
INTRODUÇÃO

A camada de transporte no modelo ISO/OSI é crucial para garantir a transferência confiável de dados entre aplicações em diferentes dispositivos. No contexto do protocolo de transmissão TCP/IP, a camada de transporte é responsável por gerenciar a comunicação entre processos de *software* executando em dispositivos de rede. As aplicações podem usar o protocolo de transporte TCP para garantir que os dados sejam entregues de forma confiável e em ordem, com verificação de erros e retransmissões automáticas em caso de perda de pacotes.

Além disso, a camada de transporte também permite que as aplicações controlem o fluxo de dados durante a transmissão, evitando congestionamentos e perdas de pacotes. Isso é especialmente importante em redes de alta velocidade e alta demanda, onde o tráfego de dados pode ser intenso e a rede pode estar sujeita a interrupções e atrasos. O protocolo de transporte UDP, por outro lado, é mais adequado para aplicações que exigem transferência rápida e sem confirmação de recebimento, como streaming de vídeo ou áudio. Em resumo, a camada de transporte é essencial para garantir a confiabilidade, eficiência e segurança da comunicação de dados em redes de computadores, e sua implementação em conjunto com sockets em Python permite a criação de aplicações de rede sofisticadas e robustas.

O uso de *sockets* em Python permite a implementação de comunicação de rede orientada a conexão, em que as aplicações podem estabelecer e manter uma conexão confiável e bidirecional, permitindo a transferência de grandes volumes de dados com segurança e eficiência.

Exemplos de aplicações utilizando *sockets* em Python serão abordados nos exemplos ao longo do capítulo.

SOCKETS

Sockets são uma interface de programação de aplicativos (API) que permite a comunicação entre processos de *software* em diferentes dispositivos em uma rede de computadores. Eles são amplamente usados em aplicações de rede, incluindo navegadores *web*, clientes de e-mail, jogos *online*, aplicativos de mensagens instantâneas e muitos outros. Além disso, eles fornecem uma maneira fácil e flexível de estabelecer conexões entre diferentes dispositivos e permitir a transferência de dados entre eles.

Por exemplo, em um navegador *web*, quando você digita um endereço de site e clica em "ir", o navegador usa um *socket* para estabelecer uma conexão com o servidor que hospeda o site. Os *sockets* são usados para enviar e receber dados entre o navegador e o servidor, permitindo que o navegador exiba o conteúdo do site em sua tela. Da mesma forma, em um cliente de e-mail, os *sockets* são usados para conectar o cliente ao servidor de e-mail e permitir a transferência de mensagens entre eles.

Outro exemplo comum é em aplicativos de jogos online, onde os *sockets* são usados para permitir que vários jogadores se conectem e interajam em tempo real. Os jogadores enviam e recebem dados através deles, permitindo que esses jogadores se comuniquem uns com os outros e atualizem o estado do jogo em tempo real. Os *sockets*

também são usados em aplicativos de mensagens instantâneas para permitir a troca de mensagens entre usuários em diferentes dispositivos, como será abordado no exemplo prático do tópico abaixo.

Em resumo, os *sockets* são uma parte essencial da infraestrutura de rede que permite a comunicação de dados entre dispositivos em uma rede. Eles são amplamente usados em aplicações do dia a dia e fornecem uma maneira fácil e flexível de estabelecer conexões e transferir dados entre diferentes dispositivos.

EXEMPLO 1

CHAT DE MENSAGENS

Um exemplo de aplicação simples de implementação de um chat usando *sockets* TCP em Python pode ser feito com poucas linhas como mostrado abaixo. Nesta aplicação, os usuários podem se conectar a um servidor e enviar mensagens para outros usuários conectados.

Dê uma olhada no primeiro código apresentado abaixo, que diz respeito ao código de aplicação do lado servidor.

SERVIDOR.PY

```
import socket
import threading

# Cria um socket TCP/IP
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM )

# Define o endereço e a porta que o socket vai se vincular
# O endereço dos sockets em python é realizado via tupla (IP, PORT)
IP, PORT = 'localhost', 8000

print( f'Iniciando servidor em {IP}:{PORT}' )
server_socket.bind( (IP, PORT) )

# Escuta as conexões definindo o número de clientes que podem se conectar
server_socket.listen(5)

# Lista para armazenar os clientes conectados
clients = []

# Função para enviar uma mensagem para todos os clientes conectados
def broadcast( message ):
    for client in clients:
        client.send(message)
```

```
# Função para lidar com as conexões de clientes
def handle_client( client_socket, client_address ):
    # Adiciona o cliente à lista de clientes conectados
    clients.append(client_socket)
    while True:
        try:
            # Recebe a mensagem do cliente
            message = client_socket.recv(1024)
            if message:
                # Envia a mensagem para todos os clientes conectados
                broadcast(message)
            else:
                # Remove o cliente da lista de clientes conectados
                clients.remove( client_socket )
                client_socket.close()
                break
        except:
            # Remove o cliente da lista de clientes conectados
            # caso ocorra algum erro
            clients.remove(client_socket)
            client_socket.close()
            break

while True:
    print( 'Aguardando conexão...' )

    # Aceita a conexão do cliente
    client_socket, client_address = server_socket.accept()
    print( f'Cliente conectado: {client_address[0]}:{client_address[1]}' )

    # Inicia uma thread para lidar com a conexão do cliente
    # Roda cada solicitação paralelamente
    client_thread = threading.Thread(
        target = handle_client,
        args = ( client_socket, client_address )
    )
    client_thread.start()
```

Neste exemplo, criamos um servidor que aceita conexões de clientes e lida com as mensagens que eles enviam. O servidor escuta as conexões e, quando um cliente se conecta, cria uma nova Thread para lidar com essa conexão. Isso permite que o servidor lide com vários clientes ao mesmo tempo.

A função *broadcast* é usada para enviar uma mensagem para todos os clientes conectados. Ele percorre a lista de clientes e envia a mensagem para cada um, isso é necessário de ser feito, uma vez que o protocolo TCP não possui nativamente a função de entrega de mensagens em *broadcast*. A função *handle_client* é usada para lidar com a

conexão de um cliente. Ele recebe a mensagem do cliente e, em seguida, usa a função *Broadcast* para enviar a mensagem para todos os outros clientes conectados.

Para testar a aplicação, você pode executar o código acima em um terminal e aguardar novas conexões de clientes na aplicação. Em seguida, usar outro terminal ou outra instância do Python para se conectar ao servidor como cliente. Para se conectar, basta executar o seguinte código do lado Cliente:

CLIENTE.PY

```
import threading
import socket

# Cria um socket TCP/IP
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Define o endereço e a porta do servidor
IP, PORTA = 'localhost', 8000

# Conecta ao servidor
client_socket.connect( (IP, PORTA) )

# Função que fica aguardando o recebimento de novas mensagens
def listen( socket : socket ):
    while True:
        data = socket.recv(1024)
        print( data.decode(), end = '\n>' )

# Inicia uma Thread para ficar ouvindo o servidor
client_listen = threading.Thread(
    target = listen,
    args = (client_socket, )
)
client_listen.start()

while True:
    # Lê a mensagem do usuário
    message = input('> ')
    # Envia a mensagem para o servidor
    client_socket.send( message.encode('utf-8') )
```

Ao executar o código acima, você deve ser capaz de digitar mensagens e enviá-las para o servidor. O servidor irá retransmitir as mensagens para todos os outros clientes conectados. Dê uma boa olhada no código acima antes de prosseguir, para que seja possível entender cada linha de código da execução.

Os códigos usados acima estão disponíveis no repositório público do GitHub pelo link https://github.com/iOsnaaente/Monitoria_Redes-Industriais e estão presentes no Exemplo 1 – Chat.

TESTANDO O CHAT

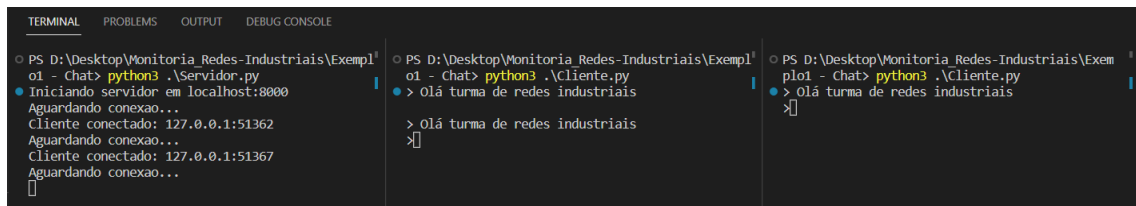
Para executar a aplicação, primeiro abre-se o script `Servidor.py` para que se cria uma porta de aplicação no endereço definido pelo socket. Nesse exemplo o endereço definido foi definido como:

```
# Define o endereço e a porta do servidor
IP, PORTA = 'localhost', 8000

# Conecta ao servidor
client_socket.connect( (IP, PORTA) )
```

O IP associado como *localhost* é um endereço usado para se referir ao endereço IP da própria máquina e geralmente ele vale `'127.0.0.1'`. Isso é útil para testar aplicativos em desenvolvimento sem precisar de uma conexão de rede real.

