

Chat assíncrono criptografado utilizando sockets TCP em Java

Sila Geroges Agiru Judick Siebert
UDESC

Wagner Luis Sousa da Luz
UDESC

November 26, 2016

Abstract

Neste artigo, uma aplicação de bate-papo para enviar mensagens criptografadas é proposta. O algoritmo de criptografia é caracterizado por inverter valor dos bits da mensagem. A aplicação é desenvolvida usando a linguagem de programação Java e utilizando sockets TCP. A seguir são resumidas as etapas de engenharia de software seguidas durante a implementação deste projeto.

1 Introdução

Aplicativos de mensagens instantâneas tornaram-se populares sendo usados diariamente pelas pessoas. A maioria dos usuários convencionais na Internet não percebe que suas conversas estão sendo transmitidas em texto claro e são vulneráveis a espionagem durante a transmissão. O projeto foi intitulado chat assíncrono criptografado e seu objetivo principal é implementar uma sala de bate-papo com criptografia nas mensagens. Objetivos secundários foram a pesquisa sobre sockets e experiência prática dos autores no desenvolvimento de uma aplicação bate-papo baseada em Java utilizando conceitos de threads e sockets aprendidos em aula.

1.1 Requisitos

Nesta seção, descrevemos alguns dos requisitos gerais do nosso aplicativo de sala de bate-papo distribuído.

- a criptografia deve inverter valor dos bits da mensagem.
- Devem utilizar uma conexão TCP/IP
- não utilizar a classe bufferedreader e bufferedwriter

A figura 1 representa uma sala de bate-papo.

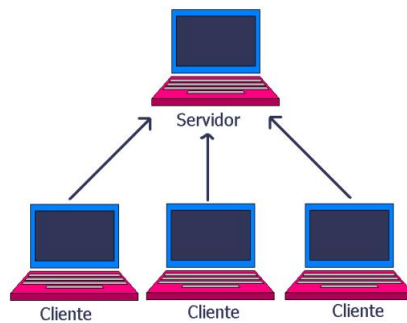


Figure 1: Sala de bate-papo

2 A sua Implementação

2.1 Visão geral da aplicação

2.1.1 Diagramas de classe

A figura 2 representa o diagrama de classe da nossa implementação.

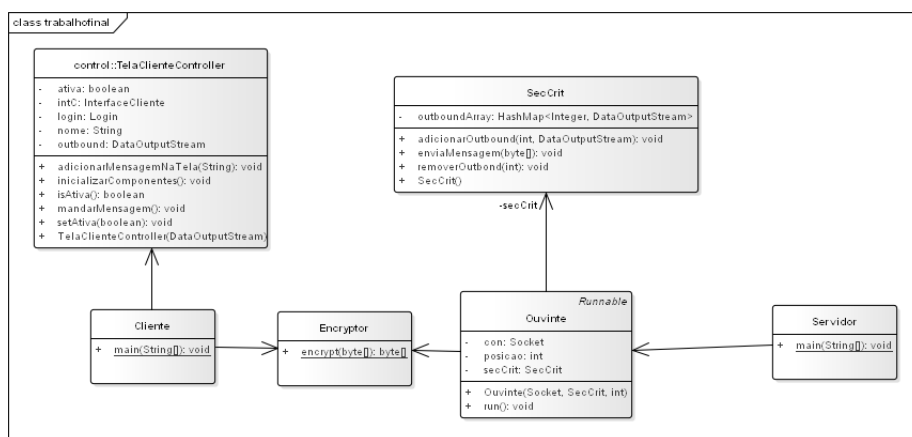


Figure 2: Diagrama de classes da aplicação

Cada cliente abre uma conexão TCP / IP para o servidor para envio e receber mensagens.

```
1      Socket clientSocket = new Socket("127.0.0.1", 6666);
2      DataInputStream inbound = new
      ↳ DataInputStream(clientSocket.getInputStream());
3      DataOutputStream outbound = new
      ↳ DataOutputStream(clientSocket.getOutputStream());
```

Figure 3: Conexão com o servidor

A tela do cliente deve ter um campo de texto no qual mensagens são exibidas à medida que chegam, e um campo para o usuário digitar mensagens de saída. Cada mensagem deve ter a hora em que foi enviada, o nome do remetente, a mensagem decriptada e uma versão encriptada de mensagem. Qualquer número de pessoas deve ser capaz de participar.

```
1      do {
2
3          if (inbound.available() > 0) {
4
5              arrayMensagem = new byte[inbound.available()];
6              for (int i = 0; i < arrayMensagem.length; i++) {
7
8                  arrayMensagem[i] = inbound.readByte();
9                  System.out.println("Byte received");
10             }
11             // ler mensagens recebidas
12             mensagemEncriptada = new String(arrayMensagem);
13             arrayMensagem = Encryptor.encrypt(arrayMensagem);
14             mensagemRecebida = new String(arrayMensagem);
15             if (!mensagemRecebida.isEmpty()) {
16                 System.out.println("Cliente recebeu mensagem " +
17                     ↳ mensagemRecebida);
18
19                 tela.adicionarMensagemNaTela(mensagemRecebida + "
20                     ↳ ----> " + mensagemEncriptada);
21             }
22         }
23     } while (tela.isAtiva());
```

Figure 4: Loop que recebe mensagens do servidor

```
1      public void adicionarMensagemNaTela(String mensagem) {
2          String textoMostrado = intC.taMensagens.getText();
3          intC.taMensagens.setText(textoMostrado + "\n" + mensagem);
4      }
```

