



PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Bacharelado em Engenharia da Computação

Ramon Vinícius Silva Corrêa

**Trabalho Prático I - Redes de Computadores I**

Belo Horizonte

2024

Ramon Vinícius Silva Corrêa

## **Trabalho Prático I - Redes de Computadores I**

Trabalho Prático I – Redes de Computadores I – Curso de Engenharia da Computação da Pontifícia Universidade Católica de Minas Gerais.

Orientador: Ricardo Carlini Sperandio

Belo Horizonte

2024

## RESUMO

Este trabalho prático tem como objetivo, implementar um sistema de envio de mensagens curtas que funcionará em um modelo multi-servidor, utilizando de técnicas de programação orientada a eventos utilizando-se a primitiva select e a temporização por sinais.

No projeto, foi desenvolvida uma solução em linguagem C, que funciona em ambiente Linux que consiste em um servidor e dois clientes, um para envio e outro recebimento das mensagens utilizando os conceitos de sockets. O estudo dessas soluções permitiu uma compreensão mais profunda sobre o funcionamento dos conceitos de sockets, aplicando na prática os conhecimentos obtidos em aula.

**Palavras-chave:** Sockets, Select, Linguagem C, Temporização.

## LISTA DE FIGURAS

FIGURA 1 – Execução do Servidor .....	19
FIGURA 2 – Execução do cliente de envio de mensagens.....	19
FIGURA 3 – Execução do cliente de envio de mensagens.....	20

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO .....</b>	<b>1</b>
1.1	Objetivo do Trabalho .....	1
1.2	Contextualização Teórica .....	1
<b>2</b>	<b>DESENVOLVIMENTO .....</b>	<b>3</b>
2.1	Descrição do Problema .....	3
2.2	Metodologia .....	4
2.3	Implementação .....	4
2.3.1	<i>Servidor</i> .....	5
2.3.2	<i>Cliente de Envio</i> .....	13
2.3.3	<i>Cliente de Recebimento</i> .....	16
2.4	Resultados Obtidos .....	19
<b>3</b>	<b>CONCLUSÃO .....</b>	<b>21</b>
3.1	Resumo dos Principais Pontos Abordados .....	21
3.2	Conclusão .....	21
	<b>REFERÊNCIAS .....</b>	<b>23</b>

## 1 INTRODUÇÃO

A implementação de sistemas de comunicação é um campo essencial na engenharia de computação, onde são aplicados conhecimentos de redes, sistemas distribuídos e programação orientada a eventos. Em sistemas complexos, como redes sociais e plataformas de mensagens, o envio e recebimento de dados em tempo real é fundamental para garantir a interação entre os usuários de maneira eficiente e organizada. Neste contexto, o projeto apresentado explora o desenvolvimento de um sistema simples de mensagens em um ambiente multi-servidor, utilizando comunicação via protocolo TCP e controlando múltiplas conexões simultâneas.

Para lidar com a complexidade do sistema, o servidor será programado para gerenciar várias conexões e mensagens de clientes simultaneamente, utilizando a primitiva *select* para controle de múltiplos sockets e temporização baseada em sinais. Esta abordagem permite ao servidor monitorar diversas conexões em paralelo, processando eventos de entrada e saída de forma assíncrona, característica fundamental para sistemas de comunicação em tempo real.

### 1.1 Objetivo do Trabalho

O objetivo deste trabalho prático é proporcionar aos alunos uma compreensão das técnicas de programação orientada a eventos e temporização em sistemas multi-conexão. Através da implementação de um sistema de troca de mensagens utilizando sockets TCP, os alunos serão incentivados a aplicar conceitos de controle de concorrência e gerenciamento de conexões, além de desenvolver habilidades práticas na manipulação de sinais e na gestão de múltiplos clientes conectados simultaneamente.

### 1.2 Contextualização Teórica

A comunicação entre processos e sistemas distribuídos representa um dos desafios centrais na engenharia de redes e de sistemas. Em ambientes onde diversos clientes interagem de forma concorrente, é crucial a utilização de técnicas de multiplexação, como a primitiva *select*, para gerenciar as conexões e garantir que cada evento seja tratado

com eficiência. Neste projeto, são aplicados conceitos de temporização baseada em sinais e monitoramento de eventos para criar um servidor capaz de enviar mensagens periódicas a todos os clientes conectados, com o objetivo de replicar a funcionalidade básica de sistemas de mensagens contemporâneos.

