Desenvolver Implementação de Jogo Multiplayer Usando RPC em Go

Amanda Wilmsen[*], Killian Domingues[*], Luís Trein[*] e Maria Rita Rodrigues[*]

Escola Politécnica — PUCRS

24 de abril de 2025

Objetivo

Este trabalho tem como objetivo implementar a funcionalidade multiplayer, utilizando Remote Procedure Call (RPC), no jogo desenvolvido no Trabalho 1 da disciplina de Fundamentos de Processamento Paralelo e Distribuído. A implementação teve como base o código disponibilizado pelo professor no Moodle, enquanto a lógica das caixas foi reaproveitada diretamente do código original do primeiro trabalho. O jogo completo está disponível no repositório Trabalho 2.

1. Descrição do Jogo

O jogo consiste em uma arena compartilhada, representada por um mapa bidimensional, onde múltiplos jogadores competem na busca por tesouros escondidos em caixas misteriosas. A **interação ocorre de forma concorrente e síncrona**, exigindo controle rigoroso de **estados compartilhados** e **sincronização de ações entre os jogadores**.

Cada caixa representa um recurso exclusivo: apenas um jogador pode interagir com uma caixa por vez, desde que esteja a exatamente uma célula de distância (cima, baixo, esquerda ou direita). As caixas podem conter um tesouro, uma bomba ou nada.

À medida que os **jogadores interagem com as caixas, elas são removidas do mapa,** alterando dinamicamente o ambiente e exigindo **constante atualização dos clientes conectados**. **O jogador que encontrar o maior número de tesouros vence**. No entanto, caso todos os jogadores acionem bombas e sejam eliminados, o jogo termina com derrota coletiva, incentivando abordagens cooperativas mesmo em um ambiente competitivo.

[1] amanda.wilmsen@edu.pucrs.br
[2] killian.d@edu.pucrs.br
[3] luis.trein@edu.pucrs.br
[4] maria.rita04@edu.pucrs.br

Esse jogo serve como base para a aplicação prática de conceitos como:

- **Concorrência entre múltiplos agentes** sobre recursos limitados;
- **Consistência de estado global** em sistemas distribuídos;
- **Comunicação via RPC** entre processos independentes;

2. Arquitetura do Sistema

2.1. Servidor

O servidor, implementado em **server.go**, opera como um processo em segundo plano, sem interface gráfica. Sua responsabilidade é gerenciar o estado global do jogo, incluindo:

- A estrutura completa do mapa ([][]Celula);
- Posição dos jogadores;
- Estado atual do mapa;
- Conteúdo das caixas (tesouro, bomba ou vazio);
- O controle de comandos já processados para garantir execução única;
- Comunicação com os clientes via RPC.

A lógica principal é conduzida por uma estrutura central (**GameManager**) que contém os dados do jogo, a lista de jogadores remotos, controle dos comandos já processados e mecanismos de sincronização:

```
type GameManager struct {
    jogo                *Jogo
    jogadorID           string
    jogadoresRemotos    map [string]PosicaoJogador
    comandosProcessados map [string]int64
    mutex sync.RWMutex
}
```

Ele atende às requisições dos clientes via RPC, processa a lógica do jogo (e.g., interações com caixas, movimentos dos jogadores) e propaga as atualizações necessárias para manter a consistência entre todos os clientes conectados.

2.2. Cliente

Cada instância do cliente, implementado em **client.go**, é responsável pela interface gráfica e pela interação do jogador. O cliente se conecta ao servidor RPC para inicializar sua sessão de jogo, obter o estado inicial do mapa e enviar comandos de movimento e interação.

Uma característica essencial do cliente é a utilização de uma goroutine que busca periodicamente atualizações do estado do jogo no servidor. Isso garante que a representação local do jogo no cliente esteja sempre sincronizada com o estado global mantido pelo servidor, refletindo as ações de outros jogadores e as mudanças no ambiente.

