CEFET - RJ

Centro Federal de Educação Tecnológica Celso Suckow da Fonseca

AEDS 1 -Relatório *Final*

Replicação do *BlockChain*: Descrição das Funções, Testes da Aplicação e Simulação de Ataque à Cadeia

---

Carlos Eduardo Dias Silva
Pedro Paulo Marques Carneiro
Anisio Matheus Alves Da Fonseca

Pedro Henrique Gasparetto Lugão (docente)

6 de março de 2025

Sumário

1 Introdução

Este relatório apresenta o desenvolvimento de uma aplicação em linguagem C que implementa uma blockchain funcional. O sistema utiliza o algoritmo Proof of Work (PoW) como mecanismo de validação de blocos, garantindo a integridade da cadeia por meio da resolução de desafios computacionais. Além disso, cada bloco incorpora uma Merkle Tree para organizar e estruturar suas transações de maneira eficiente, permitindo verificações rápidas e seguras.

A aplicação também conta com uma funcionalidade para verificar se uma transação específica está presente em um determinado bloco por meio do processo de Proof of Inclusion, reforçando a confiabilidade do sistema. Ao longo deste relatório, serão descritos os principais conceitos, a implementação das funcionalidades, os testes realizados e uma simulação de possíveis ataques à blockchain.

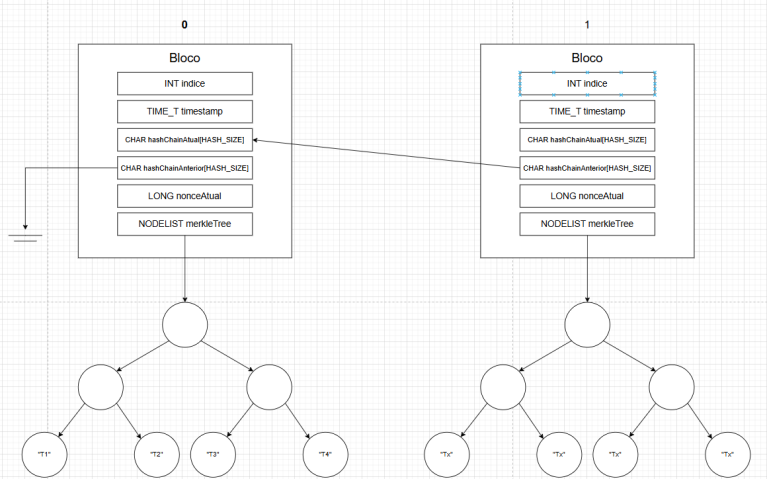

Figura 1: Diagrama da estrutura

De acordo com [**?**], O bloco é armazenado na *blockChain*, contendo informações essenciais para a validação e ligação entre os blocos. Cada bloco na *blockChain* armazena informações que garantem sua integridade e sua conexão com o bloco anterior. O hash do bloco atual é gerado a partir de suas informações internas e do hash do bloco anterior, garantindo a imutabilidade da cadeia. Além disso, a inclusão de uma árvore de Merkle permite que as transações sejam verificadas de maneira eficiente, sem a necessidade de armazenar todas as transações individualmente dentro do bloco.

A estrutura MerkleNode representa um nó dentro de uma árvore de Merkle, utilizada para armazenar em um blockchain. A árvore de Merkle é uma estrutura binária onde cada nó contém um hash derivado das transações, e os nós superiores armazenam os hashes combinados de seus filhos, garantindo a integridade dos dados.

Por fim, A estrutura NodeList é responsável por armazenar uma lista de nós da árvore de Merkle.

2 Metodologia

Utilizando a linguagem C, em conjunto com a biblioteca *openSsL*, foram desenvolvidas funções para atender aos requisitos funcionais do trabalho. Para fins de versionamento e trabalho em equipe, foi utilizado a plataforma do *GitHub*.