A implementação desse servidor e dos programas clientes envolve a criação de um protocolo de comunicação sobre TCP, onde cada mensagem possui um formato padronizado para envio de identificadores de origem e destino e controle do tamanho das mensagens. Este trabalho permite aos alunos explorar conceitos de arquitetura de sistemas de comunicação e reforça a importância de um controle preciso de eventos e temporização para a criação de sistemas de mensagens confiáveis e escaláveis.

## 2 DESENVOLVIMENTO

Nesta seção, abordamos o processo de desenvolvimento do sistema de mensagens cliente-servidor em linguagem C. Primeiramente apresentaremos o problema a ser solucionado com o sistema, depois a metodologia utilizada para desenvolver esse sistema, depois explicaremos como cada parte do sistema está funcionando e por fim apresentaremos os resultados obtidos na execução do sistema.

### 2.1 Descrição do Problema

Como dito anteriormente, o "problema" é um desenvolvimento de um sistema de troca de mensagens utilizando comunicação via protocolo TCP. Para isso será necessário desenvolver três programas:

- **Servidor:** Um programa que será responsável pelo controle de troca de mensagens.
- **Cliente Receptor:** Um programa que será responsável por exibir as mensagens recebidas no servidor.
- **Cliente Transmissor:** Um programa que será responsável por enviar as mensagens para o servidor.

Para as mensagens que serão enviadas e recebidas pelo servidor, foi definido o seguinte formato:

```
typedef struct {  
    unsigned short int type;  
    unsigned short int orig_uid;  
    unsigned short int dest_uid;  
    unsigned short int text_len;  
    unsigned char text[141];  
} msg_t;
```

As seguintes mensagens são definidas (identificadas pelo valor inteiro associado):



- **OI (0):** Primeira mensagem de um programa cliente (tanto de teclado quanto de exibição) para se identificar para o servidor. Caso a conexão seja aceita, o servidor deve enviar de volta uma mensagem idêntica; caso contrário, basta que ele feche a conexão.
- **TCHAU (1):** Última mensagem de um cliente de envio de mensagens para registrar a sua saída. A partir dessa mensagem o servidor fecha a conexão para aquele cliente e qualquer cliente de exibição que esteja associado a ele.
- **MSG (2):** Mensagem enviada pelo usuário para o servidor e do servidor para o exibidor. O campo de identificador da origem indica o cliente que enviou a mensagem e o campo de destino deve conter o número do programa de exibição alvo ou zero para indicar que a mensagem deve ser enviada a todos os clientes. O servidor, ao receber tal mensagem, deve confirmar que o identificador do cliente de origem corresponde ao que foi usado na mensagem de OI daquele cliente.

## 2.2 Metodologia

Para o desenvolvimento do projeto especificado, abordamos a seguinte metodologia:

1. **Configuração do Servidor:** Configuração do servidor para escutar conexões, aceitar múltiplos clientes e enviar mensagens de status periodicamente.
2. **Desenvolvimento de Cliente de Envio:** Desenvolver um cliente que irá se conectar ao servidor para enviar mensagens digitadas pelo usuário.
3. **Desenvolvimento de Cliente de Recebimento:** Desenvolver um cliente que irá se conectar ao servidor para ficar escutando as mensagens recebidas nele e exibi-las para o usuário.
4. **Realização de testes de funcionamento:** Ao final do desenvolvimento dos três códigos, cada um foi testado individualmente, para garantir seu funcionamento correto.

## 2.3 Implementação

Para iniciar a implementação, primeiro organizamos a estrutura do projeto para que ela ficasse mais simples de se entender e realizar possíveis manutenções. Assim chegamos na seguinte arquitetura de projeto:

Estrutura:

```

Server/
|-- Server.c
|-- Server.h
|-- msg_protocol.h
|-- main.c
|-- Makefile

```

```

SendClient/
|-- SendClient.c
|-- SendClient.h
|-- msg_protocol.h
|-- main.c
|-- Makefile

```

```

ReceiveClient/
|-- ReceiveClient.c
|-- ReceiveClient.h
|-- msg_protocol.h
|-- main.c
|-- Makefile

```

Nessa estrutura apresentada temos os arquivos *main.c* de cada projeto que apenas inicia as respectivas funções de cada código, ou seja, inicia o servidor ou os clientes, que estão presentes nos arquivos de nome *Server.c*, *SendClient.c* e *ReceiveClient.c*. Também temos o arquivo header *msgprotocol.h* que possui o struct de mensagens definido anteriormente.

