Fundamentos de Processamento Paralelo e Distribuído

Relatório do Trabalho 2
Programação Distribuída

Arthur Testa(19180759), Lívia Nöer(24102422), Luísa Zarth(24102448) e Luthero Vargas(24102458)

Link Github: https://github.com/LuisaZarth/T2_FPPD.git

1. Introdução

O Trabalho 2 da disciplina de Fundamentos de Processamento Paralelo e Distribuído tem como objetivo estender o jogo desenvolvido no Trabalho 1 (versão single-player) para um ambiente multiplayer distribuído, a aplicar os principais conceitos de concorrência, comunicação entre processos e tolerância a falhas.

Nesta nova versão, cada jogador executa uma instância independente do cliente em seu próprio computador e interage com um servidor central, responsável por manter o estado global do jogo. O servidor gerencia as sessões e as informações compartilhadas (lista de jogadores e suas posições), sem executar a lógica interna de movimentação ou regras. Todo o processamento permanece no cliente, que envia comandos e busca atualizações de forma periódica.

A comunicação entre cliente e servidor é realizada por meio de chamadas de procedimento remoto (RPC), implementadas sobre o protocolo TCP usando a biblioteca padrão net/rpc do Go. Todas as interações são iniciadas pelos clientes, que enviam comandos de atualização de estado e realizam requisições de leitura ao servidor.

Para garantir consistência e evitar duplicação de comandos em cenários de retransmissão, foi implementado um mecanismo de garantia de execução única (exactly-once). Cada comando que altera o estado do servidor contém um número de sequência associado, e o servidor mantém um controle interno dos números de sequência já processados por cliente, assegurando que cada atualização seja aplicada apenas uma vez.

Com essa abordagem, o projeto alcança um modelo de arquitetura distribuída simples e funcional, com separação clara entre responsabilidades:

- Servidor de jogo: responsável por gerenciar o estado global e responder às requisições RPC.
- Cliente de jogo: responsável pela lógica de movimentação e sincronização periódica do estado

2. Objetivo

O principal objetivo deste trabalho é transformar o jogo single-player desenvolvido no Trabalho 1 em uma aplicação multiplayer distribuída, utilizando a linguagem Go e o modelo de chamadas de procedimento remoto (RPC – Remote Procedure Calls) para comunicação entre processos.

A proposta consiste em permitir que múltiplos jogadores se conectem simultaneamente a um servidor central, responsável por gerenciar e sincronizar o estado global do jogo — incluindo informações como posições dos jogadores —, enquanto toda a lógica de movimentação, regras e interface permanece exclusivamente no cliente.

Dessa forma, cada cliente executa localmente o processamento do jogo e se comunica periodicamente com o servidor para enviar comandos e obter atualizações do estado compartilhado. Toda a comunicação é iniciada pelos clientes, e o servidor atua apenas como intermediário, respondendo às requisições recebidas.

Um aspecto central do trabalho é a implementação da garantia de execução única (exactly-once) para os comandos que modificam o estado do servidor. Essa garantia assegura que cada comando seja aplicado apenas uma vez, mesmo em casos de retransmissão ou falha de rede, reforçando a robustez e a consistência da comunicação RPC em um ambiente distribuído.

3. Desenvolvimento

Servidor RPC

O servidor foi implementado em Go usando o pacote net/rpc e escuta conexões na porta 1234. Sua estrutura central, GameServer, contém um mutex (sync.Mutex) para garantir exclusão mútua, um mapa players que armazena as posições dos jogadores e um mapa processed que guarda o último número de sequência (SeqNum) processado por cada cliente, garantindo execução única (exactly-once).

Responsabilidades do servidor:

- Manter o estado global dos jogadores.
- Não processar lógica de jogo, apenas sincronizar dados.
- Responder às requisições RPC dos clientes.
- Garantir execução única dos comandos usando SeqNum.

Métodos principais:

1. `RegisterPlayer` – adiciona um novo jogador ao mapa players. Protege o acesso com mutex para evitar condições de corrida quando vários clientes se conectam simultaneamente.
2. `UpdatePlayerState` – atualiza a posição de um jogador, verificando se o comando é novo ou duplicado. Se o SeqNum for menor ou igual ao último processado, ignora (duplicata); se for maior, aplica a atualização e grava o novo valor em processed.
3. `GetGameState` – retorna uma cópia do estado global, garantindo leitura consistente durante atualizações.
4. `UnregisterPlayer` – remove um jogador quando ele desconecta, limpando o estado.

