Universidade Estadual de Campinas

Instituto de Computação - MC833 - A 2s2020

Atividade 2.2: Servidor TCP Concorrente

Luana Felipe de Barros **RA: 201705**

1. Adicionando-se a função sleep com tempo de 10 segundos, antes que o servidor fechasse a conexão TCP, fez com que o servidor conseguisse conectar vários clientes na mesma porta, porém de forma sequencial em vez de concorrente.

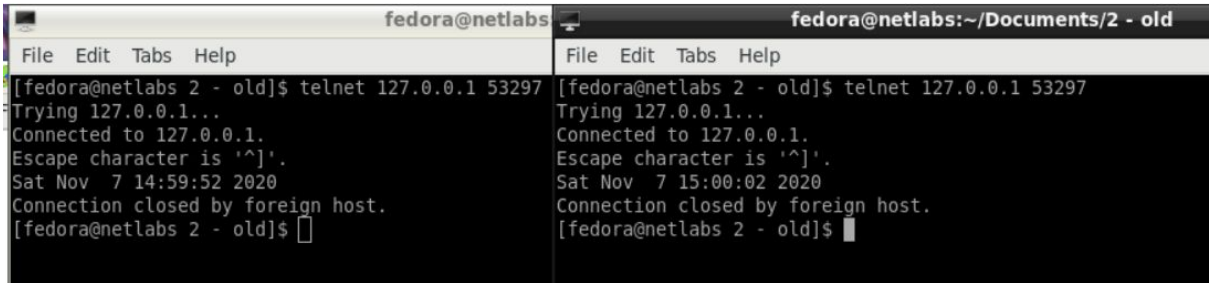
   Se um cliente requisitasse um pedido de conexão, e logo em seguida outro cliente fizesse o mesmo, o servidor respondia o handshake para os dois clientes, mas trocava os dados da data e hora apenas um de cada vez. Logo, quando a conexão com o cliente 1 se encerrava, o servidor atendia o cliente 2. Portanto, o servidor atende cada cliente em uma porta específica de cada vez, onde atender remete a trocar dados.

   O servidor forneceu a porta 51895 e aceitava conexões em todos os IPs, conectamos ao 127.0.0.1. Utilizei netstat em 4 momentos diferentes.

   a) No primeiro, o cliente 1 e o cliente 2 requisitaram a conexão na porta 51895. Podemos ver os estados da conexão TCP, e vemos que o servidor tem um socket que escuta conexões o tempo todo, independente se há uma conexão ativa. Vemos também que houve mensagens de handshake trocadas de entre os clientes e o servidor e as conexões foram estabelecidas.
   b) Nesta fase, passou-se 10 segundos (tempo de espera para o servidor fechar a conexão) e então o cliente 1 teve sua conexão encerrada por parte do servidor, ficando com o estado TIME_WAIT. Depois disso, há uma espera de tempo para ter certeza de que o TCP remoto (cliente 1) recebeu a confirmação de sua solicitação de encerramento de conexão. Se o servidor fosse concorrente, creio que não iria existir o cliente 2 com o estado ESTABLISHED nesta fase, pois ele teria sido executado no mesmo momento que o cliente 1, e teria o estado TIME_WAIT.
   c) Neste momento, o cliente 2 obtém resposta do servidor, já que o cliente 1 teve sua conexão encerrada. Além disso, depois de 10 segundos o servidor também encerra a sua conexão, ficando como TIME_WAIT.
   d) Aqui podemos ver que não há conexões estabelecidas e nem esperando respostas de ACK. Todas as conexões foram atendidas pelo servidor, que continua esperando por novas conexões.

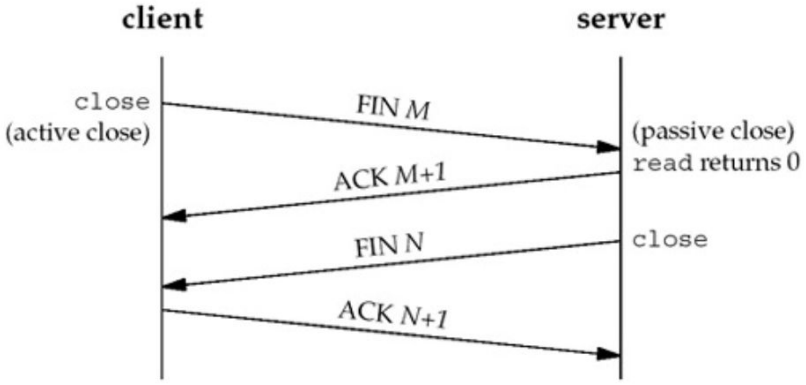
```
[fedora@netlabs ~]$ netstat -an | grep 51895
tcp        1      0 0.0.0.0:51895           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:48916         127.0.0.1:51895         ESTABLISHED
tcp        0      0 127.0.0.1:51895         127.0.0.1:48916         ESTABLISHED
tcp        0      0 127.0.0.1:48914         127.0.0.1:51895         ESTABLISHED
tcp        0      0 127.0.0.1:51895         127.0.0.1:48914         ESTABLISHED
[fedora@netlabs ~]$ netstat -an | grep 51895
tcp        0      0 0.0.0.0:51895           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:48916         127.0.0.1:51895         ESTABLISHED
tcp        0      0 127.0.0.1:51895         127.0.0.1:48916         ESTABLISHED
tcp        0      0 127.0.0.1:51895         127.0.0.1:48914         TIME_WAIT
[fedora@netlabs ~]$ netstat -an | grep 51895
tcp        0      0 0.0.0.0:51895           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:51895         127.0.0.1:48916         TIME_WAIT
tcp        0      0 127.0.0.1:51895         127.0.0.1:48914         TIME_WAIT
[fedora@netlabs ~]$ netstat -an | grep 51895
tcp        0      0 0.0.0.0:51895           0.0.0.0:*               LISTEN
[fedora@netlabs ~]$
```

Depois disso, optei pela utilização do comando telnet, visto que ele mostra a resposta com o timestamp. Na figura abaixo, ao conectar 2 clientes quase simultaneamente à mesma porta 53297, vemos que o cliente da direita recebe o retorno apenas 10 segundos depois do à esquerda.



