

Acordo de Chaves de Diffie-Hellman

Autor

- Thiago Senhorinha Rose (12100774)

O que é?

O método da troca de chaves de Diffie-Hellman permite que duas partes que não possuem conhecimento a priori de cada uma compartilhar uma chave secreta sob um canal de comunicação inseguro. Tal chave pode ser usada para encriptar posteriores mensagens usando uma esquema de cifra de chave simétrica.

A comunicação procede da seguinte maneira.

1. Um dos lados da comunicação gera um primo e procura uma raiz primitiva do mesmo.
2. Envia o primo e a raiz primitiva encontrada para o outra ponta da comunicação.
3. Ambos os lados geram um inteiro secreto.
4. O inteiro secreto é utilizado para calcular a "chave pública": $(\text{raizPrimitiva}^{\text{inteiroSecreto}}) \bmod \text{primo}$.
5. Ambos trocam a "chave pública" e realizam a o cálculo a seguir a fim de obter a chave secreta: $(\text{chavePublicaRecebida}^{\text{inteiroSecreto}}) \bmod \text{primo}$

Servidor

```
public static void main(String[] args) {
    if (args.length > 0) {
        decimalDigits = Integer.valueOf(args[0]);
    } else {
        decimalDigits = 20;
    }
    try {
        initializeServer();
        while (true) {
            waitConnection();
            waitNewRequest();
            sendDecimalDigits();
            BigInteger prime = calculatePrime();
            sendPrime(prime);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```

        BigInteger primitiveRoot = calculatePrimitiveRoot(prime);
        sendPrimitiveRoot(primitiveRoot);
        BigInteger result = readResult();
        BigInteger resultToSend = calculateFormule(prime, primitiveRoot);
        sendResult(resultToSend);
        BigInteger secret = calculateSecret(result, secretInteger, prime);
        System.out.printf("[Secret: %s]\n", secret);
        connection.close();
    }
} catch (Exception e) {
    e.printStackTrace();
}
}

```

initializeServer()

Inicializa o servidor para troca de chaves Diff-Hellman. Representa um dos lados da troca de chaves, ou seja, Beto ou Ana por exemplo.

O servidor é iniciado na porta **9988**.

```

[-] private static void initializeServer() throws Exception {
    serverSocket = new ServerSocket(9988);
    System.out.println("--- Diffie-Hellman Server Initialized ---");
}

```

waitConnection()

Aguarda um pedido externo de conexão, ou seja, pedido lado do cliente.

```

[-] private static void waitConnection() throws IOException {
    System.out.println("Waiting Connection request...");
    connection = serverSocket.accept();
}

```

waitNewRequest()

Aguarda uma solicitação para troca de chaves.

```

[-] private static boolean waitNewRequest() throws IOException {
    while (true) {
        BufferedInputStream is = new BufferedInputStream(connection.getInputStream());
        InputStreamReader isr = new InputStreamReader(is);
        StringBuffer process = new StringBuffer();
        int character;
        while ((character = isr.read()) != 13) {

```

```

        process.append((char) character);
    }
    if (process.toString().contains("newRequest")) {
        return true;
    }
}
}

```

sendDecimalDigits()

Envia ao cliente a ordem de grandeza a qual os números irão trabalhar. Por padrão está configurado em 20 dígitos. É possível, passando por parâmetro na inicialização (java executável ordemDeGrandeza), alterar essa ordem de grandeza.

```

private static void sendDecimalDigits() throws IOException {
    System.out.printf("Sending decimal digits: %s\n", decimalDigits);
    BufferedOutputStream bos = new BufferedOutputStream(connection.getOutputStream
());
    OutputStreamWriter osw = new OutputStreamWriter(bos, "US-ASCII");
    String process = "decimaldigits:" + decimalDigits + (char) 13;
    osw.write(process);
    osw.flush();
}

```

calculatePrime()

Calcula um primo utilizando como teste de primalidade o algoritmo de **Miller Rabin**

Algoritmo de Miller Rabin

```

public class MillerRabin {

    private static final BigInteger ZERO = BigInteger.ZERO;
    private static final BigInteger ONE = BigInteger.ONE;
    private static final BigInteger TWO = BigInteger.valueOf(2);

    public static boolean isProbablyPrime(BigInteger n, int precision) {
        if (n.compareTo(ONE) <= 0 || n.mod(TWO) == ZERO) {
            return false;
        } else if (n.intValue() == 3) {
            return true;
        }

        BigInteger nLessOne = n.subtract(ONE);
        BigInteger x;
        int s = 0;
        BigInteger d = nLessOne;
        BigInteger randomBigInteger;
        while (d.mod(TWO).equals(ZERO)) {

