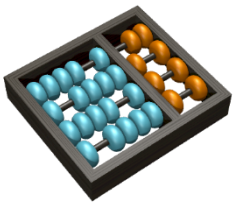
Universidade Estadual de Campinas
Instituto de Computação



Disciplina do 2º Semestre de 2022

IC - UNICAMP

Curso: Bacharelado em Ciência da Computação

MC833 - Laboratório de Redes de Computadores

Relatório do Trabalho Final

Alunos: Gabriel Volpato Giliotti **RA**: 197569
Pedro Barros Bastos **RA**: 204481

Professor: Nelson Luis Saldanha da Fonseca

Campinas – SP
2022

Este relatório tem como principal objetivo detalhar o processo de desenvolvimento do trabalho final da disciplina MC833, cuja tarefa foi construir uma aplicação cliente-servidor de chat simples em que o servidor é responsável pela gestão dos clientes disponíveis para chat.

- Execução dos programas
  - servidor:
    - ./servidor <Porta>
  - cliente:
    - ./cliente <IP Servidor> <Porta Servidor>

No processo do servidor, a lista de clientes conectados foi pensada em ser armazenada em uma lista ligada em que cada nó contém informações sobre cada cliente.

```c
typedef struct {
    int clientID; // ID único do client
    int connfd; // file descriptor da conexão TCP do cliente com o servidor
    char client_ip[16]; // endereço IP do cliente
    unsigned int client_socket_port; // porta da conexão TCP do cliente
    unsigned int client_udp_port; // porta UDP do cliente
    int in_chat; // flag para verificar se cliente está em chat ou disponível
} ClientInformation;

typedef struct ClientNode_struct {
    ClientInformation clientInformation;
    struct ClientNode_struct* nextNode;
} ClientNode;

typedef struct {
    ClientNode* head;
    ClientNode* tail;
    int size;
} ClientLinkedList;
```

Os IDs de cada cliente são definidos a partir da variável global inteira *referenceIDForClients,* sendo esta variável atribuída a cada novo cliente conectado ao servidor e posteriormente incrementada. A cada nova conexão de um cliente ao servidor uma instância de ClientInformation é criada e um nó ClientNode é adicionado à lista ligada de clientes ClientLinkedList.

A comunicação entre cliente e servidor se deu por seus respectivos TCP *file_descriptors* para diversos fins como a obtenção de porta UDP do cliente ao

se conectar, listagem de clientes disponíveis para conversa, listagem de comandos disponíveis, iniciação de chat com outro cliente e desconexão do servidor.

Para viabilizar a implementação da comunicação entre os clientes, pensamos em fazer com que cada cliente tenha seu próprio socket UDP ao invés de ter que determinar um cliente como o "servidor UDP" e outro como o "cliente UDP" usando um mesmo socket no momento de chat entre ambos. Assim, logo na inicialização de cada cliente é criado um socket UDP em uma porta definida pelo sistema operacional. Como o sistema operacional definiu a porta em questão, usou-se a função *getsockname* para saber qual porta foi atribuída para o socket UDP.

```
206  int main(int argc, char **argv) {
207      int socket_file_descriptor, maxfdp;
208      struct sockaddr_in udpaddr;
209      int udpSockFileDescriptor;
210      unsigned int currentUdpPort;
211
212      checkProgramInput(argc, argv);
213
214      socket_file_descriptor = conectToServer(argv[1], argv[2]);
215      udpSockFileDescriptor = initiateUDPSocket(&udpaddr);
216      currentUdpPort = retrieveCurrentUDPPort(udpSockFileDescriptor);
```

```
-bash-4.4$
-bash-4.4$ netstat -tupln | grep cliente
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
udp        0      0 0.0.0.0:46183           0.0.0.0:*                           1511832/./cliente
udp        0      0 0.0.0.0:46743           0.0.0.0:*                           1511911/./cliente
-bash-4.4$ netstat -tupln | grep servidor
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
tcp        0      0 0.0.0.0:44333           0.0.0.0:*               LISTEN      1511755/./servidor
-bash-4.4$
```

