Documentação - TP2 - Redes

Gabriel Cerqueira e Silva - 2017105567

Introdução

O trabalho prático 2 teve como proposta a implementação de um um sistema que coordene múltiplas conexões simultâneas entre os clientes e permita a comunicação entre eles.

Desafios

Ao realizar a implementação do trabalho , alguns desafios vieram a tona , sendo eles:

O primeiro desafio envolveu a compreensão da estrutura de sockets em C , na qual requer um entendimento de estruturas de dados, funções e chamadas do sistema relacionadas aos sockets. Foi necessário compreender os conceitos como criação de socket, associação de endereços, escuta por conexões, estabelecimento e encerramento de conexões, envio e recebimento de dados, entre outros.

Em seguida temos um outro desafio que foi o estabelecimento e desconexão de conexões, que envolveu a configuração correta dos parâmetros do socket, como família de protocolos (IPv4 ou IPv6), o socket e a porta. Além disso, foi necessário lidar com possíveis erros e tratar exceções durante o processo de conexão.

Em seguida houve o desafio da implementação das threads , tanto para permitir múltiplas conexões no lado do servidor, quanto para realizar o paralelismo entre o envio de comandos do teclado e recebimento de mensagens por parte do cliente.Foi desafiador também o compartilhamento de informação entre as threads criadas.

Solução

Servidor

Por parte do servidor, foram utilizadas threads para estabelecer múltiplas conexões, limitadas pelo contador de threads "threadCount" e a definição "MAX_CONNECTIONS" com valor 15. A função "connectionHandler" é executada em uma thread para lidar com uma conexão de cliente. Ela recebe um argumento do tipo `void*`, convertido para um ponteiro do tipo `ThreadArgs`, que contém informações relevantes sobre a conexão e o estado dos usuários, como o socket do cliente e a lista de usuários definidos pela estrutura "User". Essa função também executa a lógica de recebimento e tratamento de mensagens do cliente através da função "listenToClient".

A função "listenToClient" é responsável por lidar com os diferentes comandos do cliente, tais como "close connection", "list users", "send to" e "send all", realizando os tratamentos apropriados para cada um.

Antes do envio de mensagens, tanto em broadcast quanto em unicast, é realizada sua formatação por meio da função "formatMessage", que recebe os parâmetros `idMessage`, `idSender`, `idReceiver` e `message`, e os transforma no seguinte formato: "idMessage idSender idReceiver message".

A seguir temos as funções auxiliares e estruturas utilizadas no lado do servidor.Estas funções e estruturas são essenciais para o funcionamento do servidor, gerenciando os usuários conectados, enviando e recebendo mensagens e realizando outras operações relacionadas à comunicação cliente-servidor.

Funções auxiliares:

- `void initializeUserArray()`: Inicializa a matriz de usuários `users` definida globalmente, preenchendo-a com usuários vazios.
- `void usage(int argc, char **argv)`: Exibe informações sobre o uso correto do servidor, mostrando o tipo de IP (v4 ou v6) e a porta como argumentos.
- `void sendResponse(int csock, char *msg)`: Envia uma mensagem de resposta para o cliente através do socket `csock`, formatando a mensagem antes do envio.
- `User *findFirstDisconnectedUser(User **usersPtr)`: Encontra o primeiro usuário desconectado na matriz de usuários `usersPtr` e retorna um ponteiro para ele.
- `User *findUserByCsock(int csock, User **usersPtr)`: Localiza um usuário pelo socket `csock` na matriz de usuários `usersPtr` e retorna um ponteiro para ele.
- `User *findUserById(int userID, User **usersPtr)`: Localiza um usuário pelo seu ID na matriz de usuários `usersPtr` e retorna um ponteiro para ele.
- `void finalizeUserByCsock(int csock, User **usersPtr)`: Finaliza um usuário identificado pelo socket `csock` na matriz de usuários `usersPtr`, definindo o socket como -1 e o status de conexão como falso.
- `char* formatMessage(int idMessage, int idSender, int idReceiver, const char* message)`: Formata uma mensagem recebida como uma string com ID da mensagem, ID do remetente, ID do destinatário e conteúdo da mensagem, retornando a string formatada.
- `void broadcast(char *msg, int sendToUserID)`: Envia uma mensagem para todos os usuários conectados, exceto o usuário com ID especificado em `sendToUserID`.
- `void unicast(int csock, char *msg)`: Envia uma mensagem para um usuário específico identificado pelo socket `csock`.
- `char* getMessageInQuotes(const char* inputString)`: Extrai uma mensagem entre aspas duplas em uma string de entrada, retornando a mensagem extraída.
- `char* getCurrentTime()`: Obtém o horário atual no formato "[hh:ss]" e retorna a string formatada.
- `char* listConnectedUsers(User **usersPtr, int ownUserID, bool considerOwnId)`: Cria uma mensagem contendo os IDs dos usuários conectados, excluindo ou incluindo o ID do próprio usuário, e retorna a string formatada.

