Implementação em Python de um Sistema de Chat em Grupo Utilizando TCP e Multithreading

Dairon R. Silva[1], Filipe C. Silva[2], Gabriela M. O. Costa[3]

[1]Faculdade de Engenharia da Computação e Telecomunicações – Instituto de Tecnologia
Universidade Federal do Pará (UFPA) – Belém, PA – Brazil

dairon.silva@itec.ufpa.br, filipe.correa.silva@itec.ufpa.br,
gabriela.oliveira.costa@itec.ufpa.br

Abstract. *This report describes a Python Implementation of a Group Chat System Using TCP and Socketed Client/Server Multithreading. The application consists of a server that accepts multiple client connections. Each client sends messages to the server, which in turn relays it to all clients connected to it. This system demonstrates the fundamental concepts of network programming, including socket communication, multithreading, and message broadcasting.*

Resumo. *Este relatório descreve uma Implementação em Python de um Sistema de Chat em Grupo Utilizando TCP e Multithreading cliente/servidor com socket. A aplicação consiste em um servidor que aceita múltiplas conexões de clientes. Cada cliente envia mensagens ao servidor, que por sua vez, retransmite para todos os clientes conectados ao mesmo. Este sistema demonstra os conceitos fundamentais de programação de rede, incluindo comunicação via socket, multithreading e retransmissão de mensagens.*

1. Introdução

De início, é importante mencionar que a Implementação em Python de um Sistema de Chat em Grupo Utilizando TCP e Multithreading consiste em um servidor que aceita conexões de múltiplos clientes, que é análogo ao funcionamento de um hub. Nesse sentido, cada cliente pode enviar mensagens ao servidor, que as retransmite para todos os outros clientes conectados, e seu principal objetivo é funcionar como um chat em grupo. Diante do exposto, é possível afirmar que o servidor atuará como um centralizador de dados, disponibilizando seus recursos, organização e segurança, dessa forma, é possível que vários usuários acessem e manipulem ao mesmo tempo a aplicação, possibilitando que todos os dados fiquem sincronizados. Dessa maneira, a implementação possibilitará a troca de mensagens instantâneas pelos usuários do software.

Em conclusão, no chat em grupo, as funções do servidor são: criar um socket e vinculá-lo a um endereço IP e porta; aceitar conexões de clientes e criar uma nova thread para cada cliente; receber mensagens dos clientes e retransmiti-las para todos os outros clientes conectados. Por outro lado, as funções do cliente são: criar um socket e conectar-se ao servidor; iniciar uma thread para receber e enviar mensagens ao servidor em um loop contínuo.

2. Documentação do projeto

2.1. Topologia:

A topologia do programa é a da categoria estrela, que consiste em ter um centralizador e todos se conectarem à ele para realizar a troca de informações. Na implementação, o servidor opera como centralizador e gerencia todas as comunicações dos clientes conectados nele. Na topologia estrela, o servidor é o ponto central da rede, ele aceita conexões de múltiplos clientes e mantém uma lista de todos os clientes conectados, ou seja, cada cliente se conecta ao servidor utilizando o endereço IP e a porta especificada. Logo, os clientes enviam mensagens ao servidor, que as retransmite para todos os outros clientes conectados, isso garante que todos os clientes recebam as mensagens enviadas por qualquer outro cliente na rede.

A vantagem da topologia estrela é a facilidade de gerenciamento, pois como o servidor centraliza todas as comunicações, é mais fácil monitorar e controlar o tráfego de dados na rede. Além disso, se um cliente falhar, isso não afetará os outros clientes ou o servidor, proporcionando um determinado grau de isolamento de falhas. Esse tipo de topologia também é escalável, visto que permite que novos clientes sejam facilmente adicionados à rede conectando-se ao servidor. No entanto, a topologia estrela também apresenta algumas desvantagens, haja visto que o servidor é um ponto único de falha, se ele falhar, toda a rede ficará inoperante. Ademais, o servidor pode se tornar um gargalo, ou seja, um ponto no sistema, no qual a capacidade de processamento é limitada, se muitos clientes estiverem conectados e trocando mensagens simultaneamente pode afetar o desempenho da rede.

