

VINÍCIUS ALVES DE FARIA RESENDE - 2021039972

TRABALHO PRÁTICO 2

Redes de Computadores

1. Introdução

Esta documentação é referente ao segundo trabalho prático da disciplina de Redes de Computadores, e será dividida em 4 sessões, sendo elas introdução, implementação, testes e conclusão.

O trabalho foi desenvolvido com o propósito de ser uma experiência prática com a programação com sockets utilizando a conexão UDP, de forma que os conhecimentos teóricos adquiridos nas aulas expositivas, pudessem ser colocados em perspectiva. O trabalho se resume no desenvolvimento de um serviço simples similar ao da Netflix, onde um cliente realiza uma solicitação de um dos filmes do catálogo a um servidor, que então retornará, de forma cadenciada, as frases do filme. Então, as frases do filme serão exibidas para o cliente com recorrência de 3 segundos, dando a opção de seleção de um novo filme quando o atual acabar, o cliente também pode escolher por terminar a conexão. Também é válido mencionar que o Servidor irá manter um contato de clientes conectados sendo exibido pelo lado do servidor, a exibição será atualizada a cada 4 segundos.

A implementação foi feita na linguagem C, utilizando das bibliotecas de socket que o sistema Linux disponibiliza, seguindo a implementação do material fornecido pelo professor, o livro *TCP/IP Sockets in C: Practical Guide for Programmers*. O código está disponível em conjunto com esta documentação e, por si só, complementa-a, uma vez que o código também possui documentação própria detalhando seu funcionamento.

2. Implementação

2.1. Servidor

Ao falar do servidor implementado, a primeira coisa necessária é descrever a interface de inicialização deste. A interface foi dada no enunciado do trabalho, e foi implementada de acordo, sendo que o servidor recebe o tipo de protocolo IP a ser utilizado (IPv4 ou IPv6) e, como segundo parâmetro, a porta a ser inicializada. Portanto, a primeira parte da implementação verifica se o número de parâmetros passado condiz com os parâmetros esperados, caso contrário, a execução é interrompida e uma mensagem instruindo o usuário dos parâmetros e sua ordem é exibida. Passando dessa etapa, os parâmetros são lidos e armazenados, sendo que o tipo de IP é traduzido para um código inteiro que dispensa a necessidade de armazenar uma string, e a porta é convertida e armazenada como uma string.

Tendo os parâmetros iniciais bem definidos e armazenados, o sistema então prossegue para a criação de um endereço de servidor, sendo que este é do tipo *addrinfo*. O servidor então, primeiro realiza a criação de um *Address Criteria*, que possui as informações de configuração

do endereço, posteriormente constroi o endereço usando as informações fornecidas usando o seguinte código:

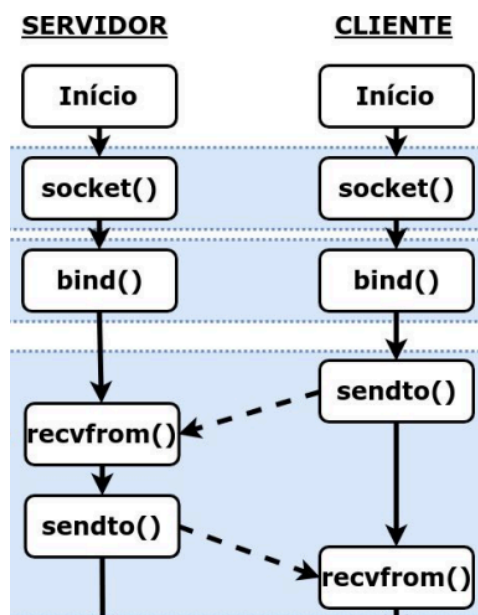
```
// Construct the server address structure
struct addrinfo addrCriteria; // Criteria for address
memset(&addrCriteria, 0, sizeof(addrCriteria)); // Zero out structure
addrCriteria.ai_family = (ipType == IPV4_CODE) ? AF_INET : AF_INET6;
addrCriteria.ai_flags = AI_PASSIVE; // Accept on any address/port
addrCriteria.ai_socktype = SOCK_DGRAM; // Only datagram socket
addrCriteria.ai_protocol = IPPROTO_UDP; // Only UDP socket

struct addrinfo *serverAddr; // Holder for returned server addr
int rtnVal = getaddrinfo(NULL, servPort, &addrCriteria, &serverAddr);
if (rtnVal != 0)
    exitWithSystemMessage("getaddrinfo() failed");
```

Podemos ver que, existe uma configuração condicional para o valor *ai_family*, que depende do tipo de IP escolhido ao iniciar o servidor, sendo configurado como *AF_INET* em caso de IPV4 e como *AF_INET6* em caso de IPV6.

A função *getaddrinfo* é utilizada para associar os critérios do endereço ao servidor, esta função é muito conveniente pois ela retorna tanto o endereço IPV4 e IPV6 e utiliza o primeiro que funciona, como está descrito na seção 3.2.1 do livro, isso a torna muito versátil.