### 2.3.1 Servidor

O código em C para a implementação do servidor é mostrado a seguir:

```

Server.c

#include "server.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static client_t

```

```

    clients[MAX_CLIENTS];          // Array para armazenar os clientes conectados
static int client_count = 0;       // Contador de clientes conectados
static time_t server_start_time;  // Tempo de início do servidor
volatile sig_atomic_t timer_expired =
    0; // Flag para indicar que o timer expirou

// Manipulador de sinal para SIGALRM
void handle_alarm(int sig) {
    (void)sig; // Suprime o aviso de parâmetro não utilizado
    timer_expired = 1;
}

// Configura o timer para expirar a cada 60 segundos
void setup_timer() {
    struct sigaction sa;
    struct itimerval timer;

    // Configura o handler para o sinal SIGALRM
    sa.sa_handler = &handle_alarm;
    sa.sa_flags = SA_RESTART;
    sigaction(SIGALRM, &sa, NULL);

    // Configura o timer para expirar a cada 60 segundos
    timer.it_value.tv_sec = 60;
    timer.it_value.tv_usec = 0;
    timer.it_interval.tv_sec = 60;
    timer.it_interval.tv_usec = 0;
    setitimer(ITIMER_REAL, &timer, NULL);
}

// Função para enviar uma mensagem de status do servidor
void send_server_status() {
    char status_message[141];
    snprintf(status_message, sizeof(status_message),
        "Servidor ativo há %ld segundos com %d clientes conectados.",
        time(NULL) - server_start_time, client_count);

    msg_t message;
    message.type = htons(MSG_TYPE_MSG);

```

```

message.orig_uid = htons(0); // UID do servidor
message.dest_uid = htons(0); // Broadcast
message.text_len = htons(strlen(status_message));
strcpy((char*)message.text, status_message);

// Envia a mensagem de status para todos os clientes conectados
for (int i = 0; i < client_count; i++) {
    send(clients[i].socket, &message, sizeof(message), 0);
}
printf("Mensagem de status enviada: %s\n", status_message);
}

// Função para processar mensagens dos clientes
void handle_client_message(client_t* client, msg_t* message) {
    switch (ntohs(message->type)) {
        case MSG_TYPE_OI:
            client->uid = ntohs(message->orig_uid);
            send(client->socket, message, sizeof(*message), 0);
            break;
        case MSG_TYPE_TCHAU:
            close(client->socket);
            *client = clients[client_count - 1];
            client_count--;
            break;
        case MSG_TYPE_MSG:
            if (client->uid == ntohs(message->orig_uid)) {
                broadcast_message(message, client);
            }
            break;
        default:
            printf("Tipo de mensagem desconhecido.\n");
            break;
    }
}

// Função para enviar uma mensagem para todos os clientes conectados
void broadcast_message(msg_t* message, client_t* sender) {
    for (int i = 0; i < client_count; i++) {
        if (clients[i].uid != sender->uid) {

```

```

        send(clients[i].socket, message, sizeof(*message), 0);
    }
}

// Função para verificar se um UID já está em uso
int is_uid_in_use(uint16_t uid) {
    for (int i = 0; i < client_count; i++) {
        if (clients[i].uid == uid) {
            return 1;
        }
    }
    return 0;
}

// Função para iniciar o servidor
void start_server(const char* port) {
    int listen_socket;
    struct sockaddr_in server_addr;

    // Cria um socket para escutar conexões
    listen_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_socket < 0) {
        perror("Falha ao criar o socket");
        return;
    }

    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(atoi(port));

    // Associa o socket ao endereço e porta especificados
    if (bind(listen_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) <
        0) {
        perror("Falha ao associar (bind)");
        close(listen_socket);
        return;
    }
}

