

Trabalho Prático

Redes de Computadores

Protocolo Olímpico

com múltiplas conexões

1 Problema a ser tratado

As Olimpíadas serão no Brasil e o Comitê Olímpico Internacional (COI) precisa de ajuda. O COI precisa coletar os dados das avaliações dos atletas e classificá-los. Baseado nessa ideia, deve-se criar um programa cliente que envia os dados dos desempenhos dos atletas para um servidor e um programa servidor que informa a classificação geral do atleta dados os valores já recebidos naquela conexão.

Os programas deverão funcionar utilizando o TCP e devem funcionar com IPv4 e IPv6. O protocolo olímpico é baseado em texto, ou seja, os valores são strings de caracteres ASCII. O cliente irá se conectar ao servidor e enviará uma sequência de valores de tempo no formato onde haverá um número seguido do indicador de tempo (se este for maior que zero), onde h indicará horas, m indicará minutos, s segundos e ms milissegundos. Todos os valores são terminados por um “\n”. Pode ocorrer mais de um caractere de espaço, mas outros caracteres de separação tais como tab ou ‘\r’ não deverão ser considerados. O servidor deverá responder, para cada valor recebido, a classificação do tempo, ordenando pelos menores valores e sendo 1 a primeira posição, considerando todos os dados recebidos naquela conexão. O servidor deve aceitar várias conexões ao mesmo tempo.

2 Implementação

Os programas foram escritos em linguagem C que é uma linguagem de programação compilada de propósito geral, estruturada, imperativa, procedural, padronizada pela ISO. É uma das linguagens de programação mais populares. Um dos recursos mais atrativos na programação em C, é o uso de Sockets. O uso de sockets permite o desenvolvimento de várias aplicações; Exploits, flooders e programas cliente/servidor são poucos exemplos de aplicativos que podemos desenvolver.

Arquivo de cabeçalho

```
funcao.c x olim.h x
/*
  Arquivo de cabeçalho usado no Protocolo olímpico.
 */

#ifndef OLIM_H
#define OLIM_H

int temp_ms(char *msg); /* função que recebe uma string com o tempo no formato "xxh xxm xxs xxms"
                        e devolve o tempo em milissegundos*/

void insercao (int n, int v[]); /* método de ordenação "inserção" */

int pos(int n, int v[]);      /* função que recebe um vetor e um número e retorna a posição desse número*/
int verif_carac(char ch);    /* função que verifica se um caractere é um número ou um espaço*/
int carac_esp(char ch);      /* função que verifica se um caractere é 'h', 'm', 's' ou 'ms'.*/
#endif
```

Foi implementado o arquivo de cabeçalho “olim.h” que contém a declaração de cinco funções que foram usadas no programa Servidor.

A primeira recebe ponteiro de uma string contendo informações do tempo que o cliente enviou ao servidor e devolve esse tempo em milissegundos.

A segunda é o método de ordenação por inserção.

A terceira procura a posição de um dado número em um vetor.

A quarta verifica se um caractere é um número ou um espaço.

A última verifica se um caractere é 'h', 'm' ou 's'.

Funções Auxiliares

int verif_carac(char ch):

Esta função faz um teste simples com números e o espaço, retorna 1 caso seja um desses e 0 caso contrário.

```
/* função que verifica se um caractere é um número ou um espaço*/
int verif_carac(char ch) {
    if(ch == '0' || ch == '1' || ch == '2' || ch == '3' || ch == '4' || ch == '5' || ch == '6' || ch == '7' || ch == '8' || ch == '9' || ch == ' ')
        return(1);
    return (0);
}
```

int carac_esp(char ch):

Esta função faz um teste simples com os caracteres de tempo, retorna 1 se for um desses caracteres e 0 caso contrário.

```
}

/* função que verifica se um caractere é 'h', 'm', 's' ou 'ms'.*/
int carac_esp(char ch) {
    if(ch == 'h' || ch == 'm' || ch == 's')
        return (1);
    return (0);
}
```

void insercao (int n, int v[])

