

Atividade 3: Servidor TCP (HTTP) Concorrente

Alunos: Elias Santos Martins e Gabriel Sanders

Pereira Sobral

RA: 247057 e 247118

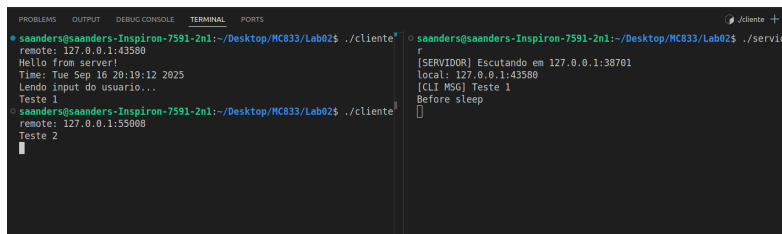
Instituto de Computação

Universidade Estadual de Campinas

Campinas, 19 de Agosto de 2025.

Sumário

1	Questão 1 — “Segurar” a primeira conexão	2
2	Questão 2 — Servidor TCP concorrente	2
3	Questão 3 — Cliente HTTP simples	2
4	Perguntas de Análise	3
4.1	1. Por que o <code>sleep()</code> no filho evidencia concorrência?	3
4.2	2. Propósito dos <code>Close</code> no trecho com <code>fork()</code>	3
4.3	3. Clientes “nunca” recebem <code>FIN</code> porque o servidor está em <code>LISTEN</code> ?	4
4.4	4. Comprovação de que manipuladores são proces- sos filhos	4
4.5	5. Quem entra em <code>TIME_WAIT</code> ao encerrar?	5



```
saanders@saanders-Inspiron-7591-2n1:~/Desktop/MC833/Lab02$ ./cliente
remote: 127.0.0.1:43588
Hello from server!
Time: Tue Sep 16 20:19:12 2025
Lendo input do usuario...
Teste 1
saanders@saanders-Inspiron-7591-2n1:~/Desktop/MC833/Lab02$ ./cliente
remote: 127.0.0.1:55088
Teste 2

saanders@saanders-Inspiron-7591-2n1:~/Desktop/MC833/Lab02$ ./servidor
r
[SERVIDOR] Escutando em 127.0.0.1:38701
local: 127.0.0.1:43588
[CLI MSG] Teste 1
Before sleep
[]
```

Figura 1: Sleep() segurando conexão

1 Questão 1 — “Segurar” a primeira conexão

O servidor **não aceita** as duas conexões de forma concorrente pois não há a criação de processos filhos para lidar com as requisições, portanto o mesmo processo que aceitou e processou a primeira conexão irá fazer o mesmo com a segunda, e faz isso sequencialmente. Portanto não há concorrência, e é preciso esperar até que o sleep() termine para que a próxima conexão possa ser estabelecida.

2 Questão 2 — Servidor TCP concorrente

Para essa questão, nós implementamos um servidor de tal modo que toda conexão estabelecida leva a um novo fork e portanto a criação de um processo filho exclusivamente dedicado para essa conexão, tornando o servidor concorrente. Além disso, implementamos as funções envelopadoras requisitadas e as saídas especificadas, como consta na Figura 2.

3 Questão 3 — Cliente HTTP simples

Todos os requisitos foram implementados.

```
saanders@saanders-Inspiron-7591-2n1:~/Desktop/MC833/Lab02$ ./cliente
remote: 127.0.0.1:49328
Hello from server!
Time: Tue Sep 23 19:58:53 2025
Lendo input do usuario...
GET / HTTP/1.0
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 91
Connection: close

<html><head><title>MC833</title></head><body><h1>MC833 TCP Concorrente </h1><
/body></html>
saanders@saanders-Inspiron-7591-2n1:~/Desktop/MC833/Lab02$ ./cliente
remote: 127.0.0.1:55864
Hello from server!
Time: Tue Sep 23 19:59:10 2025
Lendo input do usuario...
outra mensagem
400 Bad Request
saanders@saanders-Inspiron-7591-2n1:~/Desktop/MC833/Lab02$

saanders@saanders-Inspiron-7591-2n1:~/Desktop/MC833/Lab02$ ./servidor
[SERVIDOR] Escutando em 127.0.0.1:41805
local: 127.0.0.1:49328
[CLI MSG] GET / HTTP/1.0
local: 127.0.0.1:55864
[CLI MSG] outra mensagem
[]
```

Figura 2: Servidor TCP Concorrente

4 Perguntas de Análise

4.1 1. Por que o `sleep()` no filho evidencia concorrência?

