Exercício 7: Serviço de Bate-Papo

MC833AB - Programação de Redes de Computadores

Universidade Estadual de Campinas (Unicamp)
Matheus Jun Ota - RA 138889
Gustavo de Mello Crivelli - RA 136008

1 Introdução

Para o projeto final, foram desenvolvidos um cliente e um servidor para um serviço de chat, com os seguintes requisitos:

- A linguagem de programação usada para desenvolver este trabalho deverá ser excluisivamente C ou C++;
- Há um código correspondente ao servidor e um código correspondente ao cliente;
- Os clientes precisam se conectar no servidor e toda a conversa do bate-papo deve passar pelo servidor. Ou seja, cliente 1 conversa com o servidor e o servidor manda a string para o cliente 2. Essa conversa é feita via protocolo UDP;
- A transferência de arquivo texto deve acontecer de cliente para cliente sem passar pelo servidor. Essa transferência é feita via protocolo TCP;
- Um cliente, ao conectar, deve ter acesso a uma lista de usuários já conectados e escolher com quem quer conversar;
- O cliente deve informar seu nickname;
- O cliente deve poder solicitar a desconexão do servidor;
- O servidor deve ser concorrente;
- O cliente deve acessar o servidor tanto pelo nome quanto pelo endereço IP passados na linha de comando;
- O servidor deve manter um log de eventos (conexão de usuários, envío de mensagens, etc).

2 Cliente

2.1 Detalhes de implementação

Primeiramente, foram construídas funções *wrapper* para encapsular os métodos relacionados à rede: criação de sockets, *bind*, *accept*, *listen*, *close*, *select*, recebimento e envio de pacotes. Por exemplo:

```
void Bind(int sockfd, const struct sockaddr *addr, socklen_t len){
    if (bind(sockfd, addr, len) == -1) {
        perror("bind");
        exit(1);
    }
}

void Listen(int sockfd, int backlog){
    if (listen(sockfd, backlog) == -1) {
        perror("listen");
        exit(1);
    }
}
```

Além disso, implementamos algumas funções auxiliares para que o servidor possa ser especificado de três maneiras diferentes: pelo nome do processo que roda o servidor no localhost; pelo nome do servidor remoto (ex.: ssh.students.ic.unicamp.br), caso no qual é necessário realizar uma consulta DNS para resolver o endereço do servidor; ou por fim, pelo endereço IP do servidor.

O Cliente se conecta ao servidor por passagem de argumentos em sua chamada, na forma:

```
./cliente [IP_servidor | nome_servidor] [porta_servidor]
```

Caso o servidor seja encontrado, é iniciado um "aperto de mão", entre o cliente e o servidor, realizado por meio da porta pública do servidor, e através do qual é informado ao cliente qual será a porta dedicada do servidor que deverá ser usada nas comunicações futuras. Notemos que utilizamos o protocolo UDP para esse processo, e como o protocolo UDP não é orientado à conexão, não há necessidade de "handshaking". Entretanto, nesse caso o "handshaking"está sendo utilizado para que o servidor inicie uma execução concorrente do processo, e não para estabelecer a conexão como no protocolo TCP.

2.2 Funcionalidades

Uma vez conectado, o cliente poderá enviar os seguintes comandos para o servidor:

- **list** - Lista todos os usuários conectados.
- **name [nome_do_usuario]** - Registra o nickname do cliente no servidor.

- **friend [nome_do_usuario]** - Inicia uma conversa com o usuário especificado.
- **exit** - Encerra conexão do cliente com o servidor.

Os comandos **list** e **exit** estão disponíveis para o cliente durante toda a execução. Por outro lado, o comando **name** está disponível somente no início da execução, e o comando **friend** está disponível no início ou caso o antigo "amigo"tenha se desconectado.

Uma vez conectado a outro usuário, os dois poderão se comunicar através de mensagens, que serão transmitidas passando sempre pelo servidor. Haverá também um novo comando disponível para usuários que estejam conversando com outra pessoa:

- **file [nome_do_arquivo]** - Envia o arquivo especificado para o outro usuário na conversa.

Este último comando inicia uma conexão TCP diretamente entre os dois clientes envolvidos, que será utilizada para a transferência de arquivo sem passar pelo servidor. Esta transferência é realizada em um *fork* nos clientes, de modo a ser não-bloqueante e permitindo que os usuários continuem a trocar mensagens durante o processo.

Seja Alice o usuário que irá enviar o arquivo e Bob o usuário que recebe. Para utilizarmos o protocolo TCP precisamos que ao Alice requisitar uma conexão com Bob, este já esteja em estado de escuta esperando uma conexão. Assim, é necessário estabelecer um "handshaking"entre os clientes para realizar a transferência de arquivo. Implementamos esse processo da seguinte forma:

1) Alice digita **file arquivo.txt**. O processo cliente de Alice então envia uma mensagem **file1** para o servidor.