Com o servidor aberto, pode-se então executar os *scripts* definidos em `Cliente.py`. Nessa etapa, pode-se abrir quantos *scripts* de cliente se quiser, pois cada cliente conectado representa uma pessoa utilizando o chat. Na execução abaixo, a esquerda temos o terminal do lado servidor, que conecta dois clientes no endereço `127.0.0.1` (*localhost*) nas portas `51362` para o cliente 1 e porta `51367` para o cliente 2.



```
o PS D:\Desktop\Monitoria_Redes-Industriais\Exemplar> python3 .\Servidor.py
o1 - Chat> python3 .\Servidor.py
● Iniciando servidor em localhost:8000
Aguardando conexao...
Cliente conectado: 127.0.0.1:51362
Aguardando conexao...
Cliente conectado: 127.0.0.1:51367
Aguardando conexao...
[]

o PS D:\Desktop\Monitoria_Redes-Industriais\Exemplar> python3 .\Cliente.py
o1 - Chat> python3 .\Cliente.py
● > Olá turma de redes industriais
> Olá turma de redes industriais
>[]

o PS D:\Desktop\Monitoria_Redes-Industriais\Exemplar> python3 .\Cliente.py
p1o1 - Chat> python3 .\Cliente.py
● > Olá turma de redes industriais
>[]
```

Percebe-se que o cliente 1 enviou a seguinte mensagem: ‘Olá turma de redes industriais’ e essa mensagem chegou até o cliente 2 passando pelo servidor. No próximo tópico será feito uma análise dos pacotes transmitidos entre clientes e servidor para se verificar essa comunicação na prática, utilizando o *software Wireshark*.

ANALISANDO A TROCA DE MENSAGENS COM *WIRESHARK*

O *Wireshark* é uma ferramenta de análise de tráfego de rede que permite capturar e examinar pacotes de dados que estão sendo transmitidos através de uma rede selecionada dentro de um sistema operacional. Ele é utilizado para monitorar e diagnosticar problemas de rede, bem como para analisar o desempenho e a segurança da rede, além de permitir a inspeção dos pacotes transmitidos e dos seus conteúdos.

Ele é capaz de capturar e decodificar vários protocolos de rede, incluindo TCP, UDP, HTTP, DNS, entre outros. Ele permite visualizar e analisar o conteúdo dos pacotes capturados, bem como verificar os valores dos campos do cabeçalho e do *payload* dos pacotes. Além disso, ele possui recursos avançados como filtros, estatísticas e gráficos para facilitar a análise dos dados capturados.

Além de ser uma ferramenta muito útil para administradores de rede, desenvolvedores de software, especialistas em segurança e outros profissionais que precisam entender e solucionar problemas em redes de computadores, ele permite identificar gargalos na rede, detectar problemas de configuração e segurança, verificar a conformidade com os padrões de rede, entre outras funcionalidades.

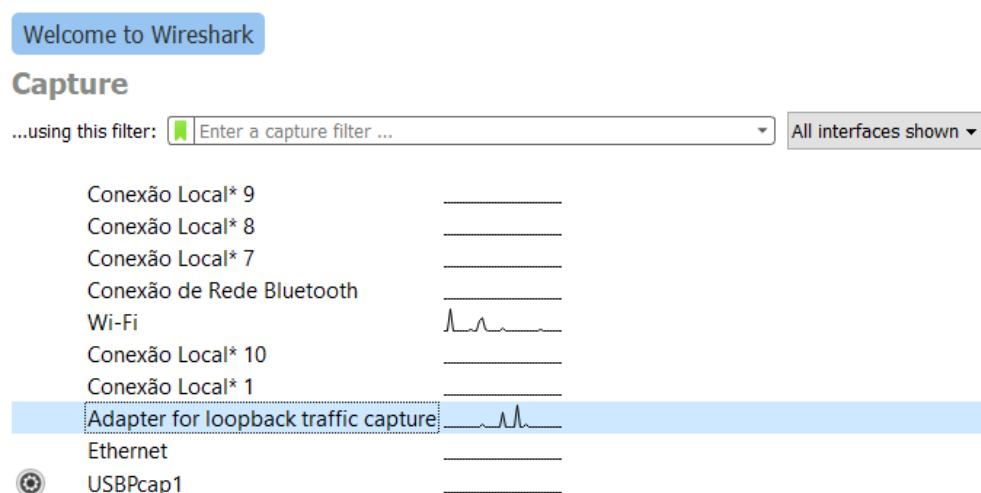
O *Wireshark* é um software livre e pode ser baixado gratuitamente no site oficial (<https://www.wireshark.org/>).

Para o nosso exemplo, o *Wireshark* será utilizado para a análise dos pacotes que estão sendo transmitidos através da nossa aplicação cliente/servidor definida no código apresentado. Com ele, podemos analisar cada quadro transmitido e analisar os campos que fazem parte do protocolo TCP/IP definido na construção do socket e os campos que fazem parte do *payload*, que diz respeito às informações transmitidas.

CAPTURANDO PACOTES

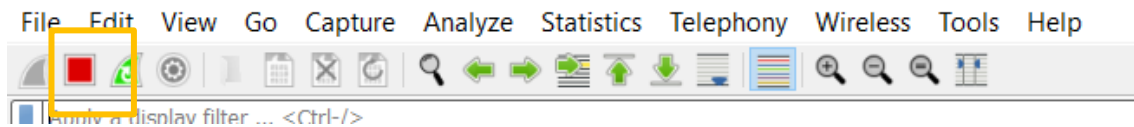
Após instalado o software, para fazer uma captura de pacotes siga os passos:

- 1- Abra o *Wireshark* e selecione a interface de rede que está sendo usada para a comunicação via socket. No nosso caso, inicialmente estávamos rodando dentro da rede local do sistema operacional, definida por *localhost*. Ele pode aparecer como *loopback* dentro das interfaces de rede disponíveis;

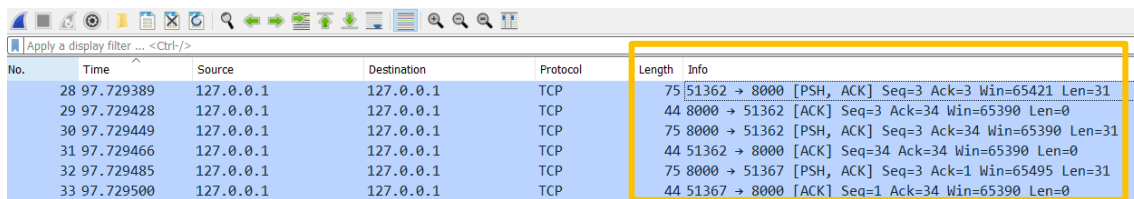


- 2- Clique no botão "Iniciar Captura" para começar a capturar os pacotes de rede;
- 3- Execute o programa servidor.py para abrir a conexão do servidor;

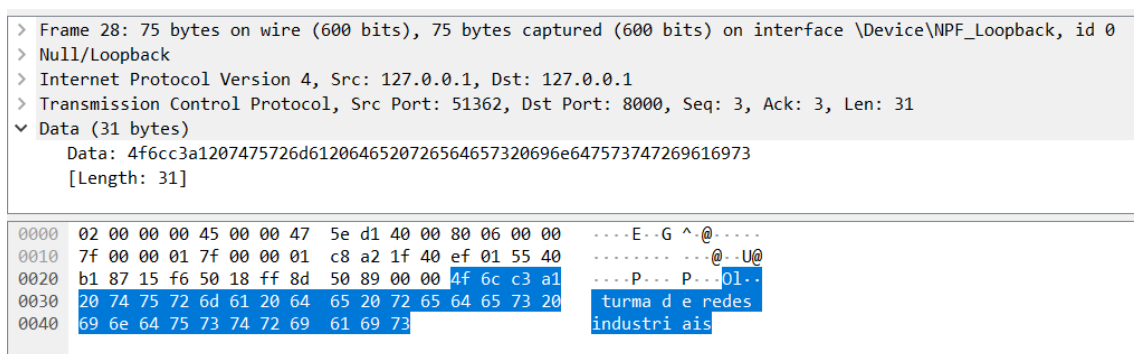
- 4- Execute o programa cliente.py e envie alguma mensagem;
- 5- Pare a captura de pacotes voltando para o *Wireshark* e clicando no botão "Parar Captura" no menu do aplicativo;



- 6- Na janela principal do *Wireshark*, você verá uma lista dos pacotes capturados. Selecione os pacotes que possuem as informações de endereço que foram utilizados para o exemplo;



- 7- Observe os campos do cabeçalho para compreender melhor o protocolo de comunicação.



ANALISANDO OS PACOTES CAPTURADOS

Observe que o pacote em destaque possui o protocolo TCP e teve como origem a porta (*src port*) 51362 e destino a porta (*dst port*) 8000. Isso quer dizer que essa mensagem foi enviada do cliente conectado numero 1 para o servidor. No quadro de Data podemos ver a mensagem transmitida: 'Olá turma de redes industriais'.