Metódo de ordenação bastante conhecido. **Insertion sort**, ou *ordenação por inserção*, é um simples algoritmo de ordenação, eficiente quando aplicado a um pequeno número de elementos. Em termos gerais, ele percorre um vetor de elementos da esquerda para a direita e à medida que avança vai deixando os elementos mais à esquerda ordenados. O algoritmo de inserção funciona da mesma maneira com que muitas pessoas ordenam cartas em um jogo de baralho como o pôquer

```
/* método de ordenação "inserção" */
void insercao (int n, int v[]) {
    /* o vetor v[0..n-1] é uma permutação do vetor original e
    o vetor v[0..j-1] está em ordem crescente.*/
    int i, j, x;
    for (j = 1; j < n; ++j) {
        x = v[j];
        for (i = j-1; i >= 0 && v[i] > x; --i)
            v[i+1] = v[i];
        v[i+1] = x;
    }
}
```

int pos(int n, int v[])

Esta função procura um elemento dentro do vetor e retorna a posição que ele está em relação ao primeiro elemento.

```
/* função que recebe um vetor e um número; retorna a posição desse número*/
int pos(int n, int v[]) {
    int i, j;
    for(i=0; i<1000; i++) { /* procura o primeiro elemento diferente de zero no vetor*/
        if(v[i] != 0) break; /* 'i' é a posição do primeiro elemento maior que zero*/
    }
    for(j=0; j<1000; j++) { /* procura dentro do vetor o elemento recebido na função */
        if(v[j] == n && v[j+1] != n) break; /* 'j' é a posição do elemento requisitado na função*/
    }
    return((j-i)+1); /* retorna a posição do elemento requisitado em relação ao primeiro elemento*/
}
```

int temp_ms(char *msg)

Esta função recebe uma *string* com os dados de tempo do cliente e transforma o tempo em milissegundos. A *string* recebida é percorrida enquanto não for encontrado o caractere de final de *string* '\0' pelo *while*. É feito um teste condicional com o *if* que ao encontrar um caractere de tempo copia toda a mensagem recebida para uma *string* auxiliar. Esta *string* auxiliar é então trabalhada até o início para eliminar caracteres não válidos e se houver um outro caractere de tempo antes do início, elimina-se então todos os elementos da *string* até a posição [0]. Feito isso a posição em que se encontra o caractere de tempo relativo ao tempo recebe um caractere finalizador de *string*. Em seguida é feita a conversão para decimal com a função *atoi*, o valor é armazenado na variável correspondente.

```
/* Esta função recebe uma string com o tempo no formato "xxh xxm xxs xxms"
e devolve o tempo em milissegundos, desde que ele seja maior que "0ms"*/
int temp_ms(char *msg){
    char str[256]; /* string auxiliar a ser chamada na função "atoi" */
    int h=0;
    int m=0, s=0, ms=0, i=0, j=0;
    int tempo=0;
    char ch;
    int test=2;
    int testc=0;

    while(msg[i] != '\0'){ /* verifica a string até o final */

        if(msg[i] == 'h'){ /* procura por hora dentro da string */
            strcpy(str, msg); /* copia a mensagem recebida*/
            for(j=i-1; j>=0; j--) { /* percorre a string antes do caractere especial de tempo */
                ch = str[j];
                test = verif_carac(ch); /* verifica se os caracteres anteriores são números ou espaço*/
                if(test == 0) str[j] = ' '; /* se o caractere for diferente de ' ' e de número, a string recebe um ' ' */
                testc = carac_esp(ch);
                if(testc>0){ /* se for encontrado um caractere especial de tempo a string é "zerada" até o início*/
                    while(j>=0) {
                        str[j] = ' ';
                        j--;
                    }
                }
            }
            str[i] = '\0'; /* finaliza a string*/
            h = atoi(str); /* valor inteiro é extraído da string*/
        }
        i++;
    }
    h += h*3600000;
    m += m*60000;
    s += s*1000;
    tempo = h + m + s + ms; /* a variável "tempo" recebe o tempo total em milissegundos */
    return(tempo);
}
```

Ao final desse processo a função retorna o tempo total em milissegundos.

```
    i++;
}
h += h*3600000;
m += m*60000;
s += s*1000;
tempo = h + m + s + ms; /* a variável "tempo" recebe o tempo total em milissegundos */
return(tempo);
}
```

