Transferência de Arquivos entre Computadores com o uso de Socket

Clarel de Menezes Spies Filho, Luiza Daiane Rabuski

Departamento de Computação – Universidade de Santa Cruz do Sul (UNISC)
96.815-900– Santa Cruz do Sul – RS – Brasil

clarel.filho@gmail.com, luiza.rabuski@gmail.com

Resumo. *Na camada de rede do modelo de Interconexão de Sistemas Abertos (OSI) há uma interface conhecida como socket, que oferece um ponto de comunicação entre a camada de transporte e a camada de aplicação. Este artigo apresenta como funciona um socket e o desenvolvimento de uma aplicação para transferir arquivos com o uso de socket.*

Abstract. *In network layer of the Open System Interconnection (OSI) is an interface known as socket, that provides a connection point of communication between the transport layer and the application layer. This article present how work the socket and the development of a application to transfer files with the usage of socket.*

1. Introdução

De maneira geral a rede TCP/IP disponibiliza dois protocolos de transporte para a camada de aplicação realizar a comunicação entre dois pontos, no caso, cliente e servidor. O UDP (Protocolo de Datagrama de Uusário) oferece um serviço não confiável para a comunicação, pois não é orientado a conexão, ou seja, apenas envia a mensagem e não espera um retorno a respeito do seu recebimento ou não. Já o TCP (Protocolo de Transmissão) é um serviço confiável de comunicação, pois estabelece uma conexão e após o envio de uma mensagem aguarda o retorno de sucesso ou não no recebimento da mesma pelo servidor [1].

Tanto o servidor como o cliente, independente do sistema operacional que usam, possuem diversas aplicações em execução e que podem estar realizando comunicação com outros computadores, mas para sabermos que a comunicação está sendo executada com a aplicação correta do outro lado, precisamos de uma identificação para ela, que seria o socket com uma porta especifica aberta para cada aplicação. Para evitar que uma mesma porta seja usada por duas aplicações diferentes no mesmo computador, o sistema operacional bloqueia a inicialização de uma aplicação que pretendia funcionar em uma porta que já está em uso por outra aplicação [2].

Uma porta é única em um computador, mas não na internet, então para efetuar uma conexão socket é necessário informar o endereço IP do computador e a porta de destino. Algumas aplicações se conectam com portas específicas já conhecidas entre 0 e 1023 e por isto na aplicação é necessário apenas informar o endereço. Por exemplo, quando se acessa um site HTTP o navegador já identifica e executa o serviço através da porta 80 e um site seguro que usa HTTPS é executado na porta 443, deixando as portas

transparentes para o usuário. Um número de porta é r composto por 16 bits na faixa de 0 a 65535 [1].

Executaremos a transferência de um arquivo por rede através de uma conexão socket entre máquinas de uma mesma rede interna através da aplicação desenvolvida que será apresentada, onde iniciaremos aplicação cliente e aplicação servidor, bem como definiremos o arquivo a ser enviado, o endereço do servidor e a porta que desejamos usar.

2. Funcionamento de um Socket

Uma interface de programação de aplicativo (API) de sockets TCP geralmente são fornecidos pelo sistema operacional para o uso de um socket de rede, que é usado como um ponto final em um fluxo de comunicação entre dois aplicativos realizada através de uma rede. Uma API de socket TCP geralmente é baseada no padrão de Berkeley [3].

Tabela 1. AS PRIMITIVAS DE SOQUETES PARA TCP (TANENBAUM, 2011, p. 518)

Primitiva	**Significado**
SOCKET	Criar um novo ponto final de comunicação
BIND	Anexar um endereço local a um soquete
LISTEN	Anunciar a disposição para aceitar conexões; mostra o tamanho da fila
ACCEPT	Bloquear o responsável pela chamada até uma tentativa de conexão ser recebida
CONNECT	Tenta estabelecer uma conexão ativamente
SEND	Enviar alguns dados através da conexão
RECEIVE	Receber alguns dados da conexão
CLOSE	Encerrar a conexão

As quatro primeiras primitivas da tabela são executadas do lado do servidor na mesma ordem descrita. Primeiramente a primitiva SOCKET aloca um espaço de tabela para ele na entidade de transporte e cria um novo ponto final. Um socket recém criado não possui um endereço de rede e este endereço é atribuído através da chamada da primitiva BIND. No momento que o servidor informou um endereço para um socket, este já pode receber uma conexão realizada por um cliente remoto. A primitiva LISTEN aloca espaço para uma fila de chamadas recebidas, caso vários clientes se conectem ao mesmo tempo. Para bloquear a espera por uma conexão de entrada é executada a primitiva ACCEPT.