Como o protocolo TCP possui confirmação de entrega, no próximo pacote temos a confirmação de recebimento do quadro feito pelo servidor identificada pela flag ACK.

Nos próximos quadros, temos então o servidor entregando a mesma mensagem para os dois clientes conectados: *scr port*: 8000 e *dst port*: 51362 e 51367 com as respectivas confirmações de recebimento.

Essa analisa é bastante rasa, outras informações presentes nos campos também são importantes de serem analisadas

Os campos do cabeçalho são importantes para analisar e dependem do protocolo de transporte que está sendo utilizado (por exemplo, TCP ou UDP). Alguns dos campos mais importantes incluem:

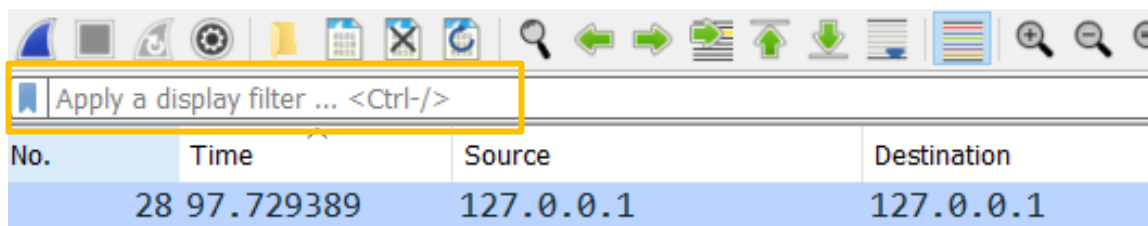
- Endereço IP de origem e destino: indica os endereços IP dos computadores que estão se comunicando.
- Número de porta de origem e destino: indica as portas usadas pelos programas que estão se comunicando.
- Número de sequência e de confirmação: usados pelo protocolo TCP para controlar a ordem e a confiabilidade da transmissão.
- Flags de controle: usados pelo protocolo TCP para controlar o fluxo de dados, reconhecimento de pacotes e outros aspectos da comunicação.

Dessa forma, encorajo que o leitor dedique tempo lendo acerca dos demais campos presentes nos protocolos utilizados e se familiarize com o software pois ele será utilizado em outros momentos durante o capítulo.

FILTROS DE PACOTES

Muitas vezes pode ser necessário filtrar os pacotes de interesse, uma vez que o sistema operacional usa a interface de rede *localhost* para executar algumas aplicações internas. Para filtrar os pacotes da aplicação em questão, podemos utilizar o parâmetro da porta definida no código como chave de filtro, assim somente os quadros que possuem a porta 12345 como origem ou destino serão apresentados.

Para utilizar os filtros o *Wireshark* dedica um menu de busca onde alguns parâmetros podem ser especificados para filtrar os pacotes.



Alguns parâmetros podem ser usados como filtros para os pacotes, como:

IP	ip.addr == "IP"
IP de origem	ip.src == "IP"
IP de destino	ip.dst == "IP"
Protocolo	"Nome do protocolo" tcp, udp, http
Excluir IP	!ip.addr == "IP"
Tráfego entre dois dispositivos	ip.addr == "IP1"/PORT1 && ip.addr == "IP2"/PORT2
Porta TCP	tcp.port == PORT
Porta TCP de origem	tcp.srcport == PORT
Porta TCP de destino	tcp.dstport == PORT
IP e Porta	tcp.port == PORT && ip.addr == "IP"
Por palavra chave	tcp contains "palavra chave"

EXEMPLO 2

CRIPTOGRAFIA DE DADOS

No *Exemplo 1 – Chat* o objetivo era exemplificar como se estabelece uma conexão cliente/servidor utilizando *sockets* em uma mesma rede local. Nesse exemplo, pequenos pacotes de texto em ASCII eram transmitidos de forma pura, ao qual era possível se identificar o conteúdo dos pacotes através do uso do software *wireshark*.

Em muitas aplicações, dados sensíveis são transmitidos entre as redes, como por exemplo em autenticações de login ou transferências bancárias. Nesses casos, não é aconselhado que os dados sejam transmitidos sem nenhum tipo de criptografia nos pacotes. Por esse motivo, no *Exemplo 2 - Criptografia*, será mostrado como se pode utilizar uma criptografia entre as transmissões.

Com a criptografia, as informações são convertidas em um formato codificado que só pode ser decodificado por alguém com a chave certa. Isso significa que, mesmo que alguém intercepte os dados, eles não poderão lê-los sem a chave adequada. Além disso, a criptografia também pode ajudar a garantir que as informações são autênticas e não foram alteradas durante a transmissão.

Ao usar criptografia em transmissões de pacotes via *socket*, você pode proteger seus dados contra ameaças como a interceptação, a modificação ou a exclusão de informações durante a transmissão. Isso é especialmente importante quando se trata de informações sensíveis ou confidenciais, como informações bancárias, senhas, dados médicos ou informações de identificação pessoal.

Para iniciar, primeiro dê uma olhada no código *Servidor_seguro.py* apresentado abaixo, que diz respeito ao código de aplicação do lado servidor e compare-o com o código do *Exemplo 1 - Chat*.

SERVIDOR_SEGURO.PY

```
import socket
import threading
from cryptography.fernet import Fernet # Biblioteca p/ criptografia

# Cria um socket TCP/IP
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM )

# Define o endereço e a porta que o socket vai se vincular
# O endereço dos sockets em Python é realizado via tupla (IP, PORT)
IP, PORT = 'localhost', 8000

print( f'Iniciando servidor em {IP}:{PORT}' )
server_socket.bind( (IP, PORT) )
```

```
# Escuta as conexões definindo o número de clientes que podem se conectar
server_socket.listen(5)

# Lista para armazenar os clientes conectados
clients = []

# Gera uma chave de criptografia única
UNIQUE_KEY = Fernet.generate_key()

# Gera o encriptador Fernet
f = Fernet( UNIQUE_KEY )

# Função para enviar uma mensagem para todos os clientes conectados
def broadcast( message ):
    for client in clients:
        client.send( message )

# Função para lidar com as conexões de clientes
def handle_client( client_socket, client_address ):
    # Envia a chave de criptografia para o cliente
    client_socket.send( UNIQUE_KEY )

    # Aguarda a resposta do cliente
    recv_data = client_socket.recv( 1024 )
    # Verifica o conteudo retornado
    if f.decrypt( recv_data ) == b'OK':
        # Se o cliente retornar a mensagem b'OK' com a
        # Criptografia certa, então eles possuem a mesma chave
        # e ele pode ser adicionado a lista de clientes ativos
        print( f'Cliente {client_address} sincronizado' )
        clients.append( client_socket )
    else:
        # Caso contrário, encerra a comunicação com o cliente
        return False

while True:
    try:
        # Recebe a mensagem do cliente
        message = client_socket.recv(1024)
        if message:
            # Envia a mensagem para todos os clientes conectados
            broadcast(message)
        else:
            # Remove o cliente da lista de clientes conectados
            clients.remove( client_socket )
            client_socket.close()
            break
```

```
except:
    # Remove o cliente da lista de clientes conectados caso
    # ocorra algum erro
    clients.remove(client_socket)
    client_socket.close()
    break

while True:
    print( 'Aguardando conexao...')

    # Aceita a conexão do cliente
    client_socket, client_address = server_socket.accept()
    print( f'Cliente conectado: {client_address[0]}:{client_address[1]}')

    # Inicia uma thread para lidar com a conexão do cliente
    # Roda cada solicitação paralelamente
    client_thread = threading.Thread(
        target = handle_client,
        args = ( client_socket, client_address )
    )
    client_thread.start()
```

Perceba que algumas mudanças foram feitas no lado Servidor, sendo elas:

1. Importação do módulo *Fernet*;

```
# Biblioteca p/ criptografia
from cryptography.fernet import Fernet
```

2. Criação de uma chave de criptografia;

```
# Gera uma chave de criptografia única
UNIQUE_KEY = Fernet.generate_key()
```

3. Sincronização da chave no início da comunicação.

```
# Envia a chave de criptografia para o cliente
client_socket.send( UNIQUE_KEY )

# Aguarda a resposta do cliente
recv_data = client_socket.recv( 1024 )

# Verifica o conteúdo retornado
if f.decrypt( recv_data ) == b'OK':
    # Inicia o tratamento do cliente
```

Neste exemplo, o servidor possui uma chave de criptografia para utilizar um método chamado de **criptografia simétrica**. Esse é um método de criptografia onde se usa uma única chave para tanto a cifragem quanto a decifragem dos dados enviados pelos pacotes *Sockets*. Esse texto cifrado pode então ser enviado com segurança através da rede, uma vez que é muito difícil de ser decifrado sem a chave secreta correspondente. O destinatário usa a mesma chave secreta para decifrar o texto cifrado e recuperar os dados originais. Essa chave é compartilhada entre as partes envolvidas na comunicação e, portanto, precisa ser mantida em segredo.

Esse é um exemplo didático, uma vez que a troca da chave acontece de maneira explícita no início de cada conexão e o servidor utiliza sempre a mesma chave secreta com todos os clientes, sendo um ponto de vulnerabilidade na aplicação, uma vez que esse pacote pode ser capturado ou um cliente malicioso pode se conectar e capturar essa chave e a comprometer.

