FACULDADE MULTIVIX VITÓRIA

Alunos:

- Arthur Alexandre Ribeiro

- Caio Zottele Mendes

- João Pedro Victor Dias

- Thor Perini Merçon

Professor: Breno Krohling

PROGRAMAÇÃO DISTRIBUÍDA E PARALELA

Vitória, ES

08/11/2025

1. Introdução

Este relatório detalha a implementação e os resultados obtidos na avalição processual da disciplina de Programação Paralela e Distribuída, focado na experimentação de sistemas distribuídos baseados na arquitetura Cliente/Servidor. O objetivo central foi aplicar o conceito de Chamada de Procedimento Remoto (RPC) para construir duas aplicações distintas.

O escopo do trabalho foi dividido em duas atividades principais:

1. **Atividade 1 (Calculadora):** Adaptação de um exemplo base para criar uma calculadora com as quatro operações básicas (soma, subtração, multiplicação e divisão), acessível via RPC.
2. **Atividade 2 (Minerador):** Construção de um protótipo de mineração de criptomoedas, utilizando Python e a biblioteca gRPC, simulando um processo de desafio criptográfico (Proof-of-Work) entre um servidor e múltiplos clientes.

Ambas as atividades foram desenvolvidas utilizando a linguagem Python e a biblioteca gRPC, aproveitando o Protobuf para a serialização eficiente dos dados.

2. Metodologia de Implementação

2.1. Atividade 1: Calculadora RPC

Para esta atividade, foi implementado um serviço RPC simples que expõe as quatro operações matemáticas básicas.

- Definição do Serviço (Protobuf): Foi definido um serviço RPC (inferido dos arquivos grpcCalc_pb2.py e grpcCalc_pb2_grpc.py) contendo quatro métodos: add, sub, mul e div. Cada método foi configurado para receber uma mensagem contendo dois operandos numéricos (numOne, numTwo) e retornar uma mensagem com o resultado (num).
- Lógica do Servidor (grpcCalc_server.py):
    - O servidor implementa um CalculatorServicer que escuta por conexões na porta 8080, utilizando um ThreadPoolExecutor para lidar com requisições concorrentes.
    - Os métodos add, sub e mul executam as operações aritméticas simples e retornam o resultado.
    - O método div implementa uma validação crítica: ele verifica se o divisor (request.numTwo) é zero. Caso afirmativo, a operação não é realizada e o servidor retorna um status de erro gRPC INVALID_ARGUMENT, com a mensagem "Divisão por zero", para o cliente.

- Lógica do Cliente (grpcCalc_client.py):
    - O cliente implementa um menu interativo em loop, que permite ao usuário selecionar a operação desejada (1-Soma, 2-Subtração, 3-Multiplicação, 4-Divisão) ou sair (0).
    - Após a escolha, o cliente coleta os dois operandos (tratados como float).
    - A aplicação cliente possui tratamento de exceções para grpc.RpcError, permitindo capturar e exibir de forma controlada os erros retornados pelo servidor, como a tentativa de divisão por zero.
    - *Obs: O cliente também utiliza a biblioteca pybreaker para implementar o padrão Circuit Breaker, aumentando a resiliência da aplicação a falhas do servidor.*

2.2. Atividade 2: Protótipo de Minerador de Criptomoedas

Esta atividade simulou um processo de mineração, onde clientes competem para resolver um desafio criptográfico proposto pelo servidor. A implementação foi focada em robustez, observabilidade e performance de concorrência.

- Arquitetura e Estrutura de Dados:
    - O sistema segue o modelo cliente/servidor. O servidor (MinerAPI) gerencia os desafios e um ou mais clientes mineradores se conectam via RPC para tentar solucioná-los.
    - O servidor mantém uma tabela de transações em memória (TxTable), composta por: TransactionID (int), Challenge (int, nível de dificuldade), Solution (string) e Winner (int, ClientID do vencedor ou -1 se pendente).
- Definição do Serviço (grpcCalc.proto):
    - A interface RPC foi definida no arquivo .proto conforme a especificação. Os métodos RPC incluem getTransactionID, getChallenge, getTransactionStatus, submitChallenge, getWinner e getSolution.
- Lógica do Servidor (grpcCalc_server.py):
    - Observabilidade (Logging): Para monitorar a saúde e o comportamento do sistema, foi implementado um LoggingInterceptor do gRPC. Este interceptor registra automaticamente todas as chamadas RPC de entrada e saída, detalhando o cliente (peer), o método, os parâmetros (transactionID, clientID), o status de retorno e o tempo de execução

em milissegundos. Os logs são direcionados tanto para o console quanto para um arquivo rotativo (server.log).