```

```

// Coloca o socket em modo de escuta para aceitar conexões
if (listen(listen_socket, SOMAXCONN) < 0) {
    perror("Falha ao escutar (listen)");
    close(listen_socket);
    return;
}

printf("Servidor iniciado na porta %s\n", port);
server_start_time = time(NULL);

fd_set read_fds;

// Configura o timer para enviar mensagens de status periodicamente
setup_timer();

while (1) {
    FD_ZERO(&read_fds);
    FD_SET(listen_socket, &read_fds);
    for (int i = 0; i < client_count; i++) {
        FD_SET(clients[i].socket, &read_fds);
    }

    struct timeval timeout;
    timeout.tv_sec = 1;
    timeout.tv_usec = 0;

    int activity = select(FD_SETSIZE, &read_fds, NULL, NULL, &timeout);
    if (activity < 0) {
        if (errno == EINTR) {
            // Se o select foi interrompido por um sinal, continue
            continue;
        } else {
            perror("Erro no select");
            break;
        }
    }

    if (timer_expired) {
        send_server_status();
    }
}

```

```

    timer_expired = 0;
}

if (FD_ISSET(listen_socket, &read_fds)) {
    // Aceita novas conexões
    int new_socket = accept(listen_socket, NULL, NULL);
    if (new_socket < 0) {
        perror("Falha ao aceitar conexão");
        continue;
    }

    // Recebe a mensagem de saudação (OI) do novo cliente
    msg_t message;
    int bytes_received = recv(new_socket, &message, sizeof(message), 0);
    if (bytes_received <= 0 || ntohs(message.type) != MSG_TYPE_OI) {
        printf("Falha ao receber mensagem OI.\n");
        close(new_socket);
        continue;
    }

    uint16_t new_uid = ntohs(message.orig_uid);

    // Verifica se é um cliente de exibição (UID entre 1 e 999)
    if (new_uid >= 1 && new_uid <= 999) {
        // Verifica se o UID
        if (is_uid_in_use(new_uid)) {
            printf(
                "UID %hu já está em uso para um cliente de exibição. Conexão "
                "recusada.\n",
                new_uid);
            message.type = htons(MSG_TYPE_ERRO);
            send(new_socket, &message, sizeof(message), 0);
            close(new_socket);
            continue;
        }
    }

    // Verifica se é um cliente de envio de mensagens (UID entre 1001 e 1999)
    else if (new_uid >= 1001 && new_uid <= 1999) {
        // Verifica se apenas o UID está em uso (sem verificar UID - 1000)

```

```

if (is_uid_in_use(new_uid)) {
    printf(
        "UID %hu já está em uso para um cliente de envio. Conexão "
        "recusada.\n",
        new_uid);
    message.type = htons(MSG_TYPE_ERRO);
    send(new_socket, &message, sizeof(message), 0);
    close(new_socket);
    continue;
}
} else {
    // UID inválido, fora do intervalo permitido
    printf("UID %hu está fora do intervalo permitido. Conexão recusada.\n",
        new_uid);
    message.type = htons(MSG_TYPE_ERRO);
    send(new_socket, &message, sizeof(message), 0);
    close(new_socket);
    continue;
}

if (client_count < MAX_CLIENTS) {
    clients[client_count].socket = new_socket;
    clients[client_count].uid = new_uid;
    client_count++;
    printf("Novo cliente conectado com UID %hu.\n", new_uid);

    // Envia a resposta "OI" de volta ao cliente
    msg_t oi_response;
    oi_response.type = htons(MSG_TYPE_OI);
    oi_response.orig_uid = htons(0); // UID do servidor
    oi_response.dest_uid = htons(new_uid);
    oi_response.text_len = htons(0);
    send(new_socket, &oi_response, sizeof(oi_response), 0);
} else {
    printf("Número máximo de clientes atingido. Conexão recusada.\n");
    close(new_socket);
}
}

```



```

// Lida com clientes existentes
for (int i = 0; i < client_count; i++) {
    if (FD_ISSET(clients[i].socket, &read_fds)) {
        msg_t message;
        int bytes_received =
            recv(clients[i].socket, &message, sizeof(message), 0);
        if (bytes_received <= 0) {
            printf("Cliente desconectado.\n");
            close(clients[i].socket);
            clients[i] = clients[client_count - 1];
            client_count--;
            i--;
        } else {
            handle_client_message(&clients[i], &message);
        }
    }
}

close(listen_socket);
}

```

### 1. Configurações Iniciais:

- Inicializa variáveis globais, como a lista de clientes conectados e o contador de clientes.
- Configura um timer para enviar mensagens de status a cada 60 segundos usando sinais (SIGALRM).

