



ENGENHARIA DA COMPUTAÇÃO
REDES DE COMPUTADORES A

ATIVIDADE 3

EQUIPE:

Agostinho Sanches de Araújo -----	RA: 16507915
Evandro Douglas Capovilla Junior -----	RA: 16023905
Lucas Tenani Felix Martins -----	RA: 16105744
Pedro Andrade Caccavaro -----	RA: 16124679

26/03/2019

SUMÁRIO

INTRODUÇÃO.....	03
OBJETIVO.....	03
DESCRIÇÃO.....	04
ANALISE PROGRAMA SERVIDOR TCP COM <i>SLEEP</i>	04
ANALISE PROGRAMA SERVIDOR TCP CONCORRENTE.....	05
IMPLEMENTAÇÃO PROGRAMA SERVIDOR TCP CONCORRENTE.....	06
1. CADASTRAR MENSAGEM.....	06
2. LER MENSAGENS.....	07
3. APAGAR MENSAGENS.....	07
4. SAIR DA APLICAÇÃO.....	07
OBSERVAÇÕES.....	07
RESULTADO.....	09
CONCLUSÃO.....	14

INTRODUÇÃO

Transmission Control Protocol, conhecido como TCP, é um dos protocolos mais utilizados da camada de transporte do modelo TCP/IP. O TCP é orientado a conexão, ou seja, é necessário estabelecer uma conexão prévia entre as máquinas antes que ocorra o envio de pacotes, denominado como *Three Way Handshake*. Dessa maneira é garantido que todos os pacotes serão entregues do emissor ao receptor, através de inúmeras confirmações e comunicações entre as máquinas e, no caso de perda de pacote, é enviado um pedido de retransmissão.

OBJETIVO

A atividade tem como principal objetivo demonstrar a interação de um servidor pai criando vários servidores filhos para atender diversos clientes ao mesmo tempo. Para efetuar a comunicação entre cliente e servidor, foi utilizado o protocolo TCP. Com relação aos servidores filhos, foi utilizado a função *fork()* para efetuar a criação.

DESCRIÇÃO

A atividade é dividida em três partes, sendo elas:

1. Executar o programa servidor TCP com *sleep*, levando em consideração pelo menos cinco instâncias do programa cliente TCP sendo executados simultaneamente. Após a análise dos dados, aumentar o número de instâncias para vinte e analisar os resultados.
2. Executar o programa servidor TCP concorrente (*fork*), levando em consideração pelo menos cinco instâncias do programa cliente TCP sendo executados simultaneamente. Após a análise dos dados, aumentar o número de instâncias para vinte e analisar os resultados. Explicar o motivo de ocorrer processos zumbis.
3. Implementar um programa servidor TCP concorrente se comunicando com o cliente TCP.

ANALISANDO PROGRAMA SERVIDOR TCP COM *SLEEP*

Primeiramente, é inicializado o programa servidor TCP com *sleep*. Após todas as informações serem configuradas para o servidor, o mesmo fica em uma espera bloqueante até que o programa cliente aceite uma conexão com o programa servidor. Com o êxito da conexão, o programa cliente envia uma mensagem “Requisição” para o programa servidor e espera a resposta do mesmo. O programa servidor por sua vez recebe a mensagem do programa cliente, mostra na tela para o usuário o endereço IP e porta que o programa cliente utilizou para entrar em contato com o programa servidor. Após isso, o programa servidor manda um *sleep* de dez segundos e depois envia uma resposta para o programa cliente.

É importante ressaltar que mesmo executando cinco programas clientes de forma simultânea, o *accept* só aceitará uma conexão por vez, e essa conexão ficará ativa até um programa cliente terminar de se comunicar com o programa servidor. Com a utilização do *sleep* fica ainda mais implícito o tempo que cada programa cliente fica em espera para conseguirem se comunicar com o programa servidor.

ANALISANDO PROGRAMA SERVIDOR TCP CONCORRENTE

Primeiramente, é inicializado o programa servidor TCP concorrente (*fork*). Após todas as informações serem configuradas para o servidor, o mesmo fica em uma espera bloqueante até que o programa cliente aceite uma conexão com o programa servidor. Com o êxito da conexão, o programa cliente envia uma mensagem “Requisição” para o programa servidor e espera a resposta do mesmo. O programa servidor, após sair de sua espera bloqueante, cria um processo filho para atender o programa cliente. Com base nessa criação, encerra a conexão com o servidor pai e o servidor filho recebe a mensagem do programa cliente. Como resposta, o servidor filho mostra na tela para o usuário o seu id (recebido na criação do *fork*), o endereço IP e porta que o programa cliente utilizou para entrar em contato com o programa servidor filho. Após isso, o programa servidor manda um *sleep* de dez segundos e depois envia uma resposta para o programa cliente.

