Anterior • Trilha I • Próximo comentar

Lição 4 - Transmitindo e recebendo dados via sockets

- **Objetivo(s):** Ensinar como enviar dados entre duas aplicações remotas usando sockets.
- **Direitos autorais e licença:** Veja notas de direitos autorais e licença no final da lição.

Conteúdo

4.1 - Streams

A comunicação entre *sockets* TCP usa *streams*, que pode ser definida como um caminho lógico representando o fluxo de dados entre uma fonte e um destino.

Uma *stream* só pode ser usada para enviar (escrita) ou receber (leitura) dados. Dessa forma, se desejamos enviar e receber (ou seja, escrever e ler dados), é preciso criar duas *streams*. As classes *java.io.OutputStream* e *java.io.InputStream* são responsáveis pela criação de Streams de escrita e leitura, respectivamente.

Sempre que um *socket* é criado já existem as *streams* associadas a ele. O que devemos fazer é obter uma referência a elas. Para isso, devemos usar os métodos *getOutputStream()* e *getInputStream()*, como no código abaixo:

```
Socket socketCliente = new Socket("localhost", 9876);
OutputStream paraServidor = socketCliente.getOutputStream();
InputStream doServidor = socketCliente.getInputStream();
```

Um dos inconvenientes da comunicação entre *sockets* TCP é que só é possível transmitir *bytes* pela *stream*. Por exemplo, se quisermos enviar uma *string* pela *stream* é preciso transformar esta *string* em um *array* de *bytes*. De maneira semelhante, quando queremos receber uma *string* de um *socket*, devemos ler o conteúdo em um *array* de *bytes* e transformar este *array* na *string*. Felizmente, na API de Java já existem métodos para realizar essas transformações.

Para entender melhor como é feita a comunicação entre *sockets*, vamos estudar uma aplicação de criptografia de dados. Nessa aplicação, o servidor recebe uma senha enviada pelo cliente, criptografa essa senha e a retorna ao cliente.

Nas próximas seções não entraremos em detalhes sobre como é feita a criação de *sockets*. Para mais informações sobre isso consulte a seção "Sockets de servidor e de cliente".

4.2 - Servidor de Criptografia

Vamos iniciar nossa aplicação desenvolvendo o servidor. A primeira coisa a fazer é criar o *socket* e as *streams* com o cliente.

```
ServerSocket socketRecepcao = new ServerSocket(PORTA);
while (true) {
        Socket socketConexao = socketRecepcao.accept();
        System.out.println("Conexão estabelecida na porta " + socketConexao.getPort());

        InputStream doCliente = socketConexao.getInputStream();
        OutputStream paraCliente = socketConexao.getOutputStream();

        socketConexao.close();
}
```

Conforme já foi dito, só é possível enviar e receber *bytes* pela *stream*. Dessa forma, devemos criar um *array* de *bytes* que servirá como *buffer*. O tamanho do *array* depende da aplicação: ele deve ser suficientemente grande para comportar os dados transmitidos. No nosso exemplo vamos usar 1024. Além disso, precisamos declarar uma *String* para armazenar a senha criptografada. A princípio, precisaríamos também de uma *String* para armazenar a senha enviada pelo cliente. Porém, como veremos adiante, o método de criptografia recebe como entrada um *array* de *bytes*. Portanto, não precisaremos fazer a conversão.

```
byte[] buffer = new byte[1024];
String senhaCriptografada;
```

Agora devemos implementar o funcionamento do servidor, que deve ser o seguinte:

1. Recebe a senha do cliente;
2. Criptografa a senha recebida gerando uma nova senha;
3. Envia a nova senha ao cliente.

Para receber a senha do cliente iremos usar a *stream* de leitura e o método *read*. Como entrada do método devemos passar o *array* de *bytes* onde os dados lidos serão armazenados.

```
doCliente.read(buffer);
```

Após a chamada do método acima, os dados enviados pelo cliente serão armazenados no *array buffer*. Note que isso é feito somente

quando os dados são enviados pelo cliente, ou seja, o servidor fica bloqueado até que o cliente envie algum dado.

Agora que a senha já foi recebida, o servidor deverá criptografá-la gerando uma nova senha, o que poderá ser feito usando o código abaixo. Usamos o algoritmo MD5 para criptografar a senha. Para maiores detalhes consulte a API de Java.

```
MessageDigest md = MessageDigest.getInstance("MD5");
md.update(buffer);
BigInteger hash = new BigInteger(1, md.digest());
senhaCriptografada = hash.toString(16);
```

O código acima faz com que a senha contida no *array* de *bytes* seja criptografada e armazenada na *String* referente à nova senha. O próximo passo é enviar essa nova senha para o cliente. Como foi dito, antes de enviar devemos transformar a *String* em um *array* de *bytes*, o que é feito pelo método *getBytes* da classe String. Para facilitar podemos usar o mesmo *array*. Depois de transformar podemos enviar os dados usando a *stream* de escrita e o método *write*. O código abaixo faz as duas operações:

```
buffer = senhaCriptografada.getBytes();
paraCliente.write(buffer);
```