Tendo a estrutura do endereço do servidor bem definida, o código então parte para a criação do *socket* propriamente dito, sendo este o primeiro passo do lado do servidor descrito no diagrama fornecido na disciplina:



Para tal o seguinte código é utilizado:

```
// Create socket for incoming connections
int serverSock = socket(serverAddr->ai_family, serverAddr->ai_socktype,
serverAddr->ai_protocol);
if (serverSock < 0) {
    exitWithSystemMessage("socket() failed");
}
```

Neste trecho de código, o novo socket é criado, o primeiro parâmetro é o domínio do socket, onde já foi configurado o tipo de IP fornecido pelo usuário, então atribui-se o valor correto. Então preenchemos o segundo parâmetro que é o domínio do socket como *SOCK_DGRAM*, o que configura uma transmissão de dados como sendo exclusivamente por datagramas. Por fim, o último parâmetro define o protocolo de comunicação, e nesse caso estamos configurando para a utilização do UDP. Verifica-se a existência de erro (retorna -1), se não, atribuí o identificador do socket à variável *serverSock*.

A próxima etapa é o **bind**, que é feita a partir do seguinte código:

```
// Bind to the local address
if (bind(serverSock, serverAddr->ai_addr, serverAddr->ai_addrlen) <
0) {
    exitWithSystemMessage("bind() failed");
}
```

Neste trecho, realizamos o **bind**, do endereço e da porta. O socket do servidor precisa ser associado com o endereço local e com a porta, e é isso que o método **bind** faz, recebendo o identificador do socket recém criado como primeiro parâmetro, a estrutura de configuração do endereço do servidor como segundo parâmetro e o tamanho da estrutura como terceiro. Também é feita uma verificação de erros para o **bind**, para o caso de uma porta já estar sendo utilizada, ou a porta passada requerer permissões especiais, por exemplo.

As próximas etapas em uma conexão normalmente se caracterizam pelo **listen** e pelo **accept**, contudo estamos lidando com um protocolo sem conexão, portanto não se faz necessário. Antes de prosseguir para a próxima etapa, é válido mencionar a necessidade da chamada da função *freeaddrinfo* conforme é mencionado na seção 3.1.1 do livro, de forma a evitar problemas com *memory leak*.

Antes de seguir para a comunicação com o cliente, existe um requerimento do trabalho para que seja exibido, a cada 4 segundos, os clientes conectados. Isto deve ser feito mesmo se algum filme estiver sendo transmitido no momento. Portanto, faz-se necessária a criação de uma *thread* especializada nesta tarefa, a qual realizará a exibição do contador de clientes. Este

contador é uma variável global, então, requer que um *mutex* seja implementado de forma que esta variável possa ser alterada por *threads* diferentes sem o risco de valores conflitarem.

```
// Handles exhibition of connected clients
pthread_mutex_init(&mutex, NULL);

pthread_t printThread;
int ret = pthread_create(&printThread, NULL, printClients, NULL);
if(ret != 0) perror("pthread_create() failed");
```

O código acima cria a nova *thread* ao mesmo tempo que inicializa o *mutex*, a função *printClients* é associada à *thread* e é uma função simples com um *loop* infinito que trava o *mutex* exibe o contato de clientes com um comando *printf* e então libera o *mutex*, após estes passos a *thread* dorme por 4 segundos e executa novamente as mesmas ações.

Agora, de fato, podemos prosseguir para a conexão com os clientes que será feita em um loop infinito que lidará com os vários clientes do servidor. A primeira parte é a criação de uma estrutura que irá armazenar informações relevantes para o tratamento de uma conexão, como o socket usado, o filme selecionado e o endereço do socket. Inicialmente o espaço para armazenar as informações é alocado mas ainda não preenchido: `};`

```
struct handlerData *connectionData = malloc(sizeof(struct handlerData)
if (connectionData == NULL) {
    perror("ERROR: malloc() failed");
    continue;
}
connectionData->serverSocket = serverSock;

// Set length of client address structure (in-out parameter)
socklen_t clntAddrLen = sizeof(connectionData->sockaddr);
```

Pelo fato de estarmos lidando com um protocolo UDP, o próximo passo já se caracteriza por uma chamada à função *recvfrom*, que faz com que o servidor fique aguardando algum dado por parte de um cliente. Neste momento, as informações necessárias para preencher a variável *connectionData* são fornecidas.

```
char buffer[MESSAGE_SIZE]; // I/O buffer
ssize_t numBytesRcvd = recvfrom(
    serverSock,
    buffer,
    MESSAGE_SIZE,
    NO_FLAGS,
    (struct sockaddr *) &(connectionData->sockaddr),
    &clntAddrLen
```

```

);
if (numBytesRcvd < 0) {
    free(connectionData);
    perror("recvfrom() failed");
}

connectionData->selectedMovie = atoi(buffer);
pthread_mutex_lock(&mutex);
connectedClients++;
pthread_mutex_unlock(&mutex);

```

Nesta etapa já é possível obter as informações do cliente, como o seu endereço e também o buffer é preenchido com a escolha do filme realizada pelo cliente, este valor é adicionado à estrutura do *connectionData*. A mensagem recebida é sempre uma string, por isso foi necessária a utilização do método *atoi* para converter o filme selecionado para um inteiro. Note que, ao final é realizado um travamento e destravamento do *mutex* para realizar a alteração da variável global *connectedClients* que é responsável por manter o contador de clientes conectados atualizado no lado do servidor.

