MC833 - 2s2017

Bruno Orsi Berton **RA 150573 - Turma B**
Fábio Takahashi Tanniguchi **RA 145980 - Turma A**

Detalhes da implementação

Para compilar o código, vá até a pasta raiz e execute o comando **make**.
Para iniciar o servidor, execute o comando **./servidor <número da porta>**, substituindo o número da porta que deseja que o servidor ouça por conexões de clientes.
Para iniciar o cliente, execute o comando **./cliente <IP do servidor> <número da porta>**, substituindo o IP que o servidor se encontra e o número da porta.

Após se conectar com um cliente no servidor, basta executar qualquer comando linux que o servidor era retornar a resposta para o cliente.

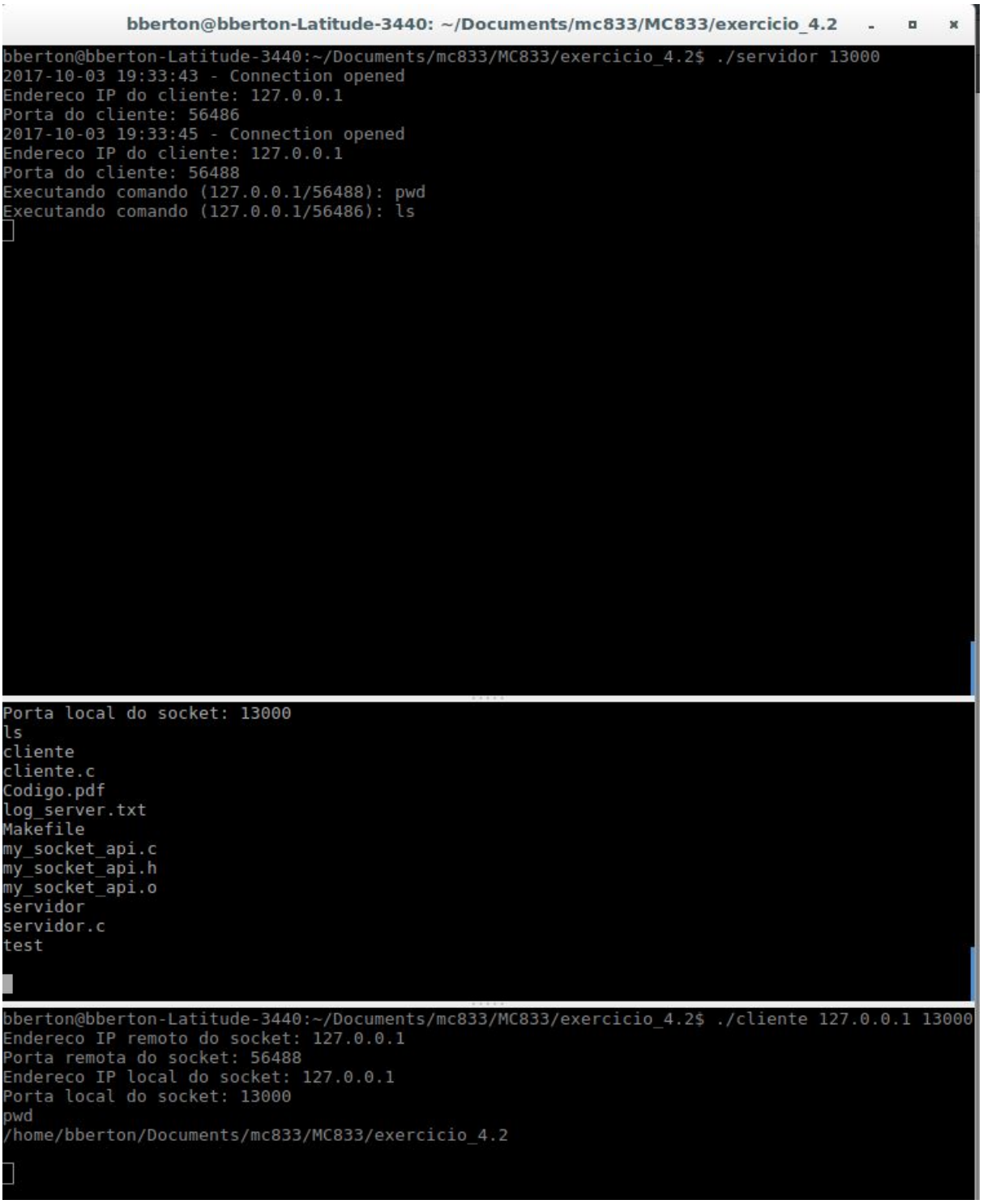
Utilizamos a função **popen** para executar os comandos do linux, pois permite ter um controle maior sobre o resultado do comando executado. No caso, o retorno do comando é armazenado em uma variável e retornado ao cliente.

