Configurações Necessárias:

1- Instalar Python.

2- Baixar RADMIN VPN em todos os dispositivos.

(Nome da rede: Projeto Final – Chat / Senha: 123456)

Testar dando um ping em cada dispositivo.

3- Executar o servidor:

No dispositivo que vai rodar o servidor, abrir o cmd, ir até a pasta onde está o arquivo .py do servidor e executar: pyhton servidor.py

3- Executar os clientes:

No dispositivo que vai rodar o cliente, abrir o cmd, ir até a pasta onde está o arquivo .py do cliente e executar: pyhton cliente.py

Requisitos:

1- Comunicação Entre Múltiplos Dispositivos:

O sistema permite que diversos dispositivos (incluindo o servidor) estejam conectados simultaneamente, a comunicação é feita usando VPN.

2- Mensagem Broadcast (Quando um cliente se conectar, todos devem receber uma mensagem informando)

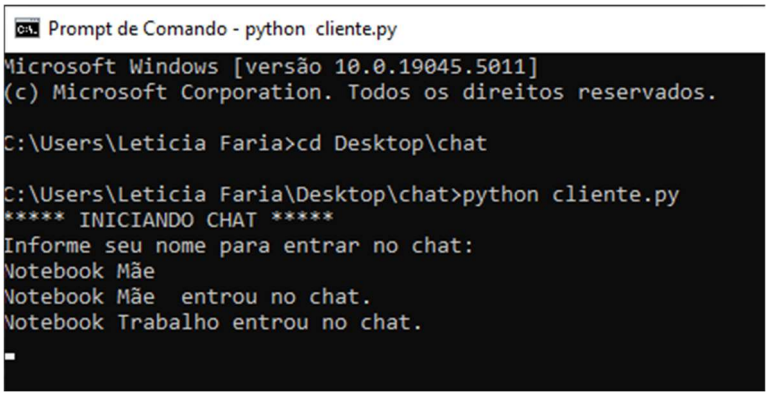
Servidor.py: Adicionei uma lista para armazenar os clientes conectados. Implementei uma função (broadcast) para enviar mensagens para todos os clientes conectados, também utilizei essa função para notificar todos os clientes quando um novo usuário entra no chat e também para enviar mensagens para todos os participantes.

Cliente.py: Adicionei thread para receber mensagens do servidor de forma contínua e exibir elas no terminal.

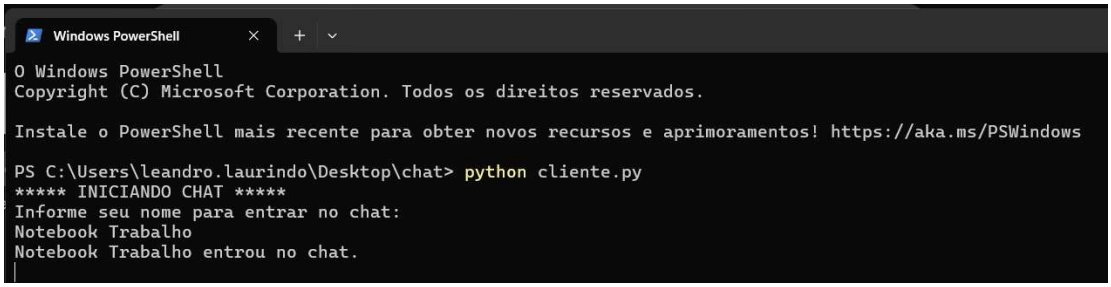
Servidor:

```
Microsoft Windows [versão 10.0.22631.4317]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\leand>cd desktop/chat

C:\Users\leand\Desktop\Chat>python servidor.py
O servidor 26.253.198.232:9999 está aguardando conexões...
Conexão com sucesso com Notebook Mãe  : ('26.228.82.64', 57186)
Conexão com sucesso com Notebook Trabalho : ('26.248.193.183', 57881)
```

Notebook 1 (primeiro a entrar no chat):



Notebook 2:

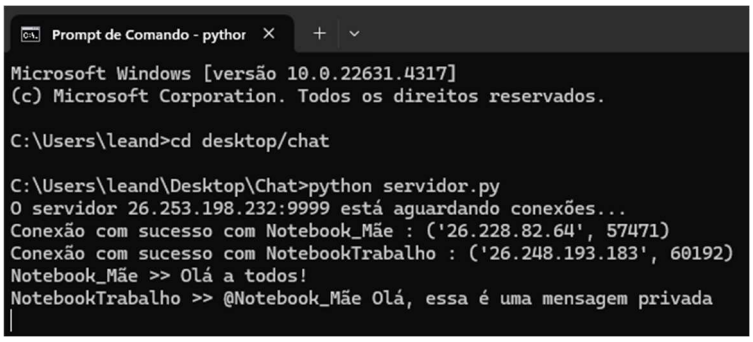


3- Mensagem Unicast (Mensagem privada para um usuário no chat)

Servidor.py: Adicionei uma condição na função de receber dados que verifica se a mensagem começa com @, se começar, ela é uma mensagem Unicast. Também criei uma função que formata a mensagem e envia para o destinatário certo. Como estava com muito erro nessa parte do nome, adicionei um loop de tratamento de erro que ao entrar no servidor, ele não permite e informa que o nome precisa ser separado por '_' ou precisa ser tudo junto.

