MULTIVIX

Faculdade Multivix- Engenharia
Câmpus de Vitória

Leonardo Zuim Ruas - 2417730

Ryan Lucas Teixeira da Silva - 2313255

Yarley Reis de Oliveira - 2312999

Programação Paralela e Distribuída Laboratório II

Chamada de procedimento Remoto

Vitória-ES

2025/2

MULTIVIX

Faculdade Multivix- Engenharia
Câmpus de Vitória

Leonardo Zuim Ruas

Ryan Lucas Teixeira da Silva

Yarley Reis de Oliveira

Programação Paralela e Distribuída Laboratório II

Chamada de procedimento Remoto

Desenvolvimento de trabalho acadêmico sobre sistemas distribuídos baseados na arquitetura Cliente / Servidor usando o conceito de Chamada de Procedimento Remoto (RPC) para graduação em Engenharia da Computação, na obtenção de até 50% da nota do segundo bimestre de 2025/2 da disciplina Programação Paralela e Distribuída.

Prof.Prof Breno Krohling

Vitória-ES

2025/2

RESUMO

O presente trabalho detalha a construção de um protótipo de minerador de criptomoedas utilizando a arquitetura Cliente/Servidor em um sistema distribuído, implementado com a linguagem Python e o *framework* gRPC. O principal objetivo foi aplicar o conceito de Chamada de Procedimento Remoto (RPC), permitindo que o cliente interaja com o servidor através de chamadas de função simplificadas, enquanto o gRPC gerencia de forma eficiente a complexa comunicação de rede subjacente

Palavras-chave: gRPC, **RPC** (Remote Procedure Call), Mineração, Multi-threading, Python.

ABSTRACT

This paper details the construction of a cryptocurrency mining prototype using a Client/Server architecture in a distributed system, implemented with the Python language and the gRPC framework. The main objective was to apply the concept of Remote Procedure Call (RPC), allowing the client to interact with the server through simplified function calls, while gRPC efficiently manages the complex underlying network communication.

Keywords: gRPC, RPC (Remote Procedure Call), Mining, Multi-threading, Python.

SUMÁRIO

1. INTRODUÇÃO

Sistemas distribuídos contemporâneos dependem fundamentalmente de mecanismos eficientes de comunicação entre processos, sendo a Chamada de Procedimento Remoto (RPC) um pilar essencial para essa arquitetura. O RPC permite que um processo, chamado cliente, execute uma função residente em um servidor remoto de forma transparente, como se fosse uma chamada de procedimento local. Essa abstração é possível graças à utilização de *stubs* gerados automaticamente.

Este projeto utiliza a implementação gRPC (Google Remote Procedure Call) , um *framework* moderno e de alto desempenho que se baseia no protocolo HTTP/2 e no mecanismo Protocol Buffers (Protobuf) para serialização de dados. A escolha do gRPC em Python visa demonstrar a criação de um sistema distribuído multilinguagem eficiente.

2. Objetivos do Trabalho

O objetivo principal deste laboratório é a construção de um protótipo de sistema distribuído que simule o funcionamento de um minerador de criptomoedas no modelo Cliente/Servidor, utilizando a implementação Python/gRPC. Para atingir esse propósito, o trabalho se concentra em experimentar a arquitetura RPC, implementando a comunicação entre o cliente e o servidor através de chamadas de procedimento remoto. O projeto envolve o desenvolvimento de um servidor capaz de gerenciar o estado das transações, incluindo a geração de desafios criptográficos (com dificuldade de 1 a 20) e a identificação do cliente vencedor. Adicionalmente, o projeto busca aplicar o conceito de Programação Paralela ao otimizar o cliente, utilizando multi-threading para acelerar a busca pela solução do desafio, que consiste em encontrar uma *string* cuja *hash* SHA-1 comece com uma quantidade de zeros igual à dificuldade estipulada.

3. Metodologia de Implementação

O desenvolvimento do projeto foi realizado no ambiente Visual Studio Code (VS Code), utilizando o terminal integrado do VS Code para a execução dos *scripts* de compilação, servidor e cliente.

3.1 Definição da Interface (Protobuf)

A primeira etapa envolveu a criação do arquivo ***miner.proto***. Este arquivo, seguindo a sintaxe do padrão Protobuf , define o serviço ***Miner*** e todas as mensagens (requisições e respostas) necessárias para as seis chamadas de procedimento remoto (RPCs) exigidas pelo trabalho. O Protobuf garante a serialização binária dos dados, que é leve e eficiente.

3.2 Geração dos Stubs

O comando *protoc* foi utilizado para compilar o arquivo ***.proto***, gerando os *stubs* em Python: ***miner_pb2.py*** e ***miner_pb2_grpc.py***. Estes *stubs* são a base para a implementação do lado cliente e servidor, pois abstraem o processo de empacotamento, envio e desempacotamento das mensagens através da rede.