Quando é solicitada uma conexão, a entidade de transporte cria um novo socket com as mesmas propriedades do socket que originou a requisição e retorna um descritor de arquivos normal, que pode ser usado para ler e gravar de maneira padrão. O servidor então desvia uma thread ou processo para tratar essa nova conexão [3].

Do lado do cliente também é necessário primeiramente criar um socket através do uso da primitiva SOCKET, iniciando então o processo de conexão através da chamada da primitiva CONNECT. A conexão é concluída no momento em que recebemos do servidor uma chamada de segmento e o processo do cliente é desbloqueado para que a conexão seja estabelecida. Quando a conexão estiver estabelecida tanto o cliente como o servidor podem utilizar as primitivas SEND e RECEIVE para enviar e receber dados através de uma conexão full-duplex. O encerramento da conexão é simétrico, acontece quando ambos os lados executam a primitiva CLOSE [3].

3. Sockets em Java

Para a nossa aplicação foi usada a linguagem de programação Java com o uso da biblioteca java.net, que faz a abstração para o desenvolvedor da comunicação por sockets.

Para o cliente foi usada a classe java.net.Socket que representa a conexão entre cliente e servidor. A instanciação da classe recebe um endereço IP e a porta pela qual se conectará ao servidor e seus principais métodos são getInputStream, que representa os dados que serão lidos do arquivo no canal de entrada e getOutputStream, que representa os dados sendo escritos no canal de saída e que serão transportados [2].

Para o servidor foi usada a classe java.net.ServerSocket, que representa a comunicação do lado do servidor. A instanciação da classe recebe como entrada uma porta que será a qual a aplicação deverá escutar e receber os dados transferidos. Um dos métodos importantes implementados pelo ServerSocket seria o accept, que espera que uma conexão seja estabelecida e então encapsula uma instância da classe Socket que seria a comunicação sendo recebida do cliente [5].

Para a informação ser transferida foi usada a biblioteca java.io, que são usadas para a entrada e a saída de dados. Para a entrada de dados é usado a classe java.io.BufferedInputStream, que representa um fluxo de dados contínuo e para a sua instanciação recebe o retorno de Socket.getInputStream. Para a leitura dos dados usados o método read que retorna -1 quando o fluxo de dados esperado foi totalmente lido [6].

A classe java.io.BufferedOutputStream representa a saída e a escrita final dos dados e sua instanciação recebe o retorno de Socket.getOutputStream. Seus métodos são write, que recebe bytes ou vetor de bytes para a escrita, e flush para forçar a saída dos dados dos buffer e então realizar a saída dos dados [7].

4. Aplicação cliente

```
12 public class ClienteSocket {
13
14     public static void main(String[] args) throws IOException {
15         // Abre uma conexão socket com o endereço e a porta especificados
16         Socket socket = new Socket("localhost", 2016);
17
18         File arquivoSeraTransferido = new File("/home/clarel/Documentos/apache-tomcat-7.0.69.tar.gz");
19
20         // Lenndo todos os bytes do arquivo, para depois serem enviados.
21         byte[] arrayDeBytesDoArquivo = new byte[(int) arquivoSeraTransferido.length()];
22         FileInputStream fis = new FileInputStream(arquivoSeraTransferido);
23         BufferedInputStream bis = new BufferedInputStream(fis);
24         DataInputStream dis = new DataInputStream(bis);
25         dis.readFully(arrayDeBytesDoArquivo, 0, arrayDeBytesDoArquivo.length);
26         OutputStream os = socket.getOutputStream();
27
28         // Enviando o NOME e o TAMANHO do arquivo para o server
29         DataOutputStream dos = new DataOutputStream(os);
30         dos.writeUTF(arquivoSeraTransferido.getName());
31         dos.writeLong(arrayDeBytesDoArquivo.length);
32         dos.write(arrayDeBytesDoArquivo, 0, arrayDeBytesDoArquivo.length);
33         dos.flush();
34
35         // Enviando o ARQUIVO para o server
36         os.write(arrayDeBytesDoArquivo, 0, arrayDeBytesDoArquivo.length);
37         os.flush();
38
39         // Fecha as conexões com o socket
40         os.close();
41         dos.close();
42         socket.close();
43     }
```