```

```

        s++;
        d = d.divide(TWO);
    }

    for (BigInteger k = ZERO; k.intValue() <= precision; k = k.add(ONE)) {
        randomBigInteger = BigIntegerRandomGenerator.generate(TWO, n.subtract(
TWO));

        x = randomBigInteger.modPow(d, n);
        if (x.compareTo(ONE) == 0 || x.compareTo(nLessOne) == 0) {
            continue;
        }
        for (int r = 0; r < s - 1; r++) {
            x = x.modPow(TWO, n);
            if (x.compareTo(ONE) == 0) {
                return false;
            }
            if (x.compareTo(nLessOne) == 0) {
                continue;
            }
        }
        return false;
    }
    return true;
}
}

```

calculatePrime()

```

private static BigInteger calculatePrime() {
    BigInteger prime = null;
    boolean primeNotFound = true;
    while (primeNotFound) {
        prime = BigIntegerRandomGenerator.generate(decimalDigits);
        primeNotFound = !MillerRabin.isProbablyPrime(prime, 5);
    }
    return prime;
}

```

sendPrime()

Envia para outra ponta da conexão o primo calculado previamente.

```

private static void sendPrime(BigInteger prime) throws IOException {
    System.out.printf("Sending prime: %s\n", prime);
    BufferedOutputStream bos = new BufferedOutputStream(connection.getOutputStream
());
    OutputStreamWriter osw = new OutputStreamWriter(bos, "US-ASCII");
    String process = "prime:" + prime + (char) 13;
    osw.write(process);
    osw.flush();
}

```

calculatePrimitiveRoot()

Calcula uma raiz primitiva.

PrimitiveRootCalculator.calculate(prime);

A fim de possibilitar a troca de chaves com primos grandes foi necessário procurar uma maneira alternativa, mais performática, para o cálculo de raízes primitivas. A maneira encontrada foi explorar uma derivação da definição de raízes primitivas.

Definição alternativa: Se a ordem multiplicativa de um número g módulo n é igual a $\phi(n)$, então g é uma raiz primitiva. O inverso também é considerado verdadeiro, ou seja, g é um módulo raiz primitiva n se a ordem multiplicativa de g é $\phi(n)$. Deste axioma, obtemos um modo para testar se, para qualquer g e n , g é uma raiz primitiva módulo n , como se segue:

1. Primeiro precisamos calcular $\phi(n)$.
2. Fatoramos $\phi(n)$ em k diferentes fatores primos p_1, p_2, \dots, p_k
3. Por fim, calculamos $g^{(\phi(n)/p_i)} \bmod n$; {Para $i = 1, 2, 3, \dots, k$ }. Se todos os resultados k forem diferentes de 1, então g é uma raiz primitiva módulo n .

```
public class PrimitiveRootCalculator {

    private static final BigInteger ONE = BigInteger.ONE;
    private static final BigInteger ZERO = BigInteger.ZERO;

    public synchronized static BigInteger calculate(BigInteger value) {
        BigInteger q = value.subtract(ONE);
        List<BigInteger> primeFactors = primeFactorsOf(q);
        List<BigInteger> coprimesAlreadyTested = new ArrayList<BigInteger>();
        BigInteger coprime = ONE;
        while (true) {
            coprime = getOneCoprimeOfValueThatIsNotInList(value, coprimesAlreadyTested);
            if (coprime == null) {
                return null;
            }
            if (verify(value, primeFactors, q, coprime)) {
                return coprime;
            }
            coprimesAlreadyTested.add(coprime);
        }
    }

    private static boolean verify(BigInteger value, List<BigInteger> primeFactors, BigInteger q, BigInteger coprime) {
        for (BigInteger primeFactor : primeFactors) {
            if (coprime.modPow(q.divide(primeFactor), value).compareTo(ONE) == 0) {
                return false;
            }
        }
        return true;
    }
}
```

```

        return false;
    }
}
return true;
}

private static BigInteger getOneCoprimeOfValueThatIsNotInList(BigInteger
value, List<BigInteger> list) {
    BigInteger probableCoprime = ZERO;
    while (probableCoprime != null) {
        probableCoprime = BigIntegerRandomGenerator.generate(ZERO, value);
        if (list.contains(probableCoprime)) {
            list.add(probableCoprime);
            if (list.size() >= value.intValue()) {
                break;
            }
            continue;
        }
        if (probableCoprime.gcd(value).compareTo(ONE) == 0) {
            return probableCoprime;
        }
    }
    return null;
}

public static List<BigInteger> primeFactorsOf(BigInteger number) {
    BigInteger n = number;
    List<BigInteger> factors = new ArrayList<BigInteger>();
    for (BigInteger i = BigInteger.valueOf(2); i.compareTo(n.divide(i)) <= 0
; i = i.add(ONE)) {
        while (n.mod(i).compareTo(ZERO) == 0) {
            factors.add(i);
            n = n.divide(i);
        }
    }
    if (n.compareTo(ONE) == 1) {
        factors.add(n);
    }
    return factors;
}
}