É importante ressaltar que diferentemente do programa servidor TCP com *sleep*, com concorrência o servidor consegue atender tanto os cinco quanto os vinte programas clientes de forma simultânea, porque quando um programa servidor filho é criado, ele é conectado com o cliente e os demais clientes não ficam mais bloqueados pelo *accept*.

Outro fator importante foi a percepção dos programas servidores filhos zumbis através do comando *ps-axf*, onde é possível identificá-los pela *flag* de estado “Z” e também através da mensagem exibida na frente do processo *<defunct>*. O processo zumbi ocorre quando um processo filho terminar de executar o que era necessário e o processo pai não o finaliza. Sem a finalização, o processo filho fica em um “estado” zumbi, onde ele permanece como processo porém não executa nada.

Para a resolução desse problema foi utilizado uma estrutura de dados chamada *sigaction*. Dentro dessa estrutura, foram configurados o *handler* (definir como *SIG_DFL* - fazendo com que o processo filho que receber esse sinal seja encerrado quando terminar a execução) e a *flag* (definir como *SA_NOCLDWAIT* - fazendo com que espere o processo filho terminar a sua execução antes de finalizá-lo). Em seguida, é chamada a função *sigaction(SIGCHLD,&sigchild_action,NULL)*. Essa função define que todos os processos filhos serão interpretados de acordo com o *handler* e a *flag* passados como *default* na estrutura.

IMPLEMENTANDO PROGRAMA SERVIDOR TCP CONCORRENTE

O programa servidor é inicializado passando como parâmetro a porta que será utilizada. Assim, um *socket* é criado e o servidor é setado com as devidas informações.

Após setar as informações necessárias para o funcionamento do servidor, é feita uma conexão (*bind*) entre o *socket* criado e o próprio servidor. Com o êxito do *bind*, uma chamada *listen* é criada promovendo uma lista de mensagens. Em sequência, a função *accept* é chamada para entrar em contato com o programa cliente, fazendo com que o programa servidor fique em uma espera bloqueante até estabelecer uma conexão.

Com o programa servidor em espera bloqueante, o programa cliente é executado passando como parâmetro o endereço IP (endereço local da máquina - 127.0.0.1) ou o nome apropriado que indica o endereço IP local (localhost) e a mesma porta que foi utilizada para o programa servidor. Após a execução do programa cliente, um *socket* é criado e o servidor cliente é configurado do mesmo formato que o programa servidor. Estabelecido o *socket*, a função *connect* é chamada para efetuar a conexão entre o programa cliente e o programa servidor fazendo com que o programa servidor não fique mais em espera bloqueante. O programa servidor, após sair de sua espera bloqueante, cria um processo filho para atender o programa cliente. Com base nessa criação, encerra a conexão com o servidor pai e o servidor filho recebe a mensagem do programa cliente.

Um menu de opções é mostrado para o usuário, após a conexão ser estabelecida, sendo elas:

1. CADASTRAR MENSAGEM

O programa cliente pedirá o nome do usuário (até 19 caracteres) e a mensagem que deseja enviar (até 79 caracteres). Esses dados são colocados dentro de uma *struct* chamada mensagem e enviadas para o programa servidor por meio da função *send*. Após o envio, o programa servidor deve guardar as informações (usuário e a mensagem).

Todos os programas cliente fica em espera bloqueante até o programa servidor enviar uma resposta sobre o estado da requisição solicitada (positivo ou negativo) liberando outro cliente para executar a requisição.

O servidor deve permitir alocar no máximo 10 mensagens.



2. LER MENSAGENS

O programa cliente fica em espera bloqueante até o programa servidor enviar todas as mensagens que foram cadastradas. É mostrado como resultado no programa cliente o número de mensagens que foram lidas e as informações recebidas pelo programa servidor.

3. APAGAR MENSAGENS

O programa cliente pedirá o nome do usuário. O nome será mandado para o programa servidor e todas as mensagens que tenham sido enviadas por esse usuário serão apagadas.

O programa cliente fica em espera bloqueante até o programa servidor enviar uma resposta sobre o estado da requisição solicitada (positivo ou negativo).