Figura 1. Algoritmo desenvolvido para a aplicação cliente

No código do ClienteSocket podemos ver como foi desenvolvida a comunicação do lado do cliente. Na linha 16 é aberta a conexão com o servidor e caso o mesmo não esteja disponível, já somos informados de erro de comunicação recusada, caso o servidor seja encontrado mas a porta não esteja disponível, ou como host desconhecido caso não encontre o servidor na rede. Da linha 21 até a 26, é lido o arquivo especificado e adquiridas as informações no modelo que o socket necessita, em forma de bytes. Também são adquiridas informações como nome do arquivo e o tamanho do mesmo.

Da linha 29 até a 33 é enviado o nome do arquivo e o tamanho total do arquivo, que o servidor precisará saber antes de salvá-lo, para saber quantos bytes precisa receber e o nome do arquivo que será salvo. A linha 36 executa o envio do arquivo, que será dividido em diversas mensagens pelas camadas mais baixas conforme a necessidade em relação ao tamanho do arquivo e nas linhas seguintes é fechado o buffer do arquivo e a conexão pelo lado do cliente.

5. Aplicação servidor

```
public class ServidorSocket extends Thread {
	public static void main(String[] args) throws IOException {
		int bytesRead;

		// Abre o socket do servidor na porta especificada
		ServerSocket serverSocket = new ServerSocket(2016);

		while (true) {
			Socket socketDoCliente = serverSocket.accept();

			InputStream in = socketDoCliente.getInputStream();

			DataInputStream dataRecebidaDoCliente = new DataInputStream(in);

			String fileName = dataRecebidaDoCliente.readUTF();
			OutputStream output = new FileOutputStream("/home/clarel/" + fileName);
			long tamanhoDoArquivo = dataRecebidaDoCliente.readLong();
			byte[] buffer = new byte[1024];
			while (tamanhoDoArquivo > 0 && (bytesRead = dataRecebidaDoCliente.read(buffer, 0, (int) Math.min(buffer.length, tamanhoDoAr
				output.write(buffer, 0, bytesRead);
				tamanhoDoArquivo -= bytesRead;
			}

			// Fecha o arquivo que foi escrito e a conexão com o cliente
			in.close();
			dataRecebidaDoCliente.close();
			output.close();
```

Figura 2. Algoritmo desenvolvido para a aplicação servidor

No código do ServidorSocket podemos ver como foi desenvolvida a comunicação pelo lado do servidor. Na linha 15 podemos ver como é instanciado o servidor, informamos a porta onde nossa aplicação através de uma thread ficará escutando e esperando uma conexão, quando a mesma é recebida na linha 19 é então instanciado o socket do lado do servidor. Na linha 22 e 23 vemos os dados da primeira mensagem enviada que continha o nome do arquivo e o tamanho do mesmo. Na linha 27 definimos onde será salvo o arquivo, e da linha 30 até a 35 é executada a leitura dos dados sendo recebidos pela porta e a escrita do buffer do arquivo, que é executada até receber o total de bytes especificados na mensagem anterior, e na linha 38 até a 40 são finalizados o buffer do arquivo e então o mesmo é salvo, e a conexão é encerrada pelo lado do servidor.

6. Ambiente de teste

Para o ambiente de teste era necessário duas máquinas para transferir o arquivo entre uma e outra, para isto foi configurado uma máquina virtual através do programa Virtual Box. Esta máquina configurada representara o servidor. Nela foi instalado o sistema operacional Windows XP, e no Virtual Box foi definido que a rede desta máquina seria recebido do host em modo bridge, o que permitiria a máquina virtual ter seu próprio endereço e fazer parte da rede interna como se fosse um computador totalmente independente e podendo se comunicar com qualquer outra máquina da rede, recebendo um endereço IP através do serviço DHCP do Router.