Uma vez definido o código do lado Servidor, basta adaptar o lado Cliente para que ele consiga se conectar. Dê uma olhada no código **Cliente_seguro.py** e compare-o com o mesmo código do *Exemplo 1 – Chat*.

CLIENTE_SEGURO.PY

```
import threading
import socket
from cryptography.fernet import Fernet

# Cria um socket TCP/IP
client_socket = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
IP, PORTA = 'localhost', 8000

# Conecta ao servidor
client_socket.connect( (IP, PORTA) )

# Aguarda a chave de criptografia única e gera o encriptador Fernet
UNIQUE_KEY = client_socket.recv( 1024 )
f = Fernet( UNIQUE_KEY )

# Responde um b'OK' criptografado para sinalizar
# o recebimento da chave
client_socket.send( f.encrypt( b'OK' ) )

# Função que fica aguardando o recebimento de novas mensagens
def listen( socket : socket ):
    while True:
        # Recebe os dados do servidor
        data = socket.recv(1024)
        # Quebra a criptografia dos dados e transforma o bytearray em str
        data = f.decrypt( data ).decode()
        # Printa na tela a mensagem
        print( data , end = '\n>' )
```

```
# Se sincronizou, inicia uma Thread para ficar ouvindo o servidor
client_listen = threading.Thread(
    target = listen,
    args = (client_socket, )
)
client_listen.start()

while True:
    # Lê a mensagem do usuário
    message = input('> ')

    # Envia a mensagem para o servidor
    client_socket.send( f.encrypt( message.encode() ) )
```

Perceba as diferenças do código Cliente_seguro.py para o Cliente.py do exemplo 1:

1 – Imediatamente após a conexão, o cliente aguarda uma mensagem do lado servidor com a chave de criptografia.

```
# Aguarda a chave de criptografia e gera o encriptador Fernet
UNIQUE_KEY = client_socket.recv( 1024 )
f = Fernet( UNIQUE_KEY )
```

2 – Após recebido a chave, ele retorna uma mensagem de sincronismo com um ‘OK’ utilizando a chave de criptografia recebido;

```
# Responde um b'OK' criptografado para sinalizar
# o recebimento da chave
client_socket.send( f.encrypt( b'OK' ) )
```

3 – Ao enviar uma mensagem ao servidor, antes ele passa a mensagem pelo encriptador;

```
# Lê a mensagem do usuário
message = input('> ')

# Envia a mensagem para o servidor
client_socket.send( f.encrypt( message.encode() ) )
```

4 – Ao receber uma mensagem ele descripta ela.

```
# Recebe os dados do servidor
data = socket.recv(1024)
# Quebra a criptografia dos dados e decodificação
data = f.decrypt( data ).decode()
```

TESTANDO O CHAT

Em um primeiro momento, podemos testar o Chat do Exemplo 2 – Criptografia, para vermos como ele irá se comportar. Para isso, inicia-se o `Servidor_seguro.py` em um terminal e em seguida os dois clientes de teste em outros terminais independentes, semelhante com o executado no Exemplo 1 – Chat.

Ao executar os scripts acima teremos algo assim:

```

PS D:\Desktop\Monitoria Redes-Industriais> C:\Python310\python.exe '.\Exemplo2 - Criptografia\Servidor_seguro.py'
Iniciando servidor em localhost:8000
Aguardando conexao...
Cliente conectado: 127.0.0.1:58403
Aguardando conexao...
Cliente ('127.0.0.1', 58403) sincronizado
Cliente conectado: 127.0.0.1:58406
Aguardando conexao...
Cliente ('127.0.0.1', 58406) sincronizado

PS D:\Desktop\Monitoria Redes-Industriais> C:\Python310\python.exe '.\Exemplo2 - Criptografia\Cliente_seguro.py'
>

PS D:\Desktop\Monitoria Redes-Industriais> C:\Python310\python.exe '.\Exemplo2 - Criptografia\Cliente_seguro.py'
>
  
```

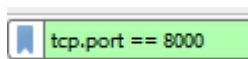
Veja o lado do servidor a esquerda conectando os dois clientes nas portas 58403 e 58406, sendo o primeiro na tela do meio e o segundo a tela a direita.

No terminal, apenas o lado servidor sinaliza a conexão como mostrado abaixo:

```

Cliente conectado: 127.0.0.1:58403
Aguardando conexao...
Cliente ('127.0.0.1', 58403) sincronizado
  
```

Como sabemos, ao iniciar a conexão há a troca das chaves de criptografia entre o lado servidor e cliente. Podemos verificar essa troca pelo *wireshark* e verificar como ocorre essa transmissão. Podemos filtrar os dados do *Wireshark* pela porta do lado servidor definida como 8000 da seguinte forma.



Com isso teremos uma troca de pacotes como a definida abaixo:

No.	Time	Source	Destination	Protocol	Length	Info
84	71.596713	127.0.0.1	127.0.0.1	TCP	52	58403 → 8000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1
85	71.596759	127.0.0.1	127.0.0.1	TCP	52	8000 → 58403 [SYN, ACK] Seq=0 Ack=1 Win=65495 Len=0 MSS=65495 SACK_PERM=1
86	71.596820	127.0.0.1	127.0.0.1	TCP	44	58403 → 8000 [ACK] Seq=1 Ack=1 Win=65495 Len=0
87	71.597833	127.0.0.1	127.0.0.1	TCP	88	8000 → 58403 [PSH, ACK] Seq=1 Ack=1 Win=65495 Len=44
88	71.597849	127.0.0.1	127.0.0.1	TCP	44	58403 → 8000 [ACK] Seq=1 Ack=45 Win=65451 Len=0
89	71.599809	127.0.0.1	127.0.0.1	TCP	144	58403 → 8000 [PSH, ACK] Seq=1 Ack=45 Win=65451 Len=100
90	71.599826	127.0.0.1	127.0.0.1	TCP	44	8000 → 58403 [ACK] Seq=45 Ack=101 Win=65395 Len=0
112	112.187025	127.0.0.1	127.0.0.1	TCP	52	58406 → 8000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1
113	112.187060	127.0.0.1	127.0.0.1	TCP	52	8000 → 58406 [SYN, ACK] Seq=0 Ack=1 Win=65495 Len=0 MSS=65495 SACK_PERM=1
114	112.187120	127.0.0.1	127.0.0.1	TCP	44	58406 → 8000 [ACK] Seq=1 Ack=1 Win=65495 Len=0
115	112.187799	127.0.0.1	127.0.0.1	TCP	88	8000 → 58406 [PSH, ACK] Seq=1 Ack=1 Win=65495 Len=44
116	112.187815	127.0.0.1	127.0.0.1	TCP	44	58406 → 8000 [ACK] Seq=1 Ack=45 Win=65451 Len=0
117	112.189800	127.0.0.1	127.0.0.1	TCP	144	58406 → 8000 [PSH, ACK] Seq=1 Ack=45 Win=65451 Len=100
118	112.189817	127.0.0.1	127.0.0.1	TCP	44	8000 → 58406 [ACK] Seq=45 Ack=101 Win=65395 Len=0

Com a troca de pacotes descritas acima, podemos verificar duas conexões ocorrendo no lado servidor, definidas pelos pacotes:

84: *Solicitação de conexão ao servidor pelo cliente de porta 58403;*

85: *Confirmação de conexão pelo lado servidor;*

86: *Confirmação de conexão pelo lado cliente;*

112: *Solicitação de conexão ao servidor pelo cliente de porta 58406;*

113: *Confirmação de conexão pelo lado servidor.*

114: *Confirmação de conexão pelo lado cliente;*

Com esses pacotes os clientes se conectaram ao servidor e estabeleceram uma conexão TCP com eles. No entanto, pode-se perceber que outros 4 pacotes foram transmitidos após cada conexão ser estabelecida. Esses pacotes são a troca da chave de criptografia que parte do lado servidor para o lado cliente. Veja:

87	71.597833	127.0.0.1	127.0.0.1	TCP	88	8000 → 58403	[PSH, ACK] Seq=1 Ack=1 Win=65495 Len=44
88	71.597849	127.0.0.1	127.0.0.1	TCP	44	58403 → 8000	[ACK] Seq=1 Ack=45 Win=65451 Len=0
89	71.599809	127.0.0.1	127.0.0.1	TCP	144	58403 → 8000	[PSH, ACK] Seq=1 Ack=45 Win=65451 Len=100
90	71.599826	127.0.0.1	127.0.0.1	TCP	44	8000 → 58403	[ACK] Seq=45 Ack=101 Win=65395 Len=0

Onde os pacotes:

87: *Envio da chave de criptografia do lado servidor ao cliente;*

88: *Confirmação de recebimento da chave pelo cliente;*

89: *Envio da mensagem 'OK' do cliente para o servidor.*

90: *Confirmação de recebimento da mensagem pelo servidor;*

Agora analisando o conteúdo do pacote 87, temos:

```
> Frame 87: 88 bytes on wire (704 bits), 88 bytes captured (704 bits) on interface \Device\NPF_{Loopback}, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 8000, Dst Port: 58403, Seq: 1, Ack: 1, Len: 44
  Data (44 bytes)
    Data: 4c4742316a763371456f4539634369524d6b5a584f575a716a4d625730707a5f3755624d...
    [Length: 44]
```

```
0000 02 00 00 00 45 00 00 54 9f 44 40 00 80 06 00 00
0010 7f 00 00 01 7f 00 00 01 1f 40 e4 23 81 c7 4e 64
0020 e2 91 8f 3d 50 18 ff d7 20 e1 00 00 4c 47 47 31
0030 6a 76 33 71 45 6f 45 39 63 43 69 52 4d 6b 5a 58
0040 4f 57 5a 71 6a 4d 62 57 30 70 7a 5f 37 55 62 4d
0050 69 4b 4f 70 67 4b 45 3d
```

Esse *bytearray* de tamanho 44 representa a chave de criptografia do Fernet e é um valor único gerado pelo servidor, portanto os valores descritos acima devem ser diferentes se testados novamente.