3. Comunicação e Consistência

Toda comunicação entre cliente e servidor é realizada via chamadas RPC. O cliente inicia todas as interações, e o servidor apenas responde. Para garantir a execução única dos comandos enviados (exactly-once), cada cliente envia um sequenceNumber junto ao comando. O servidor mantém um mapa com o último comando processado para cada jogador:

```
comandosProcessados map[string]int64
```

Antes de processar um comando, o servidor verifica:

```
if sequenceNumber <= gm.comandosProcessados[jogadorID] {
        return
}
```

Essa checagem impede a reexecução de comandos retransmitidos devido a falhas de rede ou atrasos.

4. Novos Elementos e Interações

Para viabilizar a transição de um jogo single-player para um ambiente multiplayer distribuído, diversos componentes foram desenvolvidos ou adaptados, focando na concorrência, sincronização e consistência dos dados entre os diferentes nós (clientes e servidor):

- **Sincronização de múltiplos jogadores**: As posições dos jogadores são armazenadas no servidor. A cada ciclo de atualização (execução periódica no cliente), o servidor coleta as informações de todos os jogadores conectados e as transmite para cada cliente. Isso é feito por meio de chamadas RPC "pull", onde os clientes solicitam o estado atual do jogo. No servidor, uma estrutura de dados (ex: o map[string]*Jogador) é mantida para armazenar dinamicamente a posição, status e quaisquer outros atributos relevantes de cada jogador ativo.

- **Atualização global do mapa**: Mudanças significativas no estado do mapa, como a remoção de caixas após a interação (abertura ou remoção), são gerenciadas exclusivamente pelo servidor. Após uma interação bem sucedida, o servidor atualiza seu próprio estado do mapa e, de forma semelhante à sincronização de jogadores, repassa essas mudanças para todos os clientes conectados. Isso garante que todos os jogadores tenham a mesma visão de caixas/ambientes no mapa.

- **Controle de concorrência**: Para garantir a integridade dos dados no ambiente multiplayer, foi necessário controlar o acesso simultâneo a recursos compartilhados, como as posições dos jogadores (GameServer.jogadores) e as caixas (caixasGlobal). Para isso, foi usada a sync.RWMutex, chamada mutex dentro da struct GameServer:

    - Leituras simultâneas (com RLock()), usadas em operações como ObterPosicoes, que apenas consultam o estado do jogo);

    - Escritas exclusivas (com Lock, usadas em funções que alteram o estado, como ConectarJogo, Mover, InteragirCaixa e Desconectar;

        - Antes de qualquer modificação, a função adquire o Lock() e libera com Unlock() (via defer). Isso evita condições de corrida e garante que apenas

uma requisição modifique os dados por vez. E mesmo que caixasGlobal seja uma variável global, ela é protegida pela mesma mutex, já que suas alterações ocorrem apenas dentro de métodos RPC controlados.

- **Execução periódica**: No lado do cliente, uma goroutine assíncrona foi implementada para realizar chamadas RPC periódicas ao servidor. Essa goroutine é responsável por buscar o estado mais recente do jogo (incluindo posições dos jogadores e estado do mapa) em intervalos regulares.

5. Dificuldades de Implementação

Durante a implementação do jogo, enfrentamos uma dificuldade crítica relacionada à manipulação das caixas no servidor. Identificamos que o problema pode ter sido causado pela coexistência de duas representações distintas das caixas: uma variável global caixasGlobal e o campo caixas dentro da struct GameServer. Embora o campo da struct tenha sido corretamente inicializado, ele não chegou a ser efetivamente utilizado — enquanto a variável global era acessada e modificada diretamente por diferentes partes do código, inclusive dentro de métodos protegidos por mutex.

Essa duplicidade resultou em uma condição de corrida, comprometendo a consistência da lógica de interação com as caixas. Como possível solução, consideramos concentrar todo o controle das caixas exclusivamente dentro da struct GameServer, assegurando a sincronização por meio de mutex. No entanto, não conseguimos reestruturar completamente o código a tempo da entrega, e tampouco temos certeza absoluta de que essa seja a causa exata do problema. Como consequência, observamos comportamentos inesperados durante as interações dos jogadores com o cenário, como a mensagem “não há nada para interagir aqui”, mesmo quando há uma caixa na coordenada acessada.

6. Compilar e Executar

Servidor: Go run . -server

Cliente: Go run .