4. SAIR DA APLICAÇÃO

O programa cliente é encerrado e o programa servidor finaliza a conexão com o mesmo, mas permanece executando esperando por outra conexão a menos que receba um sinal de fechamento pelo terminal linux, encerrando o semáforo implementado.

OBSERVAÇÕES

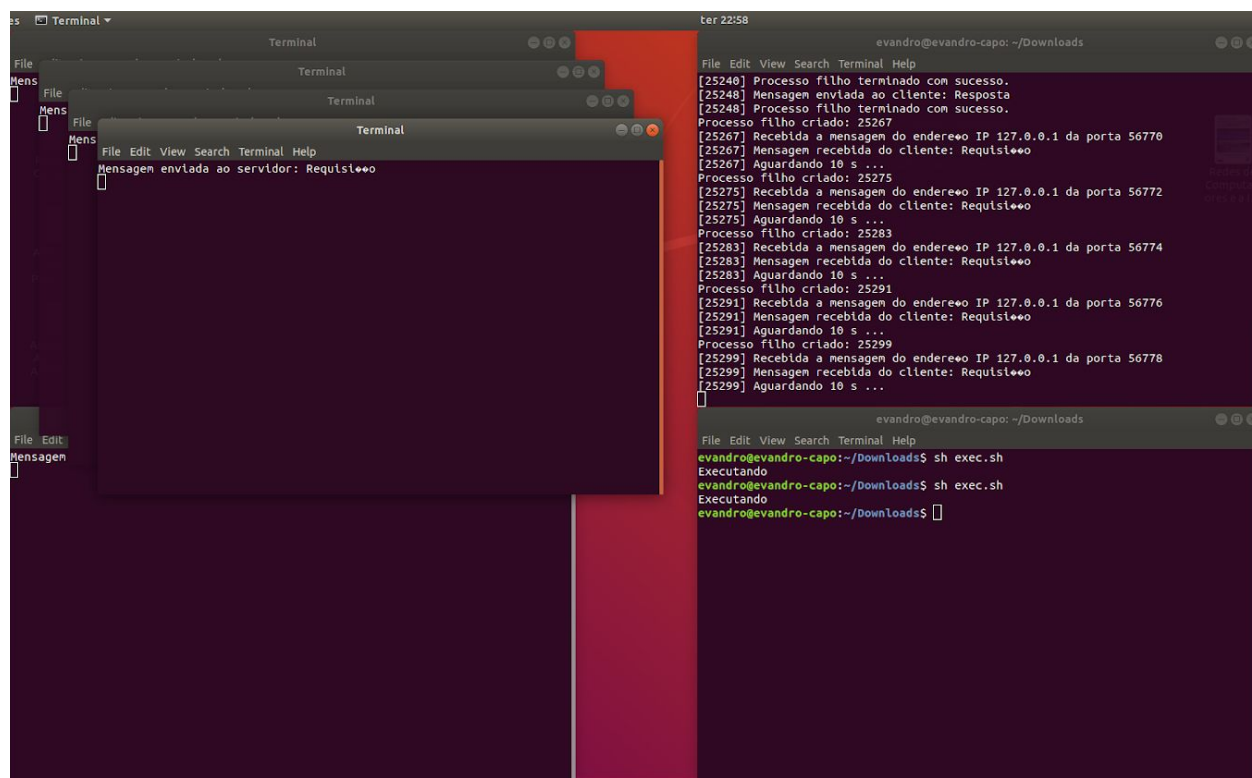
- Ao utilizar a estrutura de menus com o *switch*, observamos que houve conflito com o semáforo, fazendo-o sempre ficar liberado. Para resolver este problema, utilizamos as condições *if* e *else* para a construção dos menus.
- Para a realização dos teste com os semáforos, utilizamos a função *sleep* para que observássemos o funcionamento da mesma.
- Com a criação dos programas servidores filhos, era necessário a criação de uma memória compartilhada, para que todos os filhos pudessem obter os dados atualizados na hora de consultar qualquer opção no servidor. Essa memória compartilhada também estaria dentro do conjunto de semáforos para evitar o problema de *race condition* caso dois programas clientes quisessem fazer qualquer tipo de alteração no servidor.



- Na utilização da memória compartilhada observamos que estávamos utilizando a função “*sizeof*” de maneira equivocada, em vez de enviar o tamanho da estrutura da mensagem, estávamos enviando o tamanho em *bytes* da mensagem apenas (8 *bytes*).

