UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
DEPARTAMENTO DE INFORMÁTICA APLICADA
INF01154 – REDES DE COMPUTADORES N
PROF. VALTER ROESLER
Jéssica Rocha - 242294
Marcely Zanon Boito - 228454

Exercício UDP

Para este exercício utilizamos e modificamos o esqueleto de transmissão UDP fornecido pelo professor Valter.

Funcionamento do esqueleto:

O esqueleto fornecido pelo professor é composto de duas partes:

1) O transmissor envia mensagens de 100bytes (800 bits)
2) O receptor recebe essas mensagens e envia uma mensagem de ACK

O programa e as modificações:

1) Receptor

O objetivo do receptor é basicamente abrir uma conexão e esperar por mensagens de máquinas clientes. Para isso, é necessário definir o socket da família AF_INET e informar que tal socket será UDP (linhas 60-63).

Uma vez o socket criado, é necessário inicializar variáveis para manter a conversa (IPs aceitos, porta de acesso). Isso está ilustrado nas linhas 66 a 70 do programa abaixo.

```
59  // Cria o socket na familia AF_INET (Internet) e do tipo UDP (SOCK_DGRAM)
60      if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
61          printf("Falha na criacao do socket\n");
62          exit(1);
63      }
64
65  // Define dom?nio, IP e porta a receber dados
66      memset((void *) &peer,0,sizeof(struct sockaddr_in));
67      peer.sin_family = AF_INET;
68      peer.sin_addr.s_addr = htonl(INADDR_ANY); // Recebe de qualquer IP
69      peer.sin_port = htons(porta); // Recebe na porta especificada na linha de comando
70      peerlen = sizeof(peer);
71
72  // Associa socket com estrutura peer
73      if(bind(s,(struct sockaddr *) &peer, peerlen)) {
74          printf("Erro no bind\n");
75          exit(1);
76      }
77
78      printf("Socket inicializado. Aguardando mensagens...\n\n");
79
```

Assim, após tudo configurado, o programa receptor entra em um laço infinito, esperando mensagens e respondendo-as com mensagens de ACK. Como funções de impressão em console são demoradas, nós removemos os *printfs* que existiam nesse laço, deixando só o essencial. Outra mudança importante é o tamanho do buffer, que foi alterado para 1250 bytes (1Kbit).

```
79      while (1)
80      {
81          // Quando recebe um pacote, automaticamente atualiza o IP da estrutura peer
82          rc = recvfrom(s,buffer,sizeof(buffer),0,(struct sockaddr *) &peer,(socklen_t *)&peerlen);
83          sendto(s,buffer,sizeof(buffer),0,(struct sockaddr *)&peer, peerlen);
84      }
85
86  }
```

2) Transmissor

Semelhante ao receptor, transmissor possui em seu corpo a criação de socket (linhas 72-75) e definição das variáveis necessárias para a conversa (linhas 78-81). Em nossa versão do programa, recuperamos a taxa de transmissão da entrada (argumento -r) e utilizamos esse valor para calcular o intervalo entre as transmissões.

Isso é necessário para garantirmos que estamos enviando no máximo X pacotes por segundo, com X sendo o número de pacotes que totaliza na taxa de transmissão escolhida como parâmetro de entrada. Fazendo uma simples regra de três, chegamos ao parâmetro *sleep_time*.

```
71  // Cria o socket na familia AF_INET (Internet) e do tipo UDP (SOCK_DGRAM)
72      if((s = socket(AF_INET, SOCK_DGRAM,0)) < 0) {
73          printf("Falha na criacao do socket\n");
74          exit(1);
75      }
76
77  // Cria a estrutura com quem vai conversar
78      peer.sin_family = AF_INET;
79      peer.sin_port = htons(porta);
80      peer.sin_addr.s_addr = inet_addr(ip);
81      peerlen = sizeof(peer);
82
83  // Calcula o sleep time em mili
84      sleep_time = (float) 1.0 / (kbps);
85      sleep_time = sleep_time * 1000000; //mili
86
87      strcpy(buffer,"Enviando um pacote!");
88
```

