

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS  
INSTITUTO DE CIÊNCIAS EXATAS E COMPUTAÇÃO  
ENGENHARIA DE COMPUTAÇÃO

# Trabalho Prático Redes - Programação Socket

**Alunos:**

Alberto Magno Machado

Bruno Guimarães Bitencourt

Oscar Dias

**Professor:**

Ricardo Carlini Sperandio

## Sumário

<b>1</b>	<b>Introdução</b>	<b>4</b>
<b>2</b>	<b>Desenvolvimento</b>	<b>5</b>
2.1	Descrição do Problema . . . . .	5
2.2	Metodologia e Implementação . . . . .	5
2.2.1	Configuração do Ambiente . . . . .	6
2.2.2	Criação do Makefile . . . . .	7
2.2.3	Servidor . . . . .	8
2.2.4	Cliente Receiver . . . . .	9
2.2.5	Cliente Sender . . . . .	10
2.2.6	Transmissão Periódica do Servidor . . . . .	10
<b>3</b>	<b>Resultados</b>	<b>11</b>
<b>4</b>	<b>Conclusão</b>	<b>14</b>

## Lista de Figuras

1	Conexão entre servidor e cliente receiver com ID = 24 . . . . .	11
2	Conexão entre servidor e cliente sender com ID = 1240 . . . . .	11
3	Conexão entre servidor e cliente receiver com ID = 25 . . . . .	11
6	Encerramento de conexão de um cliente. . . . .	12
4	Envio de uma mensagem para todos os clientes . . . . .	12
5	Envio de mensagem para apenas um cliente específico. . . . .	12
7	Tentativa de retransmissão periódica . . . . .	13

## Lista de Tabelas

## 1 Introdução

Este trabalho apresenta a implementação de um sistema de envio e recepção de mensagens curtas em um ambiente multi-servidor, desenvolvido em linguagem C para a disciplina de Redes de computadores 1. O objetivo é estabelecer um modelo de comunicação entre um servidor e dois tipos de clientes — um para envio de mensagens e outro para exibição. Utilizando sockets TCP, o servidor é responsável por gerenciar conexões simultâneas e monitorar as atividades de cada cliente conectado, implementando temporizações periódicas. Espera-se que o sistema demonstre a troca eficiente e sincronizada de mensagens entre os clientes, permitindo a interação direta entre emissor e receptor, assim como a manutenção de uma comunicação estável e controlada pelo servidor. Ao final, o sistema deve mostrar resultados robustos de transmissão de dados em tempo real, com capacidade de gerenciamento e desconexão organizada dos clientes.

## 2 Desenvolvimento

### 2.1 Descrição do Problema

O problema proposto envolve a implementação de um servidor capaz de gerenciar múltiplas conexões de clientes, que trocam mensagens através de um protocolo simples e específico. O servidor deve ser capaz de:

- Aceitar conexões TCP de até 20 clientes simultaneamente.
- Identificar e registrar os clientes por meio de um identificador único (UID).
- Enviar e receber mensagens com diferentes tipos, incluindo mensagens públicas e privadas.
- Garantir que cada mensagem seja tratada adequadamente com base no seu tipo, seja para um único destinatário ou para todos os clientes conectados.
- Lidar com o encerramento das conexões de maneira eficiente, removendo os clientes desconectados da lista ativa de clientes.

O servidor deve gerenciar todos esses processos de forma eficiente e sem bloqueios, utilizando uma abordagem que permita o tratamento simultâneo de várias conexões de clientes, o que é essencial para um sistema com múltiplos usuários.

### 2.2 Metodologia e Implementação

O desenvolvimento utilizou várias bibliotecas da linguagem C para gerenciar tanto a comunicação em rede quanto as operações de tempo e controle de sinais. As principais bibliotecas e recursos utilizados incluem:

**sys/socket.h e arpa/inet.h:**

Fornecem funções e estruturas essenciais para a criação e configuração de sockets TCP/IP, permitindo a comunicação em rede entre o servidor e os clientes.

**unistd.h:**

Oferece funções de controle de execução e fechamento de sockets, como close(), importantes para o gerenciamento de conexões e para a liberação de recursos.

**fcntl.h:**

Usada para definir e controlar características dos descritores de arquivos, garantindo que as conexões dos clientes sejam tratadas de maneira não bloqueante.