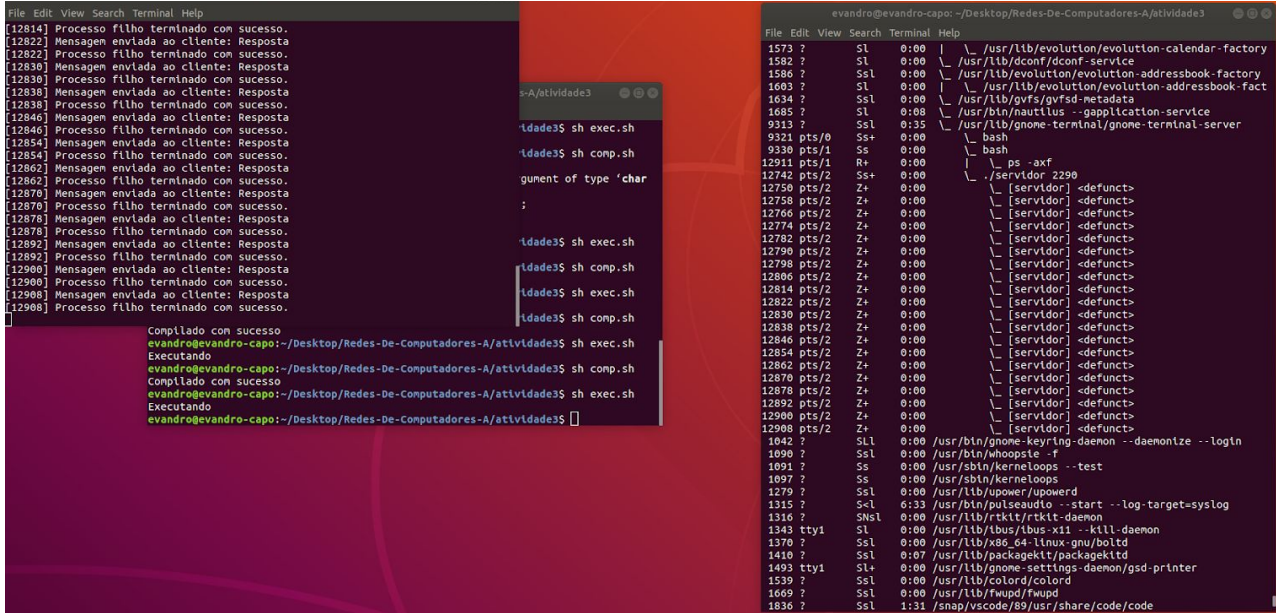
RESULTADOS

Parte 2



```
File Edit View Search Terminal Help
[25240] Processo filho terminado com sucesso.
[25248] Mensagem enviada ao cliente: Resposta
[25248] Processo filho terminado com sucesso.
Processo filho criado: 25267
[25267] Recebida a mensagem do endereço IP 127.0.0.1 da porta 56770
[25267] Mensagem recebida do cliente: Requisição
[25267] Aguardando 10 s ...
Processo filho criado: 25275
[25275] Recebida a mensagem do endereço IP 127.0.0.1 da porta 56772
[25275] Mensagem recebida do cliente: Requisição
[25275] Aguardando 10 s ...
Processo filho criado: 25283
[25283] Recebida a mensagem do endereço IP 127.0.0.1 da porta 56774
[25283] Mensagem recebida do cliente: Requisição
[25283] Aguardando 10 s ...
Processo filho criado: 25291
[25291] Recebida a mensagem do endereço IP 127.0.0.1 da porta 56776
[25291] Mensagem recebida do cliente: Requisição
[25291] Aguardando 10 s ...
Processo filho criado: 25299
[25299] Recebida a mensagem do endereço IP 127.0.0.1 da porta 56778
[25299] Mensagem recebida do cliente: Requisição
[25299] Aguardando 10 s ...
evandro@evandro-capo: ~/Downloads
File Edit View Search Terminal Help
evandro@evandro-capo:~/Downloads$ sh exec.sh
Executando
evandro@evandro-capo:~/Downloads$ sh exec.sh
Executando
evandro@evandro-capo:~/Downloads$
```

Na imagem acima, observamos o funcionamento do programa servidor TCP com concorrência se comunicando com os programas clientes (5 clientes) antes de implementar a solução para os programas zumbis.

