



ENGENHARIA DA COMPUTAÇÃO  
REDES DE COMPUTADORES A

## ATIVIDADE 4

### EQUIPE:

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

02/04/2019



## SUMÁRIO

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



## 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 *pthread\_create()* para efetuar a criação.

## IMPLEMENTANDO PROGRAMA SERVIDOR TCP COM *THREADS*

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 uma *thread* filha para atender às requisições do programa cliente. Com base nessa criação, o *socket* criado através da função *accept()* é enviado para a função da *thread* filha, fazendo com que o cliente comece a se comunicar somente com a *thread* filha enquanto a *thread* pai continua criando filhos para atender às demais requisições de futuros programas clientes.

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

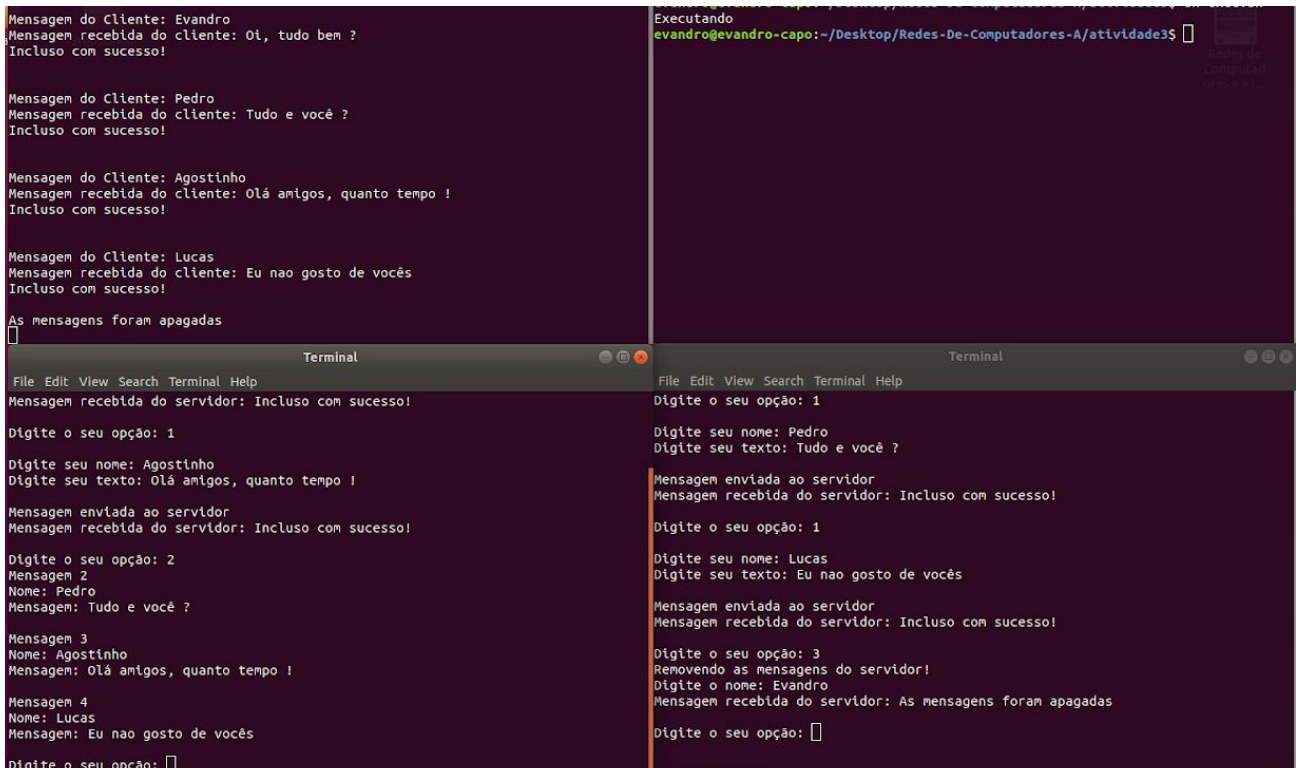
O programa cliente é encerrado e o programa servidor finaliza a conexão com o mesmo, mas permanece executando esperando por outra conexão a menos que receba um sinal de fechamento pelo terminal linux, encerrando o mutex implementado.

