Redes de Computadores
2016/02

# Trabalho Prático:
# Implementação de uma Pilha TCP/IP Simulada

Ana Cláudia
Bruno Maciel
Gustavo Borba
Thiago Alexandre
Belo Horizonte, 4 de fevereiro de 2017

# Sumário

# 1 Introdução

O presente trabalho apresenta a implementação de uma pilha TCP/IP simulada. Não é a intenção desse trabalho obter um modelo de performance, mas um modelo didático que simule razoavelmente bem os protocolos discutidos na disciplina de Redes de Computadores, do curso de Engenharia de Computação do CEFET-MG.

Proposto por Sandro Renato Dias [https://sites.google.com/site/sandrord] [http://lattes.cnpq.br/5300421458375793], este trabalho foi desenvolvido por um grupo de quatro integrantes:

- Gustavo Henrique de Souza Borba [gustavohsborba@gmail.com]
- Bruno Marques Maciel [bmarques.maciel@gmail.com]
- Thiago Alexandre de Souza Silva [thiagoalexsilva93@gmail.com]
- Ana Cláudia Gomes de Mendonça [gmanaclaudia@gmail.com]

O Código fonte deste trabalho pode ser encontrado na página do github: https github.comgustavohsborbaTCP-IP-Model
. Infelizmente, este trabalho não é multiplataforma, e só é passível de ser executável em sistemas linux.

# 2 Executando o Simulador

## 2.1 Baixando código-fonte

O Github é uma plataforma de versionamento de códigos gratuita. Dessa forma, basta acessar a referida página e realizar o download do código-fonte. Você pode optar também por clonar o projeto, via HTTPS, com o comando:

```
1 | git clone https://github.com/gustavohsborba/TCP-IP-Model.git
```

## 2.2 Instalando as Dependências

Foi criado um script para instalar automaticamente todas as dependências do projeto.
Acesse a pasta do trabalho pelo terminal. No modo root, execute o script install_dependencies.sh para instalar as linguagens e frameworks necessários para a execução das camadas.
Caso esteja ressabiado sobre executar um comando de superusuário em um

terminal, Não há motivos para preocupações: Verificado o script, nota-se que ele apenas instala as linguagens python, scala, o compilador gcc, um gerenciador de terminais chamado Terminator, e algumas outras dependências.

## 2.3 Alterando variáveis importantes

Algumas variáveis do simulador acabaram por depender do sistema operacional que o executa. O motivo é simples: a aplicação ¡ainda¿ não sabe procurar o seu IP e interface de rede (uma sugestão futura é implementar um parser do comando ifconfig que seta essas configurações automaticamente. para que o progrma do cliente funcione, os IPs devem ser manualmente setados diretamente no código:

- No arquivo /application-scala/MiniBrowser.scala, por volta da linha 170:
  var localhostAddress = "127.0.0.1"
- No arquivo /application-scala/application-client.scala, por volta da linha 18:
  var localhostAddress = "127.0.0.1"
- No arquivo /transport-python/transport-client.py, por volta da linha 7:
  LOCALHOST = "127.0.0.1"
- No arquivo /internet-php/internet-client.php, por volta da linha 7:
  $LOCALHOST_IP = '127.0.0.1';

Também é necessário alterar a variável que indica o nome de sua interface de rede, para um comando bem específico que identifica o MAC Address da máquina. Para saber o nome da sua interface de rede, execute o comando ifconfig, e verifique a interface que aparece um IP privado, como no exemplo a seguir:

```
1  ifconfig
2  enp2s0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
3          ether 50:b7:c3:00:5c:89  txqueuelen 4096  (Ethernet)
4          RX packets 0  bytes 0 (0.0 B)
5          RX errors 0  dropped 0  overruns 0  frame 0
6          TX packets 0  bytes 0 (0.0 B)
7          TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
8
9  lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
10         inet 127.0.0.1  netmask 255.0.0.0
```

```
11          inet6 ::1  prefixlen 128  scopeid 0x10<host>
12          loop  txqueuelen 1  (Loopback Local)
13          RX packets 897  bytes 68858 (67.2 KiB)
14          RX errors 0  dropped 0  overruns 0  frame 0
15          TX packets 897  bytes 68858 (67.2 KiB)
16          TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
17
18  eth0:   flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
19          inet 192.168.0.36  netmask 255.255.255.0  broadcast 192.168.0.255
20          inet6 2804:14c:5b91:967a:c685:8ff:fe68:11ef  prefixlen 64  scopeid 0x0
        <global>
21          inet6 fe80::c685:8ff:fe68:11ef  prefixlen 64  scopeid 0x20<link>
22          ether c4:85:08:68:11:ef  txqueuelen 4096  (Ethernet)
23          RX packets 428248  bytes 385214800 (367.3 MiB)
24          RX errors 0  dropped 0  overruns 0  frame 0
25          TX packets 172962  bytes 22292022 (21.2 MiB)
26          TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

E, no arquivo /physical-c/physical.h, por volta da linha 39 (um dos primeiros #define do código), altere a linha que define a interface de rede para utilizar o valor encontrado no comando ifconfig:

```
1  #define NETWORK_INTERFACE "eth0"
```

### 2.4 Executando o Programa

Em um terminal, entre na pasta test e execute o arquivo test.sh.