Estruturas de dados

Servidor

```
struct sockaddr_storage ss; /* estrutura de sockets genérica */
struct addrinfo hints, *res; /* estruturas usadas na "getaddrinfo", "*res" conterá porta, tipo de sockets etc */
socklen_t addr_size = sizeof ss; /* tamanho da estrutura genérica a ser usada na função "accept" */
char buf[MAX_LINE]; /* armazena os dados recebidos do cliente */
int sockfd, new_fd; /* socket do servidor e do cliente respectivamente */
int len;

/* variáveis adicionais */
int i=0, p=0;
int cla[1000], t=0; /* cla[1000] é o vetor de classificação */
```

ss → Estrutura de endereço genérica, usada na função **accept** para aceitar qualquer tipo de conexão do cliente tanto IPv4 quanto IPv6.

hints, *res → Estrutura de endereço para **sockets**. A estrutura **"hints"** define os tipos de **família e protocolo** a serem recebidos do **IP**. A estrutura **"*res"** é um ponteiro que contém todos os dados necessários à conexão.

Char buf[MAX_LINE] → É vetor para armazenar as linhas recebidas pelo cliente.

Cla[1000] → É um vetor para armazenar a classificação dos atletas.

socklen_t addr_size = sizeof ss → Tamanho da estrutura genérica.

Cliente

```
struct addrinfo hints, *res; /* estruturas usadas na "getaddrinfo", "*res" conterá os dados de endereço, porta, tipo de sockets etc */
char buf[MAX_LINE]; /* armazena os dados a serem enviados ao servidor */
int s; /* socket do cliente */
int len;
int i;
```

hints, *res → Estrutura de endereço para **sockets**. A estrutura **"hints"** define os tipos de **família e protocolo** a serem recebidos do **IP**. A estrutura **"*res"** é um ponteiro que contém todos os dados necessários à conexão.

Char buf[MAX_LINE] → É um vetor para armazenar as linhas a serem enviadas pelo cliente.

Funções Principais

Servidor

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol) → Cria um novo descritor para comunicação.

bind(sockfd, res->ai_addr, res->ai_addrlen) → Atribui um endereço IP e uma porta a um **socket**.

listen(sockfd, 5) → Coloca o **socket** em modo passivo, para "escutar" portas.

new_fd = accept(sockfd, (struct sockaddr *)&ss, &addr_size) → Bloqueia o servidor até a chegada de requisição de conexão.

recv(new_fd, buf, sizeof(buf), 0) → Usada para receber dados em um **socket**.

Cliente

s = socket(res->ai_family, res->ai_socktype, res->ai_protocol) → Cria um novo descritor para comunicação.

connect(s, res->ai_addr, res->ai_addrlen) → Inicia conexão com servidor.

fgets(buf, sizeof(buf), stdin) → obtém linhas de texto da entrada padrão.

close(s) → Fecha um **socket**.

send(s, buf, len, 0) → Usada para enviar uma mensagem para um **socket**.

Múltiplas Conexões

Para que o servidor possa tratar múltiplas conexões foi usado o recurso **Modelo POSIX Threads (Pthreads)**

Modelo que suporta a criação e manipulação de tarefas cuja execução possa ser intercalada ou executada concorrentemente.

O modelo Pthreads pertence à família POSIX (Portable Operating System Interface) e define um conjunto de rotinas (biblioteca) para manipulação de threads.

As definições da biblioteca Pthreads encontram-se em 'pthread.h' e sua implementação em 'libpthread.so'.

Para compilar um programa com threads é necessário incluir o cabeçalho '#include <pthread.h>' no início do programa e compilá-lo com a opção '-lpthread'.

Quando o programa inicia, uma thread (main thread) é criada. Após isso, outras threads podem ser criadas através da função:

```
--
03
04     if(pthread_create(&sniffer_thread, NULL, connection_handler, (void*) new_fd) <0){
05         perror("erro ao criar thread");
06         return 1;
07     }
08
09 }
10 }
```