Abaixo segue a imagem ilustrando como está disponibilizada a máquina virtual em relação aos outros computadores na rede em formato físico.

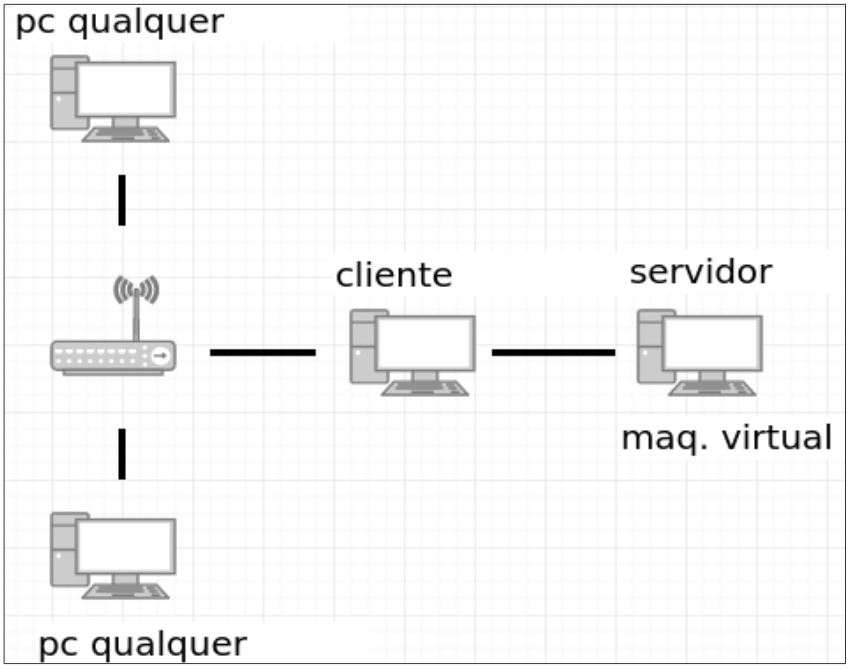


Figura 3. Visão da rede na forma física

E abaixo segue como a rede é vista pelas máquinas, e para a comunicação.

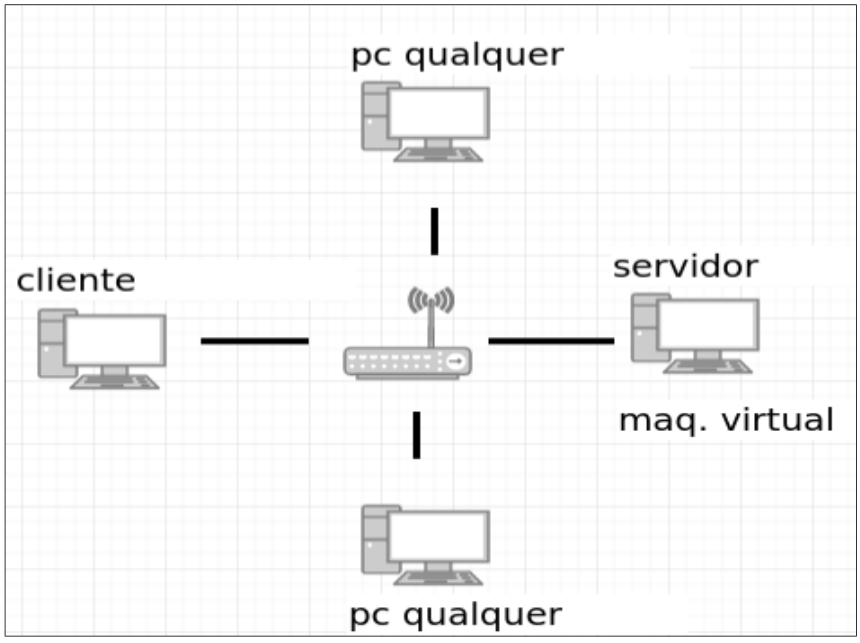
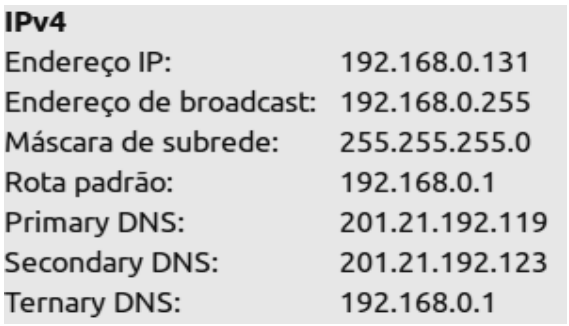


Figura 4. Visão da rede na forma virtual

Foram ilustrados os computadores chamados de PC qualquer para demonstrar claramente como seria a distribuição da rede usada para os testes caso houvesse mais máquinas. Mas há apenas a máquina cliente, e por isto foi necessária a criação da máquina virtual.

