Segurança e Auditoria de Sistemas
Atividade 3

EC38D/C81 - Eng. de Software - UTFPR-CP - 2021/2
Francisco de Carvalho Farias Junior - RA: 1581660
fariasjunior96@gmail.com

A atividade

A atividade consiste na implementação de uma aplicação de blockchain que rode de maneira local, e não distribuída. Ela deve fornecer a opção de validar e inserir blocos na blockchain, de validá-la, de minerar novos blocos (prova por trabalho para inserção), e de editar arbitrariamente os blocos a fim de testar o funcionamento da validação. O usuário deve ser capaz de definir a dificuldade de mineração de novos blocos.

O relatório deve conter também uma análise do impacto da dificuldade no tempo de mineração de novos blocos.

Descrição da solução

Tecnologias usadas

- Java 11
- Maven 4.0.0
- Spring Boot 2.6.1
- Swagger 2.9.2
- SHA-256, implementação da biblioteca java.security

Para inicializar o programa pode-se executar *mvn spring-boot:run*. Após inicializá-lo, o Swagger pode ser acessado na mesma máquina pela URL *http://localhost:8080/swagger-ui.html*.

A interface

A interface do programa assume o formato de uma API REST, com o Swagger oferecendo uma interface web embutida, além da documentação precisa para cada endpoint.

O programa oferece as seguintes opções para o usuário:

- Iniciar a blockchain, recebendo uma dificuldade de 0 a 255 e gerando o hash máximo para uma solução válida;

- Parar a blockchain, apagando todos os blocos e interrompendo qualquer minerador sendo executado;
- Receber informações da blockchain, incluindo se está ativa, se é válida, qual a dificuldade, qual a probabilidade de um dado hash ser válido e a lista de blocos;
- Iniciar um minerador, que é uma thread que minera novos blocos em background e, ao encontrá-los, solicita à sua inclusão blockchain automaticamente;
- Listar os mineradores ativos;
- Interromper um minerador ativo em específico;
- Substituir um bloco arbitrário na blockchain, para testes.

A estrutura da solução

A lógica do programa está dividida entre dois serviços, o *BlockchainService* e o *MinerService*, esse segundo feito para gerenciar mineradores, que são instâncias da classe *Miner*, feita para rodar em threads.

BlockchainService

O *BlockchainService* fornece APIs para controlar o ciclo de vida da blockchain, retornar informações sobre ela e inserir e editar blocos.

Ela possui uma rotina de validação, que percorre inversamente a chain checando se o hash que cada bloco guarda bate com o hash do bloco anterior (i.e.: a integridade da chain).

```
private boolean isValid() {
  if (!isOn()) {
    throw new IllegalStateException("Blockchain isn't initialized");
  }

  var digester :MessageDigest = getDigester();

  for (var currentIndex = chain.size() - 1; currentIndex >= 1; currentIndex-- ) {
    var currentBlock :Block = chain.get(currentIndex);
    var previousBlock :Block = chain.get(currentIndex-1);
    var previousBlockHash :byte[] = digester.digest(previousBlock.toString().getBytes());

    if (!Arrays.equals(currentBlock.getPreviousBlockHash(), previousBlockHash)) {
      return false;
    }
  }

  return true;
}
```

Figura 1: Implementação do algoritmo de validação, que checa os blocos em ordem inversa de inserção na chain.