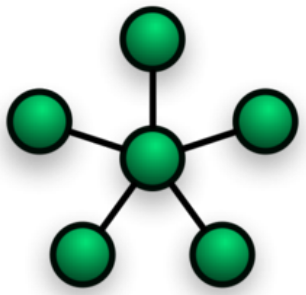


Figura 1. Ilustração da topologia estrela

2.2. Protocolo de Transporte

O TCP (Protocolo de Controle de Transmissão) é um protocolo de transporte que garante a entrega ordenada e livre de erros dos dados entre os dispositivos conectados, ele é fundamental para a comunicação entre o cliente e servidor na aplicação, pois na aplicação, o TCP é utilizado para estabelecer uma conexão confiável entre ambos. Isso ocorre, através do handshake, que é o processo pelo qual duas ou mais máquinas afirmam que reconhecem uma a outra e estão prontas para iniciar uma conexão. Quando um cliente deseja se conectar ao servidor, ele utiliza o endereço IP e a porta especificada

para iniciar uma conexão. O servidor, por sua vez, aceita essa conexão e adiciona o cliente à lista de clientes conectados.

Nesse viés, uma das principais características do TCP é a sua capacidade de garantir que os dados sejam entregues na ordem correta e sem perdas, isso é feito através de um processo de controle de fluxo e controle de congestionamento. O TCP fragmenta os dados em pacotes menores e os envia ao destino, ou seja, se algum pacote se perder no caminho, o TCP retransmite esse pacote, garantindo que todos os dados cheguem ao destino final. Além disso, o TCP utiliza um mecanismo de confirmação, no qual o receptor envia um aviso de recebimento ao remetente para confirmar que os dados foram recebidos corretamente. No entanto, se o remetente não receber essa confirmação dentro de um determinado período de tempo, ele retransmite os dados. Em suma, pode-se afirmar que esse processo garante a confiabilidade da comunicação.

No projeto, o TCP é utilizado para enviar e receber mensagens entre o cliente e o servidor. O cliente envia mensagens ao servidor utilizando a função **cliente_socket.send()**, e o servidor recebe essas mensagens utilizando a função **cliente_socket.recv()**. O servidor, então, retransmite as mensagens para todos os outros clientes conectados, garantindo que todos recebam as mensagens enviadas por qualquer cliente na rede.

Figura 2. Ilustração sobre o protocolo de controle de transmissão (TCP)

2.3 Transmissão de Dados

Do ponto de vista eletrônico, a transmissão de dados implantada na aplicação é a full-duplex, que consiste em ser um comunicação bidirecional, ou seja, A e B podem transmitir e receber dados ao mesmo tempo, no qual A e B seriam o usuário e o servidor, respectivamente. Na comunicação full-duplex, ambos os lados da conexão, podem enviar e receber dados ao mesmo tempo, isso é essencial para aplicações de chat, em que a interação em tempo real é crucial. Por exemplo, enquanto um cliente está enviando uma mensagem ao servidor, ele também pode estar recebendo mensagens de outros clientes através do servidor. Desse modo, isso garante uma experiência de comunicação fluida e contínua.

No programa, essa capacidade de comunicação simultânea é implementada através do uso de threads. No lado do cliente, a função **envia_mensagens** permite que o cliente envie mensagens ao servidor, enquanto a função **recebe_mensagens** permite que o cliente receba mensagens do servidor. Sob a perspectiva do servidor, a função

lidar_com_clientes permite que o servidor receba mensagens de um cliente específico, enquanto a função **broadcast** permite que o servidor retransmita essas mensagens para todos os outros clientes conectados. Logo, compreende-se que o servidor gerencia múltiplos clientes simultaneamente, cada um em sua própria thread, garantindo que todos os clientes possam enviar e receber mensagens ao mesmo tempo.