**sys/select.h:**

Contém a função select() que permite a multiplexação de conexões, essencial para o servidor monitorar várias conexões de clientes de forma eficiente e simultânea.

**signal.h e sys/time.h:**

Implementam a configuração de temporizadores e manipulação de sinais para que o servidor envie mensagens de status periódicas aos clientes, além de permitir uma temporização precisa nas comunicações.

Os três componentes do sistema, servidor, cliente emissor e cliente receptor, seguem um protocolo de comunicação baseado em mensagens com um formato predefinido. Cada mensagem contém um cabeçalho que indica o tipo de operação (identificação, desconexão ou mensagem de texto), um identificador único de origem e destino, o comprimento do texto e o conteúdo da mensagem propriamente dito. Essa estrutura garante que as mensagens sejam interpretadas corretamente e processadas conforme seu propósito.

### 2.2.1 Configuração do Ambiente

Para garantir um ambiente padronizado e facilitar o desenvolvimento e a execução do servidor e dos clientes, foi utilizado o Docker para configurar e isolar a aplicação. O Docker permite que a aplicação rode de maneira consistente, independente do sistema operacional utilizado, já que encapsula todas as dependências e configurações em um container.

```
1 # Usa uma imagem base com GCC instalado
2 FROM gcc:latest
3
4 # Instalar o GDB para depuração
5 RUN apt-get update && apt-get install -y gdb
6
7 # Define o diretório de trabalho dentro do contêiner
8 WORKDIR /workspace
9
10 # Copia todos os arquivos do projeto para dentro do contêiner
11 COPY . /workspace
12
13 # Exponha a porta 8080 para comunicação (caso seja necessário para seu servidor)
14 EXPOSE 8080
15
16 # Comando padrão do contêiner (pode ser substituído pelo VSCode para depuração)
17 CMD ["/bin/bash"]
```

**Programa 1:** Dockerfile utilizado

O Dockerfile configura um ambiente de desenvolvimento em C com GCC e GDB, permitindo compilar e depurar o código do servidor e cliente em um contêiner isolado. Utilizando uma imagem base com GCC pré-instalado, ele cria um diretório de trabalho dedicado, onde todos os arquivos do projeto são copiados para facilitar o acesso e a organização. O GDB é instalado para que fosse possível possam realizar depurações e análises de erros durante a execução. A porta 8080 foi exposta para permitir comunicações externas, caso o servidor seja configurado para escutá-la. Esse ambiente padronizado é iniciado no modo de linha de comando, facilitando o desenvolvimento, execução e ajustes diretamente no contêiner, proporcionando consistência e independência da máquina local.

O arquivo devcontainer.json define o ambiente de desenvolvimento necessário para o projeto em um contêiner Docker, permitindo que a configuração do ambiente de desenvolvimento seja consistente e fácil de reproduzir. Esse arquivo especifica o uso de um Dockerfile, que fornece uma imagem base com GCC e GDB instalados, garantindo que o contêiner seja adequado para a compilação e depuração de código C. Além disso,

configura o terminal padrão como `/bin/bash`, proporcionando uma interface familiar para desenvolvedores que trabalham com Linux. A configuração também inclui a pasta de trabalho principal como `/workspace`, onde os arquivos do projeto são copiados, permitindo que o código esteja imediatamente acessível quando o contêiner é iniciado.

```
1 {
2   "name": "C Server Dev Container",
3   "build": {
4     "dockerfile": "Dockerfile",
5     "context": ".."
6   },
7   "settings": {
8     "terminal.integrated.shell.linux": "/bin/bash"
9   },
10  "extensions": [
11    "ms-vscode.cpptools",
12    "ms-azuretools.vscode-docker"
13  ],
14  "workspaceFolder": "/workspace",
15  "forwardPorts": [8080]
16 }
```

**Listing 1:** Configurações do devcontainer.json

### 2.2.2 Criação do Makefile

O **Makefile** proposto automatiza a compilação e execução dos três componentes principais do projeto: o servidor (**server**), o receptor de mensagens (**receiver**) e o enviador de mensagens (**sender**). Cada executável tem uma regra de compilação dedicada que utiliza o compilador **gcc** com a flag **-Wall**, garantindo que avisos sejam mostrados durante a compilação para auxiliar na detecção de problemas. A regra **all** permite a compilação dos três executáveis com um único comando, simplificando o processo de construção do projeto. Há também regras específicas para a execução de cada programa, com **run-server** para iniciar o servidor, e **run-receiver** e **run-sender**, que executam o receptor e o enviador de mensagens com parâmetros personalizados para ID e IP, proporcionando flexibilidade na configuração de testes. Por fim, a regra **clean** remove os arquivos executáveis gerados, facilitando a limpeza do ambiente de trabalho quando necessário. Em conjunto, esse **Makefile** contribui para uma organização eficiente e fácil execução do projeto.