A comunicação entre servidor e cliente para o handshake de determinados fluxos se deu através de algumas keywords pré-definidas, além dos comandos de usuário que os clientes podem utilizar para interagir com o servidor e terminar um chat:

- Keywords transparentes para o usuário:

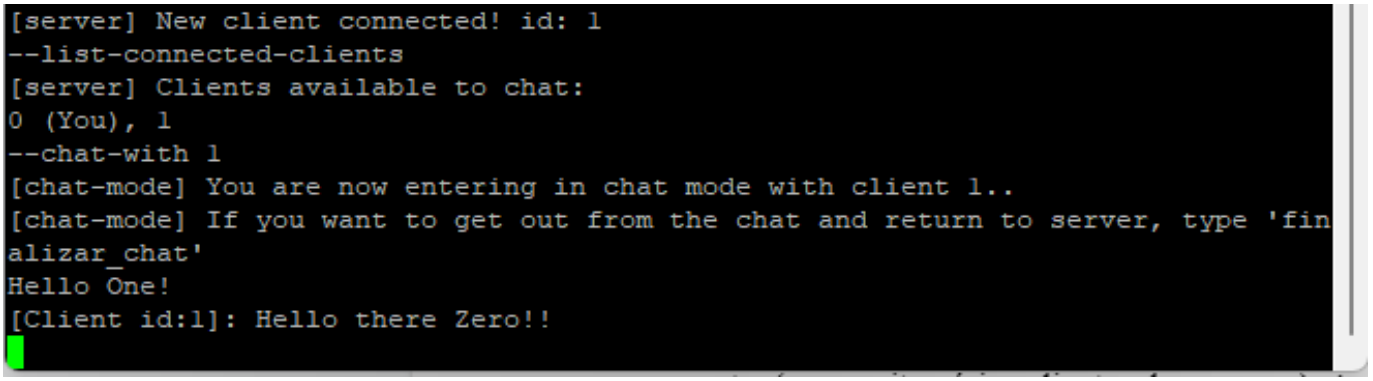
    - ***give_me_your_udp_port****:* mensagem que o servidor envia para o cliente requisitando sua porta UDP.
    - ***my_udp_port_is <udp_port>****:* mensagem que o cliente utiliza para enviar sua porta UDP para o servidor.
    - ***chat_init_with_client <client_id> <client_ip_address> <client_udp_port>****:* mensagem que o servidor envia para determinado cliente ao ser requisito por outro cliente o início de chat com o cliente em questão.
    - ***finished_chat_with_peer****:* mensagem que o cliente envia para o servidor indicando o fim de um chat.

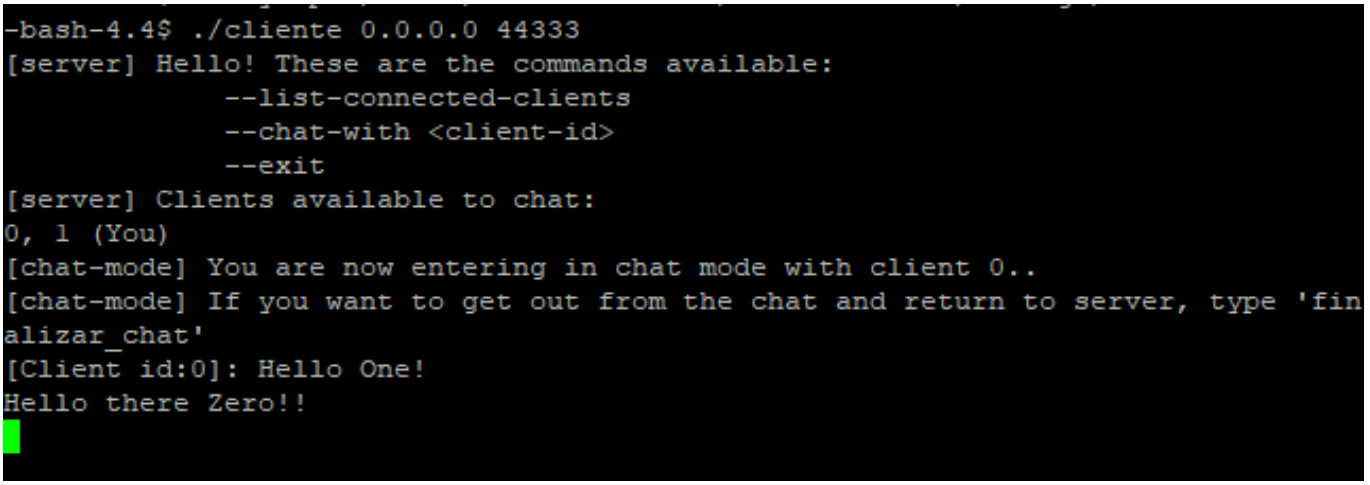
- Comandos que o usuário pode utilizar para interagir com o servidor:
    - ***--list-connected-clients***: comando para listar clientes conectados. Clientes em chat não aparecerão.
    - ***--chat-with <client-id>*:** comando para requisitar ao servidor o início de chat com o cliente de id <client-id>.
    - ***--exit:*** comando para se desconectar do servidor e encerrar cliente.