função **void *connection_handler(void *socket_desc)**

```

59
60 void *connection_handler(void *socket_desc){
61
62     int sock = *(int*)socket_desc;
63     char buf[MAX_LINE];
64     int len;
65     int i=0, p=0;
66     int cla[1000], t=0; // é o vetor de classificação
67
68     for(i=0; i<1000; i++) cla[i] = 0; //inicializa vetor de classificação
69     i=0;
70
71     while(len = recv(sock, buf, sizeof(buf), 0)) {
72
73         if(buf[0] == '-') break;
74
75         t=0;
76         t = temp_ms(buf); //a variável t recebe o tempo total em ms do cliente
77         if(t != 0) { // se tempo for diferente de '0', insere no vetor
78             cla[i] = t;
79             i++;
80             insercao(1000, cla); /* o método de ordenação "inserção" é aplicado ao vetor*/
81             p = pos(t, cla); // a variável 'p' recebe a posição depois da ordenação
82             printf("%d\n", p); //imprime a posição do atleta
83
84         }
85     }

```

Esta função é chamada quando se cria uma nova **Thread**, ela tem os parâmetros para manipular os dados do novo cliente como o descritor do **socket** e as variáveis e estruturas de dados necessárias para armazenar os dados do cliente, bem como a função de recebimento de dados e a impressão na tela dos valores de classificação dos atletas.

Parte responsável pela recepção do cliente e criação da **Thread**.

```

89
90
91 /* espera conexão, depois recebe tempo do atleta e processa os dados recebidos*/
92 while(1){
93
94     if((client_s = accept(sockfd, (struct sockaddr *)&ss, &addr_size)) < 0) {
95         perror("simplex-talk: accept");
96         exit(1);
97     }
98
99     pthread_t sniffer_thread;
100
101     new_fd = malloc(1);
102     *new_fd = client_s;
103
104     if(pthread_create(&sniffer_thread, NULL, connection_handler, (void*) new_fd) < 0){
105         perror("erro ao criar thread");
106         return 1;
107     }
108
109 }
110 }
111
112
113

```


Decisões de implementação

Levando-se em consideração a especificação relativo a espaços , o servidor aceita dados do cliente com um ou mais espaços entre os números e o seu descritor de tempo, por exemplo:

<10h 2m 3s 200ms> e <10 h 2 m 3 s 200 ms> são aceitos como entrada padronizada.

No entanto entrada como: < 1 0 h 2 m 3 s 2 0 0 ms> não é considerada entrada padronizada pois o número '10' é diferente de '1 0', aqui entendido com dois números separados, a saber '1' e '0'.

3 Tratamento do IPv6 e IPv4

Para funcionar tanto em IPv6 quanto com IPv4, os programas foram implementados de tal forma que a chamada de uma função específica “*getaddrinfo*” fornece todos os parâmetros necessários ao estabelecimento da comunicação entre o cliente e o servidor.

Servidor

```
struct sockaddr_storage ss; /* estrutura de sockets genérica */
struct addrinfo hints, *res; /* estruturas usadas na "getaddrinfo", "res" conterá porta, tipo de sockets etc */
socklen_t addr_size = sizeof ss; /* tamanho da estrutura genérica a ser usada na função "accept" */
char buf[MAX_LINE]; /* armazena os dados recebidos do cliente */
int sockfd, new_fd; /* socket do servidor e do cliente respectivamente */
int len;

/* variáveis adicionais */
int i=0, p=0;
int cla[1000], t=0; /* cla[1000] é o vetor de classificação */

if(argc != 2){
    fprintf(stderr, "usage: simplex-talk Port\n");
    exit(1);
}

/* em primeiro lugar, carrega estruturas de endereço com getaddrinfo(): */

memset(&hints, 0, sizeof hints); /* zera a estrutura */
hints.ai_family = AF_UNSPEC; /* a família escolhida é a AF_INET6, pois aceita conexões IPv4 e IPv6 */
hints.ai_socktype = SOCK_STREAM; /* o protocolo esperado para esta conexão é o TCP */
hints.ai_flags = AI_PASSIVE; /* flag sinalizando para preencher IP */

getaddrinfo(NULL, argv[1], &hints, &res); /* "argv[1]" é a porta recebida por parâmetro, res-> é um ponteiro para a estrutura */

/* prepara a abertura passiva */

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

bind(sockfd, res->ai_addr, res->ai_addrlen);

listen(sockfd, 5);

/* espera conexão, depois recebe tempo do atleta e processa os dados recebidos */
```