Analisando a resposta do cliente com a mensagem 'OK' no pacote 89, temos:

```
> Frame 89: 144 bytes on wire (1152 bits), 144 bytes captured (1152 bits) on interface \Device\NPF_{Loopback}, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 58403, Dst Port: 8000, Seq: 1, Ack: 45, Len: 100
▼ Data (100 bytes)
  Data: 674141414141426b5056504b503842356e77465f414a6653777a5073586e6d614f4d4d...
  [Length: 100]
```

0000	02 00 00 00 45 00 00 8c	9f 46 40 00 80 06 00 00
0010	7f 00 00 01 7f 00 00 01	e4 23 1f 40 e2 91 8f 3d
0020	81 c7 4e 90 50 18 ff ab	03 a3 00 00 67 41 41 41
0030	41 41 42 6b 50 56 50 4h	50 38 42 35 6e 77 46 5f
0040	41 4a 66 53 77 7a 50 73	58 6e 6d 61 4f 4f 4d 4d
0050	72 56 4f 77 46 62 31 66	75 4h 74 71 39 70 74 52
0060	71 61 38 6e 78 38 43 78	64 36 42 39 72 50 39 4h
0070	59 73 53 79 2d 37 6f 67	41 4e 55 4c 43 72 2d 55
0080	5a 71 66 53 4f 76 6f 6b	44 6f 30 36 77 67 3d 3d

Agora percebe-se que a mensagem transmitida não é apenas uma mensagem ASCII com o valor ‘OK’ visível.

Agora veremos a troca de mensagem ‘Olá turma de redes’ enviada pelo cliente 1:

```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE

PS D:\Desktop\Monitoria_Redes-Industriais> C:\Python310\python.exe '.\Exemplo2 - Criptografia\Se
rvidor_seguro.py'
Iniciando servidor em localhost:8000
Aguardando conexao...
Cliente conectado: 127.0.0.1:58403
Aguardando conexao...
Cliente ('127.0.0.1', 58403) sincronizado
Cliente conectado: 127.0.0.1:58406
Aguardando conexao...
Cliente ('127.0.0.1', 58406) sincronizado
[]

PS D:\Desktop\Monitoria_Redes-Industriais> C:\Python310\python.exe '.\Exemplo2 - Criptografia\Cl
iente_seguro.py'
> Olá turma de redes
> Olá turma de redes
> []

PS D:\Desktop\Monitoria_Redes-Industriais> C:\Python310\python.exe '.\Exemplo2 - Criptografi
a\Cliente_seguro.py'
> Olá turma de redes
> []
```

Com isso, o *wireshark* recebe os pacotes:

9065	4722.414649	127.0.0.1	127.0.0.1	TCP	164	58403 → 8000	[PSH, ACK] Seq=101 Ack=45 Win=65451 Len=120
9066	4722.414704	127.0.0.1	127.0.0.1	TCP	44	8000 → 58403	[ACK] Seq=45 Ack=221 Win=65275 Len=0
9067	4722.416981	127.0.0.1	127.0.0.1	TCP	164	8000 → 58403	[PSH, ACK] Seq=45 Ack=221 Win=65275 Len=120
9068	4722.416996	127.0.0.1	127.0.0.1	TCP	44	58403 → 8000	[ACK] Seq=221 Ack=165 Win=65331 Len=0
9069	4722.417096	127.0.0.1	127.0.0.1	TCP	164	8000 → 58406	[PSH, ACK] Seq=45 Ack=101 Win=65395 Len=120
9070	4722.417130	127.0.0.1	127.0.0.1	TCP	44	58406 → 8000	[ACK] Seq=101 Ack=165 Win=65331 Len=0

Onde os pacotes são:

- 9065: Cliente envia a mensagem;
- 9066: Servidor confirma o recebimento;
- 9067: Servidor envia a mensagem para o cliente 1;
- 9068: Cliente 1 confirma recebimento;
- 9069: Servidor envia a mensagem para o cliente 2;
- 9070: Cliente 2 confirma recebimento.

Essa estrutura de mensagens era esperada de acordo com o protocolo TCP, agora podemos verificar o conteúdo das mensagens.

```
> Frame 9065: 164 bytes on wire (1312 bits), 164 bytes captured (1312 bits) on interface \Device\NPF_{Loopback}, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 58403, Dst Port: 8000, Seq: 101, Ack: 45, Len: 120
▼ Data (120 bytes)
  Data: 674141414141426b505758316573696777774b644933486972734f51544c5566777a5834...
  [Length: 120]
```



```

0000 02 00 00 00 45 00 00 a0 aa 65 40 00 80 06 00 00
0010 7f 00 00 01 7f 00 00 01 e4 23 1f 40 e2 91 8f a1
0020 81 c7 4e 90 50 18 ff ab 2e 0c 00 00 67 41 41 41
0030 41 41 42 6h 50 57 58 31 65 73 69 67 77 77 4b 64
0040 40 33 48 69 72 73 4f 51 54 4c 55 66 77 7a 58 3d
0050 66 48 77 64 39 74 69 61 39 74 7a 64 4a 38 41 48
0060 30 79 54 43 6a 64 41 41 4e 4e 53 58 63 75 69 6e
0070 4b 44 4a 76 37 7a 69 65 53 43 38 71 6d 48 52 73
0080 36 57 37 75 57 70 4e 4b 54 4c 77 48 6f 68 56 70
0090 4f 63 38 71 7a 54 65 4c 6e 47 50 59 33 79 6c 6b
00a0 47 79 77 3d

```

Dessa forma a estrutura da mensagem enviada é apenas um *bytearray* pseudoaleatório. Se comparado com o *Exemplo 1 – Chat*, sem a criptografia:

```

> Frame 28: 75 bytes on wire (600 bits), 75 bytes captured (600 bits) on interface \Device\NPF_{Loopback}, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 51362, Dst Port: 8000, Seq: 3, Ack: 3, Len: 31
▼ Data (31 bytes)
  Data: 4f6cc3a1207475726d61206465520726564657320696e647573747269616973
  [Length: 31]

```

```

0000 02 00 00 00 45 00 00 47 5e d1 40 00 80 06 00 00  ....E..G ^..@....
0010 7f 00 00 01 7f 00 00 01 c8 a2 1f 40 ef 01 55 40  ....@...U@
0020 b1 87 15 f6 50 18 ff 8d 50 89 00 00 4f 6c c3 a1  ....P...P...01..
0030 20 74 75 72 6d 61 20 64 65 20 72 65 64 65 73 20  turma d e redes
0040 69 6e 64 75 73 74 72 69 61 69 73  industri ais

```

Assim, é possível se encriptar dados sensíveis e garantir que redes maliciosas não consigam ler os dados dos pacotes. Em contra partida, o *bytearray* de dados será consideravelmente maior com a criptografia. Comparando o tamanho dos pacotes enviados com e sem criptografia acima, tem-se que:

Com criptografia: 120 bytes

Sem criptografia: 31 bytes

A criptografia é uma técnica fundamental para garantir a segurança de informações sensíveis em sistemas computacionais. Em Python, existem diversas bibliotecas disponíveis para implementação de algoritmos criptográficos, tais como a biblioteca *cryptography* usada e o módulos como *hashlib* da biblioteca padrão do *Python*.

Além disso, a linguagem Python é amplamente utilizada em áreas que envolvem segurança da informação, como cibersegurança, desenvolvimento de criptomoedas e análise forense digital. Com o uso das bibliotecas de criptografia disponíveis no *Python*, é possível implementar algoritmos criptográficos robustos e seguros, garantindo a privacidade e a integridade das informações.

No entanto, é importante lembrar que a criptografia por si só não é suficiente para garantir a segurança das informações, sendo necessário também adotar outras práticas de segurança, como a autenticação de usuários, a aplicação de políticas de acesso e a proteção contra ameaças externas. Todavia, esse é um bom ponto de partida para se começar a estudar sobre cibersegurança.