Dessa forma para executar o servidor e dos dois clientes desenvolvidos temos os comando:

**Programa 2:** Comando Makefile para executar o servidor e dos dois clientes desenvolvidos

```
make run-server
```



```
make run-receiver IP=127.0.0.1 ID=101  
make run-sender IP=127.0.0.1 ID=124
```

O comando dos clientes passa como parâmetros o IP do endereço que será conectado e o ID com que será inserido.

### 2.2.3 Servidor

Este programa implementa um servidor TCP simples em C, configurado para gerenciar múltiplos clientes simultaneamente e lidar com a comunicação entre eles. A comunicação segue um modelo em que os clientes podem se conectar ao servidor e enviar diferentes tipos de mensagens, como "OI" (para se identificar), "TCHAU" (para desconectar), e mensagens de broadcast ou privadas. O servidor escuta por conexões TCP na porta especificada e usa select para monitorar vários sockets simultaneamente, o que permite que ele gere conexões e eventos de entrada de dados de clientes de maneira eficiente, sem necessidade de múltiplas threads.

No início do programa, o servidor configura um socket para escutar em uma porta específica e aguarda conexões de clientes. Assim que um cliente se conecta, o servidor aceita a conexão e adiciona o cliente à lista de clientes conectados. Cada cliente é identificado por um UID (identificador único), que é armazenado em um array para referência. O programa então usa select em um loop contínuo, verificando se novos eventos surgem em algum dos sockets (tanto o socket principal, onde clientes se conectam, quanto os sockets de clientes conectados).

Para as mensagens, o servidor define diferentes tipos de comandos e responde adequadamente. Quando um cliente envia uma mensagem de tipo "OI", o servidor registra o UID do cliente e envia uma resposta de confirmação. Caso o cliente envie um "TCHAU", o servidor encerra a conexão e remove o cliente da lista. Para mensagens do tipo "MSG", o servidor verifica o UID de destino e decide se deve enviar a mensagem para todos os clientes (no caso de uma mensagem pública) ou para um cliente específico (no caso de uma mensagem privada).

Em pseudocódigo, a estrutura do fluxo principal do programa é mostrada:

```
inicializar servidor;
definir clientes como um array vazio;
while verdadeiro do
    aguardar atividade em algum socket usando select;
    if houver nova conexão then
        aceitar conexão e adicionar cliente à lista;
    end
    foreach cliente em clientes do
        if há mensagem recebida then
            if tipo da mensagem é "OI" then
                registrar UID do cliente;
                enviar confirmação ao cliente;
            end
            else if tipo da mensagem é "TCHAU" then
                encerrar conexão e remover cliente da lista;
            end
            else if tipo da mensagem é "MSG" then
                if destino é 0 (broadcast) then
                    enviar mensagem para todos os clientes;
                end
                else
                    enviar mensagem para cliente específico;
                end
            end
        end
    end
end
```

#### 2.2.4 Cliente Receiver

O programa receiver é um cliente que se conecta a um servidor TCP para receber mensagens. Seu principal objetivo é estabelecer uma conexão com o servidor, identificar-se enviando uma mensagem de "OI" com seu identificador único (ID), e então entrar em um loop contínuo aguardando mensagens do servidor. Essas mensagens podem ser públicas (enviadas a todos os clientes) ou privadas (destinadas especificamente ao receiver). O programa lida com diferentes tipos de mensagens, exibindo o conteúdo e a origem de cada uma para o usuário.

Além disso, o receiver implementa tratamento para sinais do sistema, como o SIGINT (gerado ao pressionar Ctrl+C), permitindo uma finalização graciosa do programa. Ao receber esse sinal, o receiver envia

uma mensagem de "TCHAU" ao servidor para informar sua desconexão e fecha o socket de comunicação, garantindo que os recursos sejam liberados corretamente e que o servidor esteja ciente da saída do cliente.