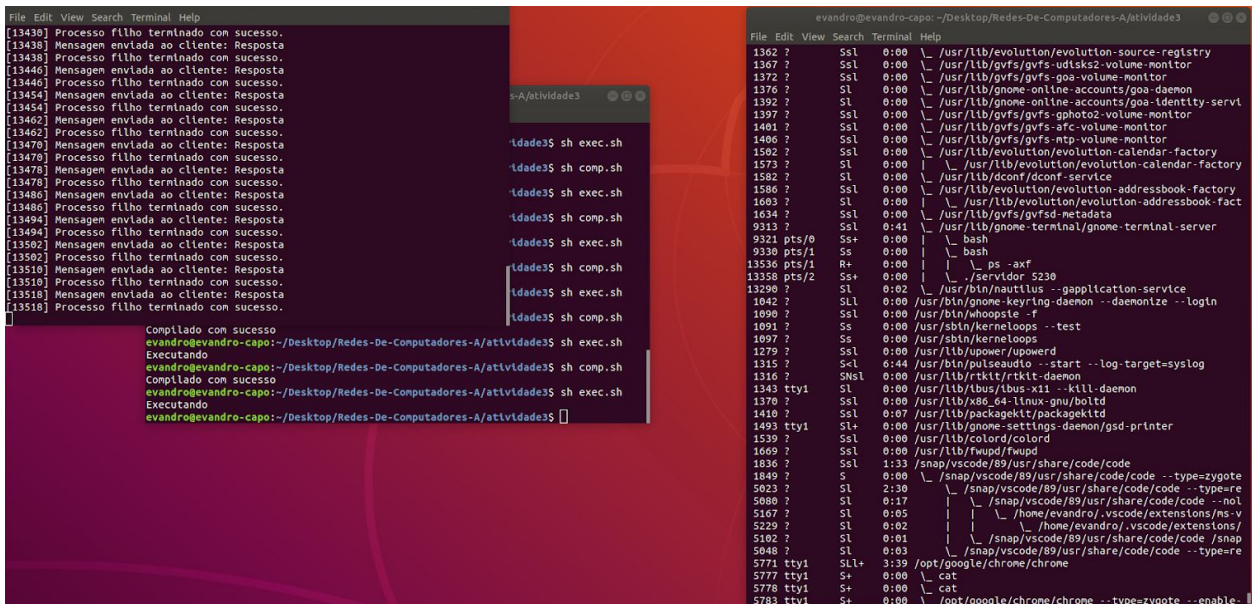


```

File Edit View Search Terminal Help
[12814] Processo filho terminado com sucesso.
[12822] Mensagem enviada ao cliente: Resposta
[12822] Processo filho terminado com sucesso.
[12830] Mensagem enviada ao cliente: Resposta
[12830] Processo filho terminado com sucesso.
[12836] Mensagem enviada ao cliente: Resposta
[12836] Processo filho terminado com sucesso.
[12846] Mensagem enviada ao cliente: Resposta
[12846] Processo filho terminado com sucesso.
[12854] Mensagem enviada ao cliente: Resposta
[12854] Processo filho terminado com sucesso.
[12862] Mensagem enviada ao cliente: Resposta
[12862] Processo filho terminado com sucesso.
[12870] Mensagem enviada ao cliente: Resposta
[12870] Processo filho terminado com sucesso.
[12878] Mensagem enviada ao cliente: Resposta
[12878] Processo filho terminado com sucesso.
[12892] Mensagem enviada ao cliente: Resposta
[12892] Processo filho terminado com sucesso.
[12900] Mensagem enviada ao cliente: Resposta
[12900] Processo filho terminado com sucesso.
[12908] Mensagem enviada ao cliente: Resposta
[12908] Processo filho terminado com sucesso.
[12908] Processo filho terminado com sucesso.
evandro@evandro-capo: ~/Desktop/Redes-De-Computadores-A/atividade3
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$ sh exec.sh
Compilado com sucesso
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$ sh comp.sh
Compilado com sucesso
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$ sh exec.sh
Compilado com sucesso
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$

```

Na imagem acima, observamos o funcionamento do programa servidor TCP com concorrência se comunicando com os programas clientes (20 clientes) antes de implementar a solução para os programas zumbis.