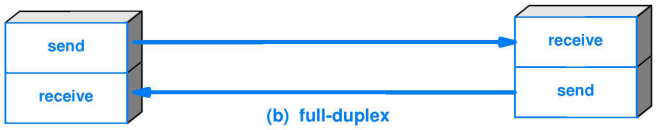


Figura 4. Ilustração da transmissão de dados full-duplex

2.4 Multithreading

Por definição é uma sequência de instruções que faz parte de um processo principal, cada processo é dividido em threads, que formam tarefas independentes, mas compartilham os recursos de forma simultânea. Ou seja, os threads formam conjuntos menores de instruções dentro de uma tarefa maior. Multithreading é o conceito em que dois ou mais threads são executados em um mesmo processo simultaneamente para aumentar a eficiência do sistema. Em consequência do compartilhamento como memória, arquivos, dados e outros, o multithreading consome menos recursos do que a execução de diversos processos simultâneos, tornando-se mais eficiente e econômico.

No projeto, esse conceito surge quando a função **threading.Thread(target=lidar_com_clientes, args=(cliente_socket,)).start()** é chamada do ponto de vista do servidor e depois surge novamente do ponto de vista do cliente em **threading.Thread(target=recebe_mensagens, args=(cliente_socket,)).start()**. As multithreading são utilizadas para que cada cliente possua uma thread independente, executando tarefas específicas.

3. Código Fonte

3.1 Servidor (Servidor.py)

```
import socket
import threading

# Configurações do servidor
HOST = '127.0.0.1'  # Endereço IP do servidor
PORT = 12345        # Porta do servidor

# Lista para armazenar os clientes conectados
clientes = []

# Função para lidar com as mensagens recebidas dos clientes
def lidar_com_clientes(cliente_socket):
    while True:
        try:
            mensagem = cliente_socket.recv(1024).decode('utf-8')
            if mensagem:
```

```
            print(f"Mensagem recebida de {cliente_socket.getpeername()}: {mensagem}")
            broadcast(mensagem, cliente_socket)
        else:
            remove(cliente_socket)
            break
    except:
        remove(cliente_socket)
        break

# Função para retransmitir mensagens para todos os clientes
def broadcast(mensagem, cliente_socket):
    for cliente in clientes:
        if cliente != cliente_socket:
            try:
                cliente.send(mensagem.encode('utf-8'))
                print(f"Mensagem retransmitida para {cliente.getpeername()}")
            except:
                remove(cliente)

# Função para remover clientes desconectados
def remove(cliente_socket):
    if cliente_socket in clientes:
        clientes.remove(cliente_socket)
        print(f"Cliente {cliente_socket.getpeername()} desconectado")

def main():
    # Configuração do socket do servidor
    servidor_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        servidor_socket.bind((HOST, PORT))
        servidor_socket.listen()

        print(f"Servidor iniciado no endereço IP: {HOST}, na porta: {PORT}")
    except:
        return print("\nNão foi possível iniciar o servidor!\n")

    while True:
        cliente_socket, endereco_do_cliente = servidor_socket.accept()
        clientes.append(cliente_socket)
        print(f"Conexão estabelecida com {endereco_do_cliente}")
        threading.Thread(target=lidar_com_clientes, args=(cliente_socket,)).start()
main()
```

3.2 Cliente (Cliente.py)

```
import socket
import threading

# Configurações do cliente
HOST = '127.0.0.1'  # Endereço IP do servidor
PORT = 12345        # Porta do servidor
```

```
# Função para receber mensagens do servidor
def recebe_mensagens(cliente_socket):
    while True:
        try:
            mensagem = cliente_socket.recv(1024).decode('utf-8')
            if mensagem:
                print(mensagem)
        except:
            print("Erro ao receber mensagem. Servidor desconectado\n")
            cliente_socket.close()
            break

def envia_menssagens(cliente_socket, usuario):

    # Loop para enviar mensagens ao servidor
    while True:
        try:
            mensagem = input().strip()
            if mensagem:  # Verifica se a mensagem não está vazia
                cliente_socket.send(f'<{usuario}> {mensagem}'.encode('utf-8'))
        except KeyboardInterrupt:
            print("Desconectando do servidor...")
            cliente_socket.close()
            break

def main():
    usuario = input("Usuário>> ")

    # Configuração do socket do cliente
    cliente_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        cliente_socket.connect((HOST, PORT))
        print(f"Conectado ao servidor no endereço IP: {HOST}, na porta: {PORT}")
    except ConnectionRefusedError:
        print(f"Não foi possível conectar ao servidor no endereço IP: {HOST}, na porta: {PORT}.
O servidor pode não estar em execução.")
        return

    # Inicia a thread para receber mensagens
    threading.Thread(target=recebe_mensagens, args=(cliente_socket,)).start()

    # Chama a função para enviar mensagens
    envia_menssagens(cliente_socket, usuario)

main()
```