Estruturas utilizadas:

- `typedef struct User`: Define a estrutura `User`, que armazena informações sobre um usuário conectado ao servidor, incluindo ID, status de conexão e socket associado.
- `typedef struct ThreadArgs`: Define a estrutura `ThreadArgs`, que é usada para passar argumentos para as threads que manipulam conexões de clientes, contendo o socket de conexão e um ponteiro para a matriz de usuários.

Cliente

O cliente pode enviar mensagens para todos os usuários conectados por meio do chat, enviar mensagens privadas para usuários específicos, listar os usuários conectados e encerrar a conexão com o servidor. Além disso, foram implementadas funções auxiliares para manipulação de arrays de inteiros, como ordenação e busca; análise e formatação de mensagens para garantir que sejam enviadas corretamente ao servidor e recebidas de forma legível pelos usuários.

A função `runClient()` é responsável por executar o cliente de chat e gerenciar a comunicação com o servidor. Ela recebe como argumento o descritor do socket que foi previamente estabelecido para a comunicação com o servidor. Dentro da função, são declaradas as variáveis necessárias para o funcionamento do cliente, como "command" para armazenar o comando digitado pelo usuário e "buf" para receber as mensagens do usuário através do terminal.

Em seguida, é criada uma thread usando a função `pthread_create()` para tratar o recebimento contínuo de mensagens do servidor. A função `receiveMessages()` é executada nessa thread para receber as mensagens assincronamente do servidor e realizar seu respectivo tratamento, permitindo que o cliente receba mensagens do servidor sem bloquear a execução do restante do programa.

Dentro da função `runClient()`, é utilizado um loop infinito para aguardar os comandos do usuário e processá-los até que a variável "disconnect" seja setada para 1, indicando a vontade do cliente de encerrar a conexão com o servidor.

Após o recebimento do comando do usuário, é realizado o chaveamento para executar a ação apropriada:

- Se o comando for "close connection" (1), o cliente envia a mensagem "close connection" ao servidor usando `send()`, sinalizando que deseja encerrar a conexão.
- Se o comando for "list users" (2), o cliente envia a mensagem "list users" ao servidor usando `send()`, solicitando a lista de usuários conectados.
- Se o comando for "send to" (3), o cliente envia a mensagem digitada pelo usuário ao servidor usando `send()`, para que seja entregue ao destinatário específico.
- Se o comando for "send all" (4), o cliente envia a mensagem digitada pelo usuário ao servidor usando `send()`, para que seja enviada a todos os usuários conectados.

Dessa forma, a função `runClient()` mantém o cliente em execução, permitindo que ele envie comandos ao servidor e receba as mensagens enviadas por outros usuários.

A seguir temos as funções auxiliares e estruturas utilizadas. Essas funções desempenham papéis essenciais no cliente, desde o gerenciamento de usuários conectados até o tratamento das mensagens recebidas do servidor e sua correta interpretação para as devidas ações.

Funções auxiliares do Cliente:

- `bool userExists(int *array, int value)`: Verifica se um elemento existe no array.
- `int insertElement(int *array, int capacity, int value)`: Insere um inteiro no array, respeitando sua capacidade máxima.
- `int removeElement(int *array, int value)`: Remove um inteiro do array.

- `void usage(int argc, char **argv)`: Exibe informações sobre o uso correto do cliente, mostrando o IP e porta do servidor como argumentos.
- `char* receiveResponse(int sfd)`: Recebe a resposta do servidor por meio do socket `sfd` e retorna a resposta em uma string.
- `ParsedMessage* parseMessage(const char* formattedMsg)`: Analisa uma mensagem formatada do servidor e preenche a estrutura `ParsedMessage` com as informações extraídas.
- `void freeParsedMessage(ParsedMessage* parsedMsg)`: Libera a memória alocada para a estrutura `ParsedMessage`.
- `void processParsedMessage(int csock, ParsedMessage* parsedMsg)`: Processa a mensagem analisada do servidor, executando a ação apropriada com base no campo `idMessage`.
- `void *msgHandler(int sfd)`: Função executada em uma thread para lidar com as mensagens recebidas do servidor.
- `void *receiveMessages(void* arg)`: Função executada em uma thread para receber e processar mensagens do servidor continuamente enquanto o cliente estiver em execução.

Estruturas utilizadas:

- `typedef struct`: Define a estrutura `ParsedMessage` que contém os campos para armazenar informações de uma mensagem analisada do servidor.