### 2.2.5 Cliente Sender

O programa sender é um cliente desenvolvido para enviar mensagens em um sistema de comunicação com um servidor via TCP. Ele permite que o usuário se conecte ao servidor utilizando um identificador único (entre 1001 e 1999) e o IP do servidor. Após a conexão inicial, o sender envia uma mensagem de saudação "OI" para se registrar, aguardando uma confirmação do servidor. A partir daí, ele entra em um loop que permite ao usuário especificar o destinatário da mensagem (um cliente específico ou todos) e o conteúdo da mensagem a ser enviada. Ao enviar uma mensagem, o sender imprime a confirmação no console. Em caso de desconexão ou encerramento (como ao receber um sinal de interrupção), ele envia uma mensagem "TCHAU" para avisar o servidor antes de encerrar a conexão de forma controlada.

### 2.2.6 Transmissão Periódica do Servidor

A transmissão periódica de mensagens do servidor envia para os clientes conectados, com intervalos definidos, neste caso, a cada 60 segundos. Essa transmissão periódica é implementada através de um temporizador (timer) configurado para disparar um sinal a cada intervalo. Quando o temporizador expira, o servidor dispara uma mensagem periódica para todos os clientes conectados. A mensagem periódica é enviada a cada cliente por meio de um loop que verifica quais clientes estão ativos e envia a mensagem de forma confiável para cada um deles.

A transmissão periódica funciona em sincronia com a rotina principal do servidor, que utiliza a função select para monitorar tanto as conexões de novos clientes quanto as mensagens de clientes já conectados. A estrutura de controle de tempo permite que o servidor identifique a expiração do temporizador e execute a transmissão periódica sem interromper o atendimento de novas conexões ou mensagens de clientes ativos. Essa abordagem garante que o servidor permaneça responsivo às atividades dos clientes e que as mensagens periódicas sejam entregues em tempo hábil.

### 3 Resultados

A Figura 1 mostra a execução do servidor e do cliente receiver no momento em que o ID = 24 se conecta:

**Figura 1:** Conexão entre servidor e cliente receiver com ID = 24



```

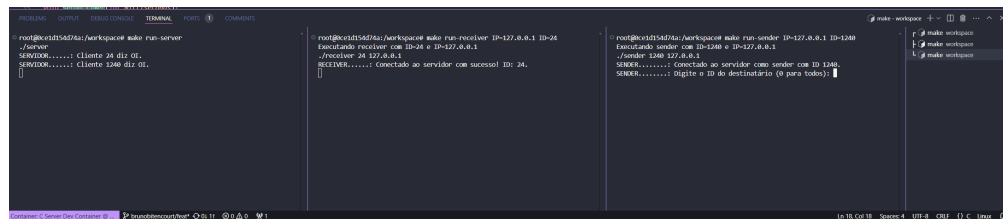
root@0ce1d154d74a:/workspace# make run-server
./server
SERVER.....: Cliente 24 diz oi.
[]

root@0ce1d154d74a:/workspace# make run-receiver IP=127.0.0.1 ID=24
Executando receiver com ID=24 e IP=127.0.0.1
./receiver 24 127.0.0.1
RECEIVER.....: Conectado ao servidor com sucesso! ID: 24.

```

A Figura 2 mostra a execução do servidor e do cliente sender no momento em que o o ID = 1240 se conecta:

**Figura 2:** Conexão entre servidor e cliente sender com ID = 1240



```

root@0ce1d154d74a:/workspace# make run-server
./server
SERVER.....: Cliente 24 diz oi.
SERVER.....: Cliente 1240 diz oi.
[]

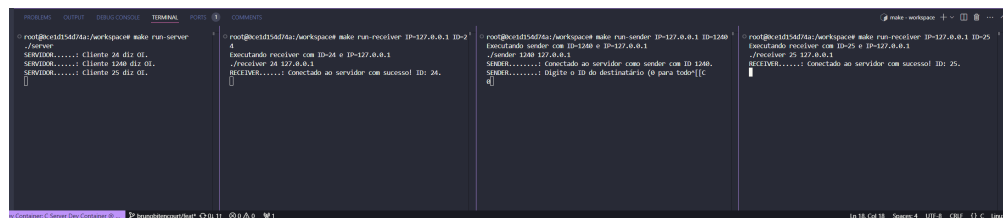
root@0ce1d154d74a:/workspace# make run-receiver IP=127.0.0.1 ID=24
Executando receiver com ID=24 e IP=127.0.0.1
./receiver 24 127.0.0.1
RECEIVER.....: Conectado ao servidor com sucesso! ID: 24.
[]

root@0ce1d154d74a:/workspace# make run-sender IP=127.0.0.1 ID=1240
Executando sender com ID=1240 e IP=127.0.0.1
./sender 1240 127.0.0.1
SENDER.....: Conectado ao servidor com sucesso com ID 1240.
SENDER.....: Digite o ID do destinatário (0 para todos):