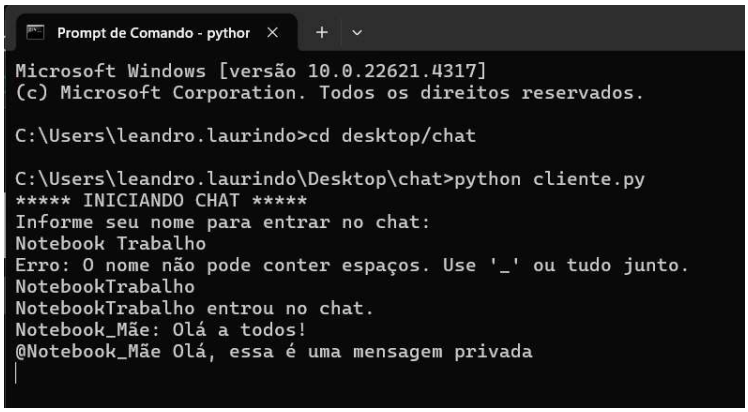
Cliente.py: Não precisei mexer.

Servidor:



Notebook 1 (primeiro a entrar):

Notebook 2:

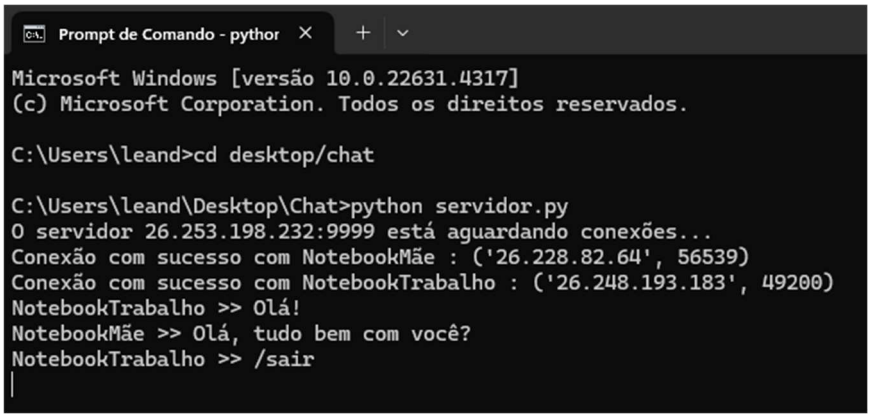


4- Comando Para Sair (Comando para o cliente sair do chat e exibir para os outros que ele saiu)

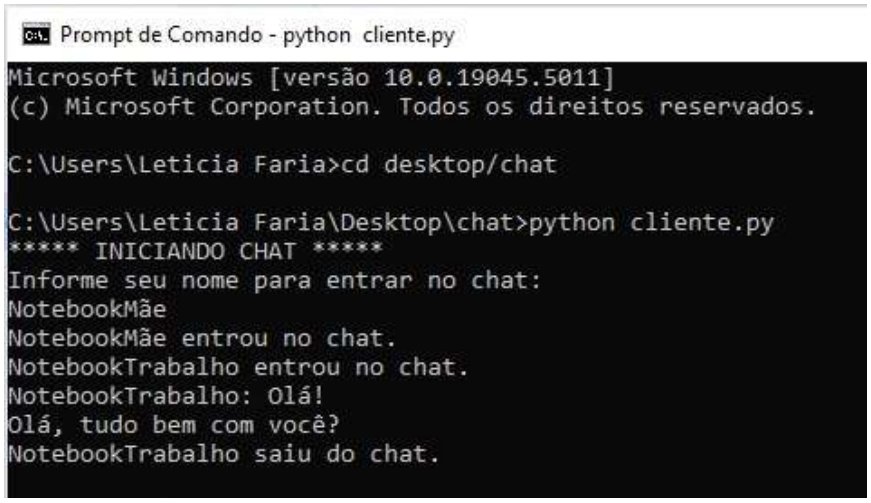
Servidor.py: Adicionei uma condição na função de receber dados que verifica se a mensagem enviada pelo cliente é o comando /sair. Se for, o servidor remove o cliente da lista de conectados, encerra sua conexão e envia uma mensagem de broadcast informando a todos os outros usuários que o cliente saiu do chat.