```
1  ./test.sh
```

Caso haja dúvidas sobre sua execução, execute o comando:

```
1  ./test.sh -h
```

Essa opção mostrará o menu de ajuda, que exibe todas as opções possíveis do arquivo test.sh

## 3 Decisões de Implementação

### 3.1 A Estrutura do Projeto

Além do código-fonte do projeto, realizado em uma linguagem diferente para cada camada, o sistema conta com uma ferramenta útil para dividir terminais, chamada Terminator. Esta ferramenta permite que se abram vários terminais simultâneos ao mesmo tempo, de forma a facilitar o processo de execução do projeto ao executar todas as camadas necessárias com apenas um comando.

O código-fonte do pojeto contém uma pasta para cada camada do protocolo, uma pasta para rodar os testes, e uma pasta que armazena os layouts do software Terminator. Cada uma das pastas contém, ao menos, dois arquivos - um para o servidor e um para cliente:

- application-scala
  - application-client.scala
  - application-server.scala
  - MiniBrowser.scala
  - index.html
  - 404.html
- transport-python
  - transport-client.py
  - transport-server.py
- internet-php
  - internet-client.php
  - internet-server.php
- physical-c
  - physical-client.c
  - physical-server.c
  - physical.h
- test
  - test.sh
- terminator-layouts
  - client.layout
  - server.layout
  - testing.layout

Outros arquivos necessários para a execução - As páginas utilizadas pela aplicação - foram adicionados nas pastas usadas pela sua respectiva camada da pilha. Layouts para a aplicação Terminator foram criados para que se pudesse rodar o programa de diferentes formas: Como cliente, como servidor, como teste (abrindo na mesma máquina as camadas para as quatro camadas de cada pilha - totalizando oito terminais), etc.

## 3.2 Abordagem utilizada na implementação

A implementação deste trabalho, primeiramente, deu-se pelo uso de arquivos para escrever e ler as informações dos PDU's de cada camada. Sendo assim, cada camada lia o arquivo resultado da camada superior, fazia as respectivas análises e alterações que lhe competia e salvava estas em um novo arquivo para envio e uso da próxima camada. Neste ponto, cada camada precisava ser executada separadamente em um terminal independente.

Diante da crescente complexidade deste método de implementação, o uso de arquivos foi adaptado e passou-se a implementar a comunicação entre as camadas por meio de chamadas por linha de comando em bash. No entanto, esta abordagem não permitia a real análise da execução das camadas. Adicionalmente, a medida que as informações dos cabeçalhos dos PDU's eram passados para as camadas inferiores, a quantidade de parâmetros incrementava, ficando, assim, inviável.

Ao ponto da implementação da camada de rede, constatou-se que a utilização de sockets era impressindivel. Deste modo, todas camadas realizavam suas comunicações por meio destes. A numeração das portas de cada socket foram determinadas a priori a execução, assim como os IP's de Localhost do cliente e servidor.

Ainda persistia-se a necessidade de abrir um terminal para cada instância de uma camada, contabilizando, portanto, 8 terminais abertos um por um. Por sua vez, se uma camada fosse aberta fora de ordem, haveria problema de comunicação entre as portas de seus sockets e todo o processo precisava ser refeito. Logo, afim de simplificar a execução e torná-la praticável, utilizou-se o Terminator. Configurando os layouts deste terminator, foi possível abrir os 8 terminais na ordem correta em apenas uma instância.

No âmbito de executar a simulação em sua plena definição primária, um script foi criado para que as máquinas cliente e servidora pudessem executar as próprias versões das camadas TCP. Como resultado, cada máquina continha uma instância do Terminator com quatro terminais. No cliente, o terminal referente à camada de aplicação demandava a inserção manual do IP da maquina servidora e da página que se deseja requerir; posteriormente,

as camadas se comunivam e o conteudo da página era impresso no terminal ao final.

Concomitantemente, para uma verdadeira experiência da camada de Aplicação, um mini-browser foi implementado em Scala para ser executado na área do cliente, de forma que o usuário pudesse utilizar de uma interface simplificada para realizar requisições de arquivos, como páginas html, por exemplo. Caso o arquivo requisitado não estivesse presente no diretório da servidora, uma página de erro 404 foi também criada como resposta ao erro.

Por conseguinte, este trabalho pode ser executado em três diferentes situações: Uma única máquina (sendo cliente e servidor como localhost - chamada de "modo teste") e com duas máquinas (cliente e servidor), onde a simulacao pode ocorrer apenas pelas instâncias do Terminator ou Termiantor/Mini-browser.