Assim que o servidor recebe a conexão de um novo cliente, é requisitado a este sua porta UDP pela mensagem ***give_me_your_udp_port.*** O cliente recebendo a mensagem com a referida keyword envia sua porta UDP utilizando a keyword ***my_udp_port_is <udp_port>.*** Quando o servidor receber a mensagem com a porta UDP do cliente, é mandada para o cliente a lista de comandos disponíveis para se executarem e a lista atual de clientes conectados. Também foi implementada a funcionalidade de mandar uma mensagem avisando a todos os clientes já conectados que um novo cliente foi conectado.

```
-bash-4.4$ ./cliente 44333
uso: ./cliente <Server-IPaddress> <Server-Port>: Success
-bash-4.4$ ./cliente 0.0.0.0 44333
[server] Hello! These are the commands available:
            --list-connected-clients
            --chat-with <client-id>
            --exit
[server] Clients available to chat:
0 (You)
[server] New client connected! id: 1
--list-connected-clients
[server] Clients available to chat:
0 (You), 1
```

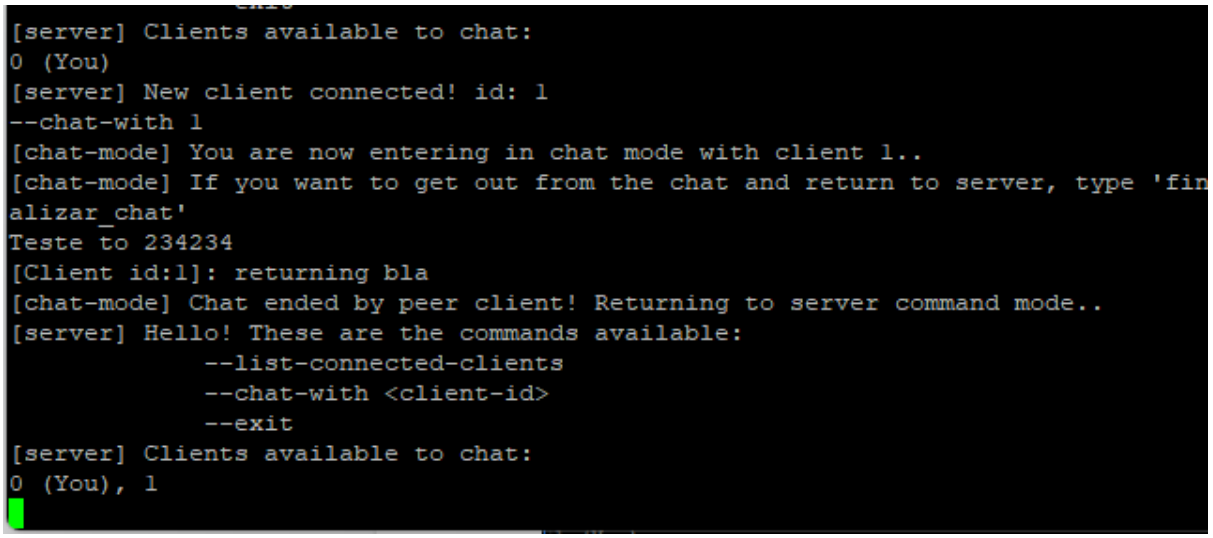
Quando um cliente desejar conversar com outro será mandado para o servidor o comando ***--chat-with <client-id>,*** passando o devido id do cliente que se deseja conversar. O servidor recebendo esta mensagem, parseia a string recebida e busca o cliente com o id fornecido. Com o objeto do cliente em mãos, é verificado se este já está em chat. Se o objeto do cliente a se iniciar o chat não foi encontrado ou este já está em chat, o servidor responde para o cliente requisitante mensagem de impossibilidade de iniciação de chat. Se o cliente buscado está livre, tanto este quanto o cliente requisitante são flagados como "**in_chat**" e o servidor envia mensagem para ambos de que estes estão entrando em modo de chat a partir da mensagem ***chat_init_with_client <client_id> <client_ip_address> <client_udp_port>.*** Assim, ambos os clientes terão as portas UDP de seus peers e poderão começar a trocar mensagens entre si via UDP.