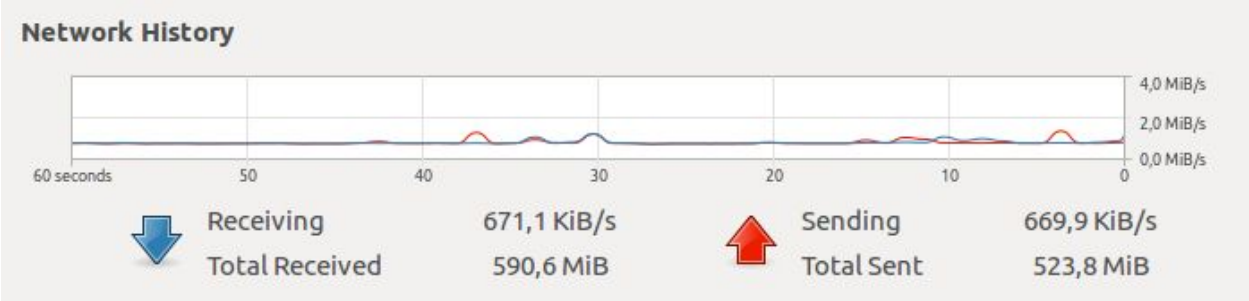
Como chamar:

./trans -h <ip da máquina receptora> -p <número de porta> -r <taxa em kbps>

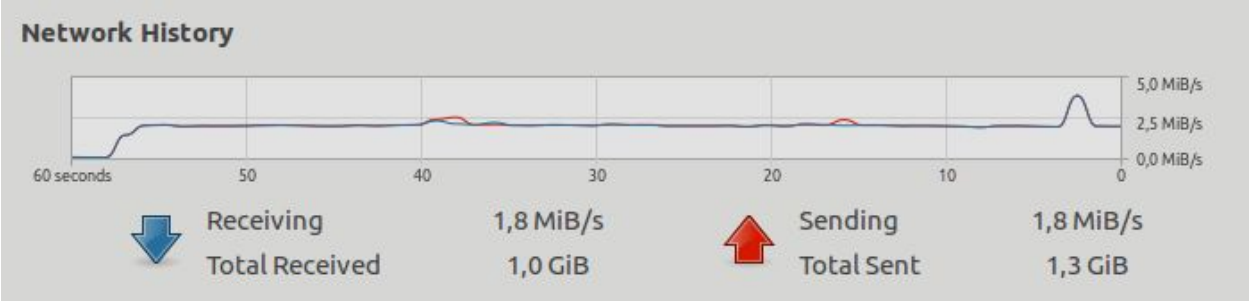
./rec -p <número de porta>

Exercícios: Abaixo listamos as transmissões solicitadas pelo professor.

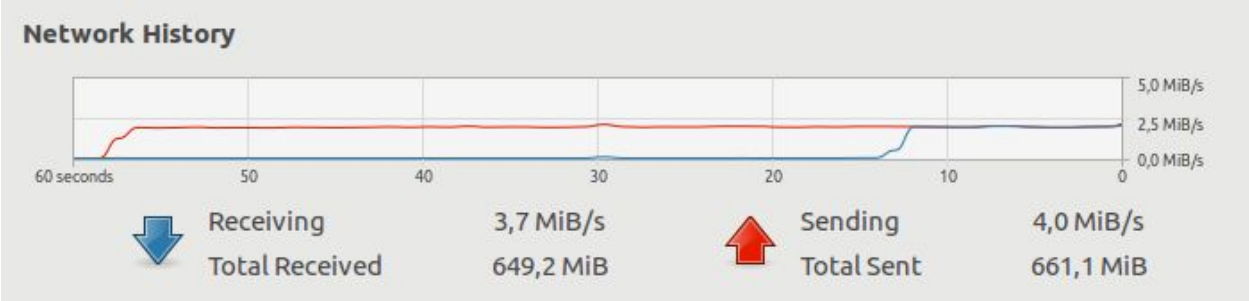
1) **-r 600Kbps** (24 (idade Marcely) * 25 (idade Jéssica))



2) **-r 2000Kbps**



3) **-r 4000Kbps**



Pela natureza do programa utilizado, não conseguimos atingir exatamente os valores desejados. Entretanto, chegamos suficientemente perto em todos os casos.

Exercício TCP

Para este exercício utilizamos e modificamos o esqueleto de transmissão TCP fornecido pelo professor Valter.

Funcionamento do esqueleto:

O esqueleto fornecido pelo professor é composto de duas partes:

3) O servidor abre uma conexão TCP em uma determinada porta, recebe mensagens e imprime essas mensagens no console. Existe a possibilidade de sair da conexão enviando uma string special (“q”).
4) O transmissor se conecta ao servidor e envia mensagens pelo stdin do console.

