Redes de computadores A

Campinas, 19 de março de 2019

2ª Atividade

NOME:	**RA:**
Ettore Biazon Baccan	16000465
Mateus Henrique Zorzi	16100661
Matheus Martins Pupo	16145559
Murilo Martos Mendonça	16063497
Victor Hugo do Nascimento	16100588

Introdução

Nessa atividade foi implementada uma aplicação cliente-servidor TCP. O TCP é do tipo orientado a conexão, ou seja, comunica-se através de conexões abertas e após o uso, as fecha.

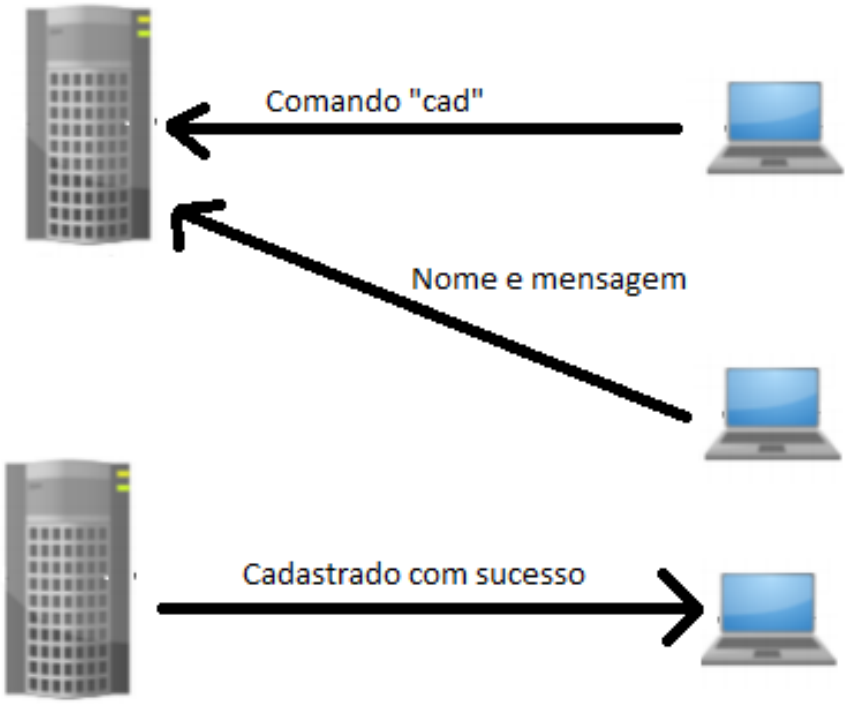
A grande vantagem das aplicações TCP é a facilidade do tratamento de problemas como perda de pacotes, envio fora de ordem e afins. O protocolo conta com recursos de confiabilidade, que é capaz de retransmitir dados que foram perdidos, verificar se os dados chegaram ao receptor sem divergências, e quando é impossível restaurar o dado perdido, informa o receptor que houve erro no pacote.

A desvantagem do uso desse protocolo é que para cada cliente, o servidor precisa de um socket separado, o que dificulta a implementação de aplicações onde há um grande número de clientes a se conectar. Nesse mesmo caso, para uma aplicação do tipo UDP, apenas um socket é usado para conectar todos esses clientes. Além disso, caso ocorra de um cliente se conectar ao servidor e cair sem se desconectar (close(socket)), esse socket não poderá ser utilizado por outro cliente, portanto, estará ocioso e ocupará recursos do servidor sem de fato utilizá-lo.