5. Implementei meu código de forma que o cliente termina a conexão TCP de forma ativa, seja quando o usuário digita ‘exit’ ou quando todas as respostas foram enviadas para o servidor. Desta forma, não há mais tarefas para o cliente realizar e por isso, o mesmo solicita o encerramento da conexão, através da função Close(). Neste momento, um byte FIN é enviado para o servidor, indicando esta solicitação. Como não há mais dados para o servidor receber, este também finaliza a conexão com Close(), tanto no processo filho quanto no pai, enviando também um pacote FIN para o cliente. Note que todo este processo é realizado sobre o socket de conexão, e não no socket listening do servidor. Inclusive, o socket listening não é finalizado de forma voluntária no código, ele sempre continua ativo escutando por futuras conexões. Então, seu funcionamento não é responsável por enviar pacotes FIN para os clientes, o que é feita na finalização de sockets de conexões. Abaixo há duas figuras ilustrando o processo de terminar a conexão. Na Figura 2.3 (Retirada do Livro), vemos que a ordem que os pacotes com a flag FIN são enviados.

Figure 2.3. Packets exchanged when a TCP connection is closed.



Na figura abaixo, é print da execução que fiz para ver se de fato os pacotes de finalização entre um cliente (porta 53262) e servidor (porta 1024) estavam sendo entregues. Neste comando, 'lo' é a interface de rede responsável por esta conexão. Utilizei ipconfig para descobrí-la anteriormente.

```
[fedora@netlabs ~]$ sudo tcpdump -i 'lo' 'tcp[13] & 1 != 0'
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
21:34:52.348013 IP localhost.1024 > localhost.53262: Flags [F.], seq 1408897154,
 ack 1133500525, win 351, options [nop,nop,TS val 1875880768 ecr 1875877768], le
ngth 0
21:34:52.348532 IP localhost.53262 > localhost.1024: Flags [F.], seq 1, ack 1, w
in 342, options [nop,nop,TS val 1875880768 ecr 1875880768], length 0
```

6. Primeiro, procurei o pid do processo do servidor que estava atendendo 2 clientes, depois mostrei a árvore deste processo, com os pids dos filhos.

```
[fedora@netlabs ~]$ ps -aux | grep './servidor 1024'
fedora    9829  0.0  0.0   2304   756 pts/7    S+   15:11   0:00 ./servidor 1024
fedora    9900  0.0  0.0 213212   944 pts/0    S+   15:15   0:00 grep --color=auto ./servidor 1024
[fedora@netlabs ~]$ pstree -p 9829
servidor(9829)─┬─servidor(9833)
               ├─servidor(9857)
               └─servidor(9891)
[fedora@netlabs ~]$
```

Tutorial para Execução do Programa

1. Compile o arquivo servidor.c desta forma:

```
gcc servidor.c -o servidor -Wall
```

2. Compile o arquivo cliente.c desta forma:

```
gcc cliente.c -o cliente -Wall -lpthread
```

3. Execute o servidor desta forma:

   ```
   ./servidor <port_number>
   ```

   Irá então aparecer a informação do socket local para que o cliente possa se conectar.

   Digite a quantidade de comandos e depois os comandos separados por enter. Exemplo:

   ```
   ~ programa: Enter a number of commands:

   ~ usuario: 3

   ~ usuario: ls

   ~ usuario: pwd

   ~ usuario: ls -l
   ```

   Depois disso, o servidor espera algum cliente se conectar, e guarda as respostas de cada cliente em um arquivo chamado “log.txt” que se encontra na mesma pasta dos programas. O código para mostrar o resultado do cliente e suas informações está comentado. Atualmente, a única saída mostrada são o ip e porta de cada cliente no momento em que o servidor recebe um resultado do mesmo, porém sem o resultado.

   No programa, há um sleep(3) que coloquei propositalmente para que fosse possível testar o “exit” do cliente, já que tudo acontece muito rápido.

4. Execute o cliente da seguinte forma:

   ```
   ./cliente 127.0.0.1 <port_number>
   ```

   A mesma porta passada para o servidor é colocada aqui.

   Neste momento, o programa irá mostrar informações do socket local e remoto.

   Em seguida, irá mostrar a cadeia de caracteres enviadas pelo servidor de forma invertida.

   Em paralelo, o terminal do cliente está escutando se o usuário digita alguma coisa. Se o mesmo digitar “exit”, o programa é encerrado, mesmo que haja comandos pendentes a serem executados. Quando um cliente termina sua execução, seja de forma interrompida ou voluntária (terminou a tarefa), o programa e a conexão entre aquele processo no servidor são encerradas. O processo pai-servidor no entanto, continua escutando futuras conexões.

5. Abrir o arquivo “log.txt” para ter acesso ao histórico de conexões, desconexões e resultados dos comandos de cada cliente.

Referências

[1] W. Richard Stevens. 1990. *UNIX network programming*. Prentice-Hall, Inc., USA.