O programa e as modificações:

(ocultamos o código de encerramento de conexão, por sua simplicidade)

1) **Servidor**

Similar ao UDP, aqui também precisamos criar um socket e definir variáveis. Entretanto, existe um número maior de processos a serem feitos, e temos uma função de tratamento de erros, deixando a comunicação mais sofisticada.

```
66    // Cria o socket na familia AF_INET (Internet) e do tipo TCP (SOCK_STREAM)
67    if ((s = socket(AF_INET, SOCK_STREAM, 0))==INVALID_SOCKET)
68       TrataErro(s, ABRESOCK);
69
70    // Define dom?nio, IP e porta a receber dados
71    addr_serv.sin_family = AF_INET;
72    addr_serv.sin_addr.s_addr = htonl(INADDR_ANY); // recebe de qualquer IP
73    addr_serv.sin_port = htons(PORTA_SRV);
74
75    // Associa socket com estrutura addr_serv
76    if ((bind(s, (struct sockaddr *)&addr_serv, sizeof(addr_serv))) != 0)
77       TrataErro(s, BIND);
78
79    // Coloca socket em estado de escuta para as conexoes na porta especificada
80    if((listen(s, 8)) != 0) // permite ateh 8 conexoes simultaneas
81       TrataErro(s, LISTEN);
82
83    // permite conexoes entrantes utilizarem o socket
84    if((s_cli=accept(s, (struct sockaddr *)&addr_cli, (socklen_t *)&addr_cli_len)) < 0)
85       TrataErro(s, ACCEPT);
86
```

O laço de recepção de mensagens é bem similar ao caso UDP. Removemos a cópia do buffer para o console (o conteúdo enviado pelo transmissor era impresso no console do servidor) pois achamos irrelevante manter isso após a remoção da iteração por console da parte do transmissor (estava basicamente só imprimindo lixo da memória).

```
// fica esperando chegar mensagem
while(1)
{
  if ((recv(s_cli, recvbuf, MAX_PACKET, 0)) < 0)
  {
    close(s_cli);
    TrataErro(s, RECEIVE);
  }

  printf("I received a message! \n");
}
```

2) Transmissor

Semelhante ao receptor, o transmissor possui em seu corpo a criação de socket (linhas 77-80), define as variáveis locais necessárias para a conversa (linhas 83-85), associa as configurações locais ao socket(linhas 88-92) e define as variáveis do servidor remoto(linhas 96-98). Por fim, conecta o cliente ao servidor(linhas 101-105).

```
76      // abre socket TCP
77      if ((s = socket(AF_INET, SOCK_STREAM, 0))==INVALID_SOCKET)
78      {
79          printf("Erro iniciando socket\n");
80          return(0);
81      }
```

```
82  // seta informacoes IP/Porta locais
83  s_cli.sin_family = AF_INET;
84  s_cli.sin_addr.s_addr = htonl(INADDR_ANY);
85  s_cli.sin_port = htons(PORTA_CLI);
86
87  // associa configuracoes locais com socket
88  if ((bind(s, (struct sockaddr *)&s_cli, sizeof(s_cli))) != 0)
89  {
90    printf("erro no bind\n");
91    close(s);
92    return(0);
93  }
94
95  // seta informacoes IP/Porta do servidor remoto
96  s_serv.sin_family = AF_INET;
97  s_serv.sin_addr.s_addr = inet_addr(STR_IPSERVIDOR);
98  s_serv.sin_port = htons(PORTA_SRV);
99
100 // connecta socket aberto no cliente com o servidor
101 if(connect(s, (struct sockaddr*)&s_serv, sizeof(s_serv)) != 0)
102 {
103   printf("erro na conexao");
104   close(s);
105   exit(1);
106 }
```

O próximo passo é verificar quantos pacotes estamos enviando por segundo. Para isso, nós estabelecemos novamente buffer de 1250 bytes (1K bits) e fizemos um laço de envio que faz o seguinte:

1) Envia um pacote de 1250 bytes
2) Atualiza o contador de pacotes enviados
3) Atualiza o tempo
4) Se já estamos enviando pacotes há um minuto, faz o seguinte:
    a) Imprime número da iteração (quantos segundos) e quantidade de bytes enviados (tamanho do pacote * contador de pacotes enviados)
    b) Zera contador de pacotes, atualiza o tempo e volta para o laço de envio

```
108   int buffer_size = 1250; // tamanho pacote
109   char str[buffer_size]; // criacao do buffer
110   int package_count =0; // count para os pacotes
111   int iter = 0; // count de iteracao
112   time_t t_now, t_init; // variaveis de controle de tempo
113   time(&t_init); //init tempo
114   while(1) {
115     if ((send(s, (const char *)&str, sizeof(str),0)) < 0) {
116       printf("erro na transmiss\357o\n");
117       close(s);
118       return 0;
119     }
120     package_count++;
121     time(&t_now);
122     if(difftime(t_now, t_init) >= 1.0){
123       iter++;
124       printf("%d\t%d\n", iter, package_count*buffer_size);
125       package_count = 0;
126       time(&t_init);
127     }
128   }
```

Como chamar:

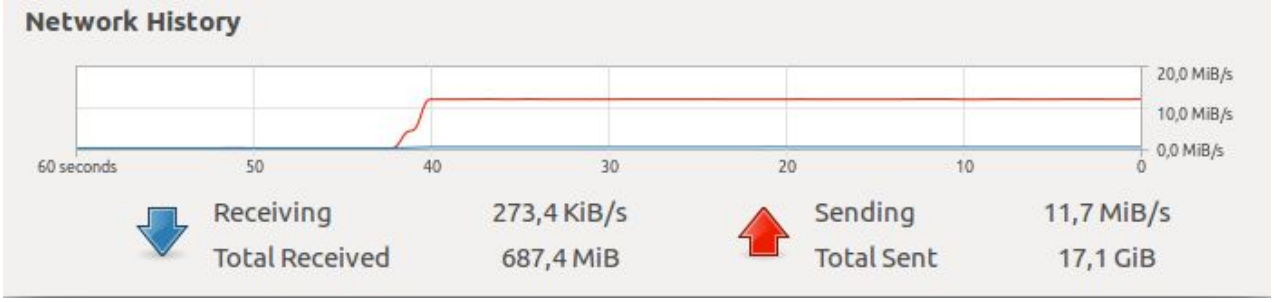
./testesrv -s <número de porta server>

./testecli -h <ip máquina receptora> -s <número de porta server> -c <número de porta cliente>

NOTA: Para essa aplicação, não conseguimos fazer ele aceitar os ifs de define de variáveis do Windows. Por causa disso, deixamos essas partes do programa comentadas, para caso o professor decida testar o programa no windows.

Exercício: O objetivo era conectar dois clientes no mesmo servidor. Conectando o primeiro, verificamos a taxa de transmissão da conexão exclusiva. Então, adicionando um segundo cliente, podemos observar a queda (em 50%) dessa taxa de transmissão. Isso é uma característica do protocolo TCP que até o momento parece um pouco “mágica” para nós. De alguma forma o protocolo é capaz de identificar quantos nós estão conectados e dividir o total disponível de recursos igualmente entre essas máquinas.

Com somente uma conexão:



Terminal com duas conexões e três máquinas:

Servidor

```
jsrrocha@s-72-206-12:~/Downloads/redes_lab3-master/transrec_tcp$ ^C
jsrrocha@s-72-206-12:~/Downloads/redes_lab3-master/transrec_tcp$ ./testesrv -s 8000
```

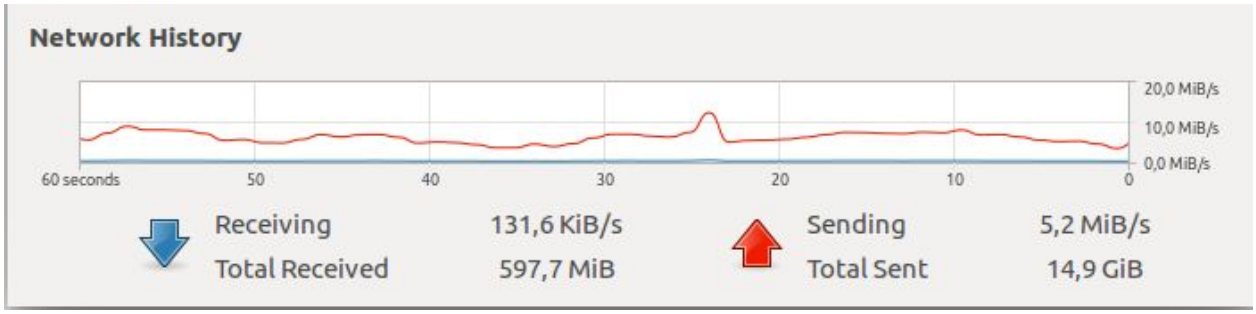
```
jsrrocha@s-72-206-12:~/Downloads/redes_lab3-master/transrec_tcp$ ^C
jsrrocha@s-72-206-12:~/Downloads/redes_lab3-master/transrec_tcp$ ./testesrv -s 8001
```