Figure 5: Metodo para mostrar a mensagem na tela

```

1      public void mandarMensagem() throws IOException {
2          LocalDateTime horaAtual = LocalDateTime.now();
3          int horas = horaAtual.getHour();
4          int minutos = horaAtual.getMinute();
5          String min = String.valueOf(minutos);
6          if (minutos < 10) {
7              min = "0" + minutos;
8          }
9          int segundos = horaAtual.getSecond();
10         String mensagem = horas + ":" + min + ":" + segundos + " -"
11             ↪ +nome+ "- says:" + intC.edMensagem.getText();
12         if (mensagem.contains("adieu")) {
13             ativa = false;
14         }
15         byte[] buffer = mensagem.getBytes();
16         buffer = Encryptor.encrypt(buffer);
17         outbound.write(buffer, 0, buffer.length);
18         intC.edMensagem.setText("");
19     }

```

Figure 6: Metodo de envio de mensagens

2.2 Servidor

O servidor aceita um cliente com a chamada ao accept, cria uma nova Thread Ouvinte para tratar cada cliente em sua própria Thread, logo adiciona um OutputStream dele e adiciona na lista de DataOutputStreams como podemos observar no código da Figura 7. Uma nova chamada ao accept aceita um novo cliente.

```

1      while (true) {
2          // declara-se pronto receber conexoes e bloqueia ate
3          ↪ recebe-las
4          clientSocket = serverSocket.accept();
5          (new Thread(new Ouvinte(clientSocket, b,
6              ↪ numConexao))).start();
7          DataOutputStream outbound = new
8              ↪ DataOutputStream(clientSocket.getOutputStream());
9          b.adicionarOutbound(numConexao, outbound);
10         numConexao++;
11     }

```

Figure 7: Método que recebe as conexões com os clientes

Agora que vários clientes podem mandar mensagens, gostaríamos que os clientes recebessem as mensagens enviadas pelos outros. Ao invés do servidor simplesmente escrever as mensagens no console, ele deve mandar cada mensagem para todos os clientes conectados.

```

1      public synchronized void enviaMensagem(byte[] mensagemArray) throws
      ↳ IOException {
2          for (DataOutputStream o : this.outboundArray.values()) {
3              o.write(mensagemArray, 0, mensagemArray.length);
4          }

```

Figure 8: Método da classe SecCrit que distribui as mensagens para os clientes

Utilizando a lista de DataOutputStreams quando chegar uma mensagem, percorremos essa lista e mandamos uma mensagem a cada cliente individualmente.

```

1      do {
2          arrayMensagem = new byte[inbound.available()];
3          for (int i = 0; i < arrayMensagem.length; i++) {
4              arrayMensagem[i] = inbound.readByte();
5          }
6          arrayMensagemDec = Encryptor.encrypt(arrayMensagem);
7          condicao = new String(arrayMensagemDec);
8          if (!condicao.isEmpty()) {
9              if (condicao.contains("adieu")) {
10                 secCrit.removerOutbond(posicao);
11                 inbound.close();
12                 this.con.close();
13                 System.out.println(Thread.currentThread().getName()
      ↳ + " fechou a conexao");
14                 String nome = condicao.split("-")[1];
15                 String msg = nome + " saiu.";
16                 byte[] msgSaida = msg.getBytes();
17                 secCrit.enviaMensagem(Encryptor.encrypt(msgSaida));
18             } else {
19
20                 secCrit.enviaMensagem(arrayMensagem);
21                 System.out.println(Thread.currentThread().getName()
      ↳ + " recebeu de " +
      ↳ + this.con.getInetAddress() + " no instante
      ↳ + " + System.currentTimeMillis() + " : " +
      ↳ + condicao);
22             }
23         }
24     } while (!condicao.contains("adieu"));

```

Figure 9: Loop da classe Ouvinte que recebe mensagens dos clientes

2.3 Criptografia

O algoritmo de criptografia é caracterizado por inverter valor dos bits da mensagem conforme ilustrado na Figura 10.

```

1      public static byte[] encrypt(byte[] mensagem) throws
      ↳ UnsupportedOperationException {
2          byte[] encriptada = new byte[mensagem.length];
3          for (int i = 0; i < mensagem.length; i++) {
4              encriptada[i] = (byte) ~mensagem[i];
5          }
6          return encriptada;
7      }

```

Figure 10: Método que realiza a criptografia

3 Avaliação da implementação

Ao lançar o servidor ele fica esperando conexões dos clientes. Quando um cliente se conecta ao servidor, este cria uma nova thread Ouvinte que fica “ouvindo” as mensagens desse cliente. Quando uma thread Ouvinte recebe uma mensagem de um cliente ela chama um método da classe SecCrit, a seção crítica compartilhada por todas as threads ouvintes, passando a mensagem por parâmetro. Este método irá mandar a mensagem para todos os clientes percorrendo a lista de Dataoutputstreams. Quando um cliente manda uma mensagem contendo a palavra “adieu”(adeus em francês), o cliente é desconectado do servidor fechando a conexão e a thread ouvinte correspondente a este cliente chama o método para retirar o DataOutputStream da seção crítica. Durante a implementação algumas dificuldades foram surgindo. A primeira delas sendo como manter todas as mensagens sincronizadas e na ordem em todas as janelas de bate-papo. Outra dificuldade identificada foi reenviar as mensagens enviados de um cliente para o servidor, para todos os outros clientes. A última dificuldade foi implementar a criptografia. Para isto algumas estratégias foram tentadas, nenhuma funcionando até finalmente trabalhar somente com vetores de bytes.

4 Conclusão

References