```

A Figura 3 é mostrado a execução do servidor e de outro cliente receiver no momento em que o ID = 25 se conecta:

**Figura 3:** Conexão entre servidor e cliente receiver com ID = 25



```

root@0ce1d154d74a:/workspace# make run-server
./server
SERVER.....: Cliente 24 diz oi.
SERVER.....: Cliente 1240 diz oi.
SERVER.....: Cliente 25 diz oi.
[]

root@0ce1d154d74a:/workspace# make run-receiver IP=127.0.0.1 ID=24
Executando receiver com ID=24 e IP=127.0.0.1
./receiver 24 127.0.0.1
RECEIVER.....: Conectado ao servidor com sucesso! ID: 24.
[]

root@0ce1d154d74a:/workspace# make run-sender IP=127.0.0.1 ID=1240
Executando sender com ID=1240 e IP=127.0.0.1
./sender 1240 127.0.0.1
SENDER.....: Conectado ao servidor com sucesso com ID 1240.
SENDER.....: Digite o ID do destinatário (0 para todos):

root@0ce1d154d74a:/workspace# make run-receiver IP=127.0.0.1 ID=25
Executando receiver com ID=25 e IP=127.0.0.1
./receiver 25 127.0.0.1
RECEIVER.....: Conectado ao servidor com sucesso! ID: 25.

```

**Figura 6:** Encerramento de conexão de um cliente.

```

root@0ce1d154d74a:/workspace# make run-server
./server
SERVIDOR.....: Cliente 24 diz Oi.
SERVIDOR.....: Cliente 1240 diz Oi.
SERVIDOR.....: Cliente 25 diz Oi.
SERVIDOR.....: Cliente 1240 publica para todos.
SERVIDOR.....: Cliente 1240 publica para 24 : T
este envio unitario.
SERVIDOR.....: Cliente 24 diz TCHAU.
[]

root@0ce1d154d74a:/workspace# make run-receiver IP=127.0.0.1 ID=24
Executando receiver com ID=24 e IP=127.0.0.1
./receiver 24 127.0.0.1
RECEIVER.....: Conectado ao servidor com sucesso! ID: 24.
RECEIVER.....: Mensagem pública de 1240: Teste envio todos os clien
tes
RECEIVER.....: Mensagem privada de 1240 para 24: Teste envio unitar
io
^CRECEIVER.....: Encerrando o sender...
RECEIVER.....: Enviando TCHAU ao servidor.

```

Nesse momento temos três clientes conectados ao nosso servidor. A Figura 4 testa o cenário de envio para todos os clientes a mensagem "Teste envio todos os clientes". Observa-se que o envio das mensagens foi feito pelo sender e todos os clientes (ID 24 e 25) receberam a mensagem.

```

root@0ce1d154d74a:/workspace# make run-server
./server
SERVIDOR.....: Cliente 24 diz Oi.
SERVIDOR.....: Cliente 1240 diz Oi.
SERVIDOR.....: Cliente 25 diz Oi.
SERVIDOR.....: Cliente 1240 publica para todos.
[]

root@0ce1d154d74a:/workspace# make run-receiver IP=127.0.0.1 ID=24
Executando receiver com ID=24 e IP=127.0.0.1
./receiver 24 127.0.0.1
RECEIVER.....: Conectado ao servidor com sucesso! ID: 24.
RECEIVER.....: Mensagem pública de 1240: Teste envio todos os clien
tes
RECEIVER.....: Mensagem privada de 1240 para 24: Teste envio unitar
io
^CRECEIVER.....: Encerrando o sender...
RECEIVER.....: Enviando TCHAU ao servidor.

root@0ce1d154d74a:/workspace# make run-receiver IP=127.0.0.1 ID=25
Executando receiver com ID=25 e IP=127.0.0.1
./receiver 25 127.0.0.1
RECEIVER.....: Conectado ao servidor com sucesso! ID: 25.
RECEIVER.....: Mensagem pública de 1240: Teste envio todos os clien
tes
RECEIVER.....: Mensagem privada de 1240 para 25: Teste envio unitar
io
^CRECEIVER.....: Encerrando o sender...
RECEIVER.....: Enviando TCHAU ao servidor.