EXEMPLO 3

TRANSFERÊNCIA DE ARQUIVOS

No *Exemplo 1 – Chat* e *Exemplo 2- Criptografia*, foram transferidos pacotes contendo mensagens de texto considerados pequenos, não passando de algumas dezenas de bytes no *payload* do pacote. No entanto, os soquetes não se limitam a transmissões de mensagens de texto, podendo ser usadas para transmitir arquivos de texto, imagens, *Streamings* de áudio e vídeo, dentre outras estruturas.

Porém, quando se transmite um arquivo entre soquetes, existe a possibilidade de que ele não seja transmitido em um único pacote, pois existem limitações nos tamanhos do *payload* utilizado pelos soquetes, sendo necessário se segmentar esses pacotes em múltiplos pacotes menores. No *Exemplo 3 – Transferência de arquivos* será visto como os soquetes executam as segmentações dos pacotes e como eles são montados na aplicação final, mostrando isso através de um *script python* que é capaz de transmitir arquivos entre sistemas. Mas antes de começar, é importante entender porque ocorre a segmentação dos dados.

SEGMENTAÇÃO DOS DADOS

A segmentação de dados é um processo utilizado na transmissão de grandes quantidades de dados via sockets para que os dados sejam divididos em pacotes menores antes de serem enviados pela rede. Cada pacote contém uma parte dos dados a serem transmitidos, juntamente com informações de cabeçalho que permitem que o receptor saiba como reconstruir os dados originais.

A segmentação é importante em redes de computadores porque os dados podem ser muito grandes para serem transmitidos como uma única unidade. Além disso, a segmentação permite que os pacotes sejam transmitidos em diferentes rotas pela rede, aumentando a confiabilidade da transmissão e garantindo que os dados cheguem ao destino mesmo se houver perda de pacotes ou congestionamento na rede.

Essa segmentação se dá devido ao tamanho máximo em bytes de um pacote TCP/IP padrão. Esse tamanho é determinado pelo MTU (*Maximum Transmission Unit*) de uma rede, que define o tamanho máximo do pacote que pode ser transmitido sem que haja fragmentação de pacotes.

O valor do MTU pode variar de rede para rede, dependendo do tipo de tecnologia de rede utilizada. Por exemplo, o MTU típico de uma rede Ethernet é de 1500 bytes, enquanto em uma rede Wi-Fi pode ser de 2304 bytes ou mais. Redes de longa distância, como redes WAN, podem ter MTUs ainda maiores.

O tamanho máximo de um pacote TCP/IP é, portanto, limitado pelo MTU da rede em que ele é transmitido. Isso significa que um pacote TCP/IP padrão pode ter no máximo

o tamanho do MTU, menos o tamanho do cabeçalho IP e do cabeçalho TCP, reduzindo ainda mais o tamanho do *payload* do quadro.

Em geral, os pacotes TCP/IP são projetados para trabalhar com MTUs de diferentes tamanhos, o que permite que eles sejam transmitidos em uma ampla variedade de redes. Quando um pacote é maior do que o MTU da rede, ele é fragmentado em pacotes menores antes de ser transmitido e, em seguida, reagrupado na extremidade de destino. No entanto, a fragmentação de pacotes pode ter impacto negativo no desempenho da rede e deve ser evitada sempre que possível.

Para ilustrar isso, o Exemplo 3 – Transferência de arquivos, irá mostrar um script onde um cliente se conecta a um servidor e solicita a transferência de um arquivo que será enviado pelo servidor. Esse arquivo deverá conter um tamanho significativo a ponto da rede precisar fragmentar ele e através do *Wireshark* seremos capazes de analisar esses pacotes.

Portanto, tente analisar o script **repositório.py** abaixo que será o nosso servidor:

REPOSITÓRIO.PY

```
import threading
import socket
import os

# Cria um socket TCP/IP
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Define o endereço e a porta que o socket vai se vincular
IP, PORT = 'localhost', 8000
print(f'Iniciando servidor em {IP}:{PORT}')
server_socket.bind((IP, PORT))
# Define o número máximo de clientes que podem se conectar
server_socket.listen(5)

# Caminho absoluto para o diretório das imagens
img_path = os.path.dirname( __file__ ) + '/imagens/'

# Função para tratar a comunicação com cada cliente paralelamente
def receive_file( client_socket ):

    # Recebe a mensagem do cliente
    data = client_socket.recv(1024)

    # Verifica se o cliente solicitou um arquivo
    if data.startswith(b'GET'):

        # Obtém o nome do arquivo solicitado
        file_name = data.split()[1].decode()
        print( f'Cliente solicitou o arquivo {file_name}' )
```

```
# Verifica se o arquivo existe
if file_name in [
    'cachorro.jpg',
    'gato.jpg',
    'iguana.jpg',
    'tucano.jpg'
]:

    print(f'Arquivo {file_name} encontrado.')
    # Abre o arquivo para leitura
    with open( img_path + file_name, 'rb') as f:
        # Envia o arquivo em pacotes de 1024 em 1024 bytes
        count = 0
        while True:
            data = f.read(1024)
            if not data:
                break
            client_socket.send(data)
            count += 1
            print( f'Enviado o segmento número {count}')

        # Envia uma mensagem final para o cliente
        print('Envio concluído' )
        client_socket.send( b'OK' )

    # Se o arquivo não existe, encerra a conexão
    else:
        # Caso o arquivo não exista
        print(f'Arquivo {file_name} NÃO encontrado.')
        data = f'ERROR "O arquivo {file_name} não existe"'
        client_socket.send( data.encode() )

# Fecha a conexão com o cliente
client_socket.close()

while True:
    # Aceita a conexão do cliente
    print('Aguardando conexão...')
    client_socket, client_address = server_socket.accept()

    print(f'Cliente conectado: {client_address[0]}:{client_address[1]}')

    # Recebe o arquivo enviado pelo cliente
    client = threading.Thread(
        target = receive_file,
        args = ( client_socket, )
    )
    client.start()
```

O *script* acima funciona de maneira semelhante aos de mais exemplos citados acima. As diferenças estão que agora o Servidor espera por uma mensagem do Cliente logo após estabelecer a conexão:

```
def receive_file( client_socket ):
    # Recebe a mensagem do cliente
    data = client_socket.recv(1024)
```

O Cliente deve enviar um comando solicitando o envio de um arquivo específico. Se a mensagem do Cliente inicia com a palavra “GET” então o servidor inicia busca pelo arquivo para enviar:

```
# Verifica se o cliente solicitou um arquivo
if data.startswith(b'GET'):

    # Obtém o nome do arquivo solicitado
    file_name = data.split()[1].decode()
    print( f'Cliente solicitou o arquivo {file_name}' )

    # Verifica se o arquivo existe
    if file_name in [
        'cachorro.jpg',
        'gato.jpg',
        'iguana.jpg',
        'tucano.jpg'
    ]:
```

Se o nome do arquivo está entre os nomes na lista, então ele inicia a transmissão dos binários do arquivo:

```
# Abre o arquivo para leitura
with open( img_path + file_name, 'rb') as f:
    # Envia o arquivo em pacotes de 1024 em 1024 bytes
    count = 0
    while True:
        data = f.read(1024)
        if not data:
            break
        client_socket.send(data)
        count += 1
    print( f'Enviado o segmento número {count}')
```

Dessa forma, o arquivo é transmitido do lado servidor, cabendo ao cliente armazenar os bytes recebidos e montar a imagem do seu lado. Esse modelo de transmissão funciona para qualquer tipo de arquivo armazenado no lado servidor, basta que o cliente saiba o tipo de arquivo que está sendo transmitido para que ele possa fazer a montagem dos pacotes após finalizada a transmissão.

Antes de iniciar a análise dos pacotes, vejamos o script **Cliente.py**:

CLIENTE.PY

```
import socket
import os

# Cria um socket TCP/IP
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Define o endereço e a porta que o servidor estará conectado
HOST, PORT = 'localhost', 8000
print(f'Iniciando conexão com o servidor em {HOST}:{PORT}')

# Conecta o socket ao endereço IP e porta do servidor
client_socket.connect((HOST, PORT))

# Envia a mensagem para o servidor solicitar um arquivo
file_name = input('Digite o nome do arquivo para fazer download: ')
client_socket.send(f'GET {file_name}'.encode())

# Caminho para fazer download dos arquivos
download_path = os.path.dirname( __file__ ) + '/downloads/'

# Abre o arquivo para escrita
with open( download_path + file_name, 'wb') as f:
    # Recebe os pacotes de dados do servidor
    while True:
        data = client_socket.recv(1024)
        if not data:
            break
        f.write(data)

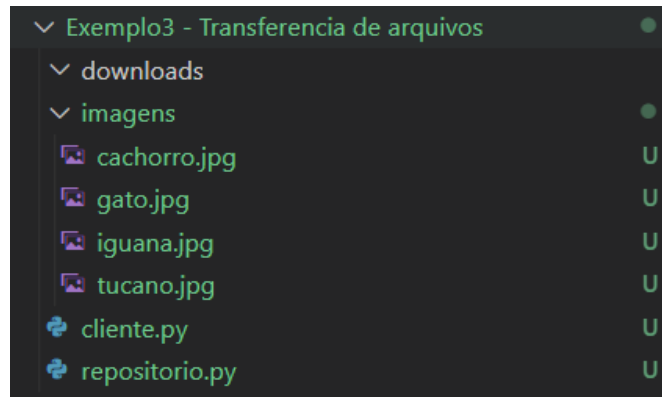
# Fecha o socket
client_socket.close()
```

O lado cliente é relativamente mais simples de ser compreendido, pois:

- 1º Inicia a conexão com o servidor pelo endereço definido;
- 2º Aguarda que o usuário dê entrada ao nome do arquivo que ele quer receber;
- 3º Monta o pacote no modelo que o servidor espera receber “GET file_name”;
- 4º Abre o arquivo para salvar os bytes recebidos;
- 5º Aguarda o recebimento de todos pacotes.