C ▾ Largura da tabulação: 8 ▾ Lin 36, Col 11

A função *getaddrinfo*:

```
int getaddrinfo( const char *hostname, const char *service, const struct addrinfo *hints, struct
addrinfo **result);
```

Por exemplo:

Cliente

```
nt main (int argc, char *argv[]){

    struct addrinfo hints, *res; /* estruturas usadas na "getaddrinfo", "*res" conterá os dados de endereço, porta, tipo de sockets etc */
    char buf[MAX_LINE]; /* armazena os dados a serem enviados ao servidor */
    int s; /* socket do cliente */
    int len;
    int i;

    if(argc != 3) {
        fprintf(stderr, "usage: simplex-talk Port/host\n");
        exit(1);
    }
    /* em primeiro lugar, carrega estruturas de endereço com getaddrinfo (): */
    memset(&hints, 0, sizeof hints); /* zera a estrutura*/
    hints.ai_family = AF_UNSPEC; /* a família não está definida, pois aceita conexões IPv4 e IPv6*/
    hints.ai_socktype = SOCK_STREAM; /* o protocolo esperado para esta conexão é o TCP */
    hints.ai_flags = AI_CANONNAME; /*ai_flags como AI_CANONNAME para que o hostname descoberto seja retornado na struct addrinfo */

    if(getaddrinfo(argv[1], argv[2], &hints, &res)) { /* "argv[1]" é o IP recebido por parâmetro, "argv[2]" é a porta, "res->" é um ponteiro para a estrutura */
        fprintf(stderr, "usage: simplex-talk getaddrinfo\n");
        exit(1);
    }
    /* abertura ativa*/
    s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

    if(s < 0) {
        perror("simplex-talk: socket");
        exit(1);
    }

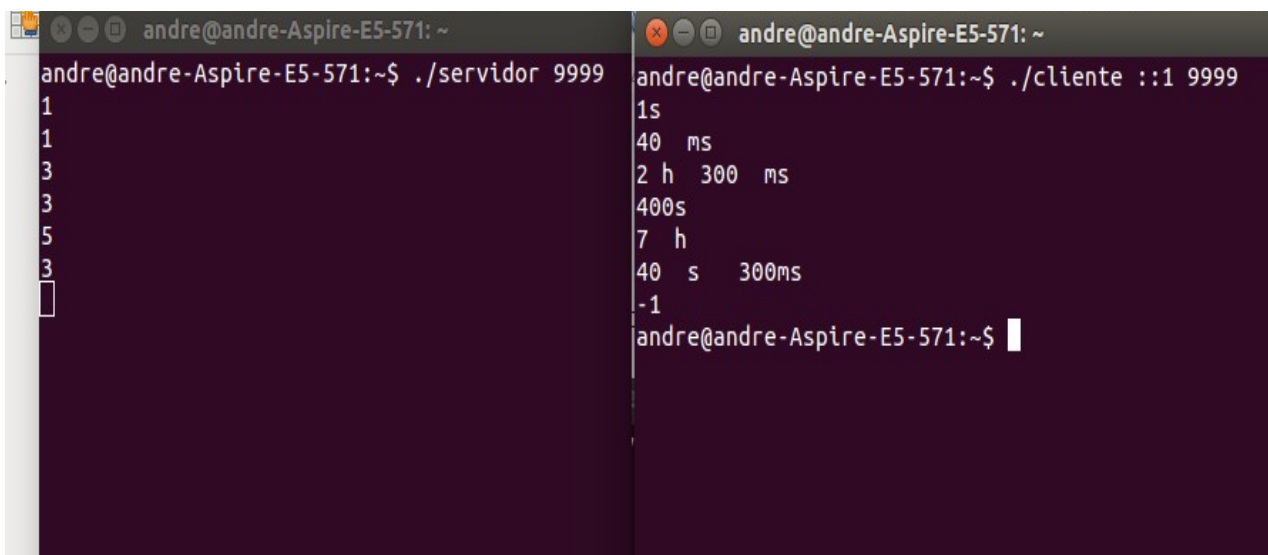
    if(connect(s, res->ai_addr, res->ai_addrlen) < 0) {
        perror("simplex-talk: connect");
        exit(1);
    }
}
```

Por exemplo:

Para se conectar, depois da abertura do `socket`, usa-se `re->ai_addr`, `res->ai_addrlen` e o inteiro relativo à abertura do `socket`.