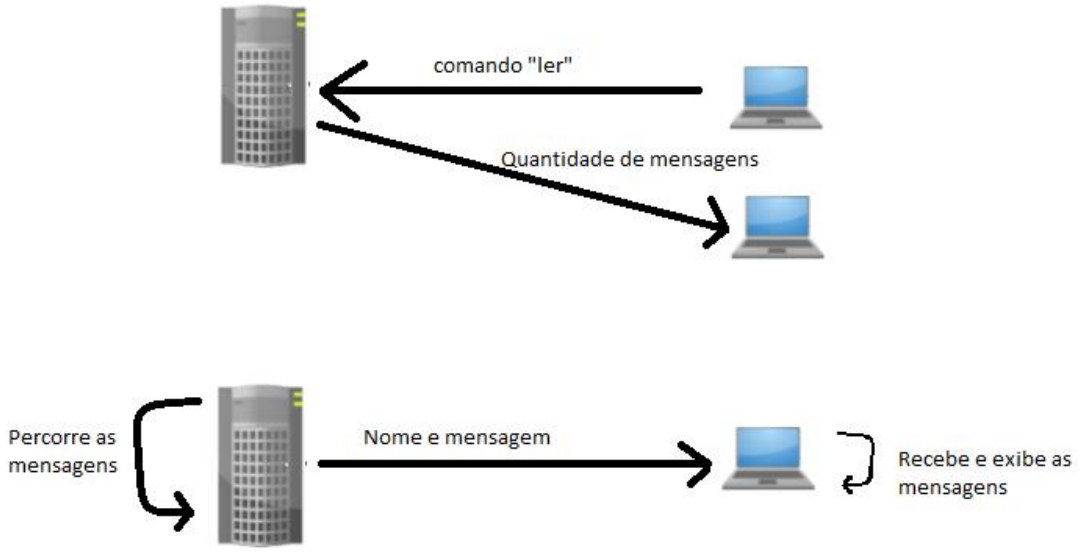
Esquematização

Legenda

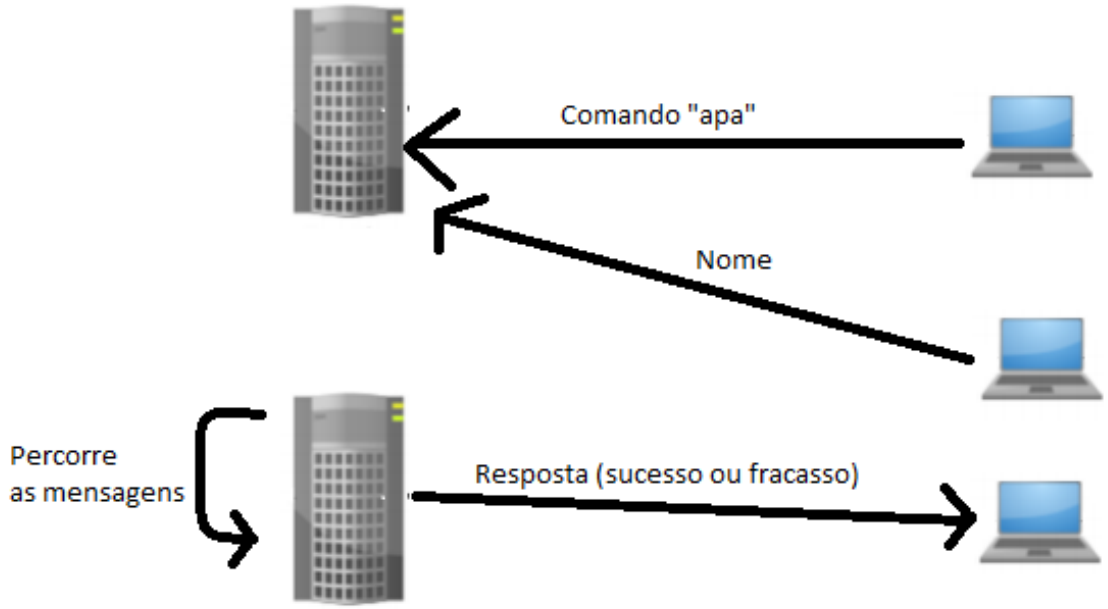
Cadastrar:



Ler:



Apagar:



Passo a passo

```
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0)
{
    perror("Socket()");
    exit(2);
}
```

Inicialmente, criamos o socket do servidor.

```
server.sin_family = AF_INET;
server.sin_port = htons(port);
server.sin_addr.s_addr = INADDR_ANY;

/*
 * Liga o servidor à porta definida anteriormente.
 */
if (bind(s, (struct sockaddr *)&server, sizeof(server)) < 0)
{
    perror("Bind()");
    exit(3);
}
```

Em seguida o conectamos (bind) a porta enviada através do ARGV.

```
if (listen(s, 1) != 0)
{
    perror("Listen()");
    exit(4);
}
printf("*** Servidor Iniciado! ***\n");
```

Agora, aguardamos por clientes através do comando “listen”.

```
namelen = sizeof(client);
if ((ns = accept(s, (struct sockaddr *)&client, &namelen)) == -1)
{
    perror("Accept()");
    exit(5);
}
```

Quando um cliente tenta se conectar, aceitamos e o servidor está pronto para receber mensagens.

```
if (recv(ns, recvbuf, sizeof(recvbuf), 0) == -1)
{
    perror("Recvbuf()");
    exit(6);
}
printf("\nMensagem recebida do cliente: %s\n", recvbuf);
```

Inicialmente, recebemos uma mensagem com a operação desejada pelo cliente:
“cad” – cadastrar mensagem.
“ler” – exibir as mensagens cadastradar.
“apa” – apagar alguma mensagem.

```c
if (strcmp(recvbuf, "cad") == 0)
{
    // Cadastrar mensagem
    if (indice == 10)
    {
        strcpy(sendbuf, "Numero maximo de mensagens atingido!");
        if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
        {
            perror("Send()");
            exit(7);
        }
    }
    else
    {
        if (recv(ns, mensagem_inteira, sizeof(mensagem_inteira), 0) == -1)
        {
            perror("Usuariobuf()");
            exit(6);
        }
        else
        {
            printf("\nMensagem inteira: %s\n", mensagem_inteira);

            strcpy(usuarios[indice], strtok(mensagem_inteira, "#"));
            strcpy(mensagens[indice], strtok('\0', "$$"));
            printf("\nMensagem inteira: %s\n", mensagem_inteira);
            printf("Usuario cadastrado: %s\n", usuarios[indice]);
            //strcpy(mensagens[indice], mensagembuf);
            printf("Mensagem cadastrada: %s\n", mensagens[indice]);
            printf("Indice: %d\n", indice);
            indice++;

            /* Envia uma mensagem ao cliente através do socket conectado */
            strcpy(sendbuf, "Mensagem cadastrada com sucesso!");
            if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
            {
                perror("Send()");
                exit(7);
            }
        }
    }
    printf("Mensagem enviada ao cliente: %s\n", sendbuf);
}
```

Caso o comando seja o cadastrar, primeiro verificamos (através da variável índice) se o número de mensagens cadastradas já atingiu o limite (10).