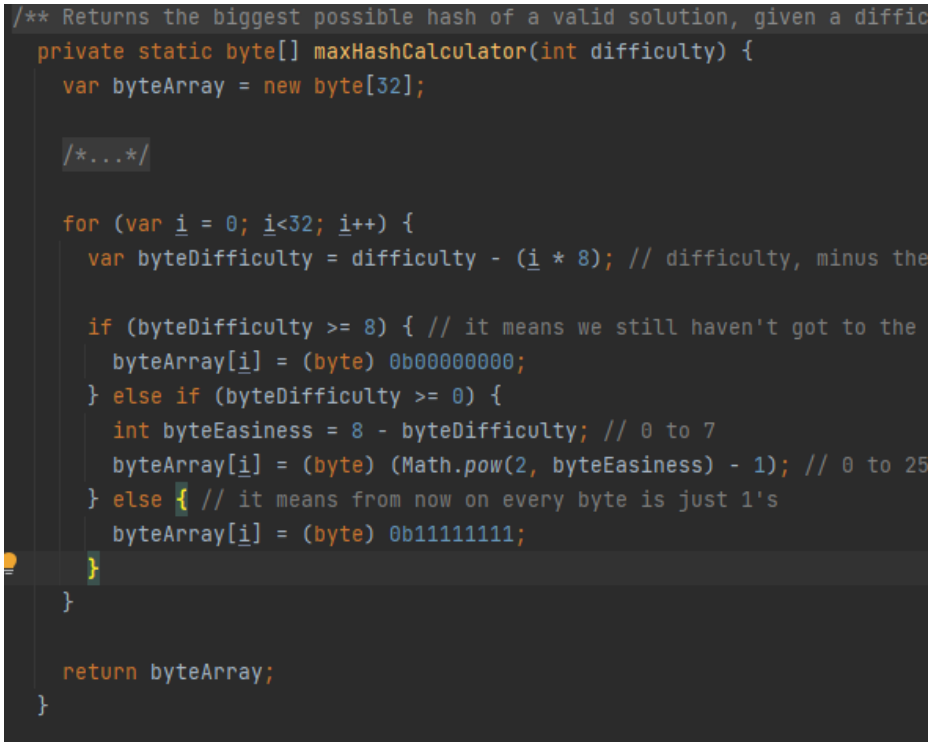
Ela também possui uma rotina para validar uma solução e adicionar um novo bloco caso ela seja válida:

```java
@Override
public void validateAndAddBlock(int minerId, byte[] solution) {
  var digester : MessageDigest = getDigester();
  var solutionHash : byte[] = digester.digest(solution);

  if (new BigInteger( signum: 1, solutionHash).compareTo(new BigInteger( signum: 1, maxHash)) > 0) {
    throw new IllegalStateException("Failed validation");
  }

  var previousBlock : Block = chain.get(chain.size() - 1);

  var newBlock : Block = Block.builder()
      .id(previousBlock.getId() + 1)
      .previousBlockHash(digester.digest(previousBlock.toString().getBytes(StandardCharsets.UTF_8)))
      .minerId(minerId)
      .dateTime(LocalDateTime.now())
      .build();

  chain.add(newBlock);
  log.info( message: "Added new block from miner={}", minerId);
}
```

Figura 2: Rotina de inserção de novos blocos com validação.

Além disso, ela possui uma rotina para calcular o hash máximo para uma solução válida dada a dificuldade informada na inicialização da blockchain. A dificuldade é um inteiro entre 0 e 255; quanto maior, mais à direita está o último bit significativo do hash, permitindo assim que o usuário ajuste a probabilidade de novos blocos serem encontrados seguindo o inverso de potência de 2 (1, 0.5, 0.25, 0.125…).

Por causa do funcionamento da biblioteca de hashing do Java, precisamos ter esse hash máximo no formato de array de bytes. Isso se provou uma tarefa não-trivial, já que o Java não é particularmente bem preparado para se trabalhar com manipulação de bits. Assim, foi elaborada uma rotina que constrói o array, byte a byte, em função da dificuldade escolhida.



```
/** Returns the biggest possible hash of a valid solution, given a diffic
  private static byte[] maxHashCalculator(int difficulty) {
    var byteArray = new byte[32];

    /*...*/

    for (var i = 0; i<32; i++) {
      var byteDifficulty = difficulty - (i * 8); // difficulty, minus the

      if (byteDifficulty >= 8) { // it means we still haven't got to the
        byteArray[i] = (byte) 0b00000000;
      } else if (byteDifficulty >= 0) {
        int byteEasiness = 8 - byteDifficulty; // 0 to 7
        byteArray[i] = (byte) (Math.pow(2, byteEasiness) - 1); // 0 to 25
      } else { // it means from now on every byte is just 1's
        byteArray[i] = (byte) 0b11111111;
      }
    }

    return byteArray;
  }
```

Figura 3: Cálculo de hash máximo de uma solução válida para uma dada dificuldade.