Após o cliente ser desligado, o servidor armazena o log de tudo o que aconteceu com esse nome **log_server.txt**, então para ver os logs, basta se desconectar do cliente e abrir o arquivo.

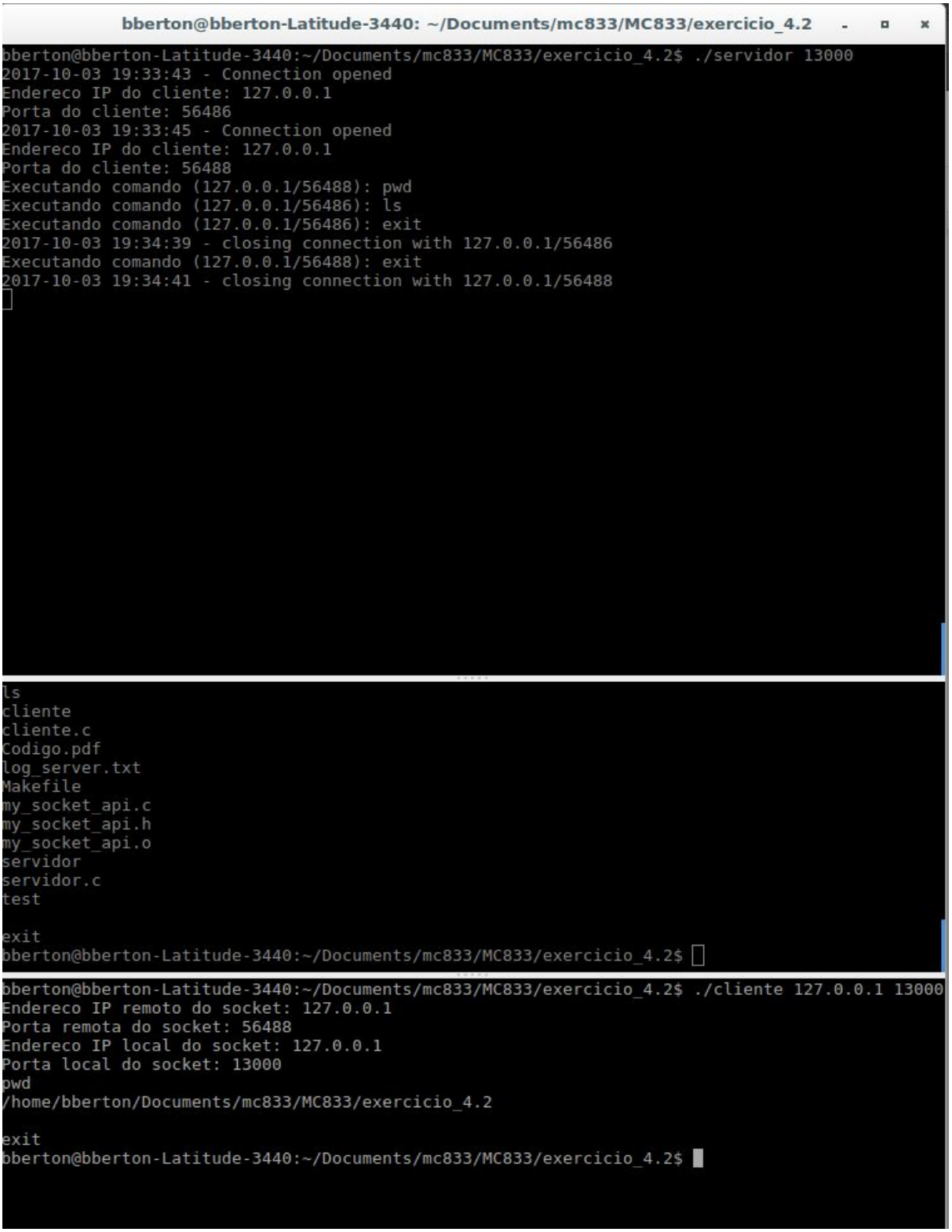
Caso o cliente queira se desconectar, basta executar o comando **exit**.