Em caso negativo, recebemos o nome e a mensagem do cliente através do ”recv”.

O modelo que escolhemos para a mensagem foi o seguinte: “NomeUsuario#Mensagem$$”, para que pudéssemos receber ambos em uma única mensagem e evitar erros.

Na sequência, salvamos a mensagem no vetor e incrementamos a variável índice, que controla a quantidade de mensagens cadastradas.

Para garantir o bom uso da aplicação, enviamos uma resposta ao cliente com o resultado da execução (sucesso ou fracasso no cadastro da mensagem).

```
if (strcmp(recvbuf, "ler") == 0)
{
    // Ler mensagem
    char qtd_msg[2];
    sprintf(qtd_msg, "%d", indice);
    strcpy(sendbuf, qtd_msg);
    printf("SENDBUF: %s\n", sendbuf);
    if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
    {
        perror("Send()");
        exit(7);
    }
    for (int i = 0; i < indice; i++)
    {
        memset(sendbuf, 0, sizeof(sendbuf));
        strcpy(mensagem_inteira, usuarios[i]);
        strcat(mensagem_inteira, "#");
        strcat(mensagem_inteira, mensagens[i]);
        strcat(mensagem_inteira, "$$");
        strcpy(sendbuf, mensagem_inteira);

        if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
        {
            perror("Send()");
            exit(7);
        }
        if (recv(ns, mensagem_inteira, sizeof(mensagem_inteira), 0) == -1)
        {
            perror("Usuariobuf()");
            exit(6);
        }

        //receber msg confirmação do cliente
    }
}
```

Caso o cliente opte por ler as mensagens cadastradas, enviamos para o cliente o número de mensagens cadastradas e percorremos o vetor de mensagens e de usuários os enviando em uma única mensagem usando a mesma convenção já citada acima (que será tratada no cliente, na exibição).

```c
if (strcmp(recvbuf, "apa") == 0)
{
    // Apaga mensagem
    char nome[20];
    if (recv(ns, recvbuf, sizeof(recvbuf), 0) == -1)
    {
        perror("Recvbuf()");
        exit(6);
    }
    strcpy(nome, recvbuf);
    strcpy(sendbuf, "Usuario nao encontrado!\n");
    for (int i = 0; i < indice; i++)
    {
        printf("Nome: %d\n", i);
        if (strcmp(nome, usuarios[i]) == 0)
        {
            printf("Nome %d localizado\n", i);
            for (int j = i; j < indice; j++)
            {
                printf("Usuario %d recebe usuario %d\n", j, j + 1);
                strcpy(usuarios[j], usuarios[j + 1]);
                strcpy(mensagens[j], mensagens[j + 1]);
            }
            indice--;
            printf("Indice: %d\nI: %d\n", indice, i);
            strcpy(sendbuf, "Usuario e mensagem apagado com sucesso!\n");
        }
    }
    if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
    {
        perror("Send()");
        exit(7);
    }
}
```

Se o cliente optar por apagar uma mensagem, recebemos o nome do usuário dono da mensagem, verificamos se existe mensagem cadastrada com esse nome, caso sim, excluímos a mensagem e informamos sucesso ao cliente. Caso não exista nenhuma mensagem cadastrada sob o nome enviado, avisamos ao cliente que o usuário não foi encontrado.

Na função “apa” recebida do cliente, nós apagamos uma mensagem de acordo com o usuário selecionado pelo cliente, começamos recebendo o nome de usuário no servidor [Figura 3.1] e em seguida passamos esse nome para uma variável dentro da função de apagar e iniciamos uma busca na matriz de usuários para ver se o usuário é válido [Figura 3.2].

```c
if (strcmp(recvbuf, "apa") == 0)
{
    // Apaga mensagem
    char nome[20];
    if (recv(ns, recvbuf, sizeof(recvbuf), 0) == -1)
    {
        perror("Recvbuf()");
        exit(6);
    }
    strcpy(nome, recvbuf);
    strcpy(sendbuf, "Usuario nao encontrado!\n");
    for (int i = 0; i < indice; i++)
    {
        printf("Nome: %d\n", i);
        if (strcmp(nome, usuarios[i]) == 0)
        {
            printf("Nome %d localizado\n", i);
            for (int j = i; j < indice; j++)
            {
                printf("Usuario %d recebe usuario %d\n", j, j + 1);
                strcpy(usuarios[j], usuarios[j + 1]);
                strcpy(mensagens[j], mensagens[j + 1]);
            }
            indice--;
            printf("Indice: %d\nI: %d\n", indice, i);
            strcpy(sendbuf, "Usuario e mensagem apagado com sucesso!\n");
        }
    }
    if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
    {
        perror("Send()");
        exit(7);
    }
}
```