As máquinas cliente e servidor estão distribuídas na rede com os endereços que serão apresentados abaixo.



IPv4	
Endereço IP:	192.168.0.131
Endereço de broadcast:	192.168.0.255
Máscara de subrede:	255.255.255.0
Rota padrão:	192.168.0.1
Primary DNS:	201.21.192.119
Secondary DNS:	201.21.192.123
Ternary DNS:	192.168.0.1

Figura 5. Endereço de rede da máquina cliente

```
Endereço físico . . . . . : 08-00-27-A8-5E-56
DHCP ativado. . . . . : Sim
Configuração automática ativada . . : Sim
Endereço IP . . . . . : 192.168.0.133
Máscara de sub-rede . . . . . : 255.255.255.0
Gateway padrão. . . . . : 192.168.0.1
Servidor DHCP . . . . . : 192.168.0.1
Servidores DNS. . . . . : 201.21.192.119
```

Figura 6. Endereço de rede da máquina servidor

7. Testes

Para fins de estudo do funcionamento da comunicação com sockets, foi feito quatro testes diferentes para obtermos resultados específicos

7.1 Erros do lado do cliente

Neste teste tentou-se primeira realizar a comunicação primeiramente com um servidor que não existe, informando um endereço de rede que não está na rede. E no segundo teste, tentou-se conectar a um servidor que existe na rede mas através de uma porta que não possui nenhum serviço em execução do lado do cliente.

Para o teste com um endereço que não existe na rede foi tentando a comunicação com o endereço 192.168.0.140, e o resultado foi o seguinte.

```
Exception in thread "main" java.net.NoRouteToHostException: Não há rota para o host
        at java.net.PlainSocketImpl.socketConnect(Native Method)
        at java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:350)
        at java.net.AbstractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.java:206)
        at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:188)
        at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392)
        at java.net.Socket.connect(Socket.java:589)
        at java.net.Socket.connect(Socket.java:538)
        at java.net.Socket.<init>(Socket.java:434)
        at java.net.Socket.<init>(Socket.java:211)
        at br.com.clarel.ClienteSocket.main(ClienteSocket.java:16)
```

Figura 7. Erro quando o servidor não existe

Para o teste com um servidor que existe, mas em uma porta que não há um serviço do lado do servidor esperando a comunicação, foi usado o endereço 192.168.0.133 e a porta 20010 e o resultado foi o seguinte.

```
Exception in thread "main" java.net.ConnectException: Conexão recusada
        at java.net.PlainSocketImpl.socketConnect(Native Method)
        at java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:350)
        at java.net.AbstractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.java:206)
        at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:188)
        at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392)
        at java.net.Socket.connect(Socket.java:589)
        at java.net.Socket.connect(Socket.java:538)
        at java.net.Socket.<init>(Socket.java:434)
        at java.net.Socket.<init>(Socket.java:211)
        at br.com.clarel.ClienteSocket.main(ClienteSocket.java:16)
```

Figura 8. Erro quando o servidor existe mas a porta não espera conexão

7.2 Erro do lado do servidor

Para o teste do lado do servidor, tentou-se abrir uma aplicação em uma porta que já é usada e reservada pelo sistema operacional, no caso a porta 80 para comunicação HTTP, e o resultado foi o seguinte.

```
Exception in thread "main" java.net.BindException: Address already in use: JVM_Bind
        at java.net.TwoStacksPlainSocketImpl.socketBind(Native Method)
        at java.net.TwoStacksPlainSocketImpl.socketBind(TwoStacksPlainSocketImpl.java:137)
        at java.net.AbstractPlainSocketImpl.bind(AbstractPlainSocketImpl.java:387)
        at java.net.TwoStacksPlainSocketImpl.bind(TwoStacksPlainSocketImpl.java:110)
        at java.net.PlainSocketImpl.bind(PlainSocketImpl.java:190)
        at java.net.ServerSocket.bind(ServerSocket.java:375)
        at java.net.ServerSocket.<init>(ServerSocket.java:237)
        at java.net.ServerSocket.<init>(ServerSocket.java:128)
        at br.com.clarel.ServidorSocket.main(ServidorSocket.java:16)
```

Figura 9. Erro ao abrir socket servidor em porta já em uso.