Tendo então o filme selecionado pelo cliente salvo, prosseguimos para a parte onde de fato lidamos com o cliente UDP, como precisamos lidar com vários clientes simultâneos, também é feita uma *thread* para lidar com a requisição em específico:

```

pthread_t thread;
int ret = pthread_create(&thread, NULL, handleUDPClient, (void *)connectionData);
if (ret != 0) {
    perror("pthread_create() failed");
    free(connectionData);
    continue;
}

```

A partir daí, o loop infinito do servidor é terminado, o que significa que toda responsabilidade de lidar com a conexão do cliente é passada agora para a função auxiliar *handlerUDPClient*, que será descrita em seguida.

A primeira parte da função *handleUDPClient* é responsável por verificar que o filme escolhido é de fato uma opção válida, exibindo uma mensagem informativa e finalizando a tratativa em caso de discordância com o padrão esperado.

```

struct handlerData *connectionData = (struct handlerData *)arg;
if (connectionData->selectedMovie <= 0 || connectionData->selectedMovie > MOVIE_OPTIONS) {

```

```

printf("Invalid movie option\n");
return 0;
}

```

No caso do filme escolhido de fato ser válido, a função precisa acessar o conteúdo do filme, que fica armazenado em um arquivo `.txt` dentro da pasta `./movies`, para encontrar o arquivo para o filme selecionado, existe uma função auxiliar `openMovieFile` que recebe o filme selecionado e retorna o ponteiro para o arquivo aberto:

```

FILE *file = openMovieFile(connectionData->selectedMovie);
if (file == NULL) {
    perror("Failed to open file");
    return 0;
}

```

A função `openMovieFile` possui o seguinte formato:

```

FILE* openMovieFile(int selectedMovie) {
    switch (selectedMovie) {
        case LORD_OF_THE_RINGS:
            return fopen("movies/lordOfTheRings.txt", "r");
        ...
    }
}

```

Neste momento, o servidor está pronto para enviar os dados ao cliente, e pode iniciar um `loop` iterativo que enviará cada uma das linhas do arquivo ao cliente, começamos primeiro lendo o conteúdo da linha do arquivo e atribuindo a um `buffer`:

```

int phrasesRead = 0;
char buffer[MESSAGE_SIZE];
while (phrasesRead < MOVIE_SIZE) {
    if (fgets(buffer, MESSAGE_SIZE, file) == NULL) {
        perror("Failed to read line from file");
        fclose(file);
        return 0;
    }
    phrasesRead++;
    ...
}

```

Então temos um `loop` que será limitado à quantidade de frases de cada filme, neste caso cinco frases, então para cada linha do arquivo utilizamos a função `fgets` para ler toda a linha e atribuí-la à variável `buffer`, além disso, após a leitura de cada frase o contador de frases lidas é incrementado.

Tendo a frase atual armazenada no buffer, basta enviá-la ao cliente por meio do socket, isso é feito usando o comando *sendto* com os seguintes parâmetros:

```
...
ssize_t numBytesSent = sendto(
    connectionData->serverSocket,
    buffer,
    strlen(buffer),
    NO_FLAGS,
    (struct sockaddr*) &(connectionData->sockaddr),
    sizeof(connectionData->sockaddr)
);
if (numBytesSent < 0) perror("sendto() failed");

sleep(3);
...
```

Desta forma, a frase lida será enviada ao cliente, e, após isso, podemos notar a chamada da função *sleep* que irá fazer a *thread* dormir até reiniciar a iteração do loop que enviará a próxima frase, isso será feito até que todas as frases sejam enviadas e o loop termine.

Após o envio de todas as frases do filme, prosseguimos então para a finalização, que se caracteriza pelo fechamento do arquivo contendo o conteúdo do filme e o envio da mensagem de finalização ao cliente, que é contida em um macro compartilhado entre cliente e servidor com o nome de *MOVIE_END_MESSAGE*:

```
fclose(file);

// Send the end of movie message
ssize_t numBytesSent = sendto(
    connectionData->serverSocket,
    MOVIE_END_MESSAGE,
    strlen(MOVIE_END_MESSAGE),
    NO_FLAGS,
    (struct sockaddr*) &(connectionData->sockaddr),
    sizeof(connectionData->sockaddr)
);
if (numBytesSent < 0) perror("sendto() failed");

pthread_mutex_lock(&mutex);
connectedClients--;
pthread_mutex_unlock(&mutex);

return 0;
```


Além do que foi descrito, podemos notar também que mais uma vez travamos o mutex e destravamos para fazer a alteração da variável global *connectedClients* que registra o número de clientes conectados ao servidor, desta vez decrementando o valor, representando o término de uma “conexão” com cliente.