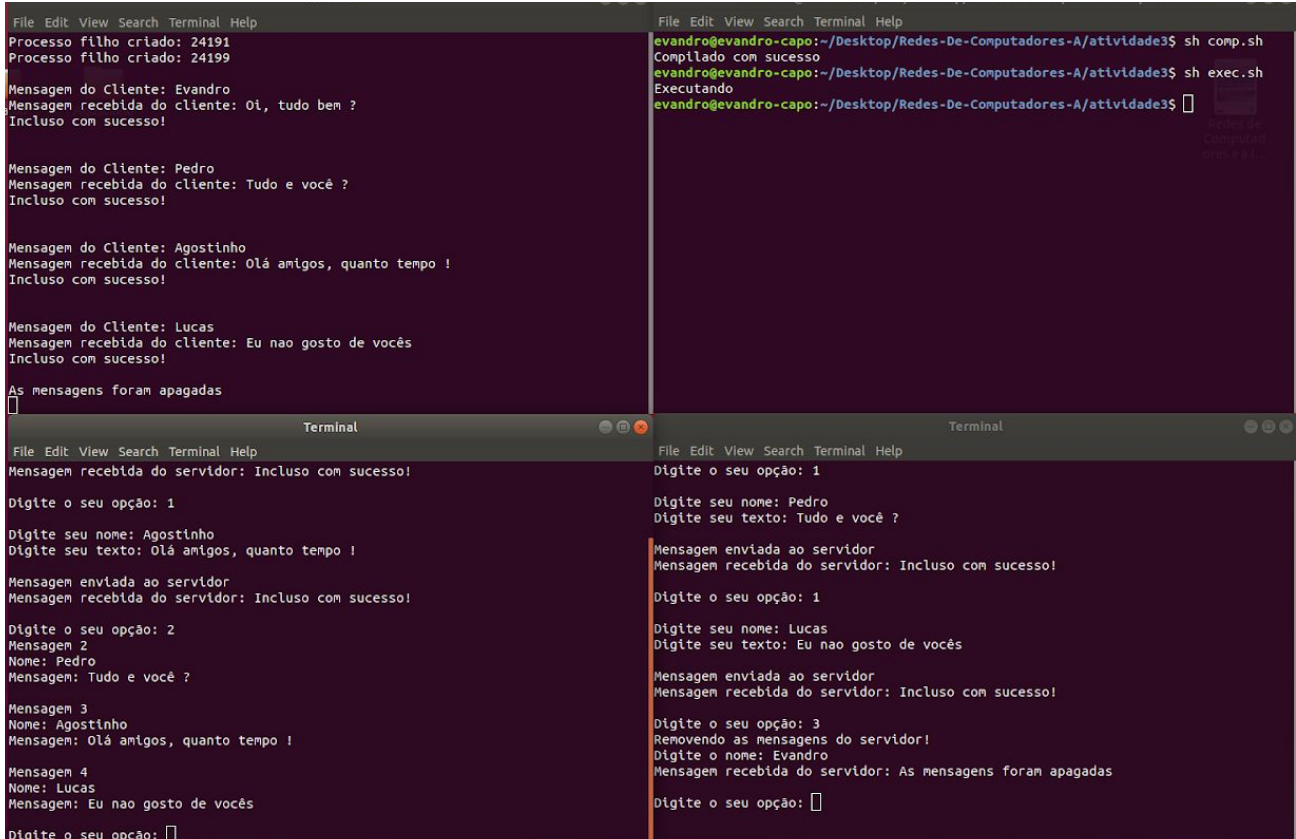
```

File Edit View Search Terminal Help
[13430] Processo filho terminado com sucesso.
[13438] Mensagem enviada ao cliente: Resposta
[13438] Processo filho terminado com sucesso.
[13446] Mensagem enviada ao cliente: Resposta
[13446] Processo filho terminado com sucesso.
[13454] Mensagem enviada ao cliente: Resposta
[13454] Processo filho terminado com sucesso.
[13462] Mensagem enviada ao cliente: Resposta
[13462] Processo filho terminado com sucesso.
[13470] Mensagem enviada ao cliente: Resposta
[13470] Processo filho terminado com sucesso.
[13478] Mensagem enviada ao cliente: Resposta
[13478] Processo filho terminado com sucesso.
[13486] Mensagem enviada ao cliente: Resposta
[13486] Processo filho terminado com sucesso.
[13494] Mensagem enviada ao cliente: Resposta
[13494] Processo filho terminado com sucesso.
[13502] Mensagem enviada ao cliente: Resposta
[13502] Processo filho terminado com sucesso.
[13510] Mensagem enviada ao cliente: Resposta
[13510] Processo filho terminado com sucesso.
[13518] Mensagem enviada ao cliente: Resposta
[13518] Processo filho terminado com sucesso.
[13518] Processo filho terminado com sucesso.
evandro@evandro-capo: ~/Desktop/Redes-De-Computadores-A/atividade3
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$ sh exec.sh
Compilado com sucesso
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$ sh comp.sh
Compilado com sucesso
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$ sh exec.sh
Compilado com sucesso
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$

```

Na imagem acima, observamos o funcionamento do programa servidor TCP com concorrência se comunicando com os programas clientes (20 clientes) depois de implementar a solução para os programas zumbis.

