Chat assíncrono criptografado utilizando sockets TCP em Java

Sila Georges Agiru Judick Siebert
UDESC

Wagner Luis Sousa da Luz
UDESC

November 29, 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.

2 A sua Implementação

2.1 Visão geral da aplicação

2.1.1 Estrutura

A figura 1 representa a estrutura da sala de bate-papo.

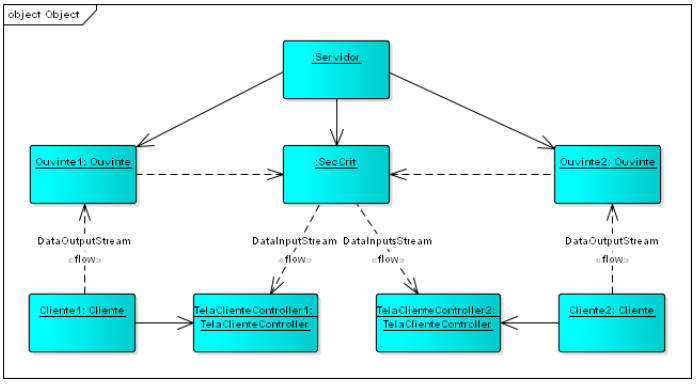

Figure 1: Sala de bate-papo

2.1.2 Diagramas de classe

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

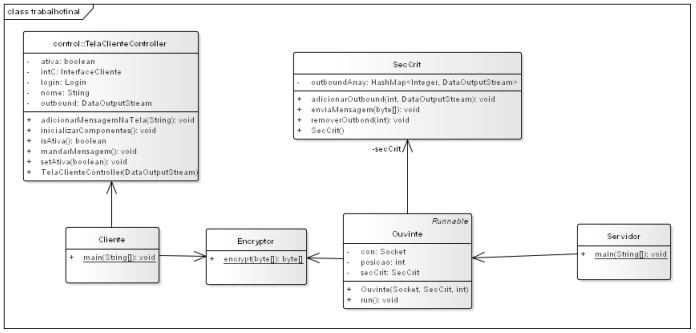

Figure 2: Diagrama de classes da aplicação

2.2 Cliente

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 - Classe Cliente

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.

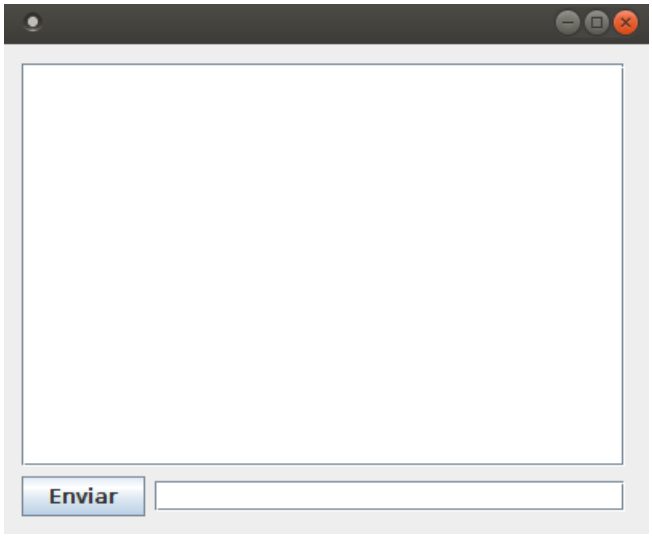

Figure 4: Tela do cliente

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.

```
            do {

                if (inbound.available() > 0) {

                    arrayMensagem = new byte[inbound.available()];
                    for (int i = 0; i < arrayMensagem.length; i++) {

                        arrayMensagem[i] = inbound.readByte();
                    }
                    mensagemEncryptada = new String(arrayMensagem);
                    arrayMensagem = Encryptor.encrypt(arrayMensagem);
                    mesagemRecebida = new String(arrayMensagem);
                    if (!mesagemRecebida.isEmpty()) {
                        tela.adicionarMensagemNaTela(mesagemRecebida + "
                            ↪ ----> " + mensagemEncryptada);
                    }
                }
            } while (tela.isAtiva());
```

Figure 5: Loop que recebe mensagens do servidor - Classe Cliente

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

Figure 6: Método para mostrar a mensagem na tela - Classe TelaClienteController

```
    public void mandarMensagem() throws IOException {
        LocalTime horaAtual = LocalTime.now();
        int horas = horaAtual.getHour();
        int minutos = horaAtual.getMinute();
        String min = String.valueOf(minutos);
        if (minutos < 10) {
            min = "0" + minutos;
        }
        int segundos = horaAtual.getSecond();
        String mensagem = horas + ":" + min + ":" + segundos + " -" +
            ↪ nome + "- says:" + intC.edMensagem.getText();
        if (mensagem.contains("adieu")) {
            ativa = false;
        }
        byte[] buffer = mensagem.getBytes();
        buffer = Encryptor.encrypt(buffer);
        outbound.write(buffer, 0, buffer.length);
        intC.edMensagem.setText("");
    }
```

Figure 7: Metodo de envio de mensagens - Classe TelaClienteController

2.3 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 8.Uma nova chamada ao accept aceita um novo cliente.

```
1               while (true) {
2                   clientSocket = serverSocket.accept();
3                   (new Thread(new Ouvinte(clientSocket, b,
                        ↪ numConexao))).start();
4                   DataOutputStream outbound = new
                        ↪ DataOutputStream(clientSocket.getOutputStream());
5                   b.adicionarOutbound(numConexao, outbound);
6                   numConexao++;
7               }
```

Figure 8: Método que recebe as conexões com os clientes - Classe Servidor

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           }
5       }
```

Figure 9: Método que distribui as mensagens para os clientes - Classe SecCrit

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                          String nome = condicao.split("-")[1];
14                          String msg = nome + " saiu.";
15                          byte[] msgSaida = msg.getBytes();
16                          secCrit.enviaMensagem(Encryptor.encrypt(msgSaida));
17                      } else {
18                          secCrit.enviaMensagem(arrayMensagem);
19                      }
20                  }
21              } while (!condicao.contains("adieu"));
```

Figure 10: Loop recebe mensagens do cliente - Classe Ouvinte

2.4 Criptografia

O algoritmo de criptografia é caracterizado por inverter valor dos bits da mensagem conforme ilustrado na Figura 11.

```
1           return encriptada;
2       }
3
4   }
```

Figure 11: Método que realiza a criptografia - Classe Encryptor

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 critica 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, a thread ouvinte correspondente

a este cliente chama o método para retirar o DataOutputStream da seção critica e fecha a thread. Durante a implementação algumas dificuldades forma surgindo. A primeira deles 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 ultima dificuldade foi implementar a criptografia. Para isto algumas estrategias foram tentadas, nenhuma funcionando ate finalmente trabalhar somente com vetores de bytes.

4 Conclusão

A implementação do aplicativo forneceu uma excelente oportunidade para o uso prático de muitas das habilidades de projeto de software que os autores desenvolveram durante os últimos 3 anos na UDESC. O aplicativo atende às especificações funcionais conforme descrito nos requisitos deste relatório. Atende ao principal objetivo de implementar uma sala de bate-papo com criptografia nas mensagens. Durante o desenvolvimento da aplicação os conceitos de sockets e threads foram aplicados na pratica e aprofundados, logo os objetivos secundários também foram atingidos.