Com isso finalizamos nosso servidor. O código abaixo representa o código completo.

```
import java.io.InputStream;
import java.io.OutputStream;
import java.math.BigInteger;
import java.net.ServerSocket;
import java.net.Socket;
import java.security.MessageDigest;

public class ServidorCriptografia {
        public static void main(String argv[]) {
                try {
                        byte[] buffer = new byte[1024];
                        String senhaCriptografada;

                        ServerSocket socketRecepcao = new ServerSocket(PORTA);
                        System.out.println("Servidor esperando conexão na porta " + PORTA);

                        while (true) {
                                Socket socketConexao = socketRecepcao.accept();
                                System.out.println("Conexão estabelecida na porta " + socketConexao.getPort());

                                InputStream doCliente = socketConexao.getInputStream();
                                OutputStream paraCliente = socketConexao.getOutputStream();

                                doCliente.read(buffer);

                                MessageDigest md = MessageDigest.getInstance("MD5");
                                md.update(buffer);
                                BigInteger hash = new BigInteger(1, md.digest());
                                senhaCriptografada = hash.toString(16);

                                buffer = senhaCriptografada.getBytes();

                                paraCliente.write(buffer);

                                socketConexao.close();
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Baixe o código-fonte acima neste link:
http://wiki.marceloakira.com/pub/GrupoJava/TransmitindoERecebendoDadosViaSockets/ServidorCriptografia.java

4.3 - Cliente de Criptografia

O cliente também precisará enviar e receber dados com o cliente. Por isso, assim como fizemos no servidor, também precisaremos de uma *stream* de leitura e outra de escrita, que devem ser criadas após a criação do socket:

```
Socket socketCliente = new Socket("localhost", 9876);

OutputStream paraServidor = socketCliente.getOutputStream();
InputStream doServidor = socketCliente.getInputStream();
```

Também precisaremos criar um *array* de *bytes* para fazer a comunicação e duas *Strings*: uma para guardar a senha original e outra a senha criptografada. O tamanho do *array* deve ser o mesmo no servidor e no cliente.

```
String senha = "admin";
String senhaCriptografada;
byte[] buffer = new byte[1024];
```

No cliente precisaremos fazer as seguintes operações:

1. Enviar a senha original ao servidor;
2. Receber a senha criptografada.

Para enviar a senha precisaremos antes transformá-la no *array* de *bytes* usando o método *getBytes* e depois enviar o *array* para o servidor usando o método *write*:

```
buffer = senha.getBytes();
paraServidor.write(buffer);
```

O próximo passo é ler a *String* criptografada do cliente. Lembre-se que o servidor a envia através de um *array* de *bytes*, por isso devemos ler usando o *buffer* e depois transformá-lo em uma String, o que pode ser feito usando o próprio construtor da classe *String*.

```
doServidor.read(buffer);
senhaCriptografada = new String(buffer);
```

Abaixo está o código completo da aplicação no lado cliente.

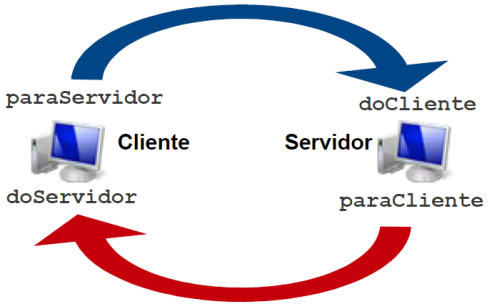
```
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;


public class ClienteCriptografia {
        public static void main(String argv[]) throws Exception {
                String senha = "admin";
                String senhaCriptografada;
                byte[] buffer = new byte[1024];

                Socket socketCliente = new Socket("localhost", 9876);

                OutputStream paraServidor = socketCliente.getOutputStream();
                InputStream doServidor = socketCliente.getInputStream();

                buffer = senha.getBytes();

                System.out.println("Cliente enviando para servidor: " + senha);
                paraServidor.write(buffer);

                doServidor.read(buffer);
                senhaCriptografada = new String(buffer);

                System.out.println("Cliente recebeu do Servidor: " + senhaCriptografada);
                socketCliente.close();
        }
}
```

Baixe o código-fonte acima neste link:
http://wiki.marceloakira.com/pub/GrupoJava/TransmitindoERecebendoDadosViaSockets/ClienteCriptografia.java

Para testar a aplicação baixe o Servidor e o Cliente. Execute primeiro o Servidor e depois o Cliente na mesma máquina.

4.4 - Entendendo melhor o exemplo

Analisando a aplicação, pode parecer a princípio que existem quatro canais de comunicação entre o servidor e o cliente: duas de entrada e duas de saída. Mas o que acontece é que existem apenas dois canais de comunicação que são "enxergados" de maneiras diferentes. A figura abaixo ilustra esse fato.



Na figura acima foram usadas duas máquinas, mas no exemplo dado o servidor e o cliente executam na mesma máquina.

Cada canal de comunicação é representado por duas *streams*: uma de leitura e outra de escrita. Sempre que se deseja enviar dados é necessário usar o método *write* com a *stream* de escrita (*OutputStream*), ao passo que para ler dados usamos o método *read* com a *stream* de leitura (*InputStream*). Sempre que uma leitura é feita, o processo fica bloqueado até que os dados sejam enviados, isto é, até que o outro lado da aplicação escreva os dados no canal de comunicação.

4.5 - Direitos autorais e licença

Anterior • Trilha I • Próximo

4.6 - Comentários

Adicionar