2.1 Proof Of Work

Utilizando métodos da biblioteca externa, recebe como parâmetro uma dificuldade, configurável de 1 a 4, e o bloco a ser minerado.

```
@brief faz o calculo do hash usando a biblioteca openSsl
@param block: Struct para gerar o hash
@param hashGerado: Output do hash

*/
void GerarHashNode(Chain *block, unsigned char *hashGerado);

/*
@brief usa um laço para exibir o hash em forma hexadecimal
@param hash: hash a ser exibido
 */
void exibirHash(unsigned char *hash);

/*
@brief Converte um hash binário em uma string hexadecimaL
@param hashBin: Ponteiro para o hash binário gerado
@param hashHex: Ponteiro para o buffer onde o hash hexadecimal será armazenado
*/
void HashParaHex(const unsigned char *hashBin, char *hashHex, size_t tamanhoHash);

/*
@brief Com base na dificuldade, realiza a mineração do nounce especifico, a fim de alterar o hash
@param bloco: struct para o POW
@param dificuldade: de 1 a 4
*/
unsigned char ProofOfWorkLinear(Chain *bloco, int dificuldade);
```

Figura 2: Cabeçalho das funções, presente em StructChain.h

Com base nesses parâmetros, realiza a mineração do nounce especifico, a fim de alterar o hash que atenda à dificuldade. Faz uso de uma função auxiliar para gerar o *hash* do bloco e para conversão de hash binário para hash hexadecimal.

2.2 Proof of Inclusion

Recebe com parâmetro o nó que será verificado, a transação que se deseja aferir, um *array* de strings que armazenará os hashs do caminho de prova e o índice que se deseja verificar.

```
/*
@brief Verifica se uma transação está incluída na árvore de Merkle e constrói o caminho de prova (proof of inclusion)
@param no: Nó da árvore de Merkle a ser verificado
@param incluido: transação que se deseja verificar se está incluída na árvore
@param proofhash: array de strings que armazenará os hashes do caminho de prova
@param indice: índice atual no array de proofhash
@return retorna 1 se a transação estiver incluída na árvore, caso contrário retorna 0
*/
int poi(MerkleNode *no, const char *incluido, char proofhash[][HASH_SIZE * 2 + 1], int *indice);
```

Figura 3: Cabeçalho da função, presente em StructChain.h

Com base nesses parâmetros, confere se uma transação está incluída na árvore de Merkle e constrói o caminho de prova (proof of inclusion).

2.3 MerkleTree

A *MerkleTree* é responsável por armazenar as transações. Desse modo, a função *buildMerkleTree* constrói uma árvore de Merkle, recebendo como parâmetros um vetor de transações e o total de transações.

Utiliza as funções auxiliares *createParent*, responsável por criar um nó pai a partir de dois nós filhos, combinando seus hashes e createLeaf, a qual cria istancia um nó folha da árvore de Merkle com base nos dados fornecidos para a transação.

```
/*
@brief Cria um nó folha da árvore de Merkle com base nos dados fornecidos
@param data: dados da transação que serão armazenados no nó folha
@return Retorna um ponteiro para o nó folha criado
*/
MerkleNode *create_leaf(const char *data);

/*
@brief Cria um nó pai na árvore de Merkle com base nos nós filhos fornecidos
@param left: nó filho esquerdo
@param right: nó filho direito
@return Retorna um ponteiro para o nó pai criado
*/
MerkleNode *create_parent(MerkleNode *left, MerkleNode *right);

/*
@brief Constrói uma árvore de Merkle a partir de uma lista de transações
@param transactions: array de strings contendo as transações
@param num_transactions: número de transações no array
@return Retorna uma estrutura NodeList contendo os nós da árvore de Merkle
*/
NodeList build_merkle_tree(char **transactions, int num_transactions);
```

Figura 4: Cabeçalho das funções, presente em StructChain.h

2.4 NovoBloco

Responsável por instanciar um bloco que irá compor a *BlockChain*. Recebe como parâmetros um vetor de transações, total de transações, índice, dificuldade, e tamanho da blockChain.

```
/*
@brief instancia um novo bloco de acordo com os parâmetros
@param transacoes vetor de strings
@param qtdTransacoes total de transações
@param indice posicao na BlockChain
@param dificuldade 0 - 4
@param blockChain BlockChain que ira receber o bloco
@param tam: Quantidade de blocos da BlockChain
*/
Chain novoBloco(char **transacoes, int qtdTransacoes, int indice, int dificuldade, Chain *blockChain, int tam);
```

