Relatório de Implementação: Jogo Concorrente em Go

Alunos: Henrique Mayer e Vittor Britis

15 de setembro de 2025

1 Introdução

Este relatório descreve o desenvolvimento realizado sobre um código-fonte base de um jogo de terminal, com o objetivo de implementar funcionalidades de concorrência. O trabalho, da disciplina de Fundamentos de Processamento Paralelo e Distribuído, exigia o uso de **goroutines** e **canais** da linguagem Go para criar elementos autônomos e gerenciar o acesso a recursos compartilhados.

O jogo original era puramente sequencial, com um loop de jogo que parava completamente enquanto esperava a entrada do jogador, tornando o ambiente de jogo estático. O trabalho consistiu em refatorar essa estrutura e adicionar novos elementos que funcionassem de forma independente.

2 Desenvolvimento e Implementação dos Requisitos

Para atender às demandas do trabalho, o projeto foi dividido em etapas, focando em cada um dos requisitos de concorrência.

2.1 Estrutura Concorrente e Exclusão Mútua

O primeiro passo foi transformar o loop de jogo bloqueante. A leitura do teclado foi movida para uma goroutine separada, que comunica os eventos por um canal. O loop principal passou a usar um `select` para gerenciar tanto os eventos do teclado quanto um `time.Ticker`, que funciona como um "pulso"para o jogo.

Para evitar **condições de corrida** ao acessar o mapa do jogo (recurso compartilhado), foi implementado um mecanismo de exclusão mútua (mutex) usando um canal com buffer de tamanho 1.

```
1 // Em jogo.go - Implementa  o do Mutex com canal
2 type Jogo struct {
3     lock chan struct{}
4 }
5 func (j *Jogo) Travar()    { j.lock <- struct{}{} }
6 func (j *Jogo) Destravar() { <-j.lock }
```

Todas as partes do código que modificam o mapa são protegidas por chamadas a `jogo.Travar()` e `jogo.Destravar()`, garantindo que apenas uma goroutine manipule o mapa por vez.

2.2 Novos Elementos Concorrentes (3 Tipos)

Foram criados três tipos de elementos que rodam em suas próprias goroutines, com comportamentos distintos:

1. **Patrulheiro:** Um inimigo que se move de forma autônoma pelo mapa. Ele possui uma IA simples que alterna entre patrulhar uma área e perseguir o jogador.
2. **Portal:** Um portão que começa fechado. Ele reage a um sinal (enviado quando o jogador usa a tecla "E"perto dele) e se abre por um tempo limitado.
3. **Armadilha:** Um obstáculo estático que, ao ser tocado pelo jogador, causa o fim imediato do jogo.

2.3 Comunicação, Escuta Múltipla e Timeout

Os requisitos mais complexos de comunicação foram implementados na interação entre o jogador, os inimigos e os portais.

Comunicação e Escuta Múltipla: O inimigo demonstra esses dois conceitos. O loop principal do jogo funciona como um "radar", que envia as coordenadas do jogador para um canal específico do inimigo quando o jogador se aproxima. A goroutine do inimigo usa `select` para escutar dois canais ao mesmo tempo: seu timer de movimento e o canal de notificação do "radar".

```
// Em jogo.go - Logica da goroutine do inimigo
select {
case alvoCoords := <-p.notificacaoCh: // Ouve o alerta do radar
    p.alvo = &alvoCoords // Muda para o modo de perseguicao

case <-tickerMovimento.C: // Ouve o seu proprio timer
    // ... logica de movimento ...
}
```

Comunicação com Timeout: O Portal implementa este requisito. Após receber o sinal para abrir, sua goroutine usa `time.After` dentro de um `select` para criar um temporizador de 5 segundos. Se o tempo esgotar, o portal se fecha automaticamente.

```
// Em jogo.go - Logica de timeout do portal
select {
case <-time.After(5 * time.Second): // Espera 5 segundos
    // ... codigo para fechar o portal ...
}
```

2.4 Condição de Fim de Jogo

Para dar um objetivo ao jogo, foi adicionada uma condição de "Game Over". O jogo termina se o jogador colidir com um inimigo ou pisar em uma armadilha. Isso foi implementado com uma variável booleana `GameOver` no estado do jogo, que é verificada a cada ciclo do loop principal.

3 Conclusão

O desenvolvimento do projeto permitiu aplicar na prática os conceitos de concorrência estudados na disciplina. Todos os requisitos obrigatórios foram cumpridos: foram criados três elementos concorrentes, a exclusão mútua foi garantida com canais, e foram implementados mecanismos de comunicação, escuta múltipla e timeout.

O resultado é um jogo funcional que demonstra claramente a diferença entre um sistema sequencial e um concorrente, onde múltiplos atores interagem de forma independente no mesmo ambiente compartilhado.