Testes

Dois clientes executando comandos no servidor



Os clientes se desconectando



e o log gerado pelo servidor

2017-10-03 19:31:07 - Connection opened
Endereco IP do cliente: 127.0.0.1
Porta do cliente: 56482
Executando comando (127.0.0.1/56482): ls

Retorno:
cliente
cliente.c
Codigo.pdf
log_server.txt
Makefile
my_socket_api.c
my_socket_api.h
my_socket_api.o
servidor
servidor.c
test

Executando comando (127.0.0.1/56482): pwd
Retorno:
/home/bberton/Documents/mc833/MC833/exercicio_4.2

Executando comando (127.0.0.1/56482): exit
2017-10-03 19:31:32 - closing connection with 127.0.0.1/56482
2017-10-03 19:31:43 - Connection opened
Endereco IP do cliente: 127.0.0.1
Porta do cliente: 56484
Executando comando (127.0.0.1/56484): ls
Retorno:
cliente
cliente.c
Codigo.pdf
log_server.txt
Makefile
my_socket_api.c
my_socket_api.h
my_socket_api.o
servidor
servidor.c
test

Executando comando (127.0.0.1/56484): exit
2017-10-03 19:31:49 - closing connection with 127.0.0.1/56484
2017-10-03 19:33:43 - Connection opened
Endereco IP do cliente: 127.0.0.1
Porta do cliente: 56486
Executando comando (127.0.0.1/56486): ls
Retorno:
cliente
cliente.c
Codigo.pdf

log_server.txt
Makefile
my_socket_api.c
my_socket_api.h
my_socket_api.o
servidor
servidor.c
test

Executando comando (127.0.0.1/56486): exit
2017-10-03 19:34:39 - closing connection with 127.0.0.1/56486
2017-10-03 19:33:45 - Connection opened
Endereco IP do cliente: 127.0.0.1
Porta do cliente: 56488
Executando comando (127.0.0.1/56488): pwd
Retorno:
/home/bberton/Documents/mc833/MC833/exercicio_4.2

Executando comando (127.0.0.1/56488): exit
2017-10-03 19:34:41 - closing connection with 127.0.0.1/56488

4) No trecho de código abaixo, que muito provavelmente estará presente nos códigos que você implementou, porque o servidor continua escutando e os clientes continuam com suas conexões estabelecidas mesmo após as chamadas dos Close? Explique o porque do uso de cada close e se algum deles está "sobrando" neste trecho de código.

```
1    for (;;) {
2       connfd = Accept (listenfd,...);
3
4       if ( (pid=Fork()) == 0) {
5          Close(listenfd);
6          doit(connfd); // Faz alguma operação no socket
7          Close(connfd);
8          exit(0);
9       }
10        Close(connfd);
11    }
```

O processo principal, ou o processo pai, fica em loop infinito esperando por conexões de clientes, como pode ser visto na linha 2.

