

## Exercício 4 de MC833 — Programação em Redes de Computadores

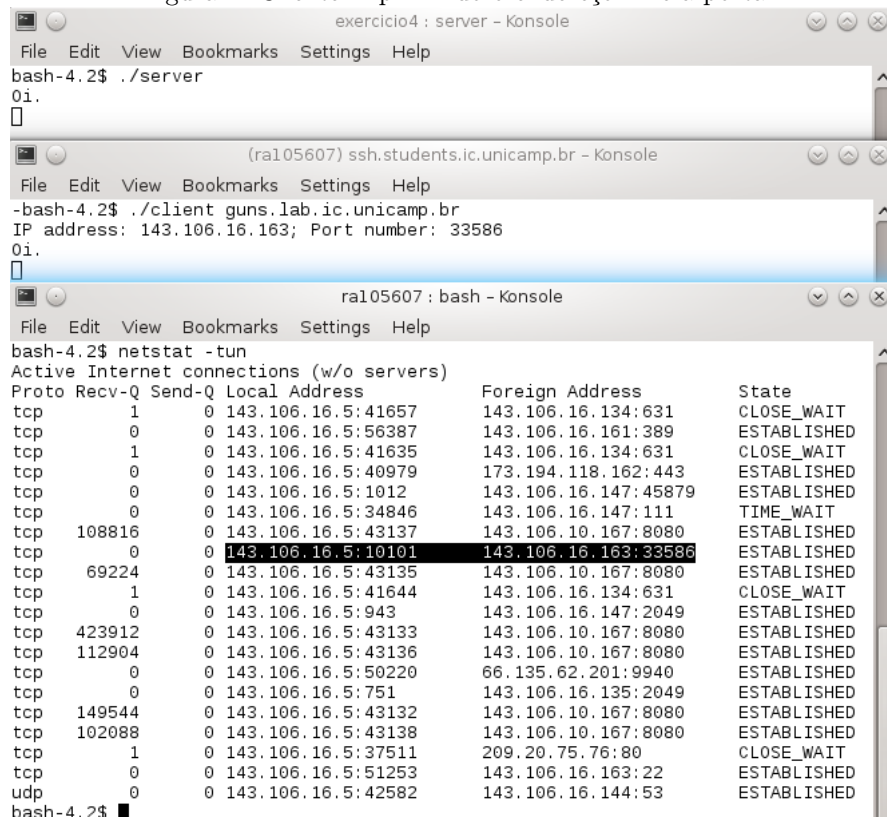
Raul Rabelo Carvalho, 105607, turma A

26 de Março de 2014

# 1

A figura 1 mostra o cliente imprimindo na saída padrão o endereço IP e a porta associados à conexão pelo sistema operacional. O servidor está sendo executado no *host* `guns.lab.ic.unicamp.br:10101` (143.106.16.5:10101) e o cliente, no *host* em 143.106.16.163. O endereço IP e a porta impressas pelo cliente são confirmadas pela linha destacada na saída do comando `netstat -tun`.

Figura 1: Cliente imprimindo o endereço IP e a porta.



# 2

A figura 2 mostra tanto o cliente como o servidor imprimindo na saída padrão o endereço IP e a porta associados ao cliente pelo sistema operacional local. O servidor está sendo executado no *host* `guns.lab.ic.unicamp.br` (no endereço IP 143.106.16.5), mas agora na porta 10102, e o cliente, no *host* em 143.106.16.163. O endereço IP e a porta impressas por ambos programas são confirmadas pela linha destacada na saída do comando `netstat -tun`.