Cliente.py: Adicionei uma verificação para o comando /sair. Quando o cliente digita esse comando, ele envia uma mensagem ao servidor para notificar sua saída, fecha a conexão e encerra o programa.

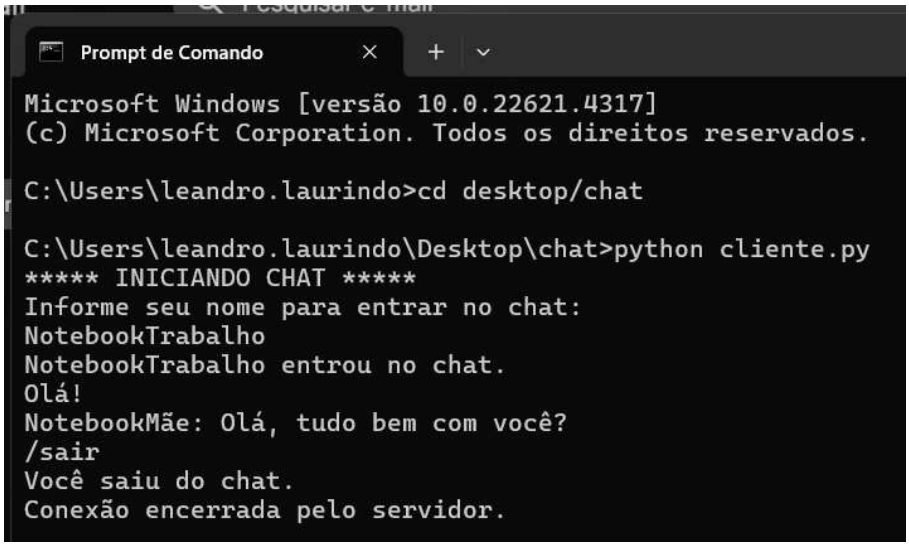
Servidor:



Notebook 1 (primeiro a entrar):



Notebook 2



5- Tratamento de Erros

Servidor.py: Implementei tratamento de exceções nas funções críticas para lidar com falhas de conexão ou envio de mensagens. Validei dados de entrada, como nomes, e garanti que clientes desconectados sejam removidos da lista.

Cliente.py: Adicionei tratamento de erros ao enviar e receber mensagens. Caso o servidor encerre a conexão ou ocorra uma falha, o cliente é desconectado de forma segura com uma mensagem explicativa.

Foram tratados os seguintes erros:

- Nome Inválido (O servidor valida e rejeita nomes que contêm espaços e envia mensagens de erro ao cliente para corrigir)
- Formato de Mensagem Inválido (Detecta mensagens unicast sem conteúdo ou formato errado e envia mensagens de erro ao cliente)
- Erro na Conexão com o Servidor (Captura falhas ao tentar conectar e exibe mensagem de erro)
- Desconexão do Servidor (Detecta quando o servidor encerra a conexão e fecha o socket e informa o usuário)

Utilização de Sockets:

Os sockets foram a base para a comunicação entre o cliente e o servidor neste projeto. Eles são utilizados para estabelecer, manter e gerenciar conexões em uma arquitetura de rede TCP/IP

1- Configuração Inicial:

Servidor: Especificamos que o socket usará o protocolo IPv4 e TCP e vinculamos o socket ao endereço IP e porta definidos, permitindo que o servidor receba conexões.

```
# Criamos o socket do servidor
try:
    socket_server = sock.socket(sock.AF_INET, sock.SOCK_STREAM)
    # Fazemos o BIND
    socket_server.bind((HOST, PORTA))
    # Entramos no modo escuta (LISTEN)
    socket_server.listen()
```

Cliente: No cliente estabelecemos a conexão com o servidor, especificando o IP e a porta.

```
# Criamos o socket do cliente
socket_cliente = sock.socket(sock.AF_INET, sock.SOCK_STREAM)

# Conecta ao servidor
try:
    socket_cliente.connect((HOST, PORTA))
    print(5 * "*" + " INICIANDO CHAT " + 5 * "*")
except Exception as e:
    print(f"Erro ao conectar ao servidor: {e}")
    exit(1)
```

2- Gerenciamento de conexões:

Servidor: Aceitamos as conexões e para cada cliente conectado, uma nova Thread é criada

```
# Loop principal para recebimento de clientes
while True:
    try:
        sock_conn, ender = socket_server.accept()
        # Nesse ponto temos uma conexão com um cliente
        # Thread para lidar com o cliente
        thread_cliente = threading.Thread(target=receber_dados, args=(sock_conn, ender))
        thread_cliente.start()
    except Exception as e:
        print(f"Erro ao aceitar nova conexão: {e}")
```

Cliente: Uma Thread separada é utilizada para receber mensagens do servidor enquanto o cliente permanece interativo para enviar mensagens:

```
# Thread para receber mensagens do servidor
thread = threading.Thread(target=receber_mensagens, args=(socket_cliente,))
thread.start()
```