Desta forma está finalizada toda a iteração de tratamento de uma requisição por parte do servidor, que irá voltar à execução do *loop* infinito principal esperando por próximas requisições.

2.2. Cliente

Ao falar do cliente implementado, a primeira coisa necessária é descrever a interface de inicialização deste. A interface foi dada no enunciado do trabalho, e foi implementada de acordo, sendo que o cliente recebe o tipo de protocolo IP a ser utilizado (IPv4 ou IPv6) o endereço IP do servidor no formato respectivo ao protocolo IP utilizado e, como terceiro parâmetro, a porta a qual conectar. Portanto, a primeira parte da implementação verifica se o número de parâmetros passado condiz com os parâmetros esperados, caso contrário, a execução é interrompida e uma mensagem instruindo o usuário dos parâmetros e sua ordem é exibida. Passando dessa etapa, os parâmetros são lidos e armazenados, sendo que o tipo de IP é traduzido para um código inteiro que dispensa a necessidade de armazenar uma *string*, e a porta é armazenada como uma *string*, o endereço IP também permanece como uma *string*.

Após esta etapa, a função *main* faz uma chamada à função *handleUDPServer*, passando os parâmetros do programa para a função (*ipFamily*, *servPort*, *address*). A partir daí, é necessária uma etapa de construção do endereço do servidor, em um processo muito similar ao que foi descrito anteriormente, onde existe a configuração do *addrCriteria* com o código abaixo:

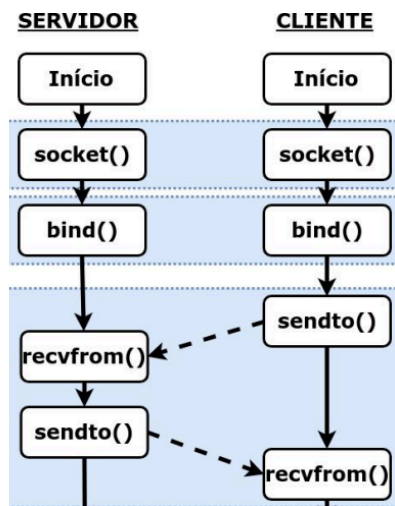
```
// Tell the system what kind(s) of address info we want
struct addrinfo addrCriteria; // Criteria for address
memset(&addrCriteria, 0, sizeof(addrCriteria)); // Zero out structure
addrCriteria.ai_family = (ipFamily == IPV4_CODE) ? AF_INET : AF_INET6;
addrCriteria.ai_socktype = SOCK_DGRAM; // Only datagram socket
addrCriteria.ai_protocol = IPPROTO_UDP; // Only UDP socket
```

Então, mais uma vez, com o auxílio da função *getaddrinfo* já mencionada é feita a construção do endereço do servidor que será essencial para o estabelecimento do socket UDP.

```
// Get address(es)
struct addrinfo *servAddr; // Holder for returned list of server addrs
int rtnVal = getaddrinfo(address, servPort, &addrCriteria, &servAddr);
if (rtnVal != 0) {
    perror("getaddrinfo() failed");
}
```

Nesse momento, começa-se um *loop* que será executado até que o usuário faça a seleção da opção de sair do programa, para tal, no início da iteração de cada *loop*, é exibida uma mensagem ao usuário dando a opção de sair do programa ou selecionar um dos filmes do catálogo. Caso seja digitado 0 o programa irá fechar sem nem criar o socket do cliente.

Caso contrário, caso o usuário selecione um dos filmes do catálogo, então a primeira etapa é a criação do socket do cliente, conforme também está enunciado no diagrama fornecido no material da disciplina, dessa vez, sob a perspectiva do cliente.



Essa etapa é feita com o seguinte código:

```
// Create a datagram/UDP socket
int clientSocket = socket(servAddr->ai_family, servAddr->ai_socktype,
servAddr->ai_protocol); // Socket descriptor for client
if (clientSocket < 0) exitWithSystemMessage("ERROR opening socket");
```

Neste trecho de código, o novo socket é criado, o primeiro parâmetro é o domínio do socket, onde o tipo de IP fornecido pelo usuário já está configurado no campo *ai_family* do endereço, então atribui-se o valor correto. Então o segundo parâmetro também já está armazenado, e é o domínio do socket (*IPPROTO_UDP*), o que configura uma transmissão de datagramas exclusivamente. Por fim, o último parâmetro define o protocolo de comunicação, e nesse caso estamos configurando para a utilização do UDP. Verifica-se a existência de erro (retorna -1), se não, atribui o identificador do socket à variável *clientSocket*. Perceba que a configuração é idêntica à do servidor, fazendo com que sejam compatíveis.