Figura 5: Cabeçalho da função, presente em StructChain.h

3 Simulação de Ataque à BlockChain

A função simula um ataque à *blockChain*, recebe como parâmetro a transação maliciosa e o índice o qual se deseja alterar. Caso o índice seja válido, realiza o cálculo de um novo hash a partir deste e recalcula o *nounce* dos blocos subsequentes.

```c
void simularAtaque(Chain *blockchain, int numblocks, int indicealterado, char *transacao)
{
    if (indicealterado < 0 || indicealterado >= numblocks)
    {
        printf("indice invalido!");
        return;
    }
    if (blockchain[indicealterado].merkleTree.nodes != NULL)
    {
        // cast necessario por conta da chamada de blockchain sendo em **.
        calculate_hash(transacao, (char *)blockchain[indicealterado].merkleTree.nodes[0]->hash);

        clock_t inicio = clock();

        for (int i = indicealterado; i < numblocks; i++)
        {
            blockchain[i].nonceAtual = 0;                    // Reseta o nonce
            ProofOfWorkLinear(&blockchain[i], MAX_SEVERIDADE); // Recalcula o nonce
        }
        clock_t fim = clock();
        double tempo_gasto = ((double)(fim - inicio)) / CLOCKS_PER_SEC;
        printf("Tempo gasto para reajustar todos os nonces: %.2f segundos\n", tempo_gasto);
        printf("transação alterada com sucesso!");
    }
    else
    {
        printf("o bloco %d não possui transações para alterar", indicealterado);
        return;
    }
}
```

Figura 6: Lógica da função de ataque

Podemos observar o processo de geração de nounces e tempo gasto em (Fig 7 e Fig 8).

```
hash minerado com sucesso em 88934 iterações.
hash minerado com sucesso em 28409 iterações.
hash minerado com sucesso em 74775 iterações.
hash minerado com sucesso em 43268 iterações.
hash minerado com sucesso em 29787 iterações.
hash minerado com sucesso em 37539 iterações.
hash minerado com sucesso em 116506 iterações.
hash minerado com sucesso em 225900 iterações.
hash minerado com sucesso em 18197 iterações.
hash minerado com sucesso em 53734 iterações.
hash minerado com sucesso em 8623 iterações.
hash minerado com sucesso em 198319 iterações.
hash minerado com sucesso em 19116 iterações.
hash minerado com sucesso em 88771 iterações.
hash minerado com sucesso em 21179 iterações.
hash minerado com sucesso em 52994 iterações.
hash minerado com sucesso em 3632 iterações.
hash minerado com sucesso em 251053 iterações.
hash minerado com sucesso em 36042 iterações.
hash minerado com sucesso em 229333 iterações.
hash minerado com sucesso em 176624 iterações.
hash minerado com sucesso em 56639 iterações.
hash minerado com sucesso em 20336 iterações.
hash minerado com sucesso em 97042 iterações.
hash minerado com sucesso em 10252 iterações.
hash minerado com sucesso em 25342 iterações.
```

Figura 7: Geração dos hashes a partir dos nounces alterados.

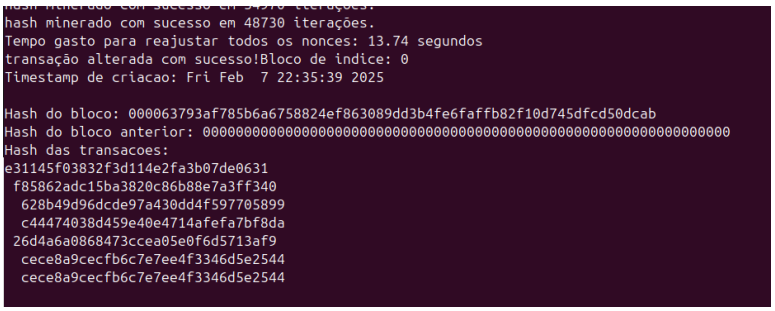

Figura 8: Tempo gasto na geração de novos hashes.

4 Referências

[1] Carlos Dias. Replicação Blockchain. Disponível em: https://github.com/CarlosDiasS/Replicacao-BlockChain

[2] OpenSSL Documentation. Guide to libcrypto. Disponível em: https://docs.openssl.org/master/man7/ossl-guide-libcrypto-introduction/#using-algorithms-in-applications