FACULDADE MULTIVIX

BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

ELYAN VINÍCIUS FERREIRA LOPES - 2213858

BRENO MERGH - 1411870

LUIZ FERNANDO SOTERIO S. DE OLIVEIRA - 2214753

GUILHERME FERNANDES DIAS - 2314073

PAULO HENRIQUE LIRA DE ANDRADE - 1714490

LABORATÓRIO II – CHAMADA DE PROCEDIMENTO REMOTO

Vitória-ES

2025

ELYAN VINÍCIUS FERREIRA LOPES - 2213858

BRENO MERGH - 1411870

LUIZ FERNANDO SOTERIO S. DE OLIVEIRA - 2214753

GUILHERME FERNANDES DIAS - 2314073

PAULO HENRIQUE LIRA DE ANDRADE - 1714490

LABORATÓRIO II – CHAMADA DE PROCEDIMENTO REMOTO

Atividade Processual da disciplina de Programação Paralela e Distribuida apresentado à Universidade Multivix de Vitória, como requisito para o recebimento da atividade processual.

Orientador(a):Prof. Breno Krohling

Vitória-ES

2025

1. INTRODUÇÃO

Este documento apresenta a consolidação do desenvolvimento e das validações realizadas para a avaliação processual da disciplina de Programação Distribuída e Paralela. O foco do trabalho foi a aplicação prática de sistemas distribuídos empregando a arquitetura Cliente/Servidor. O propósito fundamental foi empregar o paradigma de Chamada de Procedimento Remoto (RPC) para a elaboração de duas aplicações funcionais.

O escopo do projeto foi segregado em duas atividades principais:

- Atividade 1 (Calculadora): Ajuste de um código fundamental para estabelecer uma calculadora com as quatro operações aritméticas básicas (adição, subtração, multiplicação e divisão), operável via RPC.
- Atividade 2 (Minerador): Elaboração de um protótipo de mineração de criptomoeda, utilizando Python e gRPC, que emula um mecanismo de Prova de Trabalho (Proof-of-Work) entre um nó servidor e múltiplos clientes.

2. METODOLOGIA DE IMPLEMENTAÇÃO

A seguir, são descritas as arquiteturas e lógicas de implementação de cada atividade.

2.1. ATIVIDADE 1: CALCULADORA RPC

Nesta atividade, foi desenvolvido um serviço RPC elementar para expor as quatro operações matemáticas.

- Interface do Serviço (.proto): A interface foi definida pelo serviço api (no arquivo .proto), que declara os métodos add, sub, mul e div. O serviço utiliza a mensagem args (transportando numOne e numTwo) para entrada e a mensagem result (transportando num) para saída.
- Implementação do Servidor (grpcCalc_server.py): A lógica do serviço reside na classe CalculatorService, que implementa a interface apiServicer. A inicialização do servidor (função start_server) utiliza um ThreadPoolExecutor para gerenciar múltiplas requisições simultaneamente. O método div possui tratamento de exceção para divisão por zero, retornando um código INVALID_ARGUMENT ao cliente caso request.numTwo seja nulo.
- Implementação do Cliente (grpcCalc_client.py): A função start_client gerencia o loop de interação com o usuário. Para resiliência, o padrão *Circuit Breaker* (usando pybreaker) foi implementado para lidar com falhas de conexão. O código captura as entradas num1 e num2 e as envia ao stub. O cliente trata especificamente grpc.RpcError (para erros lógicos do servidor) e pybreaker.CircuitBreakerError (para falhas de conexão).

2.2. ATIVIDADE 2: PROTÓTIPO DE MINERADOR DE CRIPTOMOEDAS

Esta atividade consistiu em simular um processo de mineração onde clientes competem para solucionar um desafio criptográfico.

- Arquitetura e Gerenciamento de Estado: O sistema emprega um modelo cliente/servidor onde o MinerAPI (servidor) gerencia os desafios. O estado é mantido em memória pela classe TxTable, que emprega um threading.Lock para assegurar a atomicidade das operações (como _new_tx, get_current_id e resolve) e prevenir condições de disputa (*race conditions*).
- Interface do Serviço (.proto): O serviço MinerAPI (definido no .proto) define os pontos de entrada para a interação cliente-servidor, incluindo getTransactionID, getChallenge e submitChallenge.
- Implementação do Servidor (grpcCalc_server.py): A lógica de negócio reside na classe MinerServicer, que utiliza uma instância global (TXS) da TxTable para consultar e alterar o estado. A validação da Prova de Trabalho (função valid_solution) confere se o *hash* SHA-1 da solução candidata possui o prefixo de zeros exigido pela dificuldade (limitada internamente a 7). A função serve inicializa o servidor com um ThreadPoolExecutor.