```

**Figura 4:** Envio de uma mensagem para todos os clientes

Na Figura 5 testa o cenário de enviar apenas para o cliente desejado (passando o número do ID 24 e a mensagem "Teste envio unitario"). Nesse caso, observa-se que o cliente com ID=24 recebeu a mensagem e o cliente com ID 25 não.

**Figura 5:** Envio de mensagem para apenas um cliente específico.

```

root@0ce1d154d74a:/workspace# make run-server
./server
SERVIDOR.....: Cliente 24 diz Oi.
SERVIDOR.....: Cliente 1240 diz Oi.
SERVIDOR.....: Cliente 25 diz Oi.
SERVIDOR.....: Cliente 1240 publica para todos.
SERVIDOR.....: Cliente 1240 publica para 24 : T
este envio unitario.
[]

root@0ce1d154d74a:/workspace# make run-receiver IP=127.0.0.1 ID=24
Executando receiver com ID=24 e IP=127.0.0.1
./receiver 24 127.0.0.1
RECEIVER.....: Conectado ao servidor com sucesso! ID: 24.
RECEIVER.....: Mensagem pública de 1240: Teste envio todos os clien
tes
RECEIVER.....: Mensagem privada de 1240 para 24: Teste envio unitar
io
^CRECEIVER.....: Encerrando o sender...
RECEIVER.....: Enviando TCHAU ao servidor.

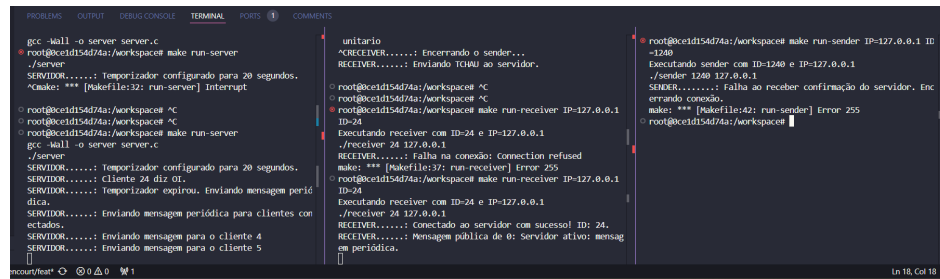
root@0ce1d154d74a:/workspace# make run-receiver IP=127.0.0.1 ID=25
Executando receiver com ID=25 e IP=127.0.0.1
./receiver 25 127.0.0.1
RECEIVER.....: Conectado ao servidor com sucesso! ID: 25.
RECEIVER.....: Mensagem pública de 1240: Teste envio todos os clien
tes
RECEIVER.....: Mensagem privada de 1240 para 25: Teste envio unitar
io
^CRECEIVER.....: Encerrando o sender...
RECEIVER.....: Enviando TCHAU ao servidor.

```

A Figura 6 mostra quando o cliente com ID=24 fecha a conexão com o servidor. Observa-se que o cliente envia a mensagem "TCHAU" para o servidor fazendo com que o mesmo desaloque seu ID. Após o envio dessa mensagem, foi colocado um período de 5 segundos para que o programa servidor consiga desalocar esse cliente, para que assim o cliente encerre sem causar impacto no servidor.