Nesse contexto, a inserção de um `sleep()` no filho evidencia concorrência uma vez que uma segunda conexão pode ser aberta e executada mesmo quando uma conexão anterior ainda não foi finalizada. O processo pai cria um processo filho, delega a conexão a ele, e logo em seguida retorna ao `accept()` para ser capaz de atender outras conexões em paralelo.

4.2 2. Propósito dos `Close` no trecho com `fork()`

Mesmo após os 'Close' o servidor segue escutando e o cliente permanece conectado uma vez que cada close apenas decrementa contador de referências, e não necessariamente fecha a conexão. A FYN é enviado somente quando contador de referências possui valor zero.

O primeiro `Close()` é necessário para que o processo filho feche o `listenfd` para não manter uma outra referência desnecessária ao socket de escuta do pai, que apenas recebeu por ser um `fork` idêntico. O segundo `Close()` serve para finalizar as interações com o cliente, fechando a conexão após o filho processar a requisição. Já o terceiro e último `Close()` é importante para que o processo pai feche a sua conexão, uma vez que agora

```
saanders@saanders-Inspiron-7591-2n1:~/Desktop/MC833/Lab02$ ./cliente 127.0.0.1 48791
91
local IP/port: 127.0.0.1:52138
remote IP/port: 127.0.0.1:48791
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 91
Connection: close
<html><head><title>MC833</title></head><body><h1>MC833 TCP Concorrente </h1></body>
</html>
saanders@saanders-Inspiron-7591-2n1:~/Desktop/MC833/Lab02$

prlimit64(0, RLIMIT_STACK, NULL, (rlim_cur=8192*1024, rlim_max=RLIM64_INF
INTTY)) = 0
munmap(0x7f7ab6cc0800, 97600) = 0
socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 3
bind(3, {sa_family=AF_INET, sin_port=htons(0), sin_addr=inet_addr("127.0.
0.1"), 16) = 0
getsockname(3, {sa_family=AF_INET, sin_port=htons(48791), sin_addr=inet_a
ddr("127.0.0.1"), 16}) = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x80, 0x1), ...}
, AT_EMPTY_PATH) = 0
getrandom("\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f", 8, GRND_NONBLOCK) = 8
brk(NULL) = 0x58b252d46900
brk(0x58b252d46900) = 0x58b252d46900
write(1, "[SERVIDOR] Escutando em 127.0.0.1:48791", 40)SERVIDOR] Escutando em
127.0.0.1:48791
l = 48
openat(AT_FDCWD, "server.info", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 4
newfstatat(4, "", {st_mode=S_IFREG|0664, st_size=0, ...}, AT_EMPTY_PATH)
= 0
write(4, "IP=127.0.0.1\nPORT=48791\n", 24) = 24
close(4) = 0
listen(3, 10) = 0
accept(3, NULL, NULL) = 4
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIG
CHLD, child_tidptr=0x7f7ab6c3a10) = 9325
local: 127.0.0.1:52138
[CLI MSG] GET / HTTP/1.0
Host: teste
close(4) = 0
... SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=9325, si_uid=10
00, si_status=0, si_utime=0, si_stime=0} ...
accept(3, NULL, NULL)
```

Figura 3: Servidor rodando com *strace*

ela já está sendo tratada pelo processo filho. Assim ele evita manter uma referência que não usa.

Portanto, percebe-se que todas os três Closes são necessários, e nenhum deles está 'sobrando'.

4.3 3. Clientes “nunca” recebem FIN porque o servidor está em LISTEN?

Não, uma vez que o FIN é enviado no momento em que o socket de conexão é fechado (quando o filho executa `close(connfd)`).

4.4 4. Comprovação de que manipuladores são processos filhos

Utilizamos a comando 'strace' para rodar o servidor e então obtivemos a imagem da Figura 3, que demonstra o clone, a criação do processo filho com o pid 9325, o processamento feito por ele e seu fim após o `close()`

4.5 5. Quem entra em TIME_WAIT ao encerrar?

O processo filho do servidor entra em TIME_WAIT quando a conexão é encerrada. Isso condiz com a implementação, já que é o processo filho no servidor que chama o close, portanto ele envia um