TESTANDO A TRANSFERÊNCIA DE ARQUIVOS

Para testar a transferência de arquivos, antes vamos olhar a árvore de repositórios da pasta *Exemplo - 3* que possuímos:



Podemos ver que dentro dela existem duas pastas. A primeira é a pasta *downloads* que esta vazia e a segunda é a pasta *imagens* que possuem 4 imagens dentro dela:

1. *Cachorro.jpg*
2. *Gato.jpg*
3. *Iguana.jpg*
4. *Tucano.jpg*

Essas quatro imagens serão as imagens disponíveis para serem solicitadas e transmitidas pelo servidor.

Além disso, temos os dois scripts dentro do repositório, que são os scripts descritos acima, sendo eles:

1. *Cliente.py*
2. *Repositório.py*

Para começar, vamos executar o script **repositório.py** em um terminal *python* que será o lado servidor. Em seguida podemos executar em outro terminal *python* o script **cliente.py**, ficando com algo assim:

```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE

PS D:\Desktop\Monitoria Redes-Industriais> & C:\Python310\python.exe "d:\Desktop\Monitoria Redes-Industriais\Exemplo3 - Transferencia de arquivos\repositorio.py"
Iniciando servidor em localhost:8000
Aguardando conexão...
Cliente conectado: 127.0.0.1:58539
Aguardando conexão...

PS D:\Desktop\Monitoria Redes-Industriais> C:\Python310\python.exe ".\Exemplo3 - Transferencia de arquivos\cliente.py"
Iniciando conexão com o servidor em localhost:8000
Digite o nome do arquivo para fazer download: 
```

Na esquerda esta a execução do *script* **repositório.py** e a direita o **cliente.py**. Perceba que o servidor já sinalizou uma conexão estabelecida e o cliente está aguardando que seja dado como entrada o nome do arquivo para ser transmitido. Podemos verificar essa conexão estabelecida com o *Wireshark*.

Time	Source	Destination	Protocol	Length	Info
1 0.000000	127.0.0.1	127.0.0.1	TCP	52	58539 → 8000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1
2 0.000066	127.0.0.1	127.0.0.1	TCP	52	8000 → 58539 [SYN, ACK] Seq=0 Ack=1 Win=65495 Len=0 MSS=65495 SACK_PERM=1
3 0.000146	127.0.0.1	127.0.0.1	TCP	44	58539 → 8000 [ACK] Seq=1 Ack=1 Win=65495 Len=0

Veja que foi estabelecida uma conexão por um cliente com porta 58539 no servidor de porta 8000. Essa é a nossa conexão estabelecida.

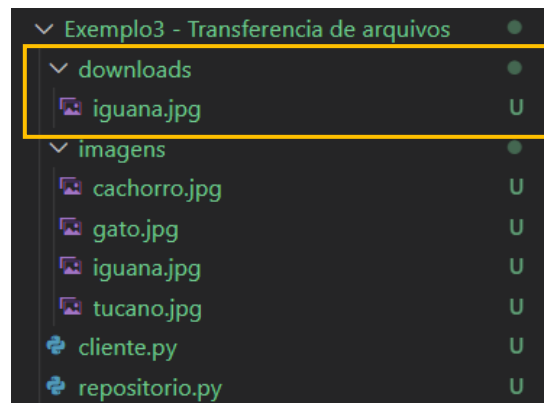
Agora vamos dar como entrada o nome do arquivo **iguana.jpg** e ver o que o acontece no terminal.

```

TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE
Enviado o segmento número 89
Enviado o segmento número 90
Enviado o segmento número 91
Enviado o segmento número 92
Enviado o segmento número 93
Enviado o segmento número 94
Enviado o segmento número 95
Enviado o segmento número 96
Enviado o segmento número 97
Enviado o segmento número 98
Envio concluido
PS D:\Desktop\Monitoria_Redes-Industriais> C:\Python310\python.exe '.\
Exemplo3 - Transferencia de arquivos\cliente.py'
Iniciando conexão com o servidor em localhost:8000
Digite o nome do arquivo para fazer download: iguana.jpg
PS D:\Desktop\Monitoria_Redes-Industriais>

```

Quando executado o comando, o servidor enviou 98 segmentos de dados para o lado cliente. Se tudo ocorreu bem, a árvore de arquivos no diretório deve estar diferente:



Perceba que agora na pasta downloads existe um arquivo chamado **iguana.jpg**, mostrando que ocorreu tudo bem.

Agora vejamos alguns pacotes transmitidos sendo capturados pelo *Wireshark*:

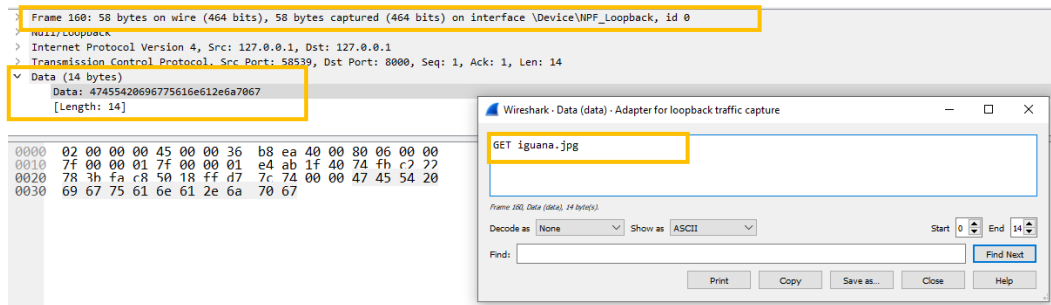
No.	Time	Source	Destination	Protocol	Length	Info
160	233.960159	127.0.0.1	127.0.0.1	TCP	58	58539 → 8000 [PSH, ACK] Seq=1 Ack=1 Win=65495 Len=14
161	233.960184	127.0.0.1	127.0.0.1	TCP	44	8000 → 58539 [ACK] Seq=1 Ack=15 Win=65481 Len=0
162	233.960848	127.0.0.1	127.0.0.1	TCP	1068	8000 → 58539 [PSH, ACK] Seq=1 Ack=15 Win=65481 Len=1024
163	233.960874	127.0.0.1	127.0.0.1	TCP	44	58539 → 8000 [ACK] Seq=15 Ack=1025 Win=64471 Len=0
164	233.961029	127.0.0.1	127.0.0.1	TCP	1068	8000 → 58539 [PSH, ACK] Seq=1025 Ack=15 Win=65481 Len=1024
165	233.961049	127.0.0.1	127.0.0.1	TCP	44	58539 → 8000 [ACK] Seq=15 Ack=2049 Win=63447 Len=0
166	233.961200	127.0.0.1	127.0.0.1	TCP	1068	8000 → 58539 [PSH, ACK] Seq=2049 Ack=15 Win=65481 Len=1024
167	233.961221	127.0.0.1	127.0.0.1	TCP	44	58539 → 8000 [ACK] Seq=15 Ack=3073 Win=62423 Len=0
168	233.961336	127.0.0.1	127.0.0.1	TCP	1068	8000 → 58539 [PSH, ACK] Seq=3073 Ack=15 Win=65481 Len=1024
169	233.961361	127.0.0.1	127.0.0.1	TCP	44	58539 → 8000 [ACK] Seq=15 Ack=4097 Win=61399 Len=0
170	233.961485	127.0.0.1	127.0.0.1	TCP	1068	8000 → 58539 [PSH, ACK] Seq=4097 Ack=15 Win=65481 Len=1024
171	233.961502	127.0.0.1	127.0.0.1	TCP	44	58539 → 8000 [ACK] Seq=15 Ack=5121 Win=60375 Len=0
172	233.961615	127.0.0.1	127.0.0.1	TCP	1068	8000 → 58539 [PSH, ACK] Seq=5121 Ack=15 Win=65481 Len=1024
173	233.961634	127.0.0.1	127.0.0.1	TCP	44	58539 → 8000 [ACK] Seq=15 Ack=6145 Win=59351 Len=0
174	233.961803	127.0.0.1	127.0.0.1	TCP	1068	8000 → 58539 [PSH, ACK] Seq=6145 Ack=15 Win=65481 Len=1024

Analisando a sequência lógica da transmissão através dos identificadores ‘No.’ de pacotes, temos:

160: Cliente envia ‘GET iguana.jpg’ para o servidor

161: O servidor confirma recebimento do pacote

Se olharmos o *payload* desse pacote podemos ler essa mensagem:



Em seguida o servidor começa a enviar os segmentos da imagem para o cliente:

162: O servidor envia o primeiro segmento da imagem

163: O cliente confirma recebimento

164: O servidor envia o segundo segmento da imagem

.....