4 Testes

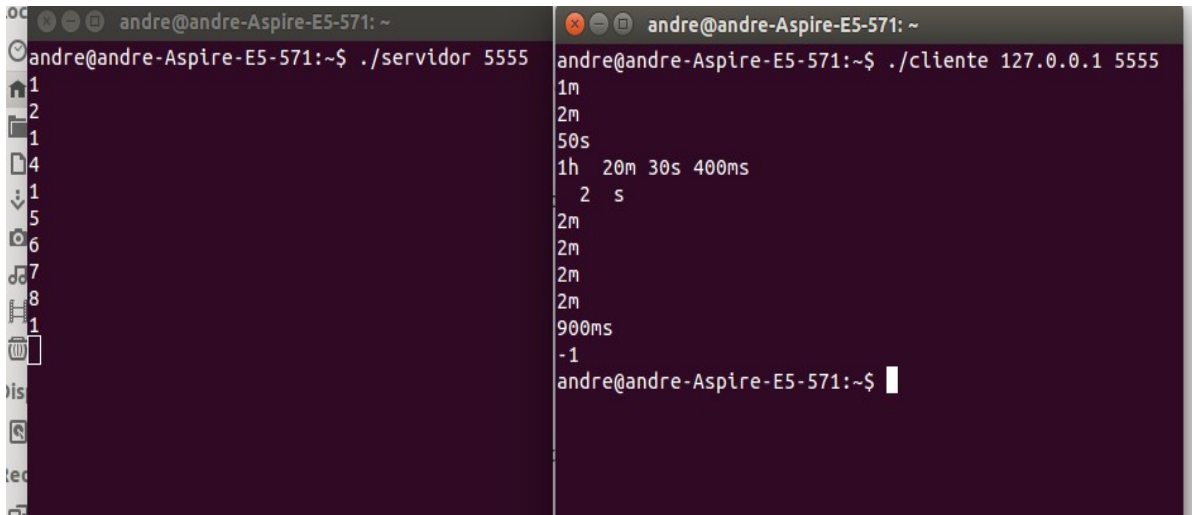
Testes realizados com cliente e servidor funcionando na mesma máquina. IPv6 local.



```
andre@andre-Aspire-E5-571: ~  
andre@andre-Aspire-E5-571:~$ ./servidor 9999  
1  
1  
3  
3  
5  
3  
[ ]  
andre@andre-Aspire-E5-571:~$  
andre@andre-Aspire-E5-571: ~  
andre@andre-Aspire-E5-571:~$ ./cliente ::1 9999  
1s  
40 ms  
2 h 300 ms  
400s  
7 h  
40 s 300ms  
-1  
andre@andre-Aspire-E5-571:~$
```

O primeiro valor recebido será sempre o primeiro classificado, neste teste '1s', em seguida o cliente manda '40ms', que agora passa a ser o primeiro, depois vem '2h 300ms' que é maior que os dois primeiros valores, assim fica na posição 3. Em seguida o cliente manda '400s' que é menor que '2h 300ms' e maior que '1s' e '40 ms', portanto passa a ocupar a posição 3. Logo após o cliente manda '7h', esse valor é o maior de todos, então fica na posição 5. O último valor enviado pelo cliente foi '40 s 300ms', que só não é menor que '1s' e '40 ms', portanto fica na posição 3. Por fim o cliente deseja fechar a conexão com o servidor, para isso ele manda um valor negativo '-1'.