A partir deste ponto, já têm-se o socket do cliente criado e o endereço do servidor corretamente configurado com o endereço IP e a porta para conexão. Então, já tendo a opção do usuário quanto ao filme selecionado, basta enviar esta escolha ao servidor, isto é feito por

meio da atribuição do valor fornecido a uma string *selectedMovieStr* com o auxílio da função *sprintf*. Então o método *sendto* é chamado passando o socket do cliente como primeiro parâmetro, a mensagem construída como segundo parâmetro, o tamanho da mensagem como terceiro e não há nenhuma *flag* como quarto parâmetro. Verifica-se o tamanho da mensagem enviada em bits, e faz-se o tratamento de erro de acordo.

```
// Send the selected movie to the server
char selectedMovieStr[MESSAGE_SIZE];
sprintf(selectedMovieStr, "%d", selectedMovie);
ssize_t numBytesSent = sendto(
    clientSocket,
    selectedMovieStr,
    strlen(selectedMovieStr),
    NO_FLAGS,
    servAddr->ai_addr,
    servAddr->ai_addrlen
);
if (numBytesSent < 0) {
    exitWithSystemMessage("ERROR sending message to server");
} else if ((size_t)numBytesSent != strlen(selectedMovieStr)) {
    exitWithSystemMessage("ERROR sending unexpected number of bytes");
}
```

Após enviar a mensagem contendo a informação sobre o filme selecionado pelo usuário, é hora de esperar uma resposta do servidor. Onde será o caso de que o servidor mandará várias respostas consecutivas, então o cliente tem que estar preparado para lidar com um número indefinido de envios. Para tal, um *loop* infinito é criado e um *buffer* é inicializado para receber o conteúdo do servidor. O seguinte código lida com o recebimento da mensagem do servidor:

```
char buffer[MESSAGE_SIZE]; // I/O buffer
while (1) {
    // Receive message from server
    memset(buffer, 0, MESSAGE_SIZE);
    numBytesRcvd = recvfrom(
        clientSocket,
        buffer,
        MESSAGE_SIZE,
        NO_FLAGS,
        (struct sockaddr *) &fromAddr,
        &fromAddrLen
    );
    if (numBytesRcvd < 0) {
        exitWithSystemMessage("ERROR receiving message from server");
    }
}
```

Nesse momento, é chamada a função **recvfrom** que recebe o socket do cliente como primeiro parâmetro, como segundo o *buffer* para registrar a mensagem recebida e o tamanho do *buffer*, bem como o indicador de *flags* vazio. Deste modo, será lida a mensagem enviada pelo servidor e armazenada ao *buffer*, bem como será armazenado na variável *numBytes* o tamanho da mensagem recebida, que será utilizado para verificar possíveis erros no recebimento da mensagem. Recebida a mensagem, existem 2 possibilidades distintas, a primeira delas sendo que esta mensagem marca o término de um filme, nesse caso teremos o seguinte:

```
// Checks for end of movie transmission
if (strcmp(buffer, MOVIE_END_MESSAGE) == 0) {
    close(clientSocket);
    break;
} else { ... }
```

Para este cenário, será verificado que a mensagem enviada pelo servidor é igual à *MOVIE_END_MESSAGE*, indicando que o filme transmitido foi finalizado. Então, o socket criado é fechado e o *loop* infinito recebe um *break* e então a execução sai do loop.

Para o segundo e último caso, temos um cenário onde o *loop* infinito de fato recebe uma mensagem com uma frase do filme solicitado. Nesse caso temos o código abaixo:

```
// Checks for end of movie transmission
if (strcmp(buffer, MOVIE_END_MESSAGE) == 0) {
    ...
} else {
    // Prints received message if not end of the movie
    printf("%s", buffer);
}
```

Para esse cenário teremos então a frase sendo exibida para o usuário, nesse caso o *loop* não é encerrado, o que fará que, novamente, espere-se pela próxima mensagem, podendo ser uma nova frase do filme ou informando que o filme foi finalizado.

Em suma, essas são as duas operações possíveis para lidar com as respostas do servidor e assim finaliza-se a explicação da implementação do cliente do sistema.

3. Testes

Nesta seção, vamos explorar exemplos práticos para entender o funcionamento do programa, observando a interação entre cliente e servidor. Através de cenários reais, vamos mostrar como executar o programa, fornecer os parâmetros necessários e acompanhar a comunicação entre cliente e servidor durante a solicitação de um filme. Cada exemplo será acompanhado por capturas de tela destacando as etapas principais da execução, oferecendo uma compreensão visual do processo. Esses exemplos serão úteis para os usuários compreenderem melhor como utilizar e interagir com a aplicação em diferentes situações.

