# Implementação de Jogo Multiplayer Usando Go e RPC

## *Introdução*

Este documento descreve a especificação para modificar um jogo de jogador único escrito em Go para um ambiente multiplayer utilizando RPC. O objetivo é permitir que múltiplos jogadores se conectem a um servidor central e joguem no mesmo mapa, cada um visualizando o mapa em seu próprio computador. Nesta arquitetura, apenas o servidor terá objetos remotos. A comunicação sempre será iniciada pelos clientes, que irão periodicamente buscar atualizações do estado do jogo a partir do servidor.

O trabalho deve ser desenvolvido a partir do código-fonte do Trabalho 1 da disciplina ou a partir da nova versão de código que será disponibilizada pelo professor no Moodle. Não serão aceitos trabalhos nos quais a implementação foi realizada do zero, a menos que seja previamente autorizado pelo professor.

## *Arquitetura do Sistema*

**Servidor de Jogo (GameServer):**

- O servidor é responsável por gerenciar o estado do jogo.
- Ele armazena a matriz do mapa e a posição de todos os elementos não estáticos do jogo, como jogadores, inimigos, itens, etc.
- O servidor também gerencia a comunicação entre os clientes (jogadores).
- Não contém interface gráfica, sendo um processo em segundo plano que mantém o estado do jogo atualizado e responde às solicitações dos clientes.

**Cliente do Jogo (GameClient):**

- O cliente possui a interface gráfica onde o jogador interage com o jogo.
- Ele se conecta ao servidor para obter o estado do jogo e envia comandos de movimento e interação ao servidor.
- Cada cliente possui uma thread dedicada para buscar periodicamente atualizações do estado do jogo no servidor e atualizar o seu estado local.
- O cliente renderiza o estado atual do jogo em sua própria interface gráfica, atualizando a visão do jogador com base nas informações recebidas do servidor.

## *Componentes e Interfaces*

**Interfaces RPC:**

- **GameServerInterface:** Define os métodos que o servidor deve implementar. Os métodos sugeridos são:
    - RegisterClient(args *RegisterArgs, reply *RegisterReply) error - Registra um novo cliente no servidor.
    - SendCommand(args *CommandArgs, reply *CommandReply) error - Recebe comandos do cliente (movimento, interação).
    - GetGameState(args *GameStateArgs, reply *GameStateReply) error - Obtém o estado atual do jogo.

**Estado do Jogo:**

- O estado do jogo contém  a posição de todos os elementos dinâmicos, como jogadores e inimigos.
- O estado também inclui informações sobre a vida dos jogadores e outros elementos relevantes do jogo.

**Servidor de Jogo (Implementação):**

- O servidor deve manter uma lista de clientes conectados e suas posições no mapa.
- Ele deve processar comandos recebidos dos clientes, como movimentos e interações, atualizando o estado do jogo de acordo.
- O servidor deve responder às solicitações de estado do jogo feitas pelos clientes.

**Cliente do Jogo (Implementação):**

- Cada cliente deve se registrar no servidor ao iniciar.
- O cliente deve enviar comandos ao servidor sempre que o jogador realizar uma ação (por exemplo, mover-se ou interagir com um item).
- O cliente deve possuir uma thread dedicada que periodicamente solicita atualizações do estado do jogo ao servidor.
- O cliente deve atualizar a interface gráfica do jogador com base no estado do jogo recebido do servidor.

***Execução do Servidor e Cliente***

**Iniciando o Servidor:**

- O servidor deve ser iniciado primeiro, configurando um servidor RPC e registrando a instância do servidor nele.
- O servidor deve então aguardar conexões de clientes e processar comandos conforme forem recebidos.

**Iniciando o Cliente:**

- O cliente deve se conectar ao servidor RPC para obter uma referência ao servidor de jogo.

- Após a conexão, o cliente deve se registrar no servidor e iniciar a interface gráfica do jogo.
- Durante o jogo, o cliente envia comandos ao servidor e, através de uma thread dedicada, busca periodicamente atualizações do estado do jogo.

***Requisitos de Idempotência***

Para garantir a semântica de execução exata de comandos (exactly-once), o método SendCommand deve incluir um identificador único (sequenceNumber) para cada comando enviado pelo cliente. Os seguintes requisitos de idempotência devem ser implementados:

**Identificação Única de Comandos:**

- Cada comando enviado por um cliente deve incluir um sequenceNumber único que incrementa sequencialmente. Este número deve ser mantido pelo cliente e nunca deve ser repetido para garantir a identificação única de cada comando.

**Armazenamento de Estado do Comando no Servidor:**

- O servidor deve manter um registro do último sequenceNumber processado para cada cliente. Se um comando com um sequenceNumber já processado for recebido, ele deve ser ignorado para evitar duplicação de execução.

**Processamento Idempotente de Comandos:**

- O servidor deve garantir que a execução de um comando seja idempotente. Isto significa que a execução repetida de um mesmo comando (com o mesmo sequenceNumber) deve resultar no mesmo estado final do jogo, sem causar efeitos adicionais.

***Entrega***

O trabalho pode ser realizado em grupos de até 3 integrantes desde que a equipe esteja cadastrada como “grupo” no Moodle. A entrega deverá ser feita como um arquivo “.zip” contendo os arquivos fonte desenvolvidos, bem como informações para sua compilação e execução. Escreva também um pequeno relatório explicando os novos elementos e interações implementadas. Atente para os prazos de entregas e apresentação definidos no Moodle.