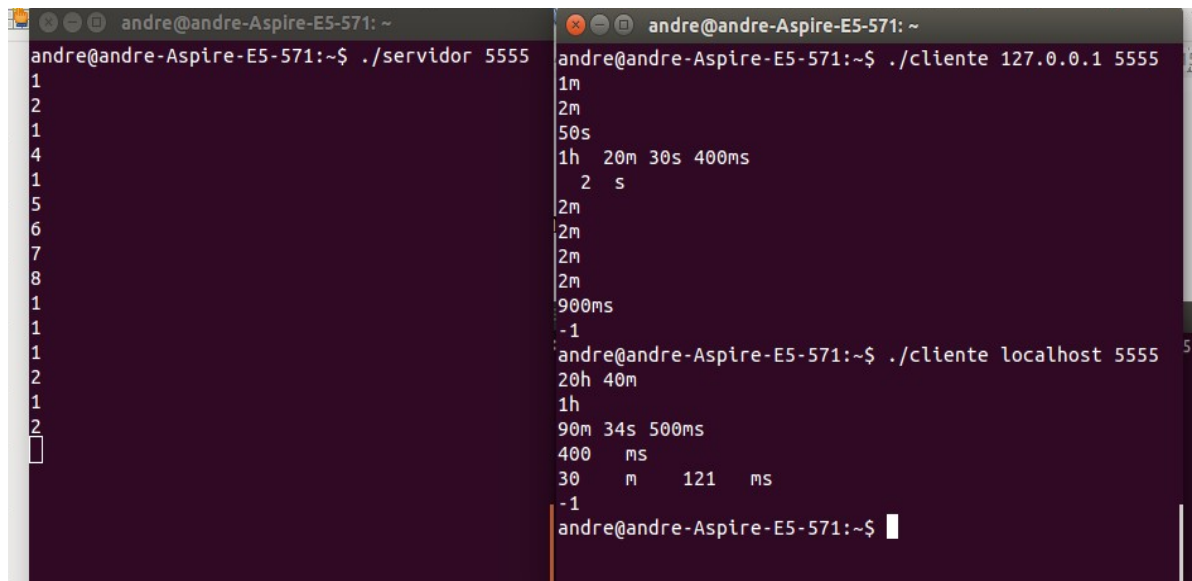
IPv4 local – Número.



```
andre@andre-Aspire-E5-571: ~  
andre@andre-Aspire-E5-571:~$ ./servidor 5555  
1  
2  
1  
4  
1  
5  
6  
7  
8  
1  
vis  
tec
```

```
andre@andre-Aspire-E5-571: ~  
andre@andre-Aspire-E5-571:~$ ./cliente 127.0.0.1 5555  
1m  
2m  
50s  
1h 20m 30s 400ms  
2 s  
2m  
2m  
2m  
2m  
900ms  
-1  
andre@andre-Aspire-E5-571:~$
```

IPv4 local – Nome.