MinerService

O *MinerService* é um serviço relativamente simples, com a função de iniciar novas threads baseadas na classe *Miner* e guardá-las para poder 1) informar ao usuário sobre elas e 2) interrompê-las quando for solicitado.

Miner

O *Miner* foi projetado para minerar em background. Quando solicitado, ele entra em um loop, buscando soluções cujo hash seja menor ou igual ao hash máximo informado pela *BlockchainService* e, caso seja, solicita que ela crie um bloco novo.

Na implementação atual ele busca valores começando do inteiro zero e adicionando um a cada loop. Foram realizados experimentos com diferentes estratégias para se definir o valor testado em cada loop e nenhuma causou diferença significativa na taxa de soluções por testes nos mineradores. Essa estratégia deve ser definida de acordo com as necessidades da aplicação; para essa aplicação, cuja meta é somente ilustrar o funcionamento de uma blockchain, as configurações atuais bastam.

```java
@SneakyThrows
@Override
public void run() {
  while (true) {
    var hash : byte[] = digest.digest(currentTestValue.toByteArray());
    var intHash = new BigInteger( signum: 1, hash);

    int comparison = intHash.compareTo(maxHash);

    if (comparison <= 0) {
      passCount++;
      SpringContext.getBean(BlockchainService.class).validateAndAddBlock(id, currentTestValue.toByteArray()
    } else {
      failCount++;
    }

    hashesCount++;

    if (hashesCount % 1000 == 0) {
      var passRatio = (double) passCount/(failCount + passCount);
      log.info( message: "miner {}: passed={}, failed={}, ratio={}", id, passCount, failCount, passRatio);
      Thread.sleep( millis: 2000); // TODO deal gracefully with this being interrupted
      hashesCount = 0;
    }

    // Thread.sleep(1000);

    currentTestValue = currentTestValue.add(BigInteger.ONE);

    if (false) { // just so the linter shut up about end conditions
      return;
    }
  }
}
```

Figura 4: Rotina de mineração.

Prova de integridade e validação

Para testar se a validação está correta foi implementada a funcionalidade de editar blocos arbitrários. Assim, realizamos aqui um teste onde editamos dados de um bloco e checamos se a chain ainda passa nas validações.

Temos a blockchain no seguinte estado:

```
{
  "difficulty": 10,
  "chance": 0.0009765625,
  "isValid": true,
  "blocks": [
    {
      "id": 0,
      "previousBlockHash": "AA==",
      "dateTime": "2021-12-11T02:07:53.0413623",
      "minerId": -1
    },
    {
      "id": 1,
      "previousBlockHash": "HbB938BKmFQnNes+qKlW+IxR3rda3GvmbChhW0mz/zI=",
      "dateTime": "2021-12-11T02:07:56.6877026",
      "minerId": 2
    },
    {
      "id": 2,
      "previousBlockHash": "ewQzIal68DqK0BaiWYemOoGcYChL/t2yOgRGtzSoUDk=",
      "dateTime": "2021-12-11T02:07:57.2981268",
      "minerId": 3
    },
    {
      "id": 3,
      "previousBlockHash": "qaNDcT4zweebc3rnpBidIEdNwiPJEntrCJ/P6jsJuYc=",
```


FIgura 5: Estado inicial da blockchain, antes das alterações.

A hipótese é que se alterarmos qualquer dado de qualquer bloco a validação passa a falhar. Assim, alteramos um dígito do campo *previousBlockHash* do bloco id=2, e conferimos que o campo *isValid* mudou para falso.


```
{
  "difficulty": 10,
  "chance": 0.0009765625,
  "isValid": false,
  "blocks": [
    {
      "id": 0,
      "previousBlockHash": "AA==",
      "dateTime": "2021-12-11T02:07:53.0413623",
      "minerId": -1
    },
    {
      "id": 1,
      "previousBlockHash": "HbB938BKmFQnNes+qKlW+IxR3rda3GvmbChhW0mz/zI=",
      "dateTime": "2021-12-11T02:07:56.6877026",
      "minerId": 2
    },
    {
      "id": 2,
      "previousBlockHash": "qwQzIal68DqK0BaiWYemOoGcYChL/t2yOgRGtzSoUDk=",
      "dateTime": "2021-12-11T02:07:57.2981268",
      "minerId": 3
    },
    {
      "id": 3,
      "previousBlockHash": "qaNDcT4zweebc3rnpBidIEdNwiPJEntrCJ/P6jsJuYc=",
```


Figura 6: Após alterar o hash do bloco anterior para o id=2, a validação passa a falhar.