Clientes

```
jsrrocha@s-72-206-13: /home/grad/jsrrocha/Downloads/redes_lab3-master/transrec_tcp
^C
jsrrocha@s-72-206-13:~/Downloads/redes_lab3-master/transrec_tcp$ ./testecli -h 143.54.6.51 -s 8000 -c 7000
1       1552500
2       9578750
3       7493750
4       7037500
5       7492500
6       6646250
7       6386250
8       5082500
9       5082500
10      5082500
11      5603750
12      5928750
13      6598750
14      5912500
15      5930000
16      5863750
17      6646250
18      5865000
19      5928750
20      4692500
21      6320000
22      7037500
```

```
mzboito@s-72-206-14: /home/grad/mzboito/Downloads/redes_lab3-master/transrec_tcp
5       4366250
6       11011250
^C
mzboito@s-72-206-14:~/Downloads/redes_lab3-master/transrec_tcp$ ./testecli -h 14
3.54.6.51 -s 8001 -c 7001
1       3181250
2       4171250
3       4691250
4       4496250
5       5342500
6       5212500
7       6646250
8       6842500
9       6906250
10      6191250
11      5537500
12      5538750
13      5538750
14      5865000
15      6190000
16      4886250
17      5865000
18      5930000
19      6841250
```

Gráficos de rede com duas conexões e três máquinas:



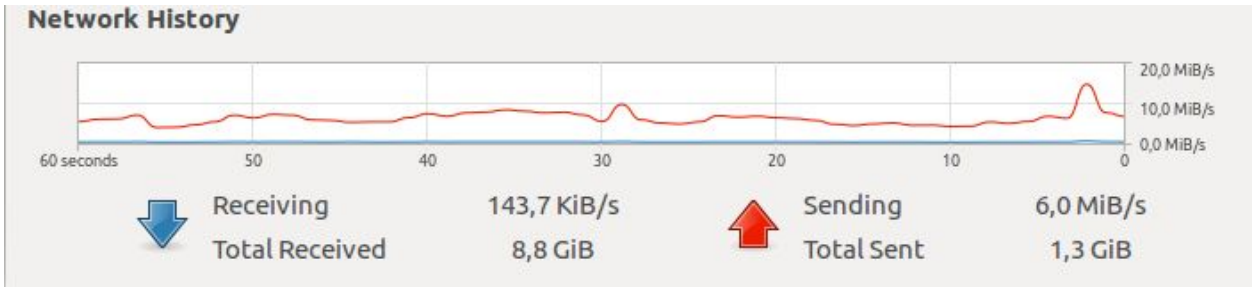
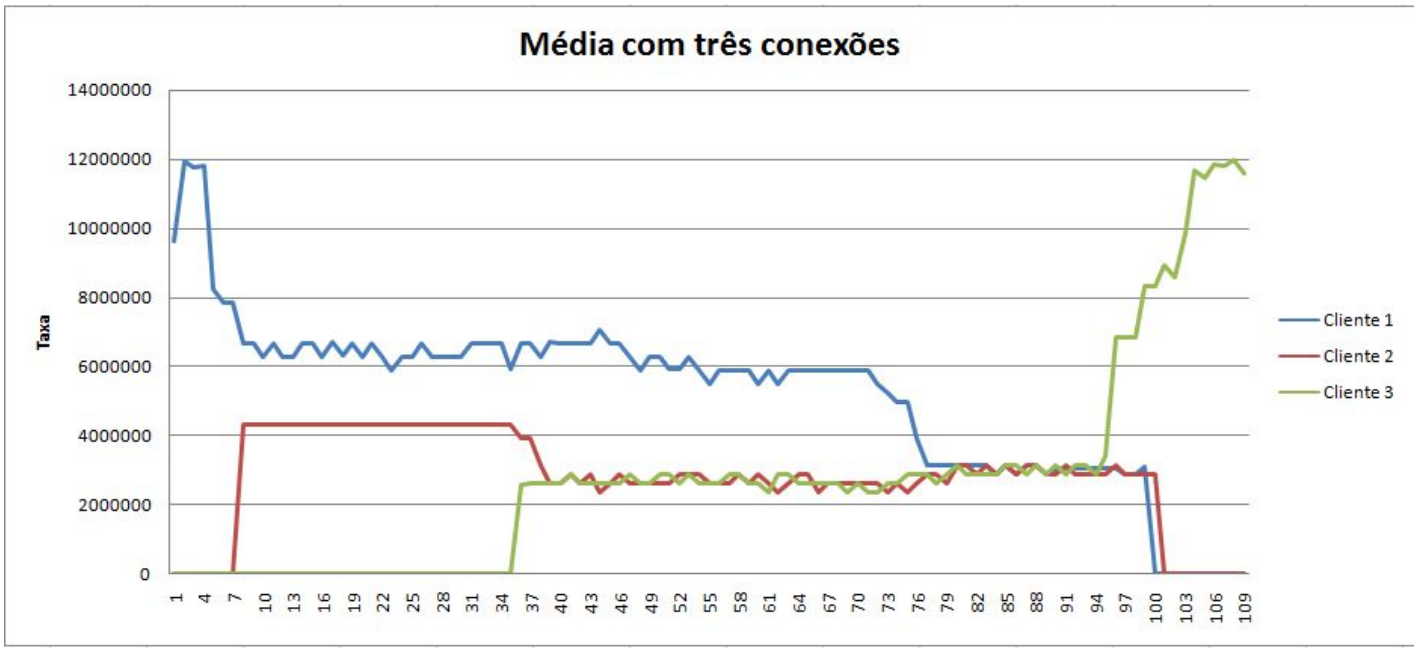