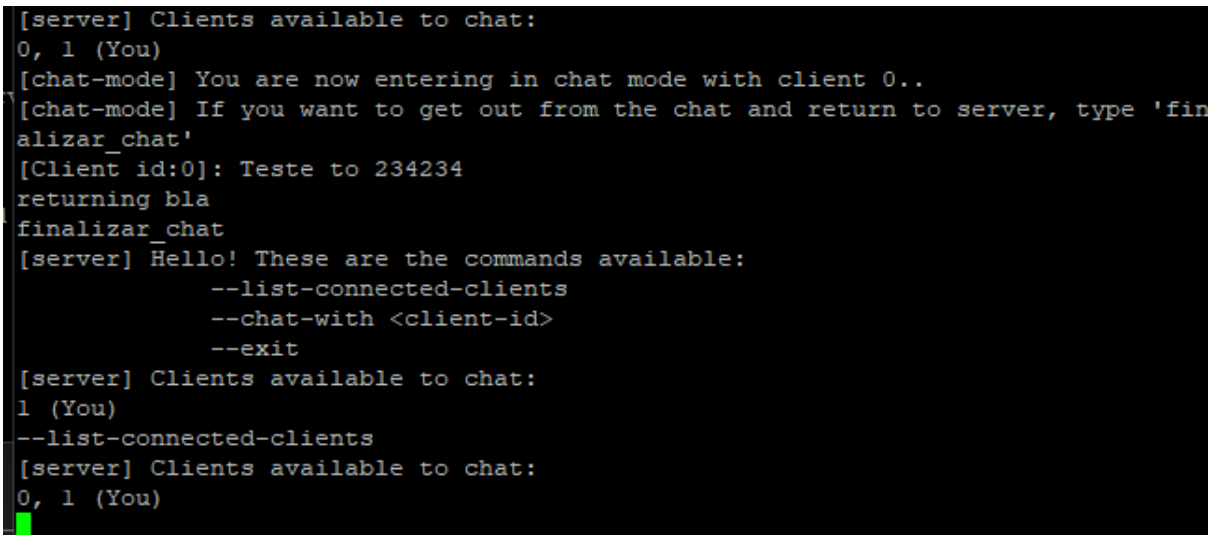




Para finalizar um chat, qualquer um dois dois clientes em chat pode enviar a palavra chave "**finalizar_chat**". O cliente que digitou a palavra chave a envia para o cliente com o qual está conversando e manda para o servidor a

mensagem ***finished_chat_with_peer***. O cliente que receber o palavra chave "**finalizar_chat**" efetuará o mesmo procedimento de enviar a mensagem ***finished_chat_with_peer*** para o servidor**.** O servidor irá marcar ***in_chat*** = 0 para cada cliente que enviar a mensagem ***finished_chat_with_peer*** e irá enviar para os clientes a lista de comandos disponíveis e usuários conectados.