### 2. Criação do Socket de Escuta:

- Cria um socket para escutar conexões de clientes.
- Associa o socket ao endereço e porta especificados.
- Coloca o socket em modo de escuta para aceitar conexões.

### 3. Loop Principal:

- Configura o conjunto de descritores de arquivo (**fd\_set**) para monitorar múltiplos sockets.
- Usa a função **select** para esperar por atividade em qualquer um dos sockets (novas conexões ou mensagens de clientes).

- Verifica se o timer expirou e, se sim, envia uma mensagem de status para todos os clientes conectados.
- Aceita novas conexões de clientes e adiciona-os à lista de clientes conectados, se o número máximo de clientes não for atingido.
- Recebe e processa mensagens dos clientes, incluindo mensagens de identificação (OI), desconexão (TCHAU) e mensagens de texto (MSG).
- Remove clientes desconectados da lista de clientes conectados.

4. **Encerramento:** Fecha o socket de escuta quando o servidor é encerrado.

O servidor aceita conexões de múltiplos clientes, gerencia essas conexões usando `select`, processa mensagens dos clientes e envia periodicamente mensagens de status para todos os clientes conectados.

### 2.3.2 *Cliente de Envio*

O código em C para a implementação do cliente de envio é mostrado a seguir:

`SendClient.c`

```
#include "SendClient.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Função que inicia o cliente de envio de mensagens
void start_send_client(const char* server_addr,
                      const char* port,
                      uint16_t uid) {
    int sock; // Descritor de socket
    struct sockaddr_in
        server; // Estrutura para armazenar informações sobre o servidor

    // Cria um socket
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("Criação do socket falhou");
        return;
    }
}
```

```

// Configura a estrutura sockaddr_in com informações sobre o servidor
server.sin_family = AF_INET;
server.sin_addr.s_addr = inet_addr(server_addr);
server.sin_port = htons(atoi(port));

// Conecta ao servidor
if (connect(sock, (struct sockaddr*)&server, sizeof(server)) < 0) {
    perror("Conexão falhou");
    close(sock);
    return;
}

// Prepara a mensagem de saudação (OI)
msg_t message;
message.type = htons(MSG_TYPE_OI);
message.orig_uid = htons(uid);
message.dest_uid = 0;
message.text_len = 0;

// Envia a mensagem de saudação (OI) ao servidor
send(sock, &message, sizeof(message), 0);

// Recebe a resposta do servidor
int bytes_received = recv(sock, &message, sizeof(message), 0);
if (bytes_received <= 0 || ntohs(message.type) != MSG_TYPE_OI) {
    printf("Falha ao receber resposta OI.\n");
    close(sock); // Fecha o socket
    return;
}

printf("Conectado ao servidor. Você pode começar a enviar mensagens.\n");

// Loop para enviar mensagens
while (1) {
    char text[141]; // Buffer para armazenar a mensagem
    uint16_t dest_uid; // UID do destinatário

    // Solicita o UID de destino do usuário

```

```

printf("Digite o UID de destino (0 para broadcast): ");
scanf("%hu", &dest_uid);

// Solicita a mensagem ao usuário
printf("Digite sua mensagem: ");
scanf(" %[^\\n]", text);

// Prepara a mensagem a ser enviada
message.type = htons(MSG_TYPE_MSG);
message.orig_uid = htons(uid);
message.dest_uid = htons(dest_uid);
message.text_len = htons(strlen(text));
strcpy((char*)message.text, text);

// Envia a mensagem ao servidor
send(sock, (char*)&message, sizeof(message), 0);
}

// Fecha o socket
close(sock);
}

```

### 1. Configurações Iniciais:

- O programa recebe argumentos de linha de comando para o identificador do cliente (`uid`), o endereço do servidor e a porta do servidor.
- Cria um socket para se conectar ao servidor.

2. **Conexão ao Servidor:** Estabelece uma conexão TCP com o servidor usando o endereço e a porta fornecidos.