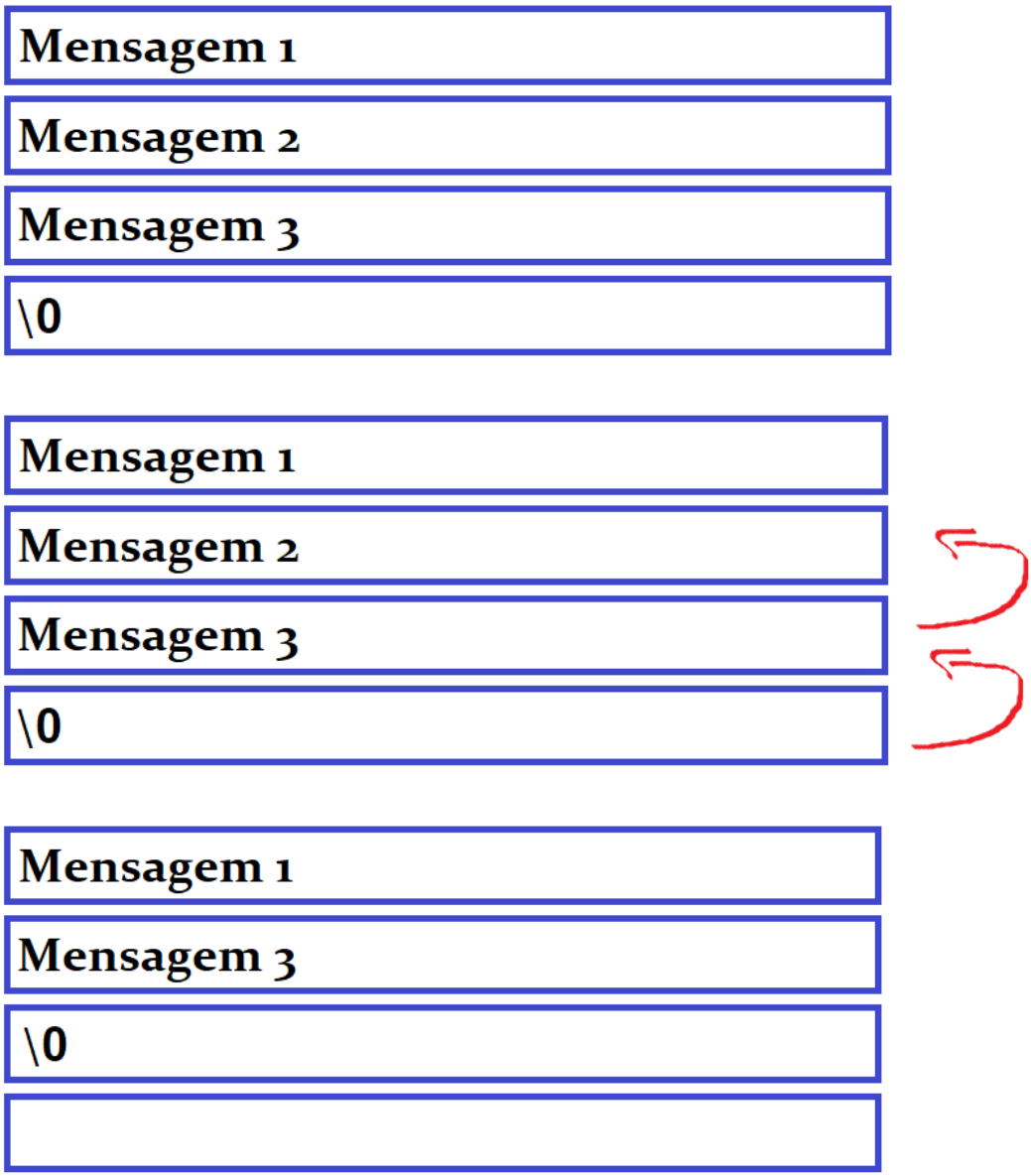
```c
if (strcmp(recvbuf, "apa") == 0)
{
    // Apaga mensagem
    char nome[20];
    if (recv(ns, recvbuf, sizeof(recvbuf), 0) == -1)
    {
        perror("Recvbuf()");
        exit(6);
    }
    strcpy(nome, recvbuf);
    strcpy(sendbuf, "Usuario nao encontrado!\n");
    for (int i = 0; i < indice; i++)
    {
        printf("Nome: %d\n", i);
        if (strcmp(nome, usuarios[i]) == 0)
        {
            printf("Nome %d localizado\n", i);
            for (int j = i; j < indice; j++)
            {
                printf("Usuario %d recebe usuario %d\n", j, j + 1);
                strcpy(usuarios[j], usuarios[j + 1]);
                strcpy(mensagens[j], mensagens[j + 1]);
            }
            indice--;
            printf("Indice: %d\nI: %d\n", indice, i);
            strcpy(sendbuf, "Usuario e mensagem apagado com sucesso!\n");
        }
    }
    if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
    {
        perror("Send()");
        exit(7);
    }
}
```