2) O servidor recebe a mensagem **file1** e envia uma mensagem para Bob informando que este deve começar a escutar pela tranferência de arquivo.

3) Bob cria um socket TCP (na mesma porta que o socket UDP, pois as portas são independentes), envia uma mensagem **file2** para o servidor e bloqueia em uma chamada a **Accept** esperando pela conexão de Alice.

4) O servidor recebe a mensagem **file2** de Bob. Ele então envia uma mensagem para Alice começar a enviar o arquivo. Essa mensagem contém também o endereço e a porta em que Bob está esperando a conexão.

5) Alice cria um processo filho com o **fork** e estabelece a conexão diretamente com Bob.

6) Após estabelecer a conexão, Bob realiza um **fork** e entra em um loop que lê do socket as mensagens de Alice e escreve em um arquivo "output.txt".

7) Quando a transferência é concluída, os sockets são fechados e os processos nos forks são encerrados.

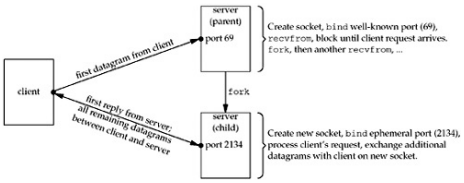

Figura 1: Arquitetura concorrente para servidores UDP.

3 Servidor

3.1 Arquitetura

Para a construção do servidor UDP concorrente, implementamos uma arquitetura na qual o servidor escuta iterativamente em uma porta pública, definida pelo usuário através dos parâmetros de execução do programa.

Toda vez que o socket do servidor (que escuta na porta pública) recebe uma mensagem de um cliente, é criada uma nova thread no servidor, responsável por processar as mensagens apenas daquele cliente. Para tal, é criado um novo socket dentro desta thread, com uma porta exclusiva para aquele cliente. O número da nova porta é enviado de volta ao cliente, completando um "aperto de mão", e todas as novas mensagens do cliente serão enviadas para esta porta. A Figura 1 representa a arquitetura descrita acima.

3.2 Detalhes de implementação

Assim como no cliente, foram escritas funções *wrapper* para lidar com os sockets, bem como funções e estruturas específicas para lidar com os usuários conectados.

Para cada cliente conectado, é preciso armazenar os dados do socket do usuário, seu nickname, e o amigo com quem ele está conversando (caso haja). Isto é feito através da estrutura **Client** abaixo.

```
typedef struct Client_aux{
char name[NAMESIZE];
struct sockaddr_in addr;
struct Client_aux *friend;
struct Client_aux *next;
struct Client_aux *prev;
} Client;
```

Além disso, a estrutura é uma lista ligada, e cada cliente armazena também um ponteiro para o cliente seguinte e um para o anterior.

Para auxiliar nas operações com a lista ligada de usuários, foram criadas funções com as assinaturas abaixo:

```
//cria uma lista ligada de clientes
void newClientList();
```

```
//insere um cliente novo na lista do chat
void insertClient(char *name, struct sockaddr_in *addr);

//encontra cliente pelo nome
Client* findClientByName(char *name);

//encontra cliente pelo endereco
Client* findClientByAddr(struct sockaddr_in addr);

//concatena em buf mensagem nomes dos clientes na list
void concatClientsList(char* buf);

//remove cliente da lista
void removeClient(Client *c);
```

O servidor detecta automaticamente o IP da máquina onde está sendo executado, de modo que o único argumento necessário é o número da porta pública a ser utilizada. A execução do programa é feita da forma:

```
./servidor [porta_servidor]
```

Como explicado na Subsecão 3.1, o servidor passa a escutar iterativamente na porta pública. Ao receber uma mensagem de algum cliente por essa porta, o servidor reserva a próxima porta ainda não utilizada para ficar dedicada àquele cliente, criando uma thread responsável por escutar nessa porta. A partir daí, o servidor processa e executa paralelamente as instruções enviadas pelos clientes, registrando as operações realizadas em um arquivo de log local.

4 Referências

[1] Kurose, James F., and Keith W. Ross. Computer networking: a top-down approach. Vol. 4. Boston, USA: Addison Wesley, 2009.

[2] Client to Client communication using select() function in c, https://stackoverflow.com/questions/38733264/client-to-client-communication-using-select-function-in-c

[3] Concurrent UDP Servers, http://www.masterraghu.com/subjects/np/introduction/unix_network_programming_v1.3/ch22lev1sec7.html

[4] Slides disponíveis em http://www.lrc.ic.unicamp.br/mc833/