7.3 Transferência de arquivo

Neste teste é finalmente realizado o envio do arquivo, foi configurado a porta e o endereço correto do servidor, no socket do cliente e informado o arquivo a ser enviado. Foi configurado o endereço IP do servidor como 192.168.0.133, e a porta 3000, e o arquivo a ser enviado está em /home/clarel/Documentos/apache-tomcat-7.0.69.tar.gz, no cliente.

No servidor configurou-se a aplicação do socket na porta 3000 para ficar no aguardo de uma conexão. Foi configurado que o arquivo recebido será salvo em C:\ no servidor. Após as configurações foi iniciado o servidor e o cliente, respectivamente e o resultados obtidos foram os seguintes.

```
Enviarei o arquivo: apache-tomcat-7.0.69.tar.gz
O arquivo tem: 8910579
```

Figura 11. Resultado do cliente ao executar aplicação

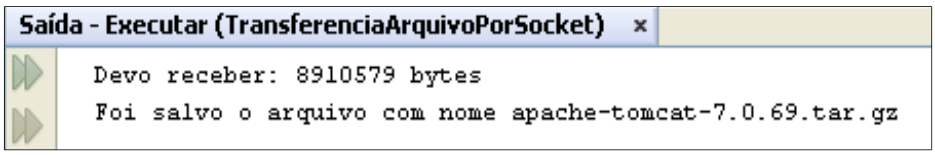


Figura 11. Resultado do servidor após receber conexão do cliente

Após rodar os testes verificou-se que o arquivo no servidor foi encontrado em C:\apache-tomcat-7.0.69.tar.gz, e pode ser aberto e executado com sucesso garantido que a transferência foi executada com sucesso.

8. Análise da transferência com Wireshark

Para fazer a análise dos dados enviados através da rede, foi instalado o Wireshark 1.10 na máquina com Windows XP. Para a análise se tornar viável foi transferido um arquivo de 4170 bytes com o nome de comprovante.html.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.0.131	192.168.0.133	TCP	74	53110 → 3000 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1…
2	0.000081	192.168.0.133	192.168.0.131	TCP	78	3000 → 53110 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 …
3	0.000258	192.168.0.131	192.168.0.133	TCP	66	53110 → 3000 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=12596077…
4	0.002890	192.168.0.131	192.168.0.133	TCP	84	53110 → 3000 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=18 TSval=12…
5	0.002920	192.168.0.131	192.168.0.133	TCP	1514	53110 → 3000 [ACK] Seq=19 Ack=1 Win=29312 Len=1448 TSval=1259…
6	0.002946	192.168.0.133	192.168.0.131	TCP	66	3000 → 53110 [ACK] Seq=1 Ack=1467 Win=65535 Len=0 TSval=53774…
7	0.003010	192.168.0.131	192.168.0.133	TCP	1514	53110 → 3000 [ACK] Seq=1467 Ack=1 Win=29312 Len=1448 TSval=12…
8	0.003018	192.168.0.131	192.168.0.133	TCP	1514	53110 → 3000 [ACK] Seq=2915 Ack=1 Win=29312 Len=1448 TSval=12…
9	0.003032	192.168.0.133	192.168.0.131	TCP	66	3000 → 53110 [ACK] Seq=1 Ack=4363 Win=65535 Len=0 TSval=53774…
10	0.003073	192.168.0.131	192.168.0.133	TCP	1514	53110 → 3000 [ACK] Seq=4363 Ack=1 Win=29312 Len=1448 TSval=12…
11	0.003082	192.168.0.131	192.168.0.133	TCP	1514	53110 → 3000 [ACK] Seq=5811 Ack=1 Win=29312 Len=1448 TSval=12…
12	0.003093	192.168.0.133	192.168.0.131	TCP	66	3000 → 53110 [ACK] Seq=1 Ack=7259 Win=65535 Len=0 TSval=53774…
13	0.003133	192.168.0.131	192.168.0.133	TCP	1174	53110 → 3000 [FIN, PSH, ACK] Seq=7259 Ack=1 Win=29312 Len=110…
14	0.003147	192.168.0.133	192.168.0.131	TCP	66	3000 → 53110 [ACK] Seq=1 Ack=8368 Win=64427 Len=0 TSval=53774…
15	0.003803	192.168.0.133	192.168.0.131	TCP	66	3000 → 53110 [FIN, ACK] Seq=1 Ack=8368 Win=64427 Len=0 TSval=…
16	0.003952	192.168.0.131	192.168.0.133	TCP	66	53110 → 3000 [ACK] Seq=8368 Ack=2 Win=29312 Len=0 TSval=12596…