Gráfico com média de tráfego por segundo de três conexões e duas máquinas:



É possível perceber que o primeiro cliente monopolizou a rede por um tempo, não sabemos exatamente por qual motivo, pode ter sido porque precisamos acertar corretamente onde cada conexão começou ao fazer o gráfico, o que pode ter gerado errados de sobreposição dos valores das conexões. Depois parece que a rede realmente é dividida entre os três.

Eficiência do código:

O cliente precisa receber por parâmetro: O número do ip do servidor, o número da porta do servidor e o número de sua porta.

```
int main(int argc, char* argv[])
{
    SOCKET s;
    char STR_IPSERVIDOR[256];
    int PORTA_SRV, PORTA_CLI, i;
    struct sockaddr_in  s_cli, s_serv;

    if(argc < 6)
    {
        printf("Utilizar:\n");
        printf("trans -h <numero_ip> -s <porta_srv> -c <porta_cli>\n");
        exit(1);
    }
```

Também é feito uma verificação, testando se todos os argumentos vieram corretos para não ocorrer erros:

```
// Pega parametros
for(i=1; i<argc; i++)
{
    if(argv[i][0]=='-')
    {
        switch(argv[i][1])
        {
        case 'h': // Numero IP
            i++;
            strcpy(STR_IPSERVIDOR, argv[i]);
            break;
        case 's': // porta servidor
            i++;
            PORTA_SRV = atoi(argv[i]);
            if(PORTA_SRV < 1024)
            {
                printf("Valor da porta invalido\n");
                exit(1);
            }
            break;
        case 'c': // porta cliente
            i++;
            PORTA_CLI = atoi(argv[i]);
            if(PORTA_CLI < 1024)
            {
                printf("Valor da porta invalido\n");
                exit(1);
            }
```

O servidor só precisa receber o número da sua porta:

```
if(argc < 3) {
        printf("Utilizar:\n");
        printf("trans -s <porta_srv>\n");
        exit(1);
    }

    // Pega parametros
    for(i=1; i<argc; i++) {
        if(argv[i][0]=='-') {
            switch(argv[i][1]) {
                case 's': // porta servidor
                    i++;
                    PORTA_SRV = atoi(argv[i]);
                    if(PORTA_SRV < 1024) {
                        printf("Valor da porta invalido\n");
                        exit(1);
                    }
                    break;
                default:
                    printf("Parametro invalido %d: %s\n",i,argv[i]);
                    exit(1);
            }
        } else {
            printf("Parametro %d: %s invalido\n",i, argv[i]);
            exit(1);
        }
    }
```