- Controle de Concorrência: Para garantir a integridade da tabela de transações sob múltiplas requisições concorrentes, foi utilizada a classe TxTable com um threading.Lock. A lógica de *lock* foi otimizada para performance: as funções críticas (como get_current_id e resolve) mantêm o *lock* apenas pelo tempo mínimo necessário para ler ou atualizar o estado. A geração de uma nova transação (_new_tx), que é uma operação mais lenta, é chamada *após* a liberação do *lock* principal, evitando contenção e *deadlocks*.
- Validação (Proof-of-Work): A validação da solução (valid_solution) é feita verificando se o *hash* SHA-1 da string proposta (formato txid:clientID:nonce) inicia com N zeros hexadecimais, onde N é o difficulty do desafio.
- Justificativa de Metodologia: A dificuldade (Challenge) foi limitada a um máximo de 7 (em vez de 20). Esta decisão metodológica foi tomada para garantir que os testes pudessem ser executados em tempo hábil em ambientes locais (CPU), preservando o comportamento exponencial da prova de trabalho em um escopo acadêmico.

- <u>Lógica do Cliente (grpcCalc_client.py):</u>
  - Configuração e Resiliência: O cliente utiliza a biblioteca argparse para permitir a configuração dinâmica do clientID, do endereço do servidor e do número de threads via linha de comando. Além disso, o canal gRPC é configurado com opções de keepalive para garantir uma conexão mais estável com o servidor.
  - Processo "Mine" (Paralelismo Local): A função de mineração (mine_locally) foi implementada para performance e eficiência.
    1. Ela utiliza os.cpu_count() para definir o número ideal de *threads* (workers) caso não seja especificado.
    2. Um threading.Event é usado como sinalizador para parar todas as *threads* de mineração imediatamente assim que uma delas encontra a solução.
    3. Uma queue.Queue é usada para que a *thread* vencedora possa depositar a solução, e a *thread* principal possa recuperá-la de forma *thread-safe*.
    4. Ao encontrar, o cliente submete a solução ao servidor usando submitChallenge().

3. Testes e Resultados Encontrados

Ambas as aplicações foram testadas para validar a implementação e o atendimento aos requisitos, conforme demonstrado no vídeo de execução.

3.1. Resultados da Atividade 1 (Calculadora)

O cliente da calculadora foi executado e testado com os seguintes cenários para validar todas as operações:

- Teste de Soma: A opção 1 foi selecionada.
    - Entrada: Número 1: 123132, Número 2: 4523.
    - Saída: Resultado: 127655.0.
- Teste de Subtração: A opção 2 foi selecionada.
    - Entrada: Número 1: 100, Número 2: 32.
    - Saída: Resultado: 68.0.
- Teste de Multiplicação: A opção 3 foi selecionada.
    - Entrada: Número 1: 23, Número 2: 65.
    - Saída: Resultado: 1495.0.
- Teste de Divisão Válida: A opção 4 foi selecionada.
    - Entrada: Número 1: 666, Número 2: 6.
    - Saída: Resultado: 111.0.

3.2. Resultados da Atividade 2 (Minerador)

Para validar a robustez e o controle de concorrência do minerador, foi executado um teste com o servidor e dois clientes (clientID=1 e clientID=2) conectados simultaneamente, simulando uma disputa, conforme demonstrado no vídeo de execução.

Uma decisão metodológica foi crucial para viabilizar estes testes: a especificação original sugeria um desafio de [1..20], mas o Challenge foi limitado a um máximo de 7. Isso garantiu que a mineração pudesse ser resolvida em tempo hábil em ambientes de CPU locais, permitindo a execução e observação de cenários de concorrência, como a disputa entre o Cliente 1 e o Cliente 2.

Além disso, a implementação de um LoggingInterceptor no servidor foi essencial para a análise dos resultados. Como observado no vídeo, os logs (em server.log e no console) permitiram validar a ordem exata dos eventos de concorrência: o log confirmou o recebimento da solução do cid=1, o registro do vencedor, a criação da nova tx 1 e, subsequentemente, a rejeição da submissão do cid=2, registrando o tempo exato de cada chamada RPC.