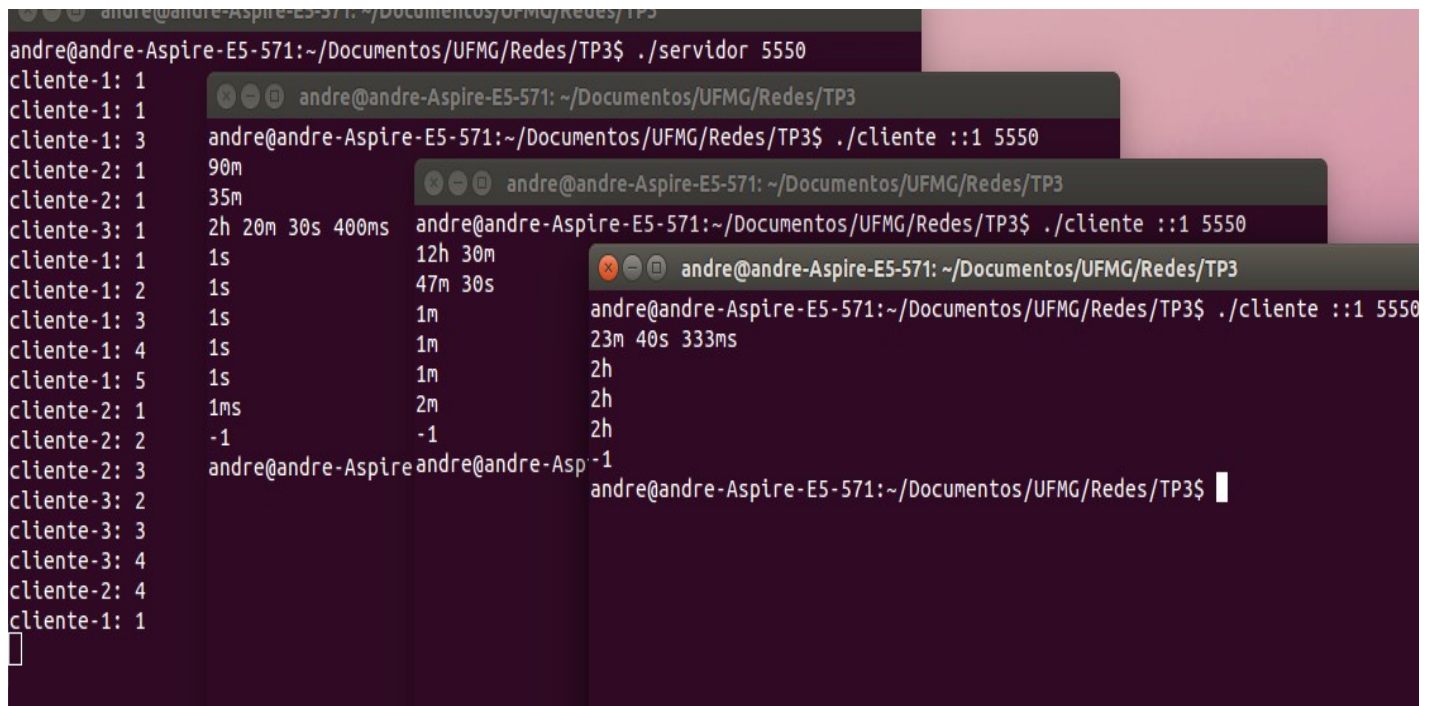


```
andre@andre-Aspire-E5-571: ~  
andre@andre-Aspire-E5-571:~$ ./servidor 5555  
1  
2  
1  
4  
1  
5  
6  
7  
8  
1  
1  
1  
2  
2
```

```
andre@andre-Aspire-E5-571: ~  
andre@andre-Aspire-E5-571:~$ ./cliente 127.0.0.1 5555  
1m  
2m  
50s  
1h 20m 30s 400ms  
2 s  
2m  
2m  
2m  
2m  
900ms  
-1  
andre@andre-Aspire-E5-571:~$ ./cliente localhost 5555  
20h 40m  
1h  
90m 34s 500ms  
400 ms  
30 m 121 ms  
-1  
andre@andre-Aspire-E5-571:~$
```

Os dois testes realizado posteriormente seguem a mesma lógica. Fica claro que depois de fechada a conexão IPv4 número o servidor continua funcionando normalmente, porém não guarda os valores recebidos do cliente anterior.

5 Testes com multiplas conexões



The screenshot shows a terminal window with a server process running in the background and several client processes running in the foreground. The server is listening on port 5550. The clients are connected to the server and are sending data. The output shows the server processing each client's data separately, without interference from other clients.

```
andre@andre-Aspire-E5-571:~/Documentos/UFGM/Redes/TP3$ ./servidor 5550
cliente-1: 1
cliente-1: 1
cliente-1: 3
cliente-2: 1
cliente-2: 1
cliente-3: 1
cliente-1: 1
cliente-1: 2
cliente-1: 3
cliente-1: 4
cliente-1: 5
cliente-2: 1
cliente-2: 2
cliente-2: 3
cliente-3: 2
cliente-3: 3
cliente-3: 4
cliente-2: 4
cliente-1: 1
```

andre@andre-Aspire-E5-571:~/Documentos/UFGM/Redes/TP3\$./cliente ::1 5550

andre@andre-Aspire-E5-571:~/Documentos/UFGM/Redes/TP3\$./cliente ::1 5550

andre@andre-Aspire-E5-571:~/Documentos/UFGM/Redes/TP3\$./cliente ::1 5550

andre@andre-Aspire-E5-571:~/Documentos/UFGM/Redes/TP3\$./cliente ::1 5550

Pelos testes é possível perceber o tratamento em separado pelo servidor a todos os clientes que se conectarem, os dados de cada cliente são separados dos demais e não interferem uns nos dos outros.

6 Conclusão

Os programas foram implementados sem maiores dificuldades levando-se em conta que o material disponível em livros e também na *internet* é suficiente para desenvolver programas de qualidade que atenda as necessidades da disciplina.

Referências Bibliográficas

Livro *Redes de computadores – uma abordagem de sistemas* – Peterson, Davie. 5ª Edição
<http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html>