Agora, restauramos o bloco 2 para o estado válido e mudaremos o ID do minerador do bloco 1, que está como -1 uma vez que ele é o bloco gênese. Assim como esperado, o campo *isValid* continua como falso.

```
{
  "difficulty": 10,
  "chance": 0.0009765625,
  "isValid": false,
  "blocks": [
    {
      "id": 0,
      "previousBlockHash": "AA==",
      "dateTime": "2021-12-11T02:07:53.0413623",
      "minerId": 2
    },
    {
      "id": 1,
      "previousBlockHash": "HbB938BKmFQnNes+qKlW+IxR3rda3GvmbChhW0mz/zI=",
      "dateTime": "2021-12-11T02:07:56.6877026",
      "minerId": 2
    },
    {
      "id": 2,
      "previousBlockHash": "ewQzIal68DqK0BaiWYemOoGcYChL/t2yOgRGtzSoUDk=",
      "dateTime": "2021-12-11T02:07:57.2981268",
      "minerId": 3
    },
    {
      "id": 3,
      "previousBlockHash": "qaNDcT4zweebc3rnpBidIEdNwiPJEntrCJ/P6jsJuYc=",
```


Figura 7: Se alterar o ID do minerador a validação também falha.

Por último, restauramos o bloco 0 e conferimos como o campo *isValid* volta a ser verdadeiro.

```
{
  "difficulty": 10,
  "chance": 0.0009765625,
  "isValid": true,
  "blocks": [
    {
      "id": 0,
      "previousBlockHash": "AA==",
      "dateTime": "2021-12-11T02:07:53.0413623",
      "minerId": -1
    },
    {
      "id": 1,
      "previousBlockHash": "HbB938BKmFQnNes+qKlW+IxR3rda3GvmbChhW0mz/zI=",
      "dateTime": "2021-12-11T02:07:56.6877026",
      "minerId": 2
    },
    {
      "id": 2,
      "previousBlockHash": "ewQzIal68DqK0BaiWYemOoGcYChL/t2yOgRGtzSoUDk=",
      "dateTime": "2021-12-11T02:07:57.2981268",
      "minerId": 3
    },
    {
      "id": 3,
      "previousBlockHash": "qaNDcT4zweebc3rnpBidIEdNwiPJEntrCJ/P6jsJuYc=",
```


Figura 8: A validação volta a passar ao restaurar os dados para o estado inicial.

Análise do impacto da dificuldade no tempo de mineração

Para esse estudo foi deixado um único minerador rodando durante 300 segundos (5 minutos).

Dificuldade	Total de tentativas	Bem-sucedidas	Bem-sucedidas/segundo
5	105.378.027	3.292.247	~10.974
10	269.863.557	264.237	~881
15	301.340.649	9.290	~31
20	286.934.889	258	~1
25	277294492	7	~0,023 (1 a cada 1,4 min.)

Tabela 1: Análise do número de soluções encontradas para cada dificuldade em uma janela de 5 minutos.

Máquina usada no teste:

- i7-10510U @ 2.3 GHz
- 32 GB RAM
- Windows 10 Pro

Referências

https://confluence.jaytaala.com/display/TKB/Super+simple+approach+to+accessing+Spring+beans+from+non-Spring+managed+classes+and+POJOs

https://docs.oracle.com/javase/8/docs/api/java/lang/Byte.html

https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html

https://www.baeldung.com/sha-256-hashing-java

https://www.treinaweb.com.br/blog/documentando-uma-api-spring-boot-com-o-swagger

https://www.baeldung.com/swagger-2-documentation-for-spring-rest-api