3.1. Um cliente solicita um filme arbitrário que é prontamente fornecido.

Para este teste, teremos o servidor configurado para receber conexões IPV4 na porta 50502. Nesse cenário, o cliente receberá o endereço do servidor (endereço local: 127.0.0.1), a porta 50502 e o tipo IPV4.

Primeiramente, o usuário escolhe um filme arbitrário, enviando então sua escolha para o servidor. Então, o servidor recebe essa informação e envia as mensagens sequenciais com cada frase do filme selecionado. Por fim, o servidor envia a mensagem informando o término do filme e o programa pode exibir o catálogo de filmes novamente ao lado do cliente, e o servidor se prepara para uma nova solicitação. Vale observar que o número de clientes conectados se mantém atualizado.

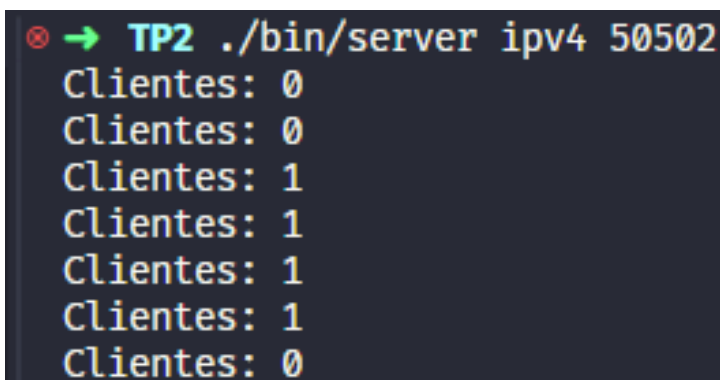
A terminal window with a dark background. The prompt is 'TP2' followed by a green arrow icon. The command entered is './bin/server ipv4 50502'. Below the command, the output 'Clientes: 0' is shown multiple times, indicating the number of connected clients. The sequence of outputs is: 'Clientes: 0', 'Clientes: 0', 'Clientes: 1', 'Clientes: 1', 'Clientes: 1', 'Clientes: 1', and finally 'Clientes: 0'.

Figura 1 - Teste 3.1 Servidor

```
• → TP2 ./bin/client ipv4 127.0.0.1 50502

0 - Sair
1 - Senhor dos anéis
2 - O Poderoso Chefão
3 - Clube da Luta

1
Um anel para a todos governar
Na terra de Mordor onde as sombras se deitam
Não é o que temos, mas o que fazemos com o que temos
Não há mal que sempre dure
O mundo está mudando, senhor Frodo

0 - Sair
1 - Senhor dos anéis
2 - O Poderoso Chefão
3 - Clube da Luta

0
```

Figura 2 - Teste 3.1 Cliente

3.2. Dois Clientes solicitam filmes distintos simultaneamente.

Neste teste, teremos o servidor configurado para receber conexões IPV6 na porta 50501. Nesse cenário, o cliente receberá o endereço do servidor (endereço local: ::1), a porta 50501 e o tipo IPV6.

Primeiramente, cada um dos dois usuários escolhe um filme distinto, enviando então suas escolhas para o servidor. Então, o servidor recebe essas informações e envia as mensagens sequenciais com cada frase dos filmes selecionados individualmente ao mesmo tempo, sem o risco de interferência entre o conteúdo de cada usuário. Por fim, o servidor envia a mensagem informando o término de cada filme e o programa pode exibir o catálogo de filmes novamente ao lado dos clientes, e o servidor se prepara para uma nova solicitação. Vale observar que o número de clientes conectados se mantém atualizado.

```
⊗ → TP2 ./bin/server ipv6 50501
Clientes: 0
Clientes: 0
Clientes: 2
Clientes: 2
Clientes: 2
Clientes: 2
Clientes: 0
```

Figura 3 - Teste 3.2 Servidor

```
TP2 ./bin/client ipv6 ::1 50501

0 - Sair
1 - Senhor dos anéis
2 - O Poderoso Chefão
3 - Clube da Luta

1
Um anel para a todos governar
Na terra de Mordor onde as sombras se deitam
Não é o que temos, mas o que fazemos com o que temos
Não há mal que sempre dure
O mundo está mudando, senhor Frodo

0 - Sair
1 - Senhor dos anéis
2 - O Poderoso Chefão
3 - Clube da Luta

0

TP2 ./bin/client ipv6 ::1 50501

0 - Sair
1 - Senhor dos anéis
2 - O Poderoso Chefão
3 - Clube da Luta

2
Vou fazer uma oferta que ele não pode recusar
Mantenha seus amigos por perto e seus inimigos mais perto ainda
É melhor ser temido que amado
A vingança é um prato que se come frio
Nunca deixe que ninguém saiba o que você está pensando

0 - Sair
1 - Senhor dos anéis
2 - O Poderoso Chefão
3 - Clube da Luta

0
```

