapassa  $-n/2$ , nunca descendo abaixo de 1

O matchmaking procura jogadores em espera com diferença máxima de um nível. Quando encontra uma correspondência adequada, o servidor cria uma nova partida e notifica ambos os jogadores.

## 4.5 Gestão de Partidas e Simulação

O servidor suporta múltiplas partidas simultâneas, cada uma com seus próprios jogadores, estados e modificadores. Ao iniciar uma partida:

1. Os jogadores são removidos da fila de espera
2. Um ID único é gerado para a partida
3. A pontuação inicial de ambos jogadores é definida como 0
4. Os jogadores são posicionados em lados opostos do mapa
5. Um processo é iniciado para gerar modificadores a intervalos regulares
6. Um temporizador é configurado para encerrar a partida após 2 minutos

A geração de modificadores é controlada para não exceder um número máximo por tipo, distribuindo-os aleatoriamente pelo mapa.

## 4.6 Processamento de Colisões

O servidor processa diferentes tipos de colisões relatadas pelos clientes:

- **Jogador-Projétil:** Adiciona 1 ponto ao atirador e notifica os clientes
- **Jogador-Parede:** Adiciona 2 pontos ao adversário, reposiciona ambos jogadores e notifica os clientes
- **Jogador-Modificador:** Aplica o efeito ao jogador, remove o modificador e notifica os clientes

Para cada tipo de colisão, o servidor atualiza o estado da partida e envia as mensagens apropriadas a todos os jogadores envolvidos.

# Capítulo 5

## Protocolos de Comunicação

A comunicação entre cliente e servidor utiliza um protocolo baseado em mensagens de texto com campos separados por ponto e vírgula. Este protocolo foi projetado para ser simples, eficiente e facilmente extensível.

### 5.1 Principais Mensagens

#### Cliente para Servidor:

- **Autenticação:** LOGIN/REGISTER com credenciais
- **Jogo:** MATCHMAKE, PID (posição), BULLET (disparo), HIT (colisão), WALL\_COLLISION, MODIFIER\_PICKUP, FORFEIT
- **Sistema:** CANCEL\_MATCHMAKING, LEADERBOARD, LOGOUT

#### Servidor para Cliente:

- **Autenticação:** LOGIN\_SUCCESS/FAILED, REGISTER\_SUCCESS/FAILED
- **Jogo:** MATCH\_FOUND, START, BULLET, HIT, MODIFIER, RESET\_POSITIONS, SCORES, END
- **Sistema:** LEADERBOARD com lista de jogadores

Este protocolo permite que todas as ações relevantes sejam comunicadas entre cliente e servidor, mantendo o estado do jogo sincronizado entre todos os participantes.

# Capítulo 6

## Conclusão

O projeto "Duelo" representa uma implementação bem-sucedida de um jogo multijogador utilizando conceitos de programação concorrente. Através da integração de um cliente Java/Processing com um servidor Erlang, conseguimos criar um sistema robusto que suporta múltiplas partidas simultâneas e oferece uma experiência de jogo fluida e interativa.

A implementação de mecanismos de concorrência em ambas as partes do sistema foi fundamental para o sucesso do projeto. No cliente Java, utilizámos uma combinação de threads explícitas, monitores, variáveis de condição e locks para garantir a sincronização adequada entre os diferentes componentes. No servidor Erlang, aproveitámos o modelo de processos leves e troca de mensagens para gerir de forma eficiente múltiplas conexões e partidas simultâneas.

Os principais desafios enfrentados incluíram a sincronização adequada entre cliente e servidor, a implementação do sistema de física e colisões, e a gestão eficiente de múltiplas conexões concorrentes. A escolha de Erlang para o servidor provou-se acertada, facilitando o desenvolvimento de um sistema concorrente robusto.

Todas as funcionalidades requeridas foram implementadas com sucesso: registo e autenticação, sistema de níveis, matchmaking, física de movimento, sistema de projéteis e modificadores, processamento de colisões e tabela de classificação.

O projeto proporcionou uma valiosa experiência prática em programação concorrente e distribuída, demonstrando como diferentes tecnologias podem ser integradas para criar um sistema funcional e interativo. As lições aprendidas sobre arquitetura cliente-servidor, protocolos de comunicação e gestão de estado em sistemas concorrentes serão certamente úteis em projetos futuros.

Em suma, o projeto "Duelo" cumpriu seus objetivos educacionais, proporcionando uma compreensão aprofundada dos princípios e desafios da programação concorrente através de uma aplicação prática e envolvente.