Parte 3



The image shows two terminal windows side-by-side. The left window is the server terminal, and the right window is the client terminal. The server terminal shows the following output:

```
File Edit View Search Terminal Help
Processo filho criado: 24191
Processo filho criado: 24199
Mensagem do Cliente: Evandro
Mensagem recebida do cliente: Oi, tudo bem ?
Incluso com sucesso!

Mensagem do Cliente: Pedro
Mensagem recebida do cliente: Tudo e você ?
Incluso com sucesso!

Mensagem do Cliente: Agostinho
Mensagem recebida do cliente: Olá amigos, quanto tempo !
Incluso com sucesso!

Mensagem do Cliente: Lucas
Mensagem recebida do cliente: Eu nao gosto de vocês
Incluso com sucesso!

As mensagens foram apagadas

```

The right window is the client terminal, showing the following output:

```
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$ sh comp.sh
Compilado com sucesso
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$ sh exec.sh
Executando
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$

```

Below the main image, there are two smaller terminal windows. The left one shows the server terminal with the following output:

```
File Edit View Search Terminal Help
Mensagem recebida do servidor: Incluso com sucesso!

Digite o seu opção: 1

Digite seu nome: Agostinho
Digite seu texto: Olá amigos, quanto tempo !

Mensagem enviada ao servidor
Mensagem recebida do servidor: Incluso com sucesso!

Digite o seu opção: 2
Mensagem 2
Nome: Pedro
Mensagem: Tudo e você ?

Mensagem 3
Nome: Agostinho
Mensagem: Olá amigos, quanto tempo !

Mensagem 4
Nome: Lucas
Mensagem: Eu nao gosto de vocês

Digite o seu opção:

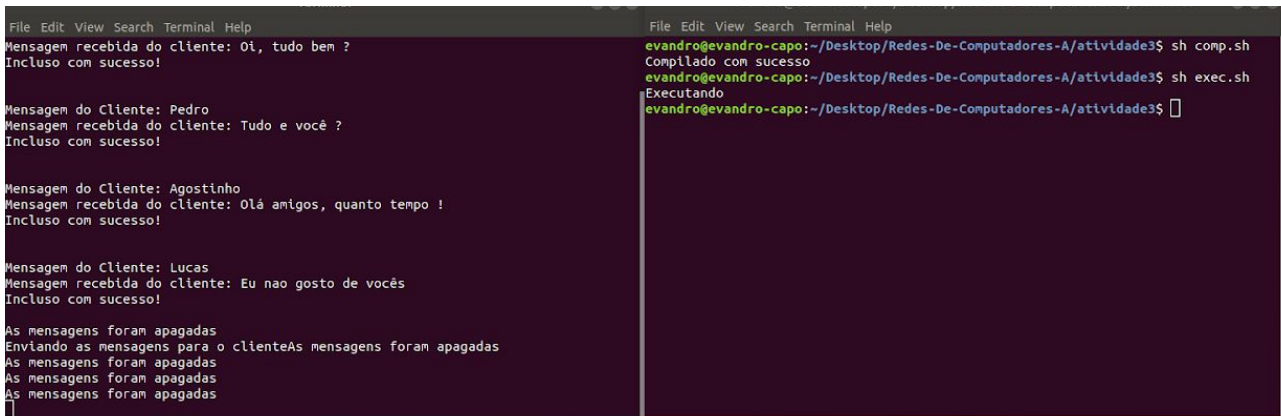
```

The right one shows the client terminal with the following output:

```
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$ sh comp.sh
Compilado com sucesso
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$ sh exec.sh
Executando
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$

```

A imagem acima ilustra o funcionamento do servidor com dois clientes concorrentes executando opções de salvar, mostrar e deletar mensagem.



The image shows two terminal windows side-by-side. The left window is the server terminal, and the right window is the client terminal. The server terminal shows the following output:

```
File Edit View Search Terminal Help
Mensagem recebida do cliente: Oi, tudo bem ?
Incluso com sucesso!

Mensagem do Cliente: Pedro
Mensagem recebida do cliente: Tudo e você ?
Incluso com sucesso!

Mensagem do Cliente: Agostinho
Mensagem recebida do cliente: Olá amigos, quanto tempo !
Incluso com sucesso!

Mensagem do Cliente: Lucas
Mensagem recebida do cliente: Eu nao gosto de vocês
Incluso com sucesso!

As mensagens foram apagadas
Enviando as mensagens para o cliente
As mensagens foram apagadas
As mensagens foram apagadas
As mensagens foram apagadas
As mensagens foram apagadas