Figura 2: Servidor imprimindo o endereço IP e a porta.

```

exercicio4 : server - Konsole
File Edit View Bookmarks Settings Help
bash-4.2$ ./server
IP address: 143.106.16.163; Port number: 55946
Olá.
^

(ra105607) ssh.students.ic.unicamp.br - Konsole
File Edit View Bookmarks Settings Help
-bash-4.2$ ./client guns.lab.ic.unicamp.br
IP address: 143.106.16.163; Port number: 55946
Olá.
^

ra105607 : bash - Konsole
File Edit View Bookmarks Settings Help
bash-4.2$ netstat -tun
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 143.106.16.5:41657      143.106.16.134:631     CLOSE_WAIT
tcp        0      0 143.106.16.5:56387      143.106.16.161:389    ESTABLISHED
tcp        0      0 143.106.16.5:40361      143.106.16.147:111    TIME_WAIT
tcp        1      0 143.106.16.5:41635      143.106.16.134:631     CLOSE_WAIT
tcp        0      0 143.106.16.5:1012       143.106.16.147:45879   ESTABLISHED
tcp    108816      0 143.106.16.5:43137      143.106.10.167:8080    ESTABLISHED
tcp    69224      0 143.106.16.5:43135      143.106.10.167:8080    ESTABLISHED
tcp        1      0 143.106.16.5:41644      143.106.16.134:631     CLOSE_WAIT
tcp        0      0 143.106.16.5:943        143.106.16.147:2049    ESTABLISHED
tcp    423912      0 143.106.16.5:43133      143.106.10.167:8080    ESTABLISHED
tcp        0      0 143.106.16.5:10102      143.106.16.163:55946   ESTABLISHED
tcp    112904      0 143.106.16.5:43136      143.106.10.167:8080    ESTABLISHED
tcp        0      0 143.106.16.5:50220      66.135.62.201:9940     ESTABLISHED
tcp        0      0 143.106.16.5:751        143.106.16.135:2049    ESTABLISHED
tcp    149544      0 143.106.16.5:43132      143.106.10.167:8080    ESTABLISHED
tcp    102088      0 143.106.16.5:43138      143.106.10.167:8080    ESTABLISHED
tcp        1      0 143.106.16.5:37511      209.20.75.76:80        CLOSE_WAIT
tcp        0      0 143.106.16.5:51253      143.106.16.163:22      ESTABLISHED
tcp        0      0 143.106.16.5:10101      143.106.16.163:33453   TIME_WAIT
udp        0      0 143.106.16.5:42582      143.106.16.144:53      ESTABLISHED
bash-4.2$

```

### 3

O código pertinente ao servidor de múltiplas conexões segue abaixo, comentado.

```

while(1) {
    /* aceita a conexao de um cliente por um socket novo new_s */
    if ((new_s = accept(s, (struct sockaddr *)&sin, &len)) < 0) {
        perror("simplex-talk: accept");
        exit(EXIT_FAILURE);
    }
    /* faz o fork do processo */
    child_pid = fork();
    if (child_pid < 0) {
        perror("simplex-talk: fork");
        exit(EXIT_FAILURE);
    }
    /* caso o processo seja o filho */
    if (child_pid == 0) {
        /* fecha o socket que esta' esperando por novos clientes */
        close(s);
        /* coleta as informacoes do socket e imprime na stdout */
        so_len = sizeof(so);
        if (getpeername(new_s, (struct sockaddr *)&so, &so_len) < 0) {
            perror("simplex-talk: getpeername");
            close(s);
            exit(EXIT_FAILURE);
        }
        inet_ntop(AF_INET, &(so.sin_addr), so_addr, INET_ADDRSTRLEN);
    }
}

```

```

        printf("IP address: %s; Port number: %d\n", so_addr,
               ntohs(so.sin_port));
        /* entra em loop imprimindo as mensagens do cliente */
        while (len = recv(new_s, buf, sizeof(buf), 0)) {
            fputs(buf, stdout);
        }
    }
    /* caso o processo seja o pai, fecha o novo socket */
    close(new_s);
}

```

---

## 4

Segue abaixo o trecho de código a ser explicado.

```

for (;;) {
    connfd = Accept (listenfd,...);

    if ( (pid=Fork()) == 0) {
        Close(listenfd);
        doit(connfd); // Faz alguma operacao no socket
        Close(connfd);
        exit(0);
    }
    Close(connfd);
}