3- Comunicação Entre Cliente e Servidor:

Servidor e Cliente: Para receber mensagens, utilizam o recv. Para enviar mensagens, utilizam o sendall.

```
while True:
    try:
        mensagem = sock_conn.recv(1024).decode()
```

```
sucesso = unicast(conteudo, destinatario, nome)
if not sucesso:
    sock_conn.sendall("Usuário não encontrado.".encode())
```

Broadcast no Servidor:

O broadcast é implementado para enviar mensagens a todos os clientes conectados, exceto o remetente. Ele é gerenciado pela função broadcast

Funcionamento:

A função recebe a mensagem e o remetente e em seguida envia a mensagem a todos os clientes da lista, exceto ao remetente.

```
for cliente, _ in clientes:
    if cliente != remetente:
        try:
            cliente.sendall(mensagem.encode())
```

Tratamento de erros: Caso um cliente desconecte, ele é removido da lista.

```
except Exception as e:
    print(f"Erro ao enviar mensagem para {cliente}: {e}")
    cliente.close()
    remover_cliente(cliente)
```

Uso no código:

```
# Notificar todos os clientes sobre a entrada
broadcast(f"{nome} entrou no chat.")
```

Threads no Servidor:

Threads foram fundamentais para implementar a comunicação simultânea entre o cliente e o servidor, permitindo que as mensagens fossem enviadas e recebidas em tempo real. Cada thread gerencia uma tarefa específica, como ouvir mensagens de outros clientes no servidor ou processar as mensagens recebidas pelo cliente.

Servidor: Quando o servidor aceita uma conexão, ele cria uma nova thread para gerenciar o cliente.

```
# Loop principal para recebimento de clientes
while True:
    try:
        sock_conn, ender = socket_server.accept()
        # Nesse ponto temos uma conexão com um cliente
        # Thread para lidar com o cliente
        thread_cliente = threading.Thread(target=receber_dados, args=(sock_conn, ender))
        thread_cliente.start()
    except Exception as e:
        print(f"Erro ao aceitar nova conexão: {e}")
```

Cada thread criada executa a função “receber_dados”, que recebe mensagens do cliente usando recv, processa as mensagens (broadcast ou Unicast) e remove o cliente e encerra sua thread ao desconectar.

```
def receber_dados(sock_conn, endereco):
    """Recebe mensagens de um cliente e gerencia sua comunicação."""
    try:
        # Loop para validar o nome do cliente
        while True:
            nome = sock_conn.recv(50).decode().strip()
            if " " in nome:
                sock_conn.sendall("Erro: O nome não pode conter espaços. Use '_' ou tudo junto.".encode())
            else:
                break  # Nome válido, sai do loop

        print(f"Conexão com sucesso com {nome} : {endereco}")

        # Adicionar o cliente à lista
        clientes.append((sock_conn, nome))

        # Notificar todos os clientes sobre a entrada
        broadcast(f"{nome} entrou no chat.")

        while True:
            try:
                mensagem = sock_conn.recv(1024).decode()
                if not mensagem:
                    raise ConnectionError("Conexão perdida com o cliente.")

                print(f"{nome} >> {mensagem}")

                if mensagem.lower() == "/sair":
                    # Cliente quer sair do chat
```

Cliente: Uma thread separada é usada para receber mensagens do servidor.

```
# Thread para receber mensagens do servidor
thread = threading.Thread(target=receber_mensagens, args=(socket_cliente,))
thread.start()
```

Função “receber_mensagens” recebe mensagens do servido usando recv

```
def receber_mensagens(socket_cliente):
    """Recebe mensagens do servidor e exibe no terminal."""
    while True:
        try:
            mensagem = socket_cliente.recv(1024).decode()
            if not mensagem:
                raise ConnectionError("Conexão encerrada pelo servidor.")
            print(mensagem)
        except (ConnectionError, Exception) as e:
            print(f"Erro na conexão: {e}")
            socket_cliente.close()
            break
```

Enquanto uma thread de recepção está ativa, o cliente permanece interativo, permitindo que o cliente envie mensagens sem bloquear a recepção:

```
# Loop para envio de mensagens
while True:
    try:
        mensagem = input('')
        if mensagem.lower() == "/sair":
            socket_cliente.sendall(mensagem.encode())  # Envia o comando para o servidor
            print("Você saiu do chat.")
            socket_cliente.close()
            break
        socket_cliente.sendall(mensagem.encode())
    except Exception as e:
        print(f"Erro ao enviar mensagem: {e}")
        socket_cliente.close()
        break
```

Resumindo:

Servidor: Cria uma thread para cada cliente conectado, garantindo comunicação simultânea.

Cliente: Utiliza uma thread para receber mensagens do servidor sem bloquear o envio de mensagens.