Figura 12. Captura realizada pelo Wireshark

Os pacotes 1 ao 3 representam o estabelecimento de uma conexão entre cliente e servidor, no primeiro pacote o cliente monta um segmento com flag SYN ligada, um número de porta fonte e um número de sequência inicial. No segundo pacote o servidor aloca recursos iniciando um número de sequência e respondendo com um segmento SYN, ACK. E no terceiro pacote o cliente reconhece o segmento SYN do servidor e respondendo com um segmento ACK. São os três passos necessários para o estabelecimento de uma conexão, e deste momento em diante os dados podem ser solicitados ou enviados [8].

No pacote 4 o cliente avisa o servidor que iniciaria o envio imediato dos dados, que ocorre logo após o estabelecimento de uma conexão e esta transferência ocorre entre os pacotes 5 e 12, com o cliente enviando os dados e o servidor respondendo com a confirmação do recebimento. No pacote 13 o servidor envia um segmento com o

cabeçalho FIN, PSH, ACK, que significa o fechamento da conexão com o cliente para o envio imediato de dados e no pacote 14 o cliente responde com a confirmação do recebimento da mensagem. No pacote 15 o cliente envia o segmento com cabeçalho FIN, ACK para finalizar a conexão do cliente com o servidor, e no pacote 16 o servidor responde com a confirmação do encerramento.

9. Considerações finais

O estudo realizado para o desenvolvimento do software foi importante para a aprendizagem e o entendimento mais amplo sobre o protocolo TCP/IP e entender como funciona a comunicação entre computadores através de sockets. O aplicativo desenvolvido juntamente com a análise feita pelo Wireshark permitiu de forma prática entender como ocorre a comunicação e transferência de dados entre máquinas, e o objetivo das portas nos sistemas operacionais.

A aplicação, o artigo e o arquivo para análise gerado pelo Wireshark estão disponíveis em https://github.com/clarel/TransferenciaArquivoPorSocket para quem tiver interesse em estudar mais a fundo o artigo e o software desenvolvido.

Referências

[1] Kurose, J.,Ross,K. (2013) Redes de computadores e a internet: uma abordagem top-down. 6.ed., Pearson.

[2] Gonzaga, T. (2015) Criando seu próprio servidor HTTP do zero (ou quase) – Parte II", http://tableless.com.br/criando-seu-proprio-servidor-http-do-zero-ou-quase-parte-ii/, acesso em 10 maio 2016.

[3] Tanenbaum, A. S. (2011) Redes de computadores. Pearson.

[4] Oracle. "Class Socket", https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html, acesso em 11 maio 2016.

[5] Oracle. "Class ServerSocket", https://docs.oracle.com/javase/7/docs/api/java/net/ServerSocket.html, acesso em 11 maio 2016.

[6] Oracle. "Class BufferedInputStream", https://docs.oracle.com/javase/7/docs/api/java/io/BufferedInputStream.html, acesso em 11 maio 2016.

[7] Oracle. "Class BufferedOutputStream", https://docs.oracle.com/javase/7/docs/api/java/io/BufferedOutputStream.html, acesso em 11 maio 2016.

[8]"Aula 17 – Redes de Computadores", http://www.mfa.unc.br/info/carlosrafael/rco/aula17-1.pdf, acesso em 12 maio 2016.