### 3. Envio de Mensagens:

- Prepara uma mensagem de identificação (OI) e a envia ao servidor para se registrar.
- Entra em um loop onde lê mensagens do usuário (do teclado).
- Para cada mensagem, prepara a estrutura da mensagem (`msg_t`) com os campos apropriados:
  - Tipo de mensagem (`MSG_TYPE_MSG`).

- Identificador de origem (`orig_uid`).
- Identificador de destino (`dest_uid`).
- Comprimento do texto (`text_len`).
- Texto da mensagem (`text`).
- Envia a mensagem preparada ao servidor.

#### 4. **Encerramento:** Fecha o socket quando o programa termina.

O *SendClient.c* é um programa cliente que se conecta a um servidor, envia uma mensagem de identificação para se registrar, e então entra em um loop onde lê mensagens do usuário e as envia ao servidor. Quando o programa termina, ele fecha o socket.

### 2.3.3 *Cliente de Recebimento*

O código em linguagem C para a implementação do cliente de recebimento é mostrado a seguir:

ReceiveClient.c

```
#include "ReceiveClient.h"
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

// Função que inicia o cliente de recebimento de mensagens
void start_receive_client(const char* server_addr,
                        const char* port,
                        uint16_t uid) {
    int sock; // Descritor de socket
    struct sockaddr_in
        server; // Estrutura para armazenar informações sobre o servidor

    // Cria um socket
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("Falha na criação do socket");
        return;
    }
}
```

```

}

// Configura a estrutura de endereço do servidor
server.sin_family = AF_INET;
server.sin_addr.s_addr = inet_addr(server_addr);
server.sin_port = htons(atoi(port));

// Conecta ao servidor
if (connect(sock, (struct sockaddr*)&server, sizeof(server)) < 0) {
    perror("Falha na conexão");
    close(sock);
    return;
}

// Prepara a mensagem de identificação inicial
msg_t message;
message.type = htons(MSG_TYPE_OI);
message.orig_uid = htons(uid);
message.dest_uid = 0;
message.text_len = 0;

// Envia a mensagem de identificação inicial
send(sock, &message, sizeof(message), 0);

// Recebe a resposta do servidor
int bytes_received = recv(sock, &message, sizeof(message), 0);
if (bytes_received <= 0 || ntohs(message.type) != MSG_TYPE_OI) {
    printf("Falha ao receber resposta OI.\n");
    close(sock);
    return;
}

printf("Conectado ao servidor. Aguardando mensagens...\n");

// Loop principal para receber mensagens do servidor
while (1) {
    bytes_received = recv(sock, &message, sizeof(message), 0);
    if (bytes_received <= 0) {
        printf("Conexão perdida.\n");
    }
}

```

```

        break;
    }

    // Verifica se a mensagem recebida é uma mensagem de texto
    if (ntohs(message.type) == MSG_TYPE_MSG) {
        uint16_t orig_uid = ntohs(message.orig_uid);
        uint16_t dest_uid = ntohs(message.dest_uid);
        if (dest_uid == uid) {
            printf("Mensagem privada de %hu para %hu: %s\n", orig_uid, dest_uid,
                message.text);
        } else {
            printf("Mensagem de %hu para %hu: %s\n", orig_uid, dest_uid,
                message.text);
        }
    }
}

// Limpa e fecha o socket
close(sock);
}

```

### 1. Configurações Iniciais:

- O programa recebe argumentos de linha de comando para o endereço do servidor, a porta do servidor e o identificador do cliente (`uid`).
- Cria um socket para se conectar ao servidor.

### 2. Conexão ao Servidor:

- Configura a estrutura de endereço do servidor (`sockaddr_in`) com o endereço e a porta fornecidos.
- Estabelece uma conexão TCP com o servidor usando o socket criado.

### 3. Identificação Inicial:

- Prepara uma mensagem de identificação (OI) com o identificador do cliente (`uid`).
- Envia a mensagem de identificação ao servidor.
- Aguarda uma resposta do servidor para confirmar a identificação.

### 4. Recepção de Mensagens:

- Entra em um loop onde espera por mensagens do servidor.
- Recebe mensagens do servidor e verifica se são mensagens de texto (`MSG_TYPE_MSG`).
- Exibe as mensagens recebidas na tela, mostrando o identificador de origem e o texto da mensagem.