Figura 3.1 e 3.2

Caso o usuário seja encontrado, iniciamos um novo loop [Figura 3.3] que moverá as mensagens e usuários em posições posteriores ao usuário desejado, fazendo com que o usuário procurado *i* e sua mensagem sejam sobrepostos pelo usuário e mensagem *i + 1* e assim em diante, como é mostrado nas figuras 3.4 e 3.5, até que as duas matrizes sejam reescritas [Figura 3.6].

```
if (strcmp(recvbuf, "apa") == 0)
{
    // Apaga mensagem
    char nome[20];
    if (recv(ns, recvbuf, sizeof(recvbuf), 0) == -1)
    {
        perror("Recvbuf()");
        exit(6);
    }
    strcpy(nome, recvbuf);
    strcpy(sendbuf, "Usuario nao encontrado!\n");
    for (int i = 0; i < indice; i++)
    {
        printf("Nome: %d\n", i);
        if (strcmp(nome, usuarios[i]) == 0)
        {
            printf("Nome %d localizado\n", i);
            for (int j = i; j < indice; j++)
            {
                printf("Usuario %d recebe usuario %d\n", j, j + 1);
                strcpy(usuarios[j], usuarios[j + 1]);
                strcpy(mensagens[j], mensagens[j + 1]);
            }
            indice--;
            printf("Indice: %d\nI: %d\n", indice, i);
            strcpy(sendbuf, "Usuario e mensagem apagado com sucesso!\n");
        }
    }
    if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
    {
        perror("Send()");
        exit(7);
    }
}
```

Figura 3.3



Figuras 3.4, 3.5 e 3.6

Para finalizar enviamos uma mensagem ao cliente caso a mensagem do devido usuário seja apagada com sucesso ou uma mensagem de usuário inválido caso a entrada do cliente não seja apta para o servidor executar [Figura 3.7].

```
if (strcmp(recvbuf, "apa") == 0)
{
    // Apaga mensagem
    char nome[20];
    if (recv(ns, recvbuf, sizeof(recvbuf), 0) == -1)
    {
        perror("Recvbuf()");
        exit(6);
    }
    strcpy(nome, recvbuf);
    strcpy(sendbuf, "Usuario nao encontrado!\n");
    for (int i = 0; i < indice; i++)
    {
        printf("Nome: %d\n", i);
        if (strcmp(nome, usuarios[i]) == 0)
        {
            printf("Nome %d localizado\n", i);
            for (int j = i; j < indice; j++)
            {
                printf("Usuario %d recebe usuario %d\n", j, j + 1);
                strcpy(usuarios[j], usuarios[j + 1]);
                strcpy(mensagens[j], mensagens[j + 1]);
            }
            indice--;
            printf("Indice: %d\nI: %d\n", indice, i);
            strcpy(sendbuf, "Usuario e mensagem apagado com sucesso!\n");
        }
    }
    if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
    {
        perror("Send()");
        exit(7);
    }
}
```

Figura 3.7

```
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0)
{
    perror("Socket()");
    exit(3);
}
```

Figura 3.8 - Já no lado do cliente, criamos o seu socket.

```
server.sin_family = AF_INET;
server.sin_port = htons(port);
server.sin_addr.s_addr = INADDR_ANY;

/*
 * Liga o servidor à porta definida anteriormente.
 */
if (bind(s, (struct sockaddr *)&server, sizeof(server)) < 0)
{
    perror("Bind()");
    exit(3);
}
```

Figura 3.9 - e o conectamos à porta enviada pelo ARGV.

```c
if (connect(s, (struct sockaddr *)&server, sizeof(server)) < 0)
{
    perror("Connect()");
    exit(4);
}
```

Figura 3.10 - Em seguida nos conectamos ao servidor que estava aguardando conexões através do comando “listen”.

```c
} while(op != 4);
```

Figura 3.11 - O cliente roda dentro de um do-while, que só para de executar quando o comando “sair” é enviado.