3.3 Bibliotecas (Servidor e Cliente)

- **Socket**: permite estabelecer conexões e troca de dados entre um cliente e um servidor de forma eficiente e flexível.
- **Threading:** cria threads independentes para executar tarefas específicas, permitindo que o programa continue executando outras tarefas em segundo plano.

4. Servidor

4.1 Configurações e Listas

- **HOST**: Endereço IP no qual o servidor será executado. '127.0.0.1' é o IP local (localhost).
- **PORT**: Porta de comunicação que será utilizada pelo servidor.
- **clientes = []**: Lista os sockets dos clientes conectados

4.2 Função para lidar com as mensagens recebidas dos clientes

- No servidor, o comando '**cliente_socket.recv(1024)**' recebe a mensagem como uma sequência de bytes e o **'decode('utf-8')'** converte os bytes em uma string legível.
- **broadcast(mensagem, cliente_socket)**: Transmite a mensagem recebida para todos os outros clientes conectados.
- **remove(cliente_socket)**: Remove o cliente da lista de 'clientes' conectados se o socket for fechado ou se ocorrer um erro.

4.3 Função para retransmitir mensagens para todos os clientes

- **cliente.send(mensagem.encode('utf-8')):** Envia a mensagem para o cliente atual, codificada em UTF-8.
- **remove(cliente):** Remove da lista o cliente que não conseguiu receber a mensagem.
- **broadcast**: permite que as mensagens de um cliente sejam transmitidas para todos os outros clientes conectados na rede.

4.4 Função para remover clientes desconectados

- **clientes.remove(cliente_socket)**: Remove o socket do cliente da lista

4.5 Configuração do socket do servidor

- **socket.AF_INET** indica que a conexão irá utilizar o protocolo IPv4.
- **socket.SOCK_STREAM** indica que será uma conexão TCP.
- **bind((HOST, PORT))** associa o socket a um endereço IP (HOST) e uma porta (PORT) específicos, tornando o servidor acessível aos clientes.

- **listen()** coloca o servidor em modo de escuta, esperando por conexões de clientes.
- **servidor_socket.accept()**: Aceita uma nova conexão de cliente, retornando um novo socket para se comunicar com o cliente.
- **threading.Thread(target=lidar_com_clientes, args=(cliente_socket,)).start()**: Cria uma nova thread para cada cliente conectado, chama a função lidar_com_clientes que gerencia as mensagens recebidas.

5. Cliente

5.1 Configurações do cliente

- **HOST**: Endereço IP do servidor no qual o cliente irá se conectar.
- **PORT**: Porta de comunicação do servidor.

5.2 Função para receber mensagens do servidor

- **mensagem = cliente_socket.recv(1024).decode('utf-8'):** cliente recebe mensagem do servidor no formato bytes e converte para uma string legível utilizando UTF-8.
- **print(mensagem):** Exibe as mensagens na tela.
- **client_socket.close( ):** Se ocorrer um erro na comunicação, o cliente é informado (ex: Erro ao receber mensagem. Servidor desconectado) e a conexão é encerrada.

5.3 Loop para enviar mensagens ao servidor

- **mensagem = input().strip():** input(): Exibe texto na tela e solicita que o usuário insira a informação desejada , strip(): remove espaços em branco do início e do final da string.
- **cliente_socket.send(f'<{usuario}> {mensagem}'.encode('utf-8'))**: Envia a mensagem para o servidor com a informação solicitada.
- **except KeyboardInterrupt:** permite que o usuário encerre a conexão com o servidor clicando **'Ctrl+C'**.
- **usuario = input("Usuário>> ")**: Solicita ao usuário o nome.