5. **Encerramento:** Fecha o socket quando a conexão é perdida ou o programa termina.

O *ReceiveClient.c* é um programa cliente que se conecta a um servidor, envia uma mensagem de identificação para se registrar, e então entra em um loop onde espera por mensagens do servidor. Quando uma mensagem de texto é recebida, ela é exibida na tela. O programa fecha o socket quando a conexão é perdida ou o programa termina.

## 2.4 Resultados Obtidos

Com os códigos apresentados funcionando, nós obtivemos um ótimo resultado dentro do que foi proposto no projeto, onde o servidor aceita e gerencia conexões de múltiplos clientes, e os clientes enviam e recebem as mensagens. A seguir temos algumas imagens do terminal mostrando a execução dos clientes e servidor:

```

ramon@Nitro-AN517-51:/mnt/c/Users/1210499/Desktop/Linux/Server$ ./Server
Servidor iniciado na porta 8080
Novo cliente conectado com UID 1001.
Novo cliente conectado com UID 1.
Mensagem de status enviada: Servidor ativo há 61 segundos com 2 clientes conectados.
Mensagem de status enviada: Servidor ativo há 121 segundos com 2 clientes conectados.
Mensagem de status enviada: Servidor ativo há 181 segundos com 2 clientes conectados.

```

Figura 1 – Execução do Servidor

Na Figura 1 é possível ver a execução do servidor, onde ele nos apresenta mensagens de qual porta o servidor foi iniciado, os clientes conectados e também a cada 60 segundos ele envia uma mensagem informando o status do servidor apresentando o tempo e o número de clientes conectados.

```

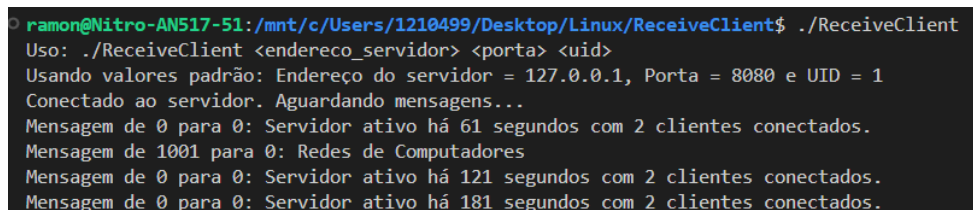
PROBLEMAS 14 SAÍDA CONSOLE DE DEPURACÃO TERMINAL PORTAS 1 COMENTÁRIOS
ramon@Nitro-AN517-51:/mnt/c/Users/1210499/Desktop/Linux/SendClient$ ./SendClient
Uso: ./SendClient <endereco_servidor> <porta> <uid>
Usando valores padrão: Endereço do servidor = 127.0.0.1, Porta = 8080 e UID = 1001
Conectado ao servidor. Você pode começar a enviar mensagens.
Digite o UID de destino (0 para broadcast): 0
Digite sua mensagem: Redes de Computadores
Digite o UID de destino (0 para broadcast): 

```

Figura 2 – Execução do cliente de envio de mensagens



Na Figura 2 vemos a execução do cliente de envio das mensagens. É possível ver que ao iniciar é mostrada uma mensagem indicando o endereço do servidor, a porta e o UID do qual ele vai se conectar. Caso a conexão seja feita com sucesso, você poderá enviar as mensagens normalmente adicionando o UID de destino e a mensagem que você deseja enviar.



```
ramon@Nitro-AN517-51:/mnt/c/Users/1210499/Desktop/Linux/ReceiveClient$ ./ReceiveClient
Uso: ./ReceiveClient <endereco_servidor> <porta> <uid>
Usando valores padrão: Endereço do servidor = 127.0.0.1, Porta = 8080 e UID = 1
Conectado ao servidor. Aguardando mensagens...
Mensagem de 0 para 0: Servidor ativo há 61 segundos com 2 clientes conectados.
Mensagem de 1001 para 0: Redes de Computadores
Mensagem de 0 para 0: Servidor ativo há 121 segundos com 2 clientes conectados.
Mensagem de 0 para 0: Servidor ativo há 181 segundos com 2 clientes conectados.
```

**Figura 3 – Execução do cliente de envio de mensagens**

Na figura 3 é possível visualizar a execução do cliente de recebimento de mensagens, que assim como o cliente de envio, apresenta as informações de conexão e também apresenta as mensagens recebidas. Podemos ver a mensagem que foi enviada pelo cliente na Figura 2 e também as mensagens de status do servidor.