Se analisado o *payload* dos pacotes, veremos apenas bytes aleatórios que compõem a imagem da iguana, no entanto, podemos analisar como os dados foram segmentados através da informação **Seq** que cada pacote possui:

	Length	Info
58	58539 → 8000	[PSH, ACK] Seq=1 Ack=1 Win=65495 Len=14
44	8000 → 58539	[ACK] Seq=1 Ack=15 Win=65481 Len=0
1068	8000 → 58539	[PSH, ACK] Seq=1 Ack=15 Win=65481 Len=1024
44	58539 → 8000	[ACK] Seq=15 Ack=1025 Win=64471 Len=0
1068	8000 → 58539	[PSH, ACK] Seq=1025 Ack=15 Win=65481 Len=1024
44	58539 → 8000	[ACK] Seq=15 Ack=2049 Win=63447 Len=0
1068	8000 → 58539	[PSH, ACK] Seq=2049 Ack=15 Win=65481 Len=1024

Dessa forma podemos perceber que cada segmento realmente possui 1024 bytes como estipulado e por estarem em uma rede sem congestionamento, cada pacote chegou sequencialmente, o que pode não acontecer se o servidor estiver fora do *loopback* ou *localhost* da rede.

Podemos ver também que o cliente confirma o recebimento de cada um dos pacotes recebidos através do indicador **ACK** indicando quantos *bytes* ele leu a partir do **Seq** recebido. Caso algum pacote não chegue até o destino, o protocolo se encarrega de reencaminhar esse pacote e monta-lo na ordem certa no destino.

Este é um exemplo simples, mas poderoso, que pode ser usado para transferir qualquer tipo de arquivos em qualquer rede, no entanto ainda não deixa de ser um modelo didático.

UTILIZANDO SOCKETS EM DIFERENTES REDES

Os exemplos acima foram modelos de como se pode utilizar *sockets* para criar aplicações que envolvam trocas de dados entre dispositivos conectados entre uma mesma máquina (*localhost*). Para usar o código em uma comunicação entre computadores em diferentes regiões físicas, é necessário fazer algumas modificações para que o servidor esteja acessível pela Internet. Veja os passos a seguir para conseguir criar um servidor acessível para qualquer computador conectado à internet:

1º Obtenha um endereço IP público: O servidor precisa estar disponível na Internet para que outros computadores possam se conectar a ele. Para isso, é necessário ter um endereço IP público, que é fornecido pelo seu provedor de internet. Você pode verificar seu endereço IP público usando sites como o Meu IP por exemplo.

2º Abra a porta do servidor no roteador: O roteador é responsável por encaminhar o tráfego de internet para o computador que está executando o servidor. Para permitir que o tráfego chegue até o servidor, é necessário configurar o roteador para encaminhar o tráfego para a porta que o servidor está usando. Isso pode ser feito acessando as configurações do roteador e criando uma regra de encaminhamento de porta, que pode aparecer em inglês como *Forwarding rules*.

- a) Para acessar as configurações do roteador e configurar as regras de roteamento, você precisa conhecer o endereço IP do roteador e as credenciais de acesso. Essas informações geralmente são fornecidas pelo manual do roteador ou pelo próprio provedor de internet.
- b) Para descobrir o endereço IP do roteador, você pode usar o comando "**ipconfig**" no Prompt de Comando (no Windows) ou o comando "**ifconfig**" no Terminal (no Linux ou no MacOS). Procure pelo endereço IP do "*Gateway Padrão*" na saída do comando. Esse é o endereço IP do roteador.
- c) Depois de obter o endereço IP do roteador, abra um navegador de internet e digite esse endereço na barra de endereço. Você será direcionado para a página de login do roteador. Digite as credenciais de acesso para fazer login.
- d) As configurações de encaminhamento de porta podem estar em diferentes lugares, dependendo do modelo e do fabricante do roteador. Geralmente, é possível encontrar as configurações de encaminhamento de porta na seção "*Firewall*", "*NAT*" ou "*Redirecionamento de porta*". Consulte o manual do roteador ou pesquise na internet por instruções específicas para o seu modelo de roteador.
- e) É importante lembrar que, ao configurar as regras de roteamento, você está abrindo uma porta no roteador e permitindo que o tráfego de internet seja encaminhado para o seu computador. Certifique-se de configurar as regras corretamente e tomar medidas de segurança para proteger o seu computador e a sua rede.

3º Modifique o endereço do servidor: Agora que o servidor está acessível pela Internet, é necessário usar o endereço IP público e a porta aberta para se conectar a ele. Para fazer isso, modifique a linha no código do servidor:

```
IP, PORTA = 'localhost', 8000
```

Para:

```
IP, PORTA = '<seu IP público>', <Porta aberta no roteador>
```

4º Modifique o endereço do cliente: Para que o cliente se conecte ao servidor na Internet, é necessário modificar o endereço do servidor para o endereço IP público e a porta aberta no roteador. Para isso, faça as mesmas modificações do passo 3 no código do cliente conectado.

Com essas modificações, o servidor estará disponível na Internet e o cliente poderá se conectar a ele a partir de qualquer lugar. É importante lembrar que, ao deixar o servidor exposto na Internet, é necessário tomar medidas de segurança para garantir que apenas usuários autorizados possam se conectar a ele.

No entanto, nem todos provedores de internet possibilitam o uso de IPs públicos para todos os seus clientes ou então eles podem cobrar por esse serviço. A maioria dos provedores usam o CGNAT (*Carrier-Grade Network Address Translation*) como uma solução para lidar com a escassez de endereços IPv4 e fornece acesso à Internet a seus clientes.

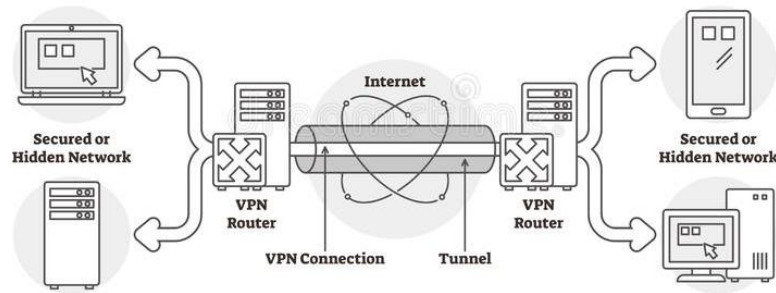
O CGNAT é uma técnica de tradução de endereço de rede que permite que vários dispositivos compartilhem um único endereço IP público. Isso é semelhante ao NAT (*Network Address Translation*) usado em redes locais usado para definir vários IPs dos dispositivos conectados em um mesmo ambiente utilizando o mesmo IP, sendo gerenciado pelo roteador. O CGNAT utiliza o mesmo processo, porém em uma escala maior.

Com ele, o provedor de internet aloca um bloco de endereços privados para seus clientes e usa um ou mais endereços IP públicos para traduzir esses endereços privados em endereços públicos para permitir que seus clientes se conectem à internet. Essa técnica permite que o provedor de internet conserve seus endereços IPv4 limitados, reduza o número de endereços IP públicos necessários e, portanto, economize dinheiro.

USANDO VPN (VIRTUAL PRIVATE NETWORK)

Caso não seja possível se obter um IP público, existem soluções que podem contornar o CGNAT dos provedores, utilizando um VPN (*Virtual Private Network*). Ao usar uma VPN, o tráfego de rede é *tunelado* para um servidor remoto antes de ser encaminhado para a internet pública.

Ao se conectar a uma VPN, o dispositivo envia suas solicitações de rede para o servidor VPN, que as encaminha para o destino pretendido. Como o servidor VPN possui um endereço IP público, as respostas do destino são enviadas de volta para o servidor VPN, que as encaminha para o dispositivo que solicitou. Dessa forma, o CGNAT do provedor de internet é contornado, permitindo que a conexão via socket seja estabelecida.



Dessa forma, uma VPN pode contornar o CGNAT, permitindo que dispositivos se comuniquem via socket em uma rede privada e segura. No entanto, é importante lembrar que o uso de uma VPN pode reduzir a velocidade da conexão e aumentar a latência, uma vez que os pacotes transmitidos percorrerão um caminho maior até chegar aos seus destinos.

O **Hamachi** é uma VPN gratuita que permite criar uma rede virtual privada para conectar computadores remotos, permitindo o compartilhamento de arquivos, jogos online e outros recursos de rede. Embora seja popular para jogos em LAN, o **Hamachi** tem algumas limitações em relação às VPNs comerciais, como a falta de recursos avançados de segurança e privacidade.

Uma das principais desvantagens do **Hamachi** é que ele utiliza servidores intermediários para encaminhar o tráfego, o que pode reduzir a velocidade e aumentar a latência. Além disso, o **Hamachi** não oferece recursos de criptografia de ponta a ponta, o que pode expor seus dados a riscos de segurança.