Em nossa análise de resultados, observou-se que a transmissão periódica de mensagens do servidor para os clientes não funcionou de maneira adequada. A implementação utilizando o temporizador e a função select para monitorar as atividades dos clientes e disparar mensagens periódicas encontrou conflitos no tratamento

Figura 7: Tentativa de retransmissão periódica



```

gcc -Wall -o server server.c
root@ceid154d74a:/workspace# make run-server
./server
SERVER.....: Temporizador configurado para 20 segundos.
^Cmake: *** [makefile:32: run-server] Interrupt

root@ceid154d74a:/workspace# ^C
root@ceid154d74a:/workspace# ^C
root@ceid154d74a:/workspace# make run-server
gcc -Wall -o server server.c
./server
SERVER.....: Temporizador configurado para 20 segundos.
SERVER.....: Cliente 24 diz Oi.
SERVER.....: Temporizador expirou. Enviando mensagem periódica.
SERVER.....: Enviando mensagem periódica para clientes conectados.
SERVER.....: Enviando mensagem para o cliente 4
SERVER.....: Enviando mensagem para o cliente 5

unitario
RECEIVER.....: Encerrando o sender...
RECEIVER.....: Enviando Tchau ao servidor.

root@ceid154d74a:/workspace# ^C
root@ceid154d74a:/workspace# make run-receiver IP=127.0.0.1 ID=24
Executando receiver com ID=24 e IP=127.0.0.1
./receiver 24 127.0.0.1
RECEIVER.....: Falha na conexão: Connection refused
make: *** [makefile:37: run-receiver] Error 255

root@ceid154d74a:/workspace# make run-receiver IP=127.0.0.1 ID=24
Executando receiver com ID=24 e IP=127.0.0.1
./receiver 24 127.0.0.1
RECEIVER.....: Conectado ao servidor com sucesso! ID: 24.
RECEIVER.....: Mensagem pública de Oi. Servidor ativo: mensagem periódica.

```

simultâneo de múltiplas operações de entrada e saída. Isso resultou em um comportamento inesperado, onde as mensagens periódicas não eram enviadas de maneira consistente, afetando a comunicação com os clientes. A Figura 7 mostra o problema em questão. Onde o servidor fica tentando enviar a mensagem periodicamente, porém apenas quando um novo cliente entra que o mesmo consegue fazer esse envio.

O conflito entre o temporizador e o select revelou que a solução proposta não era a mais apropriada para garantir uma transmissão periódica estável. A integração entre o controle de tempo e a gestão de múltiplas conexões não atingiu a eficiência esperada, destacando a necessidade de uma abordagem alternativa para assegurar a periodicidade das mensagens sem interferências. Essa análise reforça a importância de adaptar o design da solução para que o servidor consiga realizar múltiplas tarefas sem interrupções ou atrasos no envio de dados.

Uma possível solução para os problemas observados na transmissão periódica seria a utilização de threads, o que permitiria um tratamento mais eficiente e simplificado das tarefas de envio e recepção de mensagens. Com o uso de bibliotecas como pthreads, seria possível isolar a função de envio periódico em uma thread separada, enquanto outra thread gerenciaria as conexões e comunicações com os clientes. Essa abordagem evitaria os conflitos entre o temporizador e o select, permitindo que o envio das mensagens periódicas ocorresse de maneira independente, garantindo maior estabilidade e consistência na comunicação entre o servidor e os clientes.

## 4 Conclusão

O presente trabalho abordou o desenvolvimento de um sistema de comunicação em C utilizando sockets, composto por três componentes principais: um servidor, um cliente emissor de mensagens e um cliente receptor de mensagens. A implementação buscou criar um ambiente de troca de mensagens capaz de suportar múltiplos clientes simultâneos e possibilitar o envio de mensagens individuais e broadcasts. O sistema foi configurado em um contêiner Docker, o que assegura a compatibilidade do ambiente e facilita o processo de desenvolvimento e execução, tornando o código portátil e facilmente replicável em diferentes máquinas.

Um dos desafios enfrentados foi a implementação de transmissões periódicas de mensagens do servidor para todos os clientes. Esta funcionalidade, desenvolvida na branch `brunobitencourt/feat`, mostrou-se parcialmente funcional, mas apresentou alguns conflitos com o funcionamento do `select`, comprometendo o desempenho e a estabilidade da transmissão. Como continuidade deste trabalho, sugere-se o uso de threads para gerenciar a transmissão periódica de forma independente do fluxo principal de execução do servidor, o que poderá resolver os conflitos identificados e tornar o sistema mais robusto.

Para acesso ao código completo deste projeto, bem como à implementação do temporizador e transmissão periódica, acesse o repositório no GitHub: <https://github.com/brunogbitencourt/tpredes.git>. A implementação com transmissão periódica encontra-se disponível na branch `brunobitencourt/feat`.