```

The right window is the client terminal, showing the following output:

```
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$ sh comp.sh
Compilado com sucesso
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$ sh exec.sh
Executando
evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$

```

A imagem acima ilustra o funcionamento do servidor mesmo após atender os clientes do exemplo.


```

Terminal
evandro@evandro-capo: ~/Desktop/Redes-De-Computadores-A/atividade3
File Edit View Search Terminal Help
Mensagem recebida do cliente: Oi, tudo bem ?
Incluso com sucesso!

Mensagem do Cliente: Pedro
Mensagem recebida do cliente: Tudo e você ?
Incluso com sucesso!

Mensagem do Cliente: Agostinho
Mensagem recebida do cliente: Olá amigos, quanto tempo !
Incluso com sucesso!

Mensagem do Cliente: Lucas
Mensagem recebida do cliente: Eu nao gosto de vocês
Incluso com sucesso!

As mensagens foram apagadas
Enviando as mensagens para o clienteAs mensagens foram apagadas
As mensagens foram apagadas
As mensagens foram apagadas
As mensagens foram apagadas

key      msqid    owner    perms    used-bytes  messages
----- Shared Memory Segments -----
key      shmid    owner    perms    bytes       nattch     status
0x00000000 327680    evandro   600      524288      2         dest
0x00000000 4030465   evandro   600      524288      2         dest
0x00000000 294914    evandro   600      67108864    2         dest
0x00000000 6782979   evandro   600      524288      2         dest
0x00000000 557060    evandro   600      524288      2         dest
0x00000000 6815749   evandro   600      524288      2         dest
0x00000000 6750214   evandro   600      524288      2         dest
0x00000000 4128775   evandro   600      524288      2         dest
0x00000000 4227080   evandro   600      524288      2         dest
0x00000000 6553609   evandro   600      8294400     2         dest
0x00000000 6914058   evandro   600      524288      2         dest
0x00000000 6946827   evandro   666      1040        1         dest
0x00000000 5275661   evandro   600      524288      2         dest
0x00000000 5308430   evandro   600      16777216    2         dest

----- Semaphore Arrays -----
key      semid    owner    perms    nsems
0x00002034 1048576   evandro   666      1

evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$

```

Na imagem acima, utilizamos o comando `ipcs` para mostrar que o semáforo continua inicializado mesmo sem clientes para atender.

```

evandro@evandro-capo: ~/Desktop/Redes-De-Computadores-A/atividade3
File Edit View Search Terminal Help
----- Message Queues -----
key      msqid    owner    perms    used-bytes  messages

----- Shared Memory Segments -----
key      shmid    owner    perms    bytes       nattch     status
0x00000000 327680    evandro   600      524288      2         dest
0x00000000 4030465   evandro   600      524288      2         dest
0x00000000 294914    evandro   600      67108864    2         dest
0x00000000 6782979   evandro   600      524288      2         dest
0x00000000 557060    evandro   600      524288      2         dest
0x00000000 6815749   evandro   600      524288      2         dest
0x00000000 6750214   evandro   600      524288      2         dest
0x00000000 4128775   evandro   600      524288      2         dest
0x00000000 4227080   evandro   600      524288      2         dest
0x00000000 6553609   evandro   600      8294400     2         dest
0x00000000 6914058   evandro   600      524288      2         dest
0x00000000 6946827   evandro   666      1040        0         dest
0x00000000 5275661   evandro   600      524288      2         dest
0x00000000 5308430   evandro   600      16777216    2         dest

----- Semaphore Arrays -----
key      semid    owner    perms    nsems
0x00002034 1048576   evandro   666      1

evandro@evandro-capo:~/Desktop/Redes-De-Computadores-A/atividade3$

```

Na imagem acima, utilizamos o comando `ipcs` para mostrar que o semáforo é destruído após o fechamento do servidor.



CONCLUSÃO

Durante a realização deste trabalho entramos em contato com uma versão mais próxima da realidade de um sistema cliente/servidor, pois pudemos implementar vários clientes enviando ou requisitando mensagens de um único servidor, sem processos zumbis, ou seja, todos os clientes foram atendidos. Concluimos que a utilização de *forks* gera uma facilidade no sentido de controlar processos zumbis, porém tivemos que criar memória compartilhada, visto que processos não compartilham memória, e bloqueá-las com semáforos para poder controlar o acesso a fim de não ocorrer *race conditions*.