```

---

Após a execução do `fork`, existem dois processos diferentes com `pids` diferentes. O processo-pai tem esta variável definida com o número do processo-filho, portanto, ele não executa o bloco do `if`, vindo somente a fechar o *socket* `connfd` enquanto continua a aguardar novas conexões no *socket* `listenfd`. Já no caso do processo-filho, sua variável `pid` é zero, então, o bloco do `if` é executado. Dentro do bloco, o *socket* `listenfd` é fechado imediatamente, mas isso não afeta o funcionamento geral do sistema, já que o processo-pai continua a escutar neste *socket*. O processo-filho, deste modo, ficou encarregado da conexão com o cliente pelo *socket* `connfd`, o qual só é encerrado depois de executar as operações desejadas pelo cliente.

## 5

Podemos comprovar que os processos extras que o servidor cria são seus filhos usando a *tree view* do programa `htop`, o que é uma versão com mais recursos da ferramenta `top`. Na figura 3, vê-se que existem dois processos-filhos de um processo `server` sendo executando no *shell* `bash` em um *konsole* (terminal virtual do KDE).

Figura 3: htop mostrando os processos do servidor.

The image shows four terminal windows stacked vertically. The top window is titled 'exercicio4 : server - Konsole' and shows the execution of './server', which prints IP addresses and port numbers. The second window is 'exercicio4 : client - Konsole' showing './client localhost'. The third window is 'exercicio4 : client - Konsole <2>' also showing './client localhost'. The bottom window is 'exercicio4 : htop - Konsole' displaying the htop process monitor. It shows system statistics at the top (Tasks: 77, 170 thr; 3 running; Load average: 0.19 0.16 0.14; Uptime: 02:11:30) and a table of running processes below.

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
1	root	20	0	25852	3348	1824	S	0.0	0.1	0:01.50	/sbin/init \EFI\arch\vmlinux-linu
5726	rrcarvalh	20	0	568M	38792	23288	S	1.3	0.6	0:03.98	kdeinit4: konsole [kdeinit]
5759	rrcarvalh	20	0	15744	2312	1676	S	0.0	0.0	0:00.00	/bin/bash
5819	rrcarvalh	20	0	6284	672	576	S	0.0	0.0	0:00.00	└─ ./client localhost
5749	rrcarvalh	20	0	15744	2316	1680	S	0.0	0.0	0:00.00	/bin/bash
5815	rrcarvalh	20	0	6284	672	576	S	0.0	0.0	0:00.00	└─ ./client localhost
5739	rrcarvalh	20	0	15744	2316	1680	S	0.0	0.0	0:00.00	/bin/bash
5805	rrcarvalh	20	0	14112	2136	1360	R	1.3	0.0	0:01.26	└─ htop
5728	rrcarvalh	20	0	15864	2352	1712	S	0.0	0.0	0:00.01	/bin/bash
5814	rrcarvalh	20	0	4048	336	268	S	0.0	0.0	0:00.00	└─ ./server
5820	rrcarvalh	20	0	4052	72	0	S	0.0	0.0	0:00.00	└─ ./server
5816	rrcarvalh	20	0	4052	72	0	S	0.0	0.0	0:00.00	└─ ./server
5727	rrcarvalh	20	0	568M	38792	23288	S	0.0	0.6	0:00.00	kdeinit4: konsole [kdeinit]