- Decisão de Projeto (Dificuldade): O limite de dificuldade (Challenge) foi fixado em 7 (em vez de 20). Esta decisão metodológica foi tomada para viabilizar a execução de testes de mineração em tempo hábil em CPUs locais, permitindo a observação de cenários de concorrência.
- Implementação do Cliente (grpcCalc_client.py): A função run_client utiliza argparse para permitir a configuração do clientID e threads via linha de comando, e opções de keepalive para estabilidade da conexão. A mineração local (função mine_solution) é executada em paralelo:
    1. Múltiplos *workers* (mining_worker) são disparados, baseados no os.cpu_count().
    2. Um threading.Event (stop_event) é usado para sinalizar a parada imediata de todos os *workers* assim que uma solução é encontrada.
    3. Uma queue.Queue (result_queue) é usada para coletar o resultado da *thread* vencedora de forma segura.

3. TESTES E RESULTADOS ENCONTRADOS

A comprovação funcional de ambas as atividades foi efetuada através de testes práticos, conforme o vídeo de execução.

3.1. RESULTADOS DA ATIVIDADE 1 (CALCULADORA)

O cliente da calculadora (baseado em start_client) foi executado e testado com os seguintes cenários para validar as operações aritméticas:

- Teste de Soma: A opção 1 foi selecionada.
  - Entrada: Número 1: 5566, Número 2: 7655.
  - Saída: Resultado: 13221.0.
- Teste de Subtração: A opção 2 foi selecionada.
  - Entrada: Número 1: 567, Número 2: 54.
  - Saída: Resultado: 513.0.
- Teste de Multiplicação: A opção 3 foi selecionada.
  - Entrada: Número 1: 5678, Número 2: 6654.
  - Saída: Resultado: 37781412.0.

3.2. RESULTADOS DA ATIVIDADE 2 (MINERADOR)

Para validar a robustez e o controle de concorrência do minerador (baseado em TxTable e mine_solution), foi executado um teste de disputa com o servidor e dois clientes (clientID=1 e clientID=2) conectados simultaneamente.

- Consulta de Status Inicial:
  - Ambos os clientes (Cliente 1 e Cliente 2) iniciaram e consultaram o estado.
  - getTransactionID (Opção 1) retornou TransactionID atual: 0 para ambos.
  - getChallenge (Opção 2, para tx 0) retornou o mesmo desafio (Desafio (difficulty): 2) para ambos.
  - getTransactionStatus (Opção 3, para tx 0) confirmou Status: PENDENTE para ambos.
- Teste de Disputa (Race Condition):
  - Ambos os clientes (Cliente 1 e Cliente 2) selecionaram a Opção 6 (Minerar) e começaram a processar a tx=0 ao mesmo tempo.
  - O Cliente 1 encontrou a solução (0:1:380626) primeiro e a submeteu.
  - O servidor respondeu ao Cliente 1: VÁLIDA/ACEITA.
  - Imediatamente após, o Cliente 2 encontrou sua própria solução (0:2:2832) e a submeteu.
  - O servidor respondeu ao Cliente 2: JÁ SOLUCIONADO.
- Verificação Pós-Disputa:
  - O Cliente 1 consultou getTransactionID (Opção 1) e recebeu TransactionID atual: 1, confirmando que o servidor gerou a próxima tarefa.
  - O Cliente 1 consultou getWinner (Opção 4) para a tx=0 e recebeu Vencedor (ClientID): 1.
  - O Cliente 2 consultou getSolution (Opção 5) para a tx=0 e confirmou que o status era RESOLVIDO, a dificuldade 2, e a solução vencedora era a do Cliente 1 (0:1:380626).

Este teste validou com sucesso o mecanismo de lock da TxTable no servidor, que garantiu que apenas a primeira submissão válida fosse aceita, rejeitando corretamente as submissões concorrentes.

4. CONCLUSÃO

As metas estabelecidas para este laboratório foram atingidas integralmente. O desenvolvimento demonstrou a aplicação funcional do paradigma RPC para a construção de sistemas distribuídos cliente-servidor.

A primeira atividade validou a criação de serviços RPC, o correto tratamento de erros (INVALID_ARGUMENT) e a implementação de padrões de resiliência do lado do cliente (pybreaker).

A segunda atividade foi notável pela implementação de um gerenciamento de estado concorrente eficaz no servidor (através da classe TxTable e threading.Lock) e uma otimização de paralelismo local no cliente (pela função mine_solution com threading.Event e queue.Queue). O sistema resultante atende a todos os requisitos e emula com fidelidade os conceitos centrais da computação distribuída.