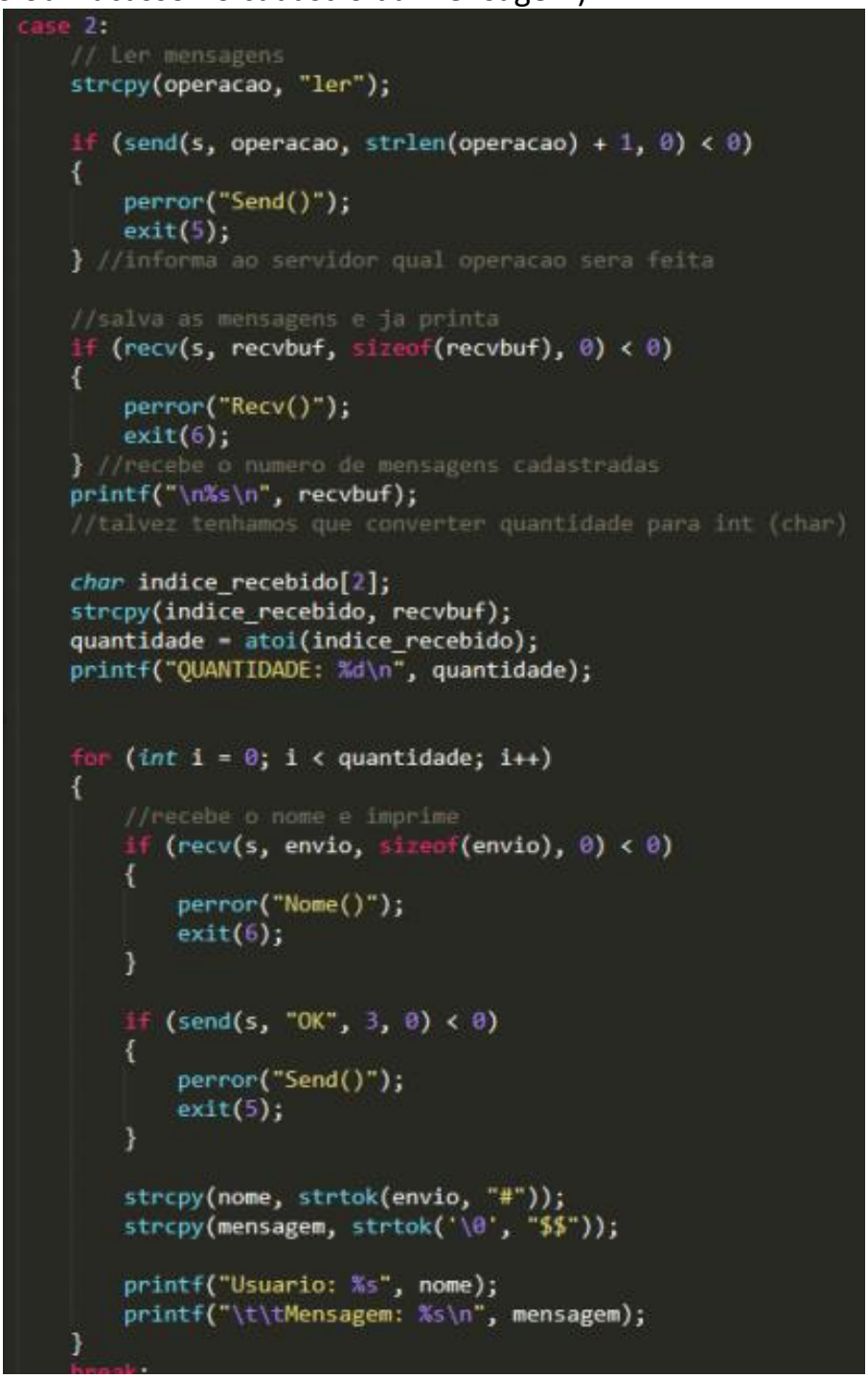
O usuário entra com o comando que deseja realizar.

```c
switch(op){

    case 1:
        // Cadastrar mensagem
        strcpy(operacao, "cad");
        if (send(s, operacao, strlen(operacao)+1, 0) < 0)
        {
            perror("Send()");
            exit(5);
        }//informa ao servidor qual operacao sera feita

        __fpurge(stdin);
        printf("\nDigite o nome: ");
        fgets(nome, sizeof(nome), stdin);
        strtok(nome, "\n"); //tira o \n inserido pelo fgets

        __fpurge(stdin);
        printf("Digite a mensagem: ");
        fgets(mensagem, sizeof(mensagem), stdin);
        strtok(mensagem, "\n"); //tira o \n inserido pelo fgets
        printf("\nNome recebido do cliente: %s\n", nome);
        printf("\nMensagem recebida do cliente: %s\n", mensagem);

        strcpy(envio, nome);
        strcat(envio, "#");
        strcat(envio, mensagem);
        strcat(envio, "$$");
        printf("\nEnvio: %s\n", envio);


        if (send(s, envio, strlen(envio)+1, 0) < 0)
        {
            perror("Send()");
            exit(5);
        }//envia o nome ao servidor

        if (recv(s, recvbuf, sizeof(recvbuf), 0) < 0)
        {
            perror("Recv()");
            exit(6);
        }//recebe a resposta do servidor
        printf("Servidor: %s\n", recvbuf);
        quantidade++;
    break;
```

Figura 3.12 - Caso o cadastrar seja selecionado, enviamos ao servidor qual a operação será realizada (“cad”).

Em seguida, o usuário digita o nome e a mensagem que serão cadastrados e os concatenamos no formato combinado, salvo na string “envio”.

