Relatório Técnico: Chamada de Procedimento Remoto (gRPC)

Alunos:

- Ana Beatriz Baltazar Boldrini (1-2012338)
- Jordian Coutinho Pereira (1-2312738)
- Matheus Gothe (1-2121135)
- Pedro Miguel Furghieri de Sousa (1-2312739)

Disciplina: Programação Paralela e Distribuída

1. Introdução

Este relatório detalha a implementação e os resultados das atividades propostas no Laboratório II, focado no conceito de Chamada de Procedimento Remoto (RPC). O objetivo principal foi construir dois sistemas distribuídos cliente-servidor utilizando o framework **gRPC** com a linguagem **Python**.

As duas atividades desenvolvidas foram:

1. **Calculadora gRPC:** Um serviço RPC para operações matemáticas básicas (soma, subtração, multiplicação e divisão).
2. **Minerador de Criptomoedas gRPC:** Um protótipo de um sistema de mineração, onde um servidor gerencia desafios e múltiplos clientes competem para resolvê-los.

2. Metodologia de Implementação

A arquitetura de ambas as atividades seguiu o padrão cliente-servidor. A definição dos serviços, métodos (RPCs) e tipos de mensagens foi realizada utilizando **Protocol Buffers (Protobuf)**, que serve como a linguagem de definição de interface (IDL) para o gRPC.

2.1. Atividade 1: Calculadora gRPC

- **Definição do Serviço (`grpcCalc.proto`):**
    - Foi definido um serviço `api` contendo quatro métodos RPC unários: `sum`, `sub`, `mult`, e `divide`.
    - As mensagens `Operacao` (para enviar dois números `float`) e `Resultado` (para retornar um `float`) foram criadas para estruturar os dados de entrada e saída.
- **Implementação do Servidor (`grpcCalc_server.py`):**
    - O servidor gRPC foi implementado para escutar na porta `8080`.
    - Uma classe `CalculatorServicer` implementa a lógica de negócio para cada uma das quatro operações. Foi incluído um tratamento específico para evitar divisão por zero, retornando `0` nesse caso.
- **Implementação do Cliente (`grpcCalc_client.py`):**

- O cliente utiliza a biblioteca `inquirer` para fornecer um menu interativo de linha de comando, permitindo ao usuário escolher a operação e inserir os operandos.
- Para aumentar a resiliência, foi implementado o padrão **Circuit Breaker** utilizando a biblioteca `pybreaker`. O circuito é configurado para abrir (parar de enviar requisições) após 2 falhas consecutivas, com um tempo de espera de 2 segundos antes de tentar se rearmar.

2.2. Atividade 2: Minerador de Criptomoedas gRPC

- **Definição do Serviço (`mine_grpc.proto`):**
  - O arquivo `.proto` define o serviço `api` com todos os métodos RPC solicitados na especificação: `getTransactionId`, `getChallenge`, `getTransactionStatus`, `submitChallenge`, `getWinner`, e `getSolution`.
  - Um método adicional `registerClient` foi implementado para que o servidor possa atribuir um ID único a cada cliente que se conecta.
- **Implementação do Servidor (`grpcMine_server.py`):**
  - O servidor mantém o estado das transações (desafio, solução, vencedor) em um dicionário Python.
  - **Gerenciamento de Concorrência:** Para garantir a integridade dos dados em um ambiente com múltiplos clientes, o acesso ao dicionário de transações e ao contador de IDs de cliente é controlado por `threading.Lock`. Isso previne *race conditions* quando múltiplos clientes tentam submeter uma solução simultaneamente.
  - **Lógica de Mineração:** Ao ser iniciado, o servidor gera a primeira transação com um desafio aleatório.
  - **Validação (SHA-1):** Ao receber uma submissão (`submitChallenge`), o servidor valida a solução. Ele calcula o *hash* **SHA-1** da solução proposta e verifica se o *hash* resultante começa com um prefixo de zeros ('0') cujo tamanho é igual ao valor do desafio.
  - **Nova Transação:** Se a solução for válida e inédita, o servidor armazena o `ClientID` vencedor e cria imediatamente uma nova transação com um novo desafio, incrementando o `current_transaction_id`.
- **Implementação do Cliente (`grpcMine_client.py`):**
  - **Registro:** Ao iniciar, o cliente primeiro chama o RPC `registerClient` para se anunciar ao servidor e receber um `ClientID` único, que é usado em submissões futuras.
  - **Mineração Local:** A opção "submitChallenge" (referida como "Mine" no PDF) é um processo de múltiplos passos: o cliente busca o `transactionID` e o `challenge` atuais, minera localmente a solução e, ao encontrar, a submete ao servidor.
  - **Mineração Paralela:** Para acelerar a busca pela solução (etapa de mineração local), o cliente utiliza a biblioteca `threading`. Múltiplas threads (4, neste caso) são disparadas para testar candidatos de solução em paralelo, cada uma verificando um subconjunto de possibilidades. A primeira thread a encontrar uma solução válida encerra a busca das demais.

