Relatório de Implementação de Jogo Multiplayer Usando RPC em Go

Nome: Guilherme Antonio Argilar

1. Introdução

Este relatório detalha o processo de modificação de um jogo de terminal de jogador único para uma arquitetura multiplayer funcional. O objetivo principal foi permitir que múltiplos jogadores se conectem a um servidor central e interajam em um mesmo ambiente de jogo em tempo real.

Para alcançar este objetivo, o projeto original foi reestruturado para um modelo cliente-servidor, e a comunicação entre as partes foi implementada utilizando o pacote de chamada de procedimento remoto (*net/rpc*) nativo da linguagem Go.

2. Justificativa da Reestruturação do Projeto

A transição de um modelo single-player para multiplayer exigiu uma mudança fundamental na arquitetura do software. A estrutura original, com todos os arquivos em um único diretório, era adequada para um programa monolítico, mas insuficiente para a nova arquitetura de rede.

A nova estrutura de pastas foi adotada para garantir a **Separação de Responsabilidades**: A divisão do projeto nas pastas **client**/, **server**/ e **comum**/ isola as responsabilidades de cada componente.

- O servidor agora lida exclusivamente com a lógica de jogo e o estado global.
- O cliente é responsável apenas pela interface do usuário e pela captura de input.
- O pacote **comum** evita a duplicação de código, centralizando as estruturas de dados usadas na comunicação.

3. Mapeamento das Funcionalidades Originais

A lógica do jogo original não foi descartada, mas sim migrada e adaptada para a nova arquitetura, com nomes de funções e tipos traduzidos para o português para manter a consistência.

- **main.go** (Original)
  O ponto de entrada único foi dividido em dois:
  - **server/main.go**: Contém a lógica de inicialização do servidor (**NovoServidorJogo**), o registro do serviço RPC e o loop que aguarda por conexões de clientes.

    - **client/main.go**: Contém a inicialização da interface com **termbox**, a conexão com o servidor e o loop principal que processa os eventos de teclado do jogador.

- **jogo.go** (Original)
  As responsabilidades de gerenciamento de estado foram transferidas para o servidor:
    - **jogoCarregarMapa**: A lógica foi movida para o método **carregarMapa** da **struct ServidorJogo** em **server/main.go**.
    - **jogoPodeMoverPara**: A lógica de validação de movimento agora está no método **podeMoverPara** em **server/main.go**, que também verifica a posição de outros jogadores.
    - As **structs Elemento** e **Jogo** foram adaptadas. **Elemento** foi mantida em **comum/tipos.go** e a antiga **struct Jogo** foi substituída pela **struct EstadoJogo**, que representa o estado global.

- **interface.go** (Original)
  Toda a lógica de interação com a biblioteca **termbox** foi movida para o cliente:
    - As funções de desenho e atualização da tela estão agora dentro do método **desenhar** da **struct ClienteJogo**, em **client/main.go.**

- **personagem.go** (Original)
  A lógica de ação do personagem foi dividida entre cliente e servidor:
    - Captura de Input: O **client/main.go** detecta as teclas pressionadas (W, A, S, D).
    - Execução da Ação: Ao invés de mover o personagem localmente, o cliente agora, através do método **mover**, faz uma chamada RPC para o método **Mover** no **ServidorJogo**. O servidor então valida e aplica a mudança de posição no estado global do jogo.

4. Novos Elementos e Interações Implementadas

Para atender aos requisitos do modo multiplayer, os seguintes elementos foram implementados:

- Comunicação via RPC: A comunicação é feita através de três métodos RPC principais expostos pelo **ServidorJogo**:
    - **ServidorJogo.Conectar**: Chamado quando um novo cliente se junta ao jogo. O servidor cria um novo **Jogador**, atribui um **IDJogador** e retorna o **EstadoJogo** atual para o cliente.
    - **ServidorJogo.Mover:** Chamado pelo cliente para solicitar um movimento.
    - **ServidorJogo.ObterEstado:** Chamado periodicamente pelo cliente para obter a versão mais recente do **EstadoJogo** (incluindo a posição de todos os outros jogadores).

- Polling de Estado pelo Cliente: No **client/main.go**, a função **buscarEstadoPeriodicamente** é executada em uma *goroutine* separada. Ela é responsável por pedir atualizações ao servidor a cada 100 milissegundos, garantindo que o jogador veja os movimentos de outros jogadores quase em tempo real.

- Garantia de Execução Única (*Exactly-Once*): Para evitar que um mesmo comando de movimento seja executado múltiplas vezes devido a retransmissões na rede, foi implementado um sistema de número de sequência.
    - A **struct ArgsMovimento** em **comum/tipos.go** inclui o campo **NumeroSequencia.**
    - O cliente incrementa este número a cada novo comando **Mover** que envia.
    - O servidor, na **struct ServidorJogo,** mantém um mapa **ultimoCmdProcessado** que armazena o último número de sequência processado para cada jogador. Qualquer comando com um **NumeroSequencia** igual ou inferior ao último registrado é ignorado, garantindo a execução única.

- Para evitar que "jogadores fantasmas" permaneçam no mapa, foi implementado um sistema duplo para gerenciar o ciclo de vida da conexão:
    - **Desconexão por Inatividade (Heartbeat):**
        - O cliente, através da goroutine **enviarPingPeriodicamente**, envia um "ping" ao servidor a cada 5 segundos para sinalizar que ainda está ativo.
          O servidor possui a goroutine **verificarJogadoresAtivos**, que remove qualquer jogador que não tenha enviado um ping nos últimos 15 segundos.

    - **Desconexão Explícita:**
        - Quando o jogador pressiona a tecla **ESC** para sair, o cliente faz uma chamada RPC final ao método **ServidorJogo.Desconectar**.
        - Este método remove imediatamente todos os dados do jogador (posição, ping, etc.) do servidor, garantindo uma saída limpa e instantânea.