Figura 4 - Teste 3.2 Cliente

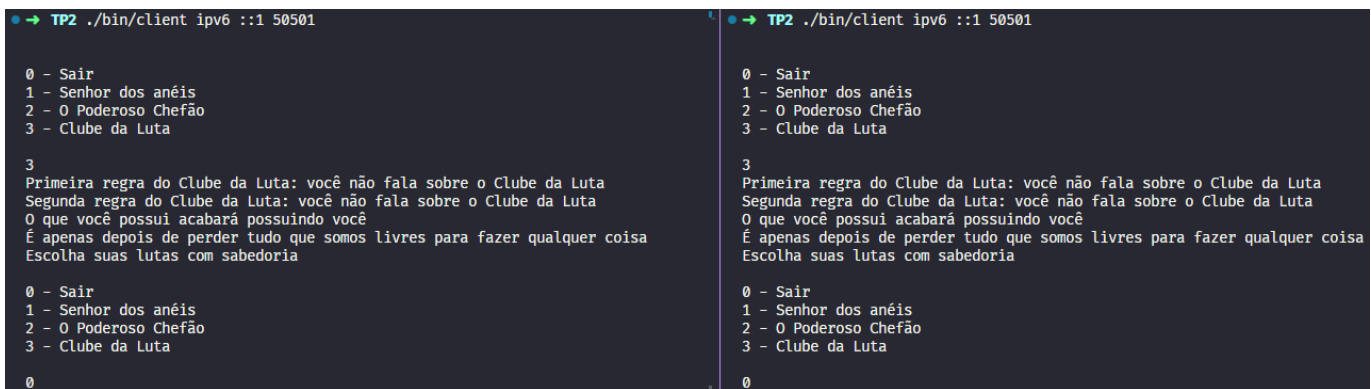
3.3. Dois Clientes solicitam filmes idênticos simultaneamente.

Neste teste, teremos o servidor configurado para receber conexões IPV6 na porta 50501. Nesse cenário, o cliente receberá o endereço do servidor (endereço local: ::1), a porta 50501 e o tipo IPV6.

Primeiramente, cada um dos dois usuários escolhem o mesmo filme, enviando então suas escolhas para o servidor. Então, o servidor recebe essas informações e envia as mensagens sequenciais com cada frase do filme selecionado individualmente ao mesmo tempo, sem o risco de interferência entre o conteúdo de cada usuário. Por fim, o servidor envia a mensagem informando o término de cada filme e o programa pode exibir o catálogo de filmes novamente ao lado dos clientes, e o servidor se prepara para uma nova solicitação. Vale observar que o número de clientes conectados se mantém atualizado.

```
TP2 ./bin/server ipv6 50501
Clientes: 0
Clientes: 0
Clientes: 1
Clientes: 2
Clientes: 2
Clientes: 2
Clientes: 0
```

Figura 5 - Teste 3.3 Servidor



The image shows two side-by-side terminal windows. Both windows have a title bar that says 'TP2' and a command prompt showing the command './bin/client ipv6 ::1 50501'. The output in both windows is identical. It starts with a menu: '0 - Sair', '1 - Senhor dos anéis', '2 - O Poderoso Chefão', '3 - Clube da Luta'. The user enters '3'. Then, two lines of text are displayed: 'Primeira regra do Clube da Luta: você não fala sobre o Clube da Luta' and 'Segunda regra do Clube da Luta: você não fala sobre o Clube da Luta'. This is followed by a paragraph: 'O que você possui acabará possuindo você. É apenas depois de perder tudo que somos livres para fazer qualquer coisa. Escolha suas lutas com sabedoria'. Then, the menu is shown again, and the user enters '0'.

```
TP2 ./bin/client ipv6 ::1 50501

0 - Sair
1 - Senhor dos anéis
2 - O Poderoso Chefão
3 - Clube da Luta

3
Primeira regra do Clube da Luta: você não fala sobre o Clube da Luta
Segunda regra do Clube da Luta: você não fala sobre o Clube da Luta
O que você possui acabará possuindo você
É apenas depois de perder tudo que somos livres para fazer qualquer coisa
Escolha suas lutas com sabedoria

0 - Sair
1 - Senhor dos anéis
2 - O Poderoso Chefão
3 - Clube da Luta

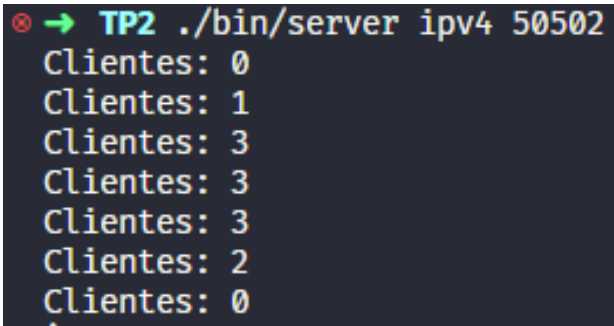
0
```

Figura 6 - Teste 3.3 Cliente

3.4. Três Clientes solicitam filmes arbitrários simultaneamente.

Para este teste, teremos o servidor configurado para receber conexões IPV4 na porta 50502. Nesse cenário, o cliente receberá o endereço do servidor (endereço local: 127.0.0.1), a porta 50502 e o tipo IPV4.