```

calculatePrimitiveRoot()

```

▶ private static BigInteger calculatePrimitiveRoot(BigInteger prime) {
    boolean primitiveRootNotFound = true;
    BigInteger primitiveRoot = null;
    while (primitiveRootNotFound) {
        primitiveRoot = PrimitiveRootCalculator.calculate(prime);

        if (primitiveRoot != null && primitiveRoot.compareTo(ONE) == 1) {
            primitiveRootNotFound = false;
        }
    }
}

```

```

    }
}
return primitiveRoot;
}

```

sendPrimitiveRoot()

Envia a raiz primitiva calculada previamente.

```

[-] private static void sendPrimitiveRoot(BigInteger primitiveRoot) throws IOException {
    System.out.printf("Sending primitive root: %s\n", primitiveRoot);
    BufferedOutputStream bos = new BufferedOutputStream(connection.getOutputStream());
    OutputStreamWriter osw = new OutputStreamWriter(bos, "US-ASCII");
    String process = "primitiveroot:" + primitiveRoot + (char) 13;
    osw.write(process);
    osw.flush();
}

```

readResult()

Recebe do cliente o valor calculado da fórmula $B = g^b \bmod p$ em que **g** é a raiz primitiva, **p** o primo e **b** um inteiro aleatório privado do cliente.

```

[-] private static BigInteger readResult() throws IOException {
    System.out.printf("Waiting for client result\n");
    BufferedInputStream bis = new BufferedInputStream(connection.getInputStream());
    InputStreamReader isr = new InputStreamReader(bis, "US-ASCII");
    int c;
    StringBuffer instr = new StringBuffer();
    while ((c = isr.read()) != (char) 13) {
        instr.append((char) c);
    }
    String result = instr.toString().split(":")[1];
    System.out.printf("Client result received: %s\n", result);
    return new BigInteger(result);
}

```

calculateFormule()

Realiza o cálculo da fórmula $A = (\text{primitiveRoot}^{\text{secretInteger}}) \bmod \text{prime}$ em que **primitiveRoot** é a raiz primitiva, **prime** o primo e **secretInteger** um inteiro aleatório privado do servidor.

```

[-] private static BigInteger calculateFormule(BigInteger prime, BigInteger primit

```

```

iveRoot) {
    secretInteger = BigIntegerRandomGenerator.generate(decimalDigits);
    return primitiveRoot.modPow(secretInteger, prime);
}

```

sendResult()

Envia o resultado da fórmula para o cliente.

```

[-] private static void sendResult(BigInteger result) throws IOException {
    System.out.printf("Sending result: %s\n", result);
    BufferedOutputStream bos = new BufferedOutputStream(connection.getOutputStream
    ());
    OutputStreamWriter osw = new OutputStreamWriter(bos, "US-ASCII");
    String process = "result:" + result.toString() + (char) 13;
    osw.write(process);
    osw.flush();
}

```

calculateSecret()

Faz o cálculo da chave secreta, comum entre cliente e servidor. Fórmula: $s = B^a \text{ mod } p$, em que **B** é o resultado enviado pelo cliente, **a** é o inteiro secreto e **p** é o primo compartilhado.

```

[-] private static BigInteger calculateSecret(BigInteger result, BigInteger secret
Integer, BigInteger prime) {
    return result.modPow(secretInteger, prime);
}

```

Cliente

Compartilha alguns métodos com o servidor. Citarei apenas os diferentes.

```

[-] public static void main(String[] args) {
    try {
        createConnection();
        sendNewRequest();
        readDecimalDigits();
        BigInteger prime = readPrime();
        BigInteger primitiveRoot = readPrimitiveRoot();
        BigInteger resultToSend = calculateFormule(prime, primitiveRoot);
        sendResult(resultToSend);
        BigInteger result = readResult();
        BigInteger secret = calculateSecret(result, secretInteger, prime);
        System.out.printf("[Secret: %s]\n", secret);
    }
}

```



```

        connection.close();
    } catch (UnknownHostException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

createConnection()

Realiza um pedido de conexão com o host localhost e na porta 9988.

```

[-] private static void createConnection() throws IOException {
    int portNumber = 9988;
    String host = "localhost";
    InetAddress address = InetAddress.getByName(host);
    connection = new Socket(address, portNumber);
    System.out.println("--- Diffie-Hellman Client Connection Established ---");
}

```

sendNewRequest()

Faz um pedido para a troca de chaves.