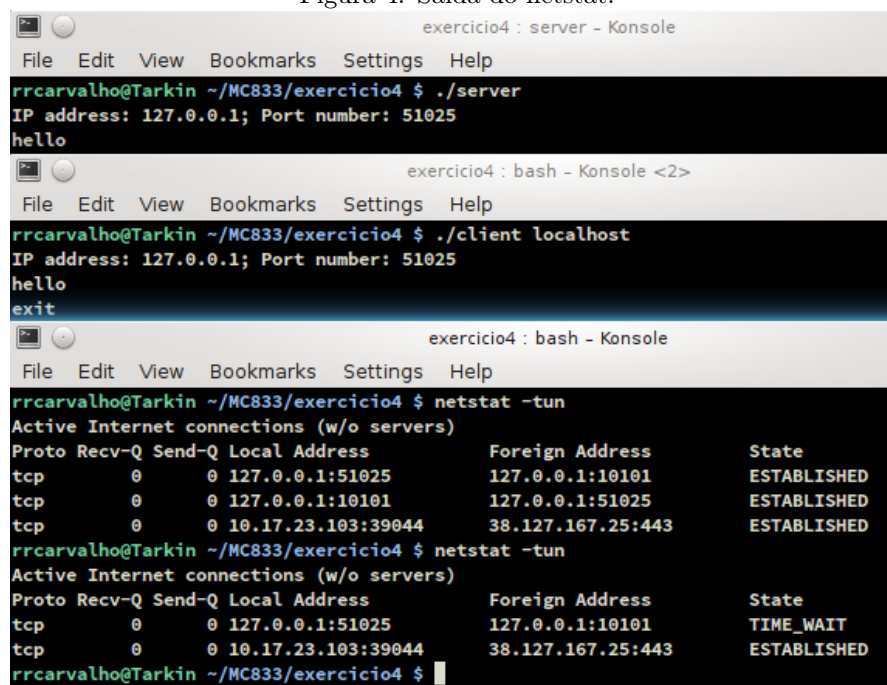
## 6

Para verificar quais dos lados da conexão entra no estado `TIME_WAIT`, foi feita uma modificação no cliente para ele encerre a conexão após a entrada da palavra `exit`. Assim, o cliente será o lado que iniciará o encerramento da conexão. O comando para verificação foi `netstat -tun`.

Como visto na figura 4, o lado que inicia o encerramento da conexão que entra em `TIME_WAIT`. Isso é condizente com a máquina de estados do protocolo TCP<sup>1</sup>. O estado `TIME_WAIT` existe garantir o encerramento adequado da conexão TCP. O cliente fica neste estado por duas vezes o `MSL` (*maximum segment lifetime*), caso o `ACK` que ele enviou em resposta ao `FIN` do servidor (note que o cliente iniciou o encerramento enviando um `FIN` ao servidor que foi por este respondido com um `ACK` seguido de um `FIN`) não seja entregue a este. O cliente precisa manter o estado da conexão para caso o servidor re-envie seu `FIN`.

<sup>1</sup>Stevens, W. Richard. UNIX Network Programming. 2nd Edition, pp. 40-41

Figura 4: Saída do netstat.



The figure consists of three vertically stacked screenshots of a terminal window titled "exercicio4 : server - Konsole".

The first screenshot shows the server program being executed: `rrcarvalho@Tarkin ~/MC833/exercicio4 $ ./server`. It displays the IP address `127.0.0.1` and port `51025`, and receives the message `hello`.

The second screenshot shows the client program being executed: `rrcarvalho@Tarkin ~/MC833/exercicio4 $ ./client localhost`. It also displays the IP address `127.0.0.1` and port `51025`, sends the message `hello`, and then exits with `exit`.

The third screenshot shows the output of the `netstat -tun` command. It displays active Internet connections (w/o servers) with columns for Protocol, Recv-Q, Send-Q, Local Address, Foreign Address, and State.

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	127.0.0.1:51025	127.0.0.1:10101	ESTABLISHED
tcp	0	0	127.0.0.1:10101	127.0.0.1:51025	ESTABLISHED
tcp	0	0	10.17.23.103:39044	38.127.167.25:443	ESTABLISHED

The second run of `netstat -tun` shows the state of the connections after the client has exited:

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	127.0.0.1:51025	127.0.0.1:10101	TIME_WAIT
tcp	0	0	10.17.23.103:39044	38.127.167.25:443	ESTABLISHED