5.4 Configuração do socket do cliente

- **cliente_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM):** Cria o socket para o cliente utilizando protocolo IPv4 e conexão TCP
- **cliente_socket.connect((HOST, PORT)):** conecta o cliente ao servidor que é identificado pelo IP e a porta.

5.5 Inicia a thread para receber mensagens

- **threading.Thread(target=recebe_mensagens, args=(cliente_socket,)).start():** cria uma nova thread, permite que duas ou mais threads trabalhem simultaneamente.
- A função **‘recebe_mensagens’** é executada na nova thread, por consequência, o cliente pode continuar enviando mensagens enquanto recebe novas mensagens, sem que uma tarefa bloqueie a outra.
- **args=(cliente_socket,):** informa que quando a thread for iniciada, o argumento **‘cliente_socket’** deve ser passado para a função **‘recebe_mensagens’** e assim, pode receber as mensagens do servidor**.**

5.6 Chama a função para enviar mensagens

- **envia_mensagens(cliente_socket, usuario):** Chama a função no qual possibilita que o usuário consiga enviar mensagens.

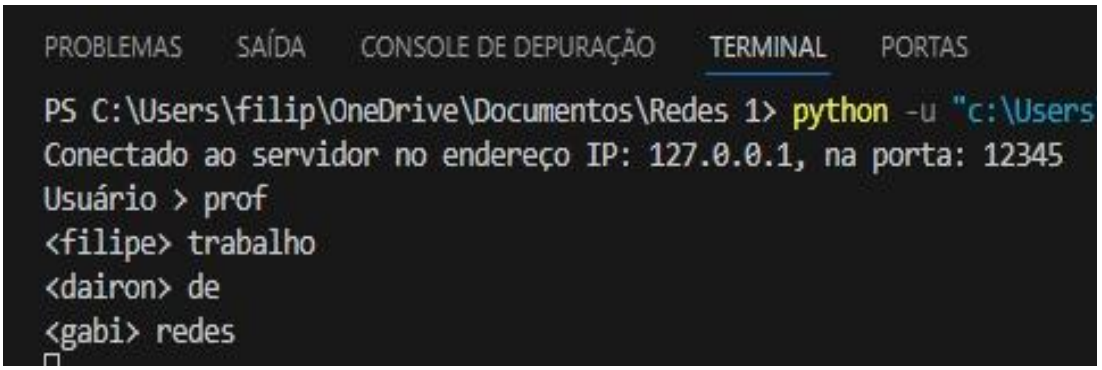


Figura 4. Funcionamento do cliente no terminal da máquina

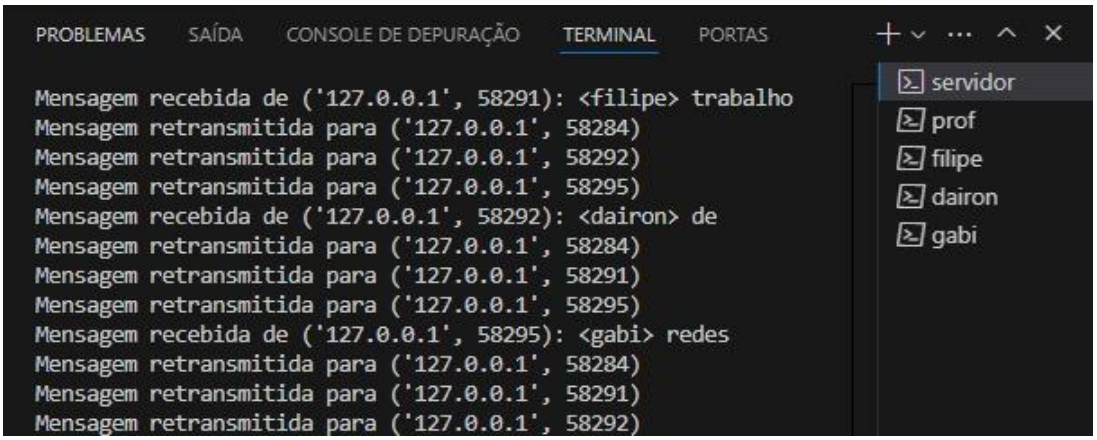


Figura 5. Funcionamento do servidor no terminal da máquina

6. Complexibilidade do cenário

Tendo em vista as múltiplas conexões, sincronização e troca de mensagens simultaneamente do projeto, é possível ressaltar algumas complexidades encontradas na implementação:

1. **O gerenciamento de conexões:** o servidor é capaz de gerenciar múltiplas conexões simultâneas, garantindo que todos os clientes recebam as mensagens no menor tempo possível.
2. **Estabilidade:** à medida que o número de usuários cresce, o servidor é escalonado para alocar mais conexões sem perder sua eficiência para com os outros usuários já conectados nele.
3. **Latência:** a função broadcast do servidor é executada toda vez que um cliente enviar uma mensagem, visando diminuir a latência, para que a experiência de troca de mensagens seja fluida.

7. Dificuldades encontradas

- Rodar vários códigos-clientes de uma vez em uma máquina
- Implementar os tratamentos de erros

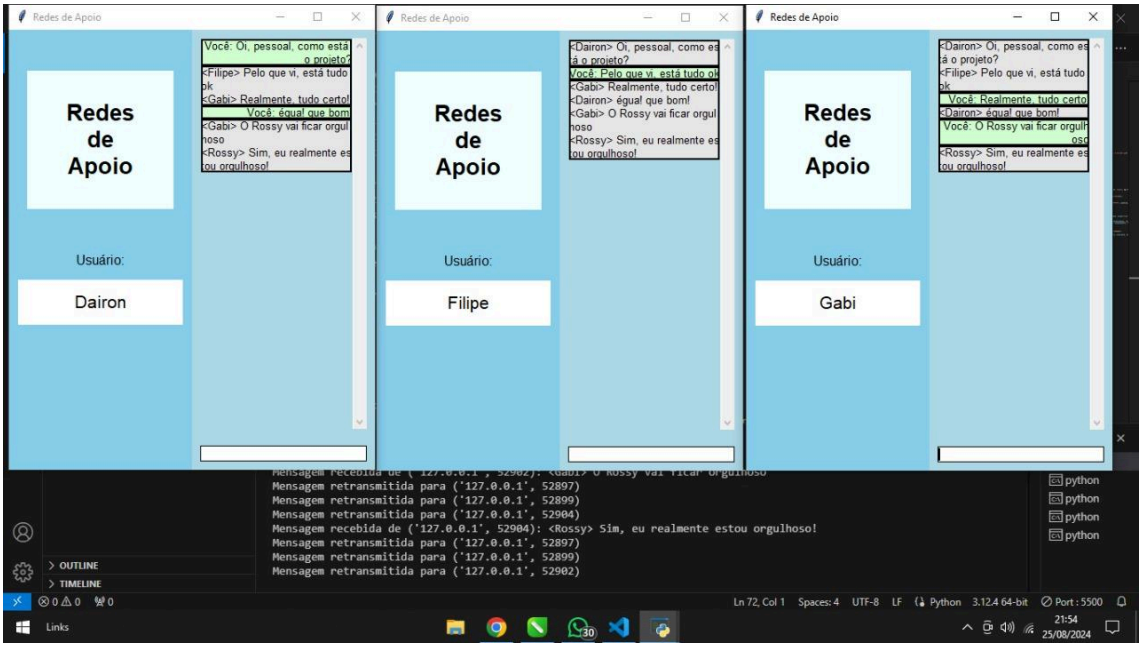
8. Referências

TORRES, Gabriel. Redes de Computadores: Curso Completo. 1° Edição. Av. Paris, 571 - Bonsucesso: Gisella Narcisi, 2001.

KUROSE, James, ROSS, Keith. Redes de computadores e a Internet. 8° Edição. Pearson Universidades: 2013.

"Cliente-Servidor, Uma Estrutura Lógica Para a Computação Centralizada." *Cliente-Servidor, Uma Estrutura Lógica Para a Computação Centralizada*, www.controle.net/faq/cliente-servidor-uma-estrutura-para-a-computacao-centralizada#:~:text=Uma%20estrutura%20cliente%2Dservidor%20%C3%A9.

9. Interface da aplicação (extra)



Biblioteca utilizada: Tkinter