Então, enviamos a string ao servidor e aguardamos sua resposta (sucesso ou fracasso no cadastro da mensagem).



```
case 2:
    // Ler mensagens
    strcpy(operacao, "ler");

    if (send(s, operacao, strlen(operacao) + 1, 0) < 0)
    {
        perror("Send()");
        exit(5);
    } //informa ao servidor qual operacao sera feita

    //salva as mensagens e ja printa
    if (recv(s, recvbuf, sizeof(recvbuf), 0) < 0)
    {
        perror("Recv()");
        exit(6);
    } //recebe o numero de mensagens cadastradas
    printf("\n%s\n", recvbuf);
    //talvez tenhamos que converter quantidade para int (char)

    char indice_recebido[2];
    strcpy(indice_recebido, recvbuf);
    quantidade = atoi(indice_recebido);
    printf("QUANTIDADE: %d\n", quantidade);


    for (int i = 0; i < quantidade; i++)
    {
        //recebe o nome e imprime
        if (recv(s, envio, sizeof(envio), 0) < 0)
        {
            perror("Nome()");
            exit(6);
        }

        if (send(s, "OK", 3, 0) < 0)
        {
            perror("Send()");
            exit(5);
        }

        strcpy(nome, strtok(envio, "#"));
        strcpy(mensagem, strtok('\0', "$$"));

        printf("Usuario: %s", nome);
        printf("\t\tMensagem: %s\n", mensagem);
    }
    break;
```

Figura 3.13 - No caso da operação de leitura, enviamos ao servidor a string “ler”, o servidor responde ao cliente a quantidade de mensagens cadastradas para que usemos na iteração.

Em seguida, recebemos um nome e uma mensagem (em uma unica string) a cada execução do “for” e as exibimos na tela (novamente na convenção adotada).

```
case 3:
    // Apagar mensagem
    strcpy(operacao, "apa");

    if (send(s, operacao, strlen(operacao)+1, 0) < 0)
    {
        perror("Send()");
        exit(5);
    }//informa ao servidor qual operacao sera feita

    printf("Digite o nome do usuario que tera a mensagem apagada: ");

    __fpurge(stdin);
    fgets(nome, sizeof(nome), stdin);
    strtok(nome, "\n"); //tira o \n inserido pelo fgets

    if (send(s, nome, strlen(nome)+1, 0) < 0)
    {
        perror("Send()");
        exit(5);
    }//envia o nome associado a mensagem que sera apagada

    if (recv(s, recvbuf, sizeof(recvbuf), 0) < 0)
    {
        perror("Recv()");
        exit(6);
    }//recebe a resposta do servidor
    if(strcmp(recvbuf, "Usuario e mensagem apagado com sucesso!\n") == 0){
        quantidade--;
    }
    printf("\n%s\n", recvbuf);

break;
```

Figura 3.14 - Para apagar alguma mensagem, enviamos ao servidor o comando “apa” e em seguida o nome do dono da mensagem. Então, aguardamos a resposta do servidor, para caso de sucesso na exclusão da mensagem, ou fracasso (não existir mensagem cadastrada sob esse nome ou não existir nenhuma mensagem cadastrada).

```
case 4: //sair
    printf("Obrigado por utilizar a aplicacao\n");
break;
```

Figura 3.15 - Se o cliente optar por sair, encerramos a execução do do-while.

```c
/* Fecha o socket */
close(s);

printf("Cliente terminou com sucesso.\n");
exit(0);
```

Figura 3.16 - E fechamos a conexão com servidor, a fim de não utilizar o socket de maneira ociosa e sobrecarregar os recursos do servidor.

Testes

```
victor@victor-TUF-GAMING-FX504GE-FX80GE:~/Redes-de-Computadores-A/Atividade 2$ .
/cliente localhost 6000

        *** Menu ***

1 - Cadastrar mensagem
2 - Ler mensagens
3 - Apagar mensagens
4 - Sair da aplicacao

Opcao: 1

Digite o nome: maria
Digite a mensagem: oi, tudo bem?

Nome recebido do cliente: maria

Mensagem recebida do cliente: oi, tudo bem?

Envio: maria#oi, tudo bem?$$
Servidor: Mensagem cadastrada com sucesso!
```

Figura 4.1 - Cliente ao cadastrar mensagem

```
/servidor 6000
*** Servidor Iniciado! ***

Mensagem recebida do cliente: cad

Mensagem inteira: maria#oi, tudo bem?$$
Usuario cadastrado: maria
Mensagem cadastrada: oi, tudo bem?
Indice: 0
Mensagem enviada ao cliente: Mensagem cadastrada com sucesso!
```