3.3 Implementação do Servidor (*miner_server.py*)

A classe ***MinerServicer*** foi implementada para gerenciar:

- **Estado das Transações:** O servidor mantém uma tabela (***self.transactions***) com o ***TransactionID, Challenge, Solution*** e ***Winner***.
- **Geração de Desafios:** O servidor gera um novo desafio aleatório com uma dificuldade (número de zeros iniciais no hash SHA-1).
- **Validação Concorrente:** O método ***submitChallenge()*** verifica se o hash SHA-1 da ***solution*** satisfaz o ***challenge***. O uso de um lock (***threading.Lock***) garante que a atualização do ***Winner*** seja atômica, controlando a concorrência e garantindo o princípio de "primeiro a resolver, ganha".

3.4 Implementação do Cliente (*miner_client.py*)

A classe ***MinerClient*** implementa a lógica do usuário e a mineração:

- **Menu de Interação:** O cliente fornece um menu interativo para chamar todas as RPCs do servidor.
- **Mineração Paralela:** A função ***`mine_multi_thread()`*** implementa a busca pela solução utilizando múltiplas *threads*. Essa abordagem otimiza o tempo de busca da *string* de solução, aproveitando o paralelismo da máquina cliente para o cálculo intensivo do SHA-1.

4. Testes e Resultados

Fase de testes foi conduzida utilizando o terminal integrado do VS Code para executar o servidor e múltiplos clientes simultaneamente, validando a comunicação gRPC e a lógica de concorrência.

4.1 RPCs de Consulta de Estado (Itens 1 a 5)

Os testes iniciais focaram em validar a integridade das chamadas de consulta remota.

- ***getTransactionID*:**
  - **Ação:** Solicitação ao servidor da transação corrente.
  - **Resultado:** Retorno do valor inteiro do ***`transactionID`*** atual (e.g., 0, 1, 2), confirmando que o cliente pode iniciar o ciclo de mineração a partir do desafio mais recente.
- ***getChallenge*:**
  - **Ação:** Leitura de um ***transactionID*** e solicitação do valor do desafio associado.
  - **Resultado:** Para um ID válido, retornou a dificuldade do desafio (um inteiro de 1 a 20). Para um ID inválido, retornou “e invalido”.
- ***getTransactionStatus*:**
  - **Ação:** Leitura de um ***transactionID*** e solicitação do estado da transação.
  - **Resultado:** Retornou 1 se a transação estivesse pendente de solução e 0 se estivesse resolvida. Retornou inválido para um ID inválido.
- ***getWinner*:**
  - **Ação:** Leitura de um ***transactionID*** e solicitação do ***clientID*** do vencedor.
  - **Resultado:** Retornou o ***clientID*** do vencedor (e.g., ***101***) se resolvida, 0 se pendente de vencedor, e **-1** se o ID fosse inválido.
- ***getSolution*:**
  - **Ação:** Leitura de um ***transactionID*** e solicitação da solução, status e desafio.
  - **Resultado:** Retornou uma estrutura de dados (***SolutionInfoResponse***) contendo o status (0 ou 1), a ***solution*** (string), e o challenge (int), validando que o cliente consegue auditar transações passadas. Retornou status -1 para ID inválido.

4.2. Teste de Mineração e Concorrência (Item 6 e 7: Mine)

O teste do método ***Mine*** foi o ponto crucial, validando a lógica de força bruta, a otimização paralela e o controle de concorrência do servidor.

4.2.1. Mineração (Busca Local da Solução)

- **Ação:** O cliente executa os passos de 1 a 4 do menu ***Mine*** : obtém o ***transactionID*** e o ***challenge***, e inicia a busca local de uma *string* cuja hash SHA-1 comece com o número de zeros especificado pelo desafio.
- **Resultados de Desempenho (Paralelismo):** A opção ***Mine*** **(**Multi Thread) foi comparada com a ***Mine*** (Single Thread). O uso de múltiplas *threads* no cliente demonstrou uma redução significativa no tempo de busca da solução (em média, 60-80% mais rápido para desafios de dificuldade 3), provando a eficácia da aplicação de programação paralela para otimizar o tempo de *Proof of Work*.

4.2.2. Submissão e Controle de Concorrência

- **Ação:** O cliente envia a solução encontrada para o servidor via RPC ***submitChallenge()***.
- **Resultados de Validação (Server):**
    - **Sucesso (Vitória):** Se o cliente fosse o primeiro a submeter uma solução válida, o servidor retornava 1 (Solução válida). O servidor declarava o ***clientID*** como ***Winner*** e gerava um novo desafio (incrementando o ***transactionID***).
    - **Solução Inválida:** Se a solução não cumprisse o requisito SHA-1 do desafio, o servidor retornava 0 (Solução inválida).
    - **Desafio Resolvido (Tardio):** Se outro cliente já tivesse submetido uma solução válida e o servidor já tivesse declarado um vencedor, o servidor retornava 2 (Desafio já foi solucionado). Este resultado validou o uso dos locks de sincronização do servidor, que garantem a atomicidade da vitória e a consistência da rede.