Cada conexão aceita é tratada em uma goroutine separada, permitindo que múltiplos clientes interajam simultaneamente. O uso de mutexes garante atomicidade em todas as operações críticas, prevenindo data races mesmo sob alta concorrência. Além disso, o servidor gera logs para cada requisição, facilitando a depuração e validação durante os testes.

Módulo Compartilhado

Foi criado um pacote shared contendo as estruturas comuns de comunicação RPC, entre elas:

- `PlayerState` (ID, linha e coluna do jogador),
- `MoveArgs` (PlayerID, nova posição e SeqNum),
- `MoveReply` (campo booleano Applied indicando se o comando foi aplicado ou ignorado),
- `GameState` (representa o mapa global de jogadores)

Ter essas definições centralizadas garante compatibilidade entre cliente e servidor, evitando erros de serialização e assegurando que ambos usem os mesmos tipos de mensagem.

Cliente

O cliente, derivado do jogo single-player do Trabalho 1, foi adaptado para operar de forma distribuída. Agora, a cada movimento, ele envia uma chamada RPC `UpdatePlayerState` ao servidor, anexando um contador de sequência local (SeqNum) que é incrementado a cada novo comando.

Se a comunicação falhar, o cliente tenta retransmitir automaticamente o mesmo comando (mantendo o mesmo SeqNum) até três vezes, garantindo confiabilidade sem duplicar ações. Em caso de resposta Applied=false, o cliente entende que a ação já foi processada e segue normalmente. Além disso, o cliente executa um polling periódico (a cada 200 ms) chamando `GetGameState` para atualizar as posições dos demais jogadores, a garantir sincronização em tempo real.

A renderização continua sendo feita com a biblioteca Termbox, e todos os jogadores são desenhados simultaneamente na interface. Ao encerrar o programa, o cliente chama `UnregisterPlayer` para informar ao servidor que está desconectando, garantindo uma saída limpa e remoção imediata do jogador da lista global.

Estratégia de Execução Única (Exactly-Once)

A garantia de execução única foi implementada através de um controle distribuído entre cliente e servidor. No cliente, o SeqNum é incrementado a cada comando novo e mantido fixo durante as retransmissões. No servidor, o mapa processed armazena o último número processado por jogador. E ao receber uma atualização, caso o SeqNum seja maior que o último processado, o comando é aplicado e o valor atualizado. Se SeqNum for menor ou igual ao último processado, o comando é considerado duplicado e ignorado. Tudo ocorre sob o mesmo lock de mutex, tornando o processo atômico e livre de condições de corrida.

Os testes práticos confirmaram que a retransmissão de comandos é ignorada corretamente e que o estado global permanece consistente mesmo com múltiplos clientes.

4. Conclusão

O desenvolvimento deste trabalho proporcionou uma aplicação prática dos conceitos de sistemas distribuídos, especialmente no contexto de comunicação entre processos, concorrência e tolerância a falhas. A transição de um jogo single-player para um ambiente multiplayer RPC exigiu a implementação de uma arquitetura cliente-servidor robusta e bem

definida, onde o servidor atua como coordenador global e os clientes executam o processamento independente.

A abordagem adotada garantiu separação clara de responsabilidades, simplicidade no design e consistência no compartilhamento de informações. O servidor manteve-se leve e eficiente, centralizando apenas o estado global, enquanto os clientes ficaram responsáveis pela renderização e pelas regras de movimentação. Essa estrutura modular não só simplificou o desenvolvimento como também reforçou a importância de arquiteturas desacopladas em sistemas distribuídos.

O mecanismo de execução única (exactly-once) foi um dos elementos mais relevantes do projeto. Ele garantiu que cada comando fosse aplicado uma única vez, mesmo em cenários de retransmissão ou falha de comunicação, reforçando a integridade do sistema. Essa técnica é essencial em aplicações distribuídas reais, onde a confiabilidade das mensagens nem sempre é garantida.

Além disso, o uso das bibliotecas padrão do Go — com net/rpc para comunicação e sync para controle de concorrência — facilitou a implementação e mostrou o poder das abstrações de alto nível da linguagem para lidar com paralelismo e sincronização. O uso de goroutines permitiu que o servidor e os clientes lidassem com múltiplas conexões simultâneas sem perda de desempenho ou travamentos.

Em suma, o trabalho demonstrou que é possível aplicar, de maneira prática e controlada, os fundamentos de processamento paralelo e distribuído em uma aplicação real, atingindo um equilíbrio entre simplicidade, desempenho e consistência. O resultado final foi um jogo multiplayer funcional e estável, capaz de ilustrar de forma concreta os princípios teóricos da disciplina — desde a comunicação RPC até o controle de concorrência e a execução confiável em ambientes distribuídos.