Primeiramente, cada um dos três usuários escolhem seus filmes, sendo dois iguais e um diferente, enviando então suas escolhas para o servidor. Então, o servidor recebe essas informações e envia as mensagens sequenciais com cada frase de cada um dos filmes selecionados individualmente ao mesmo tempo, sem o risco de interferência entre o conteúdo de cada usuário. Por fim, o servidor envia a mensagem informando o término de cada filme e o programa pode exibir o catálogo de filmes novamente ao lado dos clientes, e o servidor se prepara para uma nova solicitação. Vale observar que o número de clientes conectados se mantém atualizado.



The image shows a terminal window with a title bar that says 'TP2' and a command prompt showing the command './bin/server ipv4 50502'. The output shows the number of clients connected, with the following sequence: 'Clientes: 0', 'Clientes: 1', 'Clientes: 3', 'Clientes: 3', 'Clientes: 3', 'Clientes: 2', 'Clientes: 0'.

```
TP2 ./bin/server ipv4 50502
Clientes: 0
Clientes: 1
Clientes: 3
Clientes: 3
Clientes: 3
Clientes: 2
Clientes: 0
```

Figura 7 - Teste 3.4 Servidor


```
TP2 ./bin/client ipv4 127.0.0.1 50502
0 - Sair
1 - Senhor dos anéis
2 - O Poderoso Chefão
3 - Clube da Luta

1
Um anel para a todos governar
Na terra de Mordor onde as sombras se deitam
Não é o que temos, mas o que fazemos com o que temos
Não há mal que sempre dure
O mundo está mudando, senhor Frodo

0 - Sair
1 - Senhor dos anéis
2 - O Poderoso Chefão
3 - Clube da Luta

0
TP2

TP2 ./bin/client ipv4 127.0.0.1 50502
0 - Sair
1 - Senhor dos anéis
2 - O Poderoso Chefão
3 - Clube da Luta

3
Primeira regra do Clube da Luta: você não fala sobre o
Clube da Luta
Segunda regra do Clube da Luta: você não fala sobre o
Clube da Luta
O que você possui acabará possuindo você
É apenas depois de perder tudo que somos livres para f
azer qualquer coisa
Escolha suas lutas com sabedoria

0 - Sair
1 - Senhor dos anéis
2 - O Poderoso Chefão
3 - Clube da Luta

0
TP2

TP2 ./bin/client ipv4 127.0.0.1 50502
0 - Sair
1 - Senhor dos anéis
2 - O Poderoso Chefão
3 - Clube da Luta

3
Primeira regra do Clube da Luta: você não fala sobre
o Clube da Luta
Segunda regra do Clube da Luta: você não fala sobre o
Clube da Luta
O que você possui acabará possuindo você
É apenas depois de perder tudo que somos livres para
fazer qualquer coisa
Escolha suas lutas com sabedoria

0 - Sair
1 - Senhor dos anéis
2 - O Poderoso Chefão
3 - Clube da Luta

0
TP2
```

Figura 8 - Teste 3.4 Cliente

4. Conclusão

O trabalho prático mostrou-se como uma ótima oportunidade para ter contato com os fundamentos da comunicação servidor-cliente no contexto de redes de computadores, pudemos utilizar *sockets* nativos do sistema operacional Linux por meio da linguagem C para implementar um servidor que utiliza comunicação UDP(Transporte) e IP(Rede). O tema do trabalho é super interessante, fazendo uma simulação do sistema Netflix, o que habilitou o entendimento de vários aspectos fundamentais na troca de mensagem, especialmente pela dificuldade em sincronizar a solicitação de vários clientes simultâneos. Para lidar com tudo isso houveram grandes desafios incluindo a configuração de *sockets*, tratamentos de erros, criação de *threads*, definições de contratos entre outros.

O projeto foi também uma boa oportunidade para ter contato com as diferenças entre a utilização do endereçamento IPV4 e IPV6, propiciando uma experiência que enriqueceu a compreensão sobre o assunto, requerendo uma configuração dinâmica e confiável. Foi possível ver na prática também a característica das mensagens de datagramas sem conexão do protocolo UDP, garantindo a dinamicidade da comunicação.

Em síntese, considero que tive uma ótima experiência ao desenvolver o trabalho, pude ter uma oportunidade de vivenciar na prática vários dos conceitos estudados na disciplina. Considero que estou mais preparado para lidar com os problemas que posso enfrentar lidando com redes de computadores no dia a dia, bem como me deu confiança para explorar novos horizontes no que se refere à sistemas distribuídos e redes de computadores.