O código completo para execução também se encontra em um repositório no GitHub, com um arquivo Readme.md explicando como executá-lo. Para acessar o repositório basta *clicar aqui* e você será redirecionado para a página.

### **3 CONCLUSÃO**

#### **3.1 Resumo dos Principais Pontos Abordados**

Neste trabalho, desenvolvemos um sistema de mensagens cliente-servidor em C, composto por um servidor e dois clientes (um para envio e outro para recebimento de mensagens). O servidor foi projetado para gerenciar conexões simultâneas de clientes, permitindo a troca de mensagens em tempo real e enviando periodicamente mensagens de status. Os clientes, por sua vez, foram implementados para facilitar o envio e a recepção de mensagens, utilizando um protocolo definido que inclui mensagens de identificação e despedida.

O sistema foi validado por meio de testes práticos, onde as funcionalidades dos clientes e do servidor foram confirmadas. O cliente de recebimento demonstrou ser capaz de conectar-se ao servidor e exibir mensagens recebidas de maneira eficaz, enquanto o cliente de envio permitiu ao usuário especificar destinatários e enviar mensagens corretamente. O servidor mostrou sua eficiência ao gerenciar múltiplas conexões e retransmitir mensagens para os destinatários apropriados.

#### **3.2 Conclusão**

A implementação do sistema de mensagens atingiu os objetivos estabelecidos, destacando a robustez e a flexibilidade da programação em C para aplicações de rede. As simulações e testes confirmaram que o servidor e os clientes respondem de maneira adequada às diversas interações, garantindo a troca de mensagens de forma confiável e eficaz.

O servidor demonstrou ser uma solução eficiente para o gerenciamento de clientes e a transmissão de mensagens, assegurando que as comunicações entre os usuários ocorram de maneira organizada e dentro dos parâmetros definidos pelo protocolo. A capacidade do sistema em lidar com múltiplos clientes simultaneamente é essencial para aplicações onde a comunicação em tempo real é necessária, como em plataformas de chat e sistemas de notificações.

Os clientes, ao facilitarem a interação do usuário com o servidor, mostraram-se

fundamentais para a experiência do usuário final, permitindo a entrada e o envio de mensagens de forma intuitiva. Além disso, a estrutura de mensagens projetada permite uma fácil extensão e adaptação para futuros desenvolvimentos, como a implementação de funcionalidades adicionais ou a melhoria do protocolo de comunicação.

Em suma, o desenvolvimento deste sistema de mensagens não apenas proporcionou uma compreensão aprofundada das técnicas de programação de rede, mas também lançou as bases para projetos futuros que podem expandir e otimizar o desempenho e a usabilidade desse sistema, integrando novas funcionalidades ou melhorando a arquitetura existente.

## REFERÊNCIAS

TANENBAUM, A. S. **Organização Estruturada de Computadores**, 5a. ed. São Paulo: ... São Paulo: Makron Books, 1996. *TANENBAUM, A. S.*

HABBEMA, Hugo. **Desvendando o WSL2 no Windows 11**. Medium, 22 jan. 2024. Disponível em: <https://medium.com/@habbema/desvendando-o-wsl-2-no-windows-11-c7649545026d>. Acesso em: 26 out. 2024.

SINHA, Akshat. **Socket Programming in C**. Geeks For Geeks, 11 out. 2024. Disponível em: <https://www.geeksforgeeks.org/socket-programming-cc/>. Acesso em: 26 out. 2024.

Canal DevPro. **Como usar VSCode com Linux e WSL do jeito certo**. YouTube, 11 abr. 2024. Disponível em: <https://www.youtube.com/watch?v=oZOO5ZQ9Zfg>. Acesso em: 26 out. 2024.

Sandro Ramos. **Instalação WSL2 - 2024**. YouTube, 25 jul. 2024. Disponível em: <https://www.youtube.com/watch?v=oEdIf6mB-p4> . Acesso em: 26 out. 2024.