Caso um cliente se conecte com o servidor, o processo se divide em 2, e nesse momento vão acontecer duas coisas simultaneamente:

Caso seja o processo pai, o pid é diferente de zero, então a condição do if é falso e o processo pai apenas executa o close em connfd na linha 10. Isso faz com que o processo

pai encerre a conexão com o cliente, e fica esperando por novos clientes. O processo volta para a linha 2.

Caso seja o processo filho, o pid é igual a zero, então a condição do if é verdadeiro. Nesse caso, o processo filho encerra o socket listenfd, então ele não espera por mais conexões de outros clientes TCPs. O processo filho faz o que tem que fazer e então encerra a sua conexão com o cliente.

Dessa maneira, com processos separados, o pai não atrapalha o socket na qual o filho está usando para se conectar, e o filho não atrapalhar o pai, ouvindo um socket desnecessário. Então não há nenhum close sobrando nesse trecho de código.

5) Com base ainda no trecho de código acima, é correto afirmar que os clientes nunca receberão FIN neste caso já que o servidor sempre ficará escutando (LISTEN)? Justifique.

Não, pois em algum momento um dos processos do servidor irá fechar seu socket com o cliente sem que outro processo do servidor tenha socket aberto com o cliente. Neste caso, o servidor enviará FIN para o cliente e aguardará o ACK do cliente e o FIN do cliente.

6) Comprove, utilizando ferramentas do sistema operacional, que os processos criados para manipular cada conexão individual do servidor aos clientes são filhos do processo original que foi executado.

Com o servidor executando, podemos executar o comando netstat -lntp para descobrir seu PID, como a imagem abaixo mostra

```
bberton@bberton-Latitude-3440:~/Documents/mc833/MC833/exercicio_4.2$ netstat -lntp
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:13000           0.0.0.0:*               LISTEN      11969/servidor
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN      -
```

Então temos que o PID do processo pai é 11969.

Agora executando o comando netstat -ntp, podemos descobrir o PID do processo filho do servidor que está conectado com algum cliente

```
bberton@bberton-Latitude-3440:~/Documents/mc833/MC833/exercicio_4.2$ netstat -ntp
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 177.220.84.153:59574    64.233.186.189:443      ESTABLISHED 2535/chrome
tcp        0      0 127.0.0.1:42218         127.0.0.1:13000         ESTABLISHED 11991/cliente
tcp        0      0 177.220.84.153:39202    172.217.30.46:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:54050    172.217.30.35:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:35706    64.233.186.188:5228     ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:39196    172.217.30.46:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:39206    172.217.30.46:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:36018    172.217.29.194:443      ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:39192    172.217.30.46:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:53902    172.217.30.35:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:59576    64.233.186.189:443      ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:56066    198.252.206.25:443      ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:39204    172.217.30.46:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:34654    92.123.223.235:443      ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:45322    169.55.69.157:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:54146    172.217.30.35:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:39198    172.217.30.46:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:35466    151.101.193.69:443      ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:38938    172.217.30.46:443       ESTABLISHED 2535/chrome
tcp        0      0 127.0.0.1:13000         127.0.0.1:42218         ESTABLISHED 11992/servidor
tcp        0      0 177.220.84.153:60622    23.73.64.143:443        ESTABLISHED 2535/chrome
```

Nesse caso, temos que o PID do filho é 11992.

Executando o comando pstree -sg 11992, podemos ver toda a árvore de processo do PID passado como parâmetro ou executar o comando ps -o ppid= -p 11992 que retorna o PID do processo pai

bberton@bberton-Latitude-3440:~/Documents/mc833/MC833/exercicio_4.2$ ps -o ppid= -p 11992