Figura 4.2 - Servidor ao cadastrar mensagem

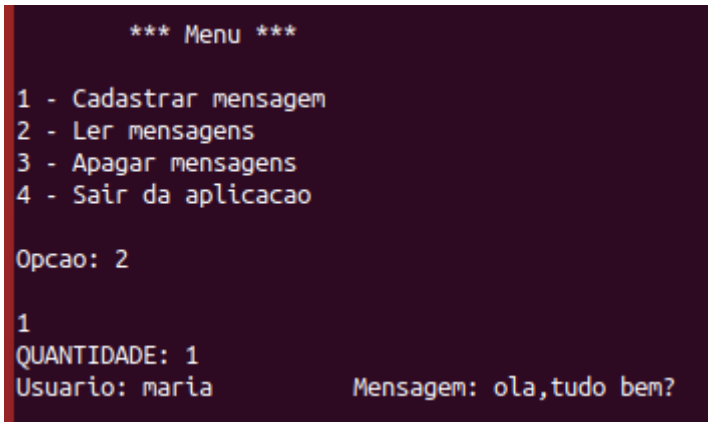


Figura 4.3 - Leitura das mensagens pelo cliente

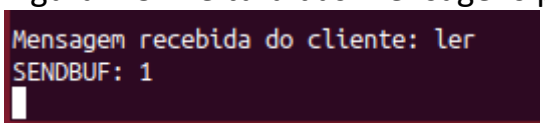


Figura 4.4 - Servidor recebendo parâmetros e passando a quantidade de mensagens

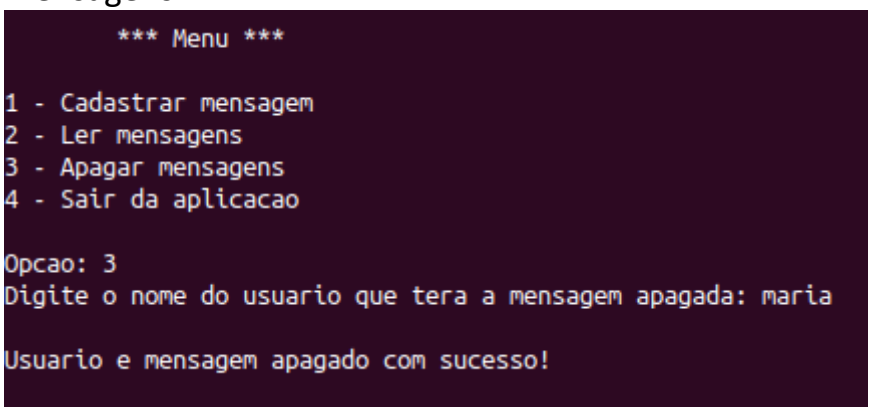


Figura 4.5 - Cliente solicita que as mensagens de um usuário específico sejam apagadas

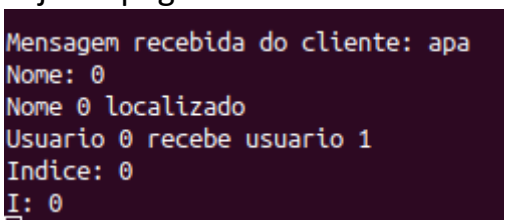
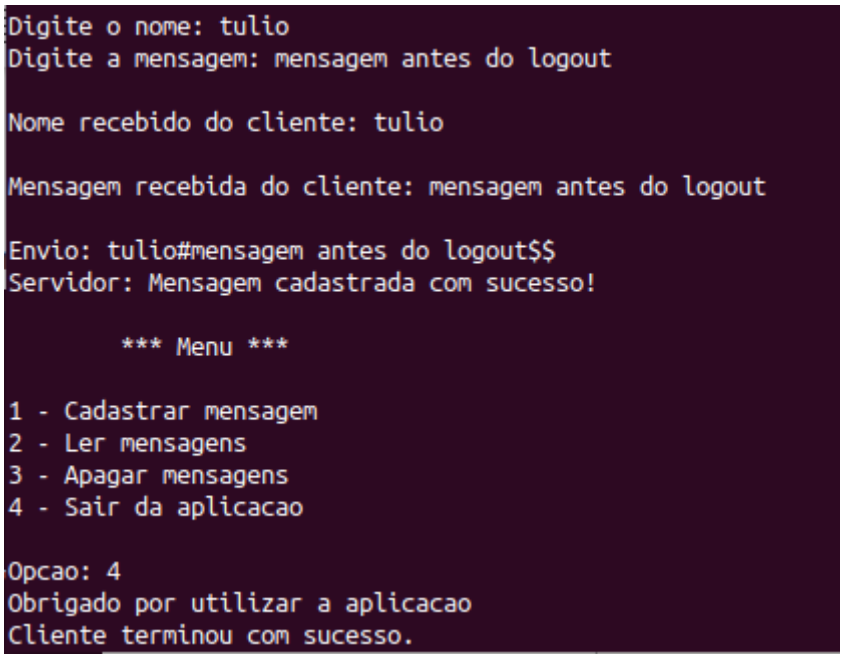
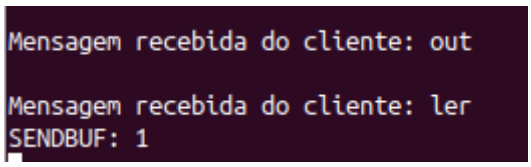


Figura 4.6 - Servidor recebe parâmetros, exclui as mensagens e realoca as mensagens remanescentes.





Figuras 4.7, 4.8 e 4.9 Como os dados são salvos sempre no servidor, o cliente tem a opção de cadastrar mensagens, fazer logout, entrar novamente, e ler as mensagens anteriormente cadastradas.

Conclusão

Para a aplicação em questão, o TCP foi interessante de se utilizar, pois, apenas um cliente se conectou, então seria próximo do impossível esgotar o buffer de conexões. Além disso, por usarmos TCP, contamos com a certeza de que nossas mensagens chegariam ao cliente quando pedisse, o que não acontece caso usássemos o UDP básico. Se optarmos pelo último, deveríamos fazer implementar manualmente o processo de confiabilidade dos envios e recebimentos das mensagens.