## OBSERVAÇÕES

- Ao tentar excluir o *mutex* observamos que devíamos deixar ele fechado, caso ao contrário, a função *pthread\_mutex\_destroy(&semaforo)* retornava um inteiro valendo 16, que corresponde ao erro *EBUSY*.
- Ao enviar parâmetros para os filhos, necessitamos fazer um *CAST* para *(void \*)*.
- Não foi mais necessário o uso de uma memória compartilhada, porque o processo de criação de *threads* permite que as *threads* filhas possam visualizar as variáveis globais sem precisar de uma memória compartilhada.
- A implementação de *mutex* foi para que não ocorresse as *race conditions*, ou seja, mais de uma *thread* filha acessando o mesmo espaço de memória

ao mesmo tempo, efetuando algum tipo de alteração ou recuperação dessa informação, ocasionando informações incertas.

## RESULTADOS



```

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

```

```

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

```

```

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: 1

Digite seu nome: Lucas
Digite seu texto: Eu nao gosto de vocês

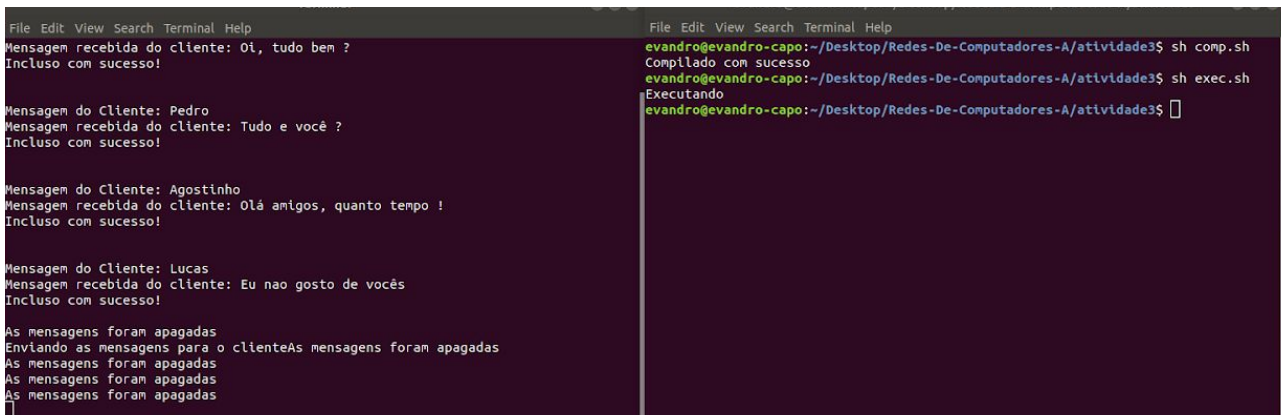
Mensagem enviada ao servidor
Mensagem recebida do servidor: Incluso com sucesso!

Digite o seu opção: 3
Renovendo as mensagens do servidor!
Digite o nome: Evandro
Mensagem recebida do servidor: As mensagens foram apagadas

Digite o seu opção:

```

A imagem acima ilustra o funcionamento do servidor com dois servidores filhos com *threads* executando opções de salvar, mostrar e deletar mensagem.



```

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

```

```

File Edit View Search Terminal Help
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.

## 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 *threads* zumbis, ou seja, todos os clientes foram atendidos. Concluimos que a utilização de *threads* facilita a manipulação das informações logo que não é necessário o uso de uma memória compartilhada para haver uma comunicação entre as *threads* filhas e a *thread* pai. Entretanto, a utilização de *threads* acaba deixando o programa muito mais suscetível a ocorrência de *races conditions* tendo que utilizar os *mutex* de uma forma mais estratégica, para evitar que duas *threads* acessem o mesmo espaço de memória e façam algum tipo de alteração ou recuperação do conteúdo da mesma.