11969

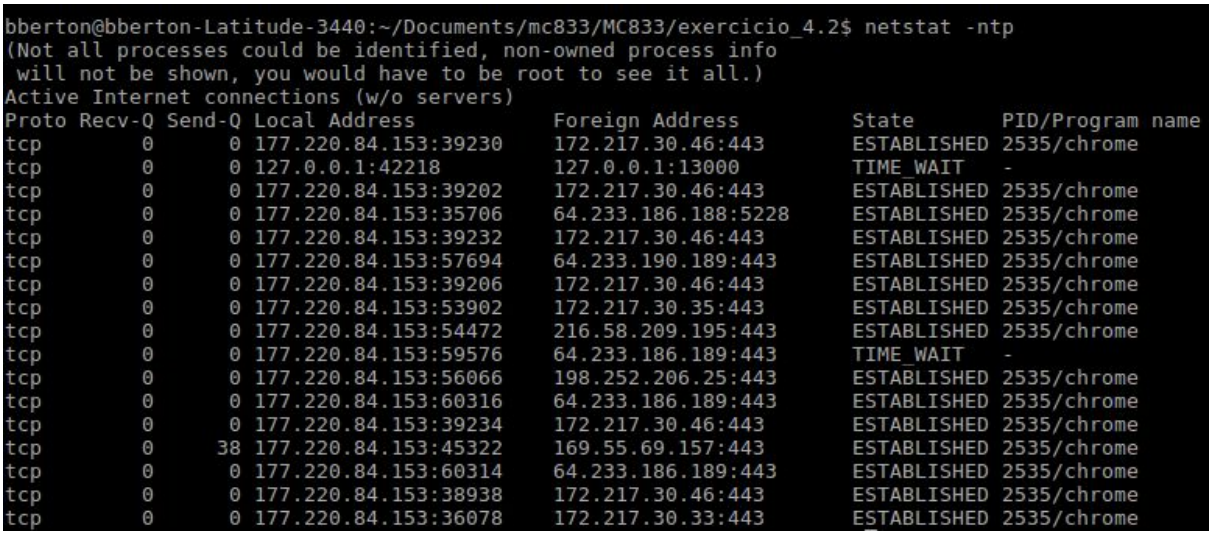
bberton@bberton-Latitude-3440:~/Documents/mc833/MC833/exercicio_4.2$ pstree -sg 11992
systemd(1)——lightdm(1050)——lightdm(1050)——cinnamon-sessio(2070)——cinnamon-launch(2070)——cinnamon(2070)——/usr/bin/termin(2070)——bash(3606)——servidor(11969)——servidor(11969)

Assim, temos certeza de que o servidor está criando processos filhos.

7) Utilizando ferramentas do sistema operacional, qual dos lados da conexão fica no estado TIME_WAIT após o encerramento da conexão? Isso condiz com a implementação que foi realizada? Justifique.

Como a gente pode ver executando o comando netstat -ntp, o cliente, conectado na porta 42218 com o servidor na porta 13000, fica em estado de TIME_WAIT.

Pois um dos motivos para que o cliente fique nesse estado é evitar que pacotes atrasados de uma conexão possam ser mal interpretados em conexões futuras, e para que a conexão possa ser encerrada com sucesso mesmo que algum ACK do encerramento da conexão se perca.



```
bberton@bberton-Latitude-3440:~/Documents/mc833/MC833/exercicio_4.2$ netstat -ntp
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 177.220.84.153:39230    172.217.30.46:443       ESTABLISHED 2535/chrome
tcp        0      0 127.0.0.1:42218         127.0.0.1:13000         TIME_WAIT   -
tcp        0      0 177.220.84.153:39202    172.217.30.46:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:35706    64.233.186.188:5228     ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:39232    172.217.30.46:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:57694    64.233.190.189:443      ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:39206    172.217.30.46:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:53902    172.217.30.35:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:54472    216.58.209.195:443      ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:59576    64.233.186.189:443      TIME_WAIT   -
tcp        0      0 177.220.84.153:56066    198.252.206.25:443      ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:60316    64.233.186.189:443      ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:39234    172.217.30.46:443       ESTABLISHED 2535/chrome
tcp        0     38 177.220.84.153:45322    169.55.69.157:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:60314    64.233.186.189:443      ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:38938    172.217.30.46:443       ESTABLISHED 2535/chrome
tcp        0      0 177.220.84.153:36078    172.217.30.33:443       ESTABLISHED 2535/chrome
```