3. Testes e Resultados

Os testes foram realizados localmente, executando o servidor e um ou mais clientes em terminais separados.

3.1. Resultados da Calculadora

- **Operações Padrão:** Todas as quatro operações foram testadas com sucesso, retornando os valores corretos. O caso de divisão por zero foi tratado conforme esperado.
- **Teste de Resiliência (Circuit Breaker):**
    1. O cliente foi iniciado e executado normalmente com o servidor online.
    2. O servidor foi deliberadamente interrompido.
    3. O cliente tentou fazer uma chamada (ex: "Adição"). A chamada falhou.
    4. O cliente tentou uma segunda chamada, que também falhou.
    5. Na terceira tentativa, o `pybreaker` interceptou a chamada antes que ela fosse enviada, e o cliente exibiu a mensagem de erro: "Erro: Circuit Breaker ativado! Servidor indisponível momentaneamente.".
    - **Resultado:** O padrão funcionou perfeitamente, protegendo a aplicação cliente de tentar se comunicar com um servidor sabidamente indisponível.

3.2. Resultados do Minerador

- **Conexão e Registro:** Múltiplas instâncias do cliente foram iniciadas. Cada uma conectou-se e recebeu um ID de cliente único (1, 2, 3, etc.), conforme registrado nos logs do servidor.
- **Teste de Concorrência (Múltiplos Clientes e *Race Condition*):**
    1. Dois clientes (`Cliente 1` e `Cliente 2`) foram iniciados.
    2. Ambos receberam o mesmo desafio (ex: `TransactionID=0`, `Challenge=4`).
    3. Ambos iniciaram a mineração local em paralelo.
    4. O `Cliente 1` encontrou e submeteu a solução primeiro. O servidor respondeu "Válida" (código 1).
    5. Imediatamente, o servidor (protegido pelo *lock*) registrou o `Cliente 1` como vencedor da `TransactionID=0` e gerou a `TransactionID=1`.
    6. Momentos depois, o `Cliente 2` submeteu sua solução (correta, porém tardia) para a `TransactionID=0`.
    7. O servidor respondeu ao `Cliente 2` com a mensagem "Já resolvida" (código 2).
    - **Resultado:** O teste confirmou que o uso de `threading.Lock` no servidor foi eficaz para garantir a atomicidade da validação e atualização do estado, tratando a *race condition* corretamente.

4. Conclusão

O laboratório permitiu aplicar na prática os conceitos de sistemas distribuídos e RPC. O gRPC, combinado com Protocol Buffers, provou ser uma ferramenta robusta e eficiente para definir e implementar serviços, tanto síncronos simples (Calculadora) quanto mais complexos (Minerador).

Os principais aprendizados foram a importância do gerenciamento de estado concorrente (usando `Lock` no servidor) e a necessidade de paralelismo no cliente (usando `threading`) para tarefas computacionalmente intensivas, como a mineração. Além disso, a implementação de padrões de resiliência como o Circuit Breaker demonstrou ser fundamental para a estabilidade de aplicações cliente.