(exemplo acima apresenta um pequeno bug. Pelo fato do cliente 1 ter iniciado a finalização do chat, o servidor responde mandando os clientes conectados, aparecendo na listagem apenas o cliente 1)

Para este trabalho final foi de extrema importância saber o funcionamento da função select e qual seria o seu objetivo dentro dos fluxos de servidor e cliente. Este ponto foi um pouco mais difícil para nós pelo fato de não termos feito a parte prática do exercício sobre multiplexação (exercício 4). O fato de termos que entender a função select acabou acontecendo quando tentamos

iniciar o desenvolvimento usando a função fork() para criação de um servidor concorrente (que aceita vários clientes de uma vez). A criação de um servidor concorrente com o fork() é possível, mas o problema foi quando tentou-se fazer um processo filho alterar uma estrutura de dados contida no pai, no caso a lista de clientes conectados ao servidor. Como criamos um processo filho, este recebe uma cópia de todos os dados do pai. Assim, se o filho alterar determinada estrutura de dados, estará apenas alterando sua própria cópia da estrutura, não alterando a do pai.

Este problema foi resolvido usando a função select no servidor para tratar diversos *file_descriptors* referentes a cada cliente que se conecta ao servidor. Com o select, foi possível construir um servidor single process que pudesse ser um servidor concorrente e assim manter todo o manuseio da lista de clientes conectados em um processo só. A implementação do lado do servidor do uso da função select foi baseada na implementação disponível no livro texto, pg. 230. A diferença foi que usamos uma lista ligada para armazenamento dos clientes ao invés de um vetor de inteiros para guardar o descritor de cada cliente por acharmos que com uma lista ligada o código poderia ficar mais limpo e mais fácil de entender.

```
453     int listenfd = initiateServer(argv[1]);
454     int maxfd = listenfd;
455 
456     fd_set rset, allset;
457     FD_ZERO(&allset);
458     FD_SET(listenfd, &allset);
459 
460     ClientLinkedList clientLinkedList;
461     initiateClientLinkedList(&clientLinkedList);
462 
463     for ( ; ; ) {
464         rset = allset; /* structure assignment */
465         select(maxfd + 1, &rset, NULL, NULL, NULL);
466         if (FD_ISSET(listenfd, &rset)) { /* new client connection */
467             handleNewConnection(&clientLinkedList, &allset, &maxfd, listenfd);
468         }
469         checkAllClientsForData(&clientLinkedList, &rset, &allset);
470     }
471 }
```

Com relação ao lado do cliente, o método select também foi muito importante pelo fato deste permitir, logo no início da implementação, registrar no fd_set o *file_descriptor* do socket de comunicação com o servidor e o

STDIN_FILENO, o *file_descriptor* do standard input. Isso possibilitou o recebimento de mensagens do servidor e o envio mensagens vindas do stdin ao mesmo tempo. Mais tarde, com a implementação da lógica de comunicação entre dois clientes, também foi adicionado ao fd_set o *file_descriptor* do socket UDP do cliente, para que caso este esteja em um chat possa ser identificado o recebimento de mensagens do seu peer.

```
select(maxfdp, &rset, NULL, NULL, NULL);

if (FD_ISSET(server_socket_fdescriptor, &rset)) {
    readMessageFromServer(server_socket_fdescriptor, currentUdpPort,
                          &chatObject);
}

if (FD_ISSET(udpSockFileDescriptor, &rset)) {
    readMessageFromChatPeer(udpSockFileDescriptor, &chatObject, server_socket_fdescriptor);
}

if (FD_ISSET(STDIN_FILENO, &rset)) {
    if (!chatObject.inChatWithAnotherClient) {
        sendMessageToServer(server_socket_fdescriptor);
    } else {
        sendMessageToAnotherClient(&chatObject, sockToSendMessagefd, server_socket_fdescriptor);
    }
}
```