```

[-] private static void sendNewRequest() throws IOException {
    System.out.println("Sending new request");
    BufferedOutputStream bos = new BufferedOutputStream(connection.getOutputStream());
    OutputStreamWriter osw = new OutputStreamWriter(bos, "US-ASCII");
    String process = "newRequest" + (char) 13;
    osw.write(process);
    osw.flush();
}

```

readDecimalDigits()

Recebe a ordem de grandeza na qual os números irão operar.

```

[-] private static void readDecimalDigits() throws IOException {
    System.out.println("Waiting for decimal digits.");
    BufferedInputStream bis = new BufferedInputStream(connection.getInputStream());
    InputStreamReader isr = new InputStreamReader(bis, "US-ASCII");
    int c;
    StringBuffer instr = new StringBuffer();
    while ((c = isr.read()) != (char) 13) {

```

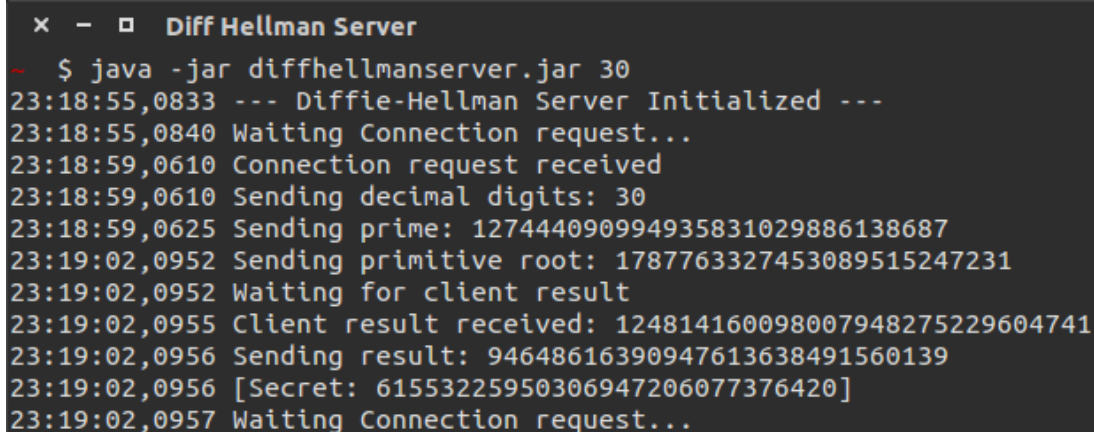
```
        instr.append((char) c);
    }
    String decimalDigitsString = instr.toString().split(":")[1];
    System.out.println("Decimal digits received: " + decimalDigitsString);
    decimalDigits = Integer.parseInt(decimalDigitsString);
}
```

Como executar?

1. Iniciar o servidor passando, opcional, o número de dígitos de trabalho. Exemplo: **java -jar diffhellmanserver.jar 30**
2. Iniciar o cliente utilizando o comando **java -jar diffhellmanclient.jar**

Em execução

Server



```
x - □ Diff Hellman Server
~ $ java -jar diffhellmanserver.jar 30
23:18:55,0833 --- Diffie-Hellman Server Initialized ---
23:18:55,0840 Waiting Connection request...
23:18:59,0610 Connection request received
23:18:59,0610 Sending decimal digits: 30
23:18:59,0625 Sending prime: 127444090994935831029886138687
23:19:02,0952 Sending primitive root: 1787763327453089515247231
23:19:02,0952 Waiting for client result
23:19:02,0955 Client result received: 124814160098007948275229604741
23:19:02,0956 Sending result: 94648616390947613638491560139
23:19:02,0956 [Secret: 61553225950306947206077376420]
23:19:02,0957 Waiting Connection request...
```

Client

```
x - □ Diff Hellman Client
~ $ java -jar diffhellmanclient.jar
23:18:59,0597 --- Diffie-Hellman Client Connection Established ---
23:18:59,0609 Sending new request
23:18:59,0610 Waiting for decimal digits.
23:18:59,0611 Decimal digits received: 30
23:18:59,0611 Waiting for prime.
23:18:59,0625 Prime received: 127444090994935831029886138687
23:18:59,0627 Waiting for primitive root.
23:19:02,0952 Primitive root received: 1787763327453089515247231
23:19:02,0954 Sending result: 124814160098007948275229604741
23:19:02,0955 Waiting for server result
23:19:02,0956 Server result received: 94648616390947613638491560139
23:19:02,0957 [Secret: 61553225950306947206077376420]
~ $ █
```