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 capitulo.

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 casa 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

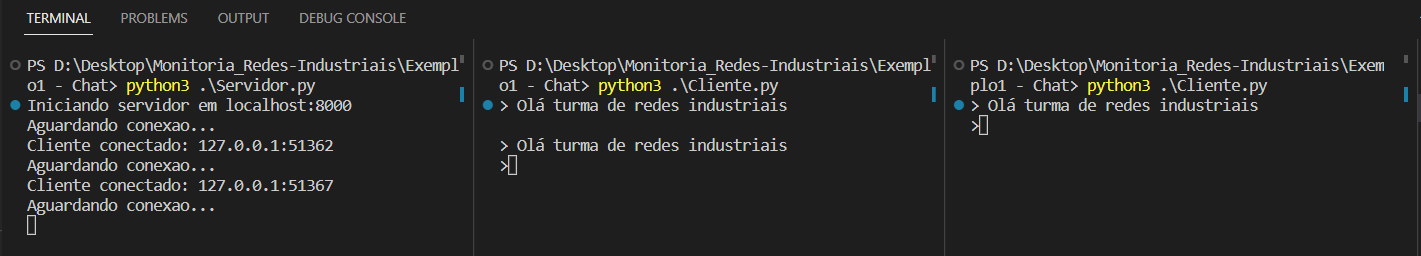
IP, PORTA = 'localhost', 8000

# 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.



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 analise 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.

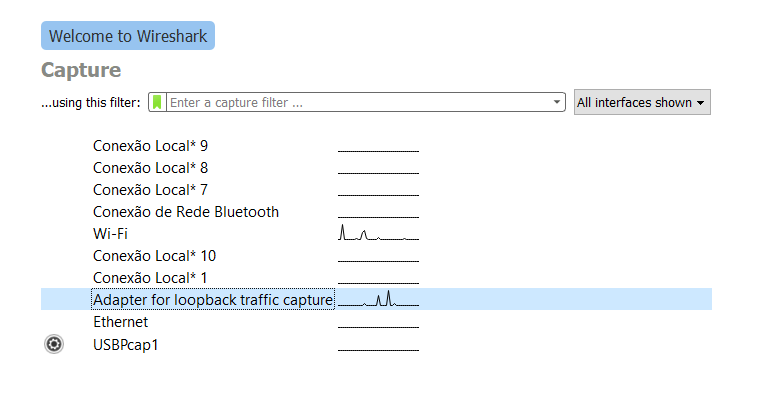
O *Wireshark* é um software livre e pode ser baixado gratuitamente no site oficial (https://www.wireshark.org/).

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.

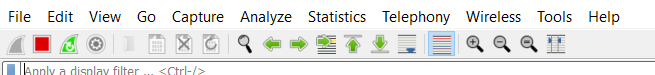
# Capturando pacotes

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

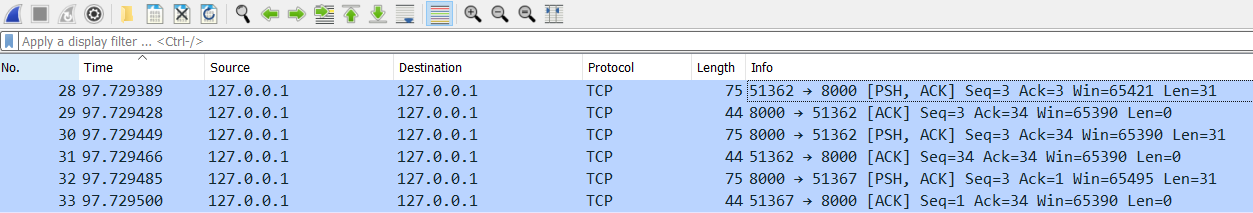
1. Abra o *Wireshark* e selecione a interface de rede que está sendo usada para a comunicação via socket. No nosso caso, inicialmente estávamos rodando dentro da rede local do sistema operacional, definida por *localhost*. Ele pode aparecer como *loopback* dentro das interfaces de rede disponíveis;



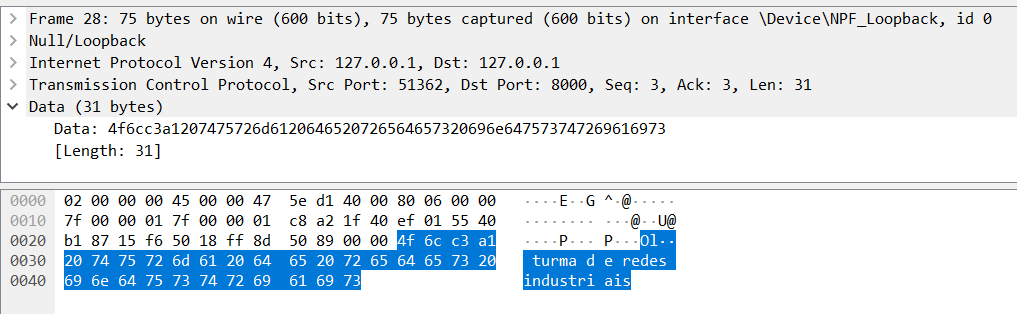
1. Clique no botão "Iniciar Captura" para começar a capturar os pacotes de rede;
2. Execute o programa servidor.py para abrir a conexão do servidor;
3. Execute o programa cliente.py e envie alguma mensagem;
4. Pare a captura de pacotes voltando para o *Wireshark* e clicando no botão "Parar Captura" no menu do aplicativo;



1. 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;



1. 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:

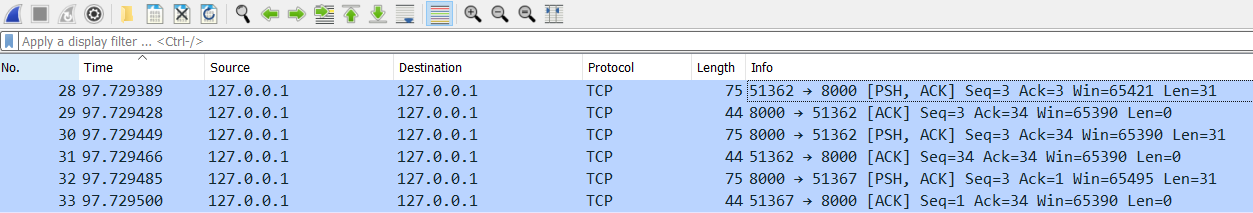
1. Endereço IP de origem e destino: indica os endereços IP dos computadores que estão se comunicando.
2. Número de porta de origem e destino: indica as portas usadas pelos programas que estão se comunicando.
3. Número de sequência e de confirmação: usados pelo protocolo TCP para controlar a ordem e a confiabilidade da transmissão.
4. 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 capitulo.

# 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 filtras os pacotes da aplicação em questão, podemos utilizar o parâmetro da porta definida no código como chave de filtro, assim somente os quadros que possuírem 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.

Para o Exemplo 2 - Criptografia, será mostrado como se pode utilizar uma criptografia entre as transmisso

# Segmentação dos dados

Ao analisar os pacotes capturados, é possível observar como alguns dados são segmentados em pacotes menores. Essas informações são importantes para compreender o desempenho e a confiabilidade da comunicação via socket.

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.

Mostrar a transferência de arquivos aqui

Utilizando sockets em diferentes redes

O exemplo acima foi um modelo 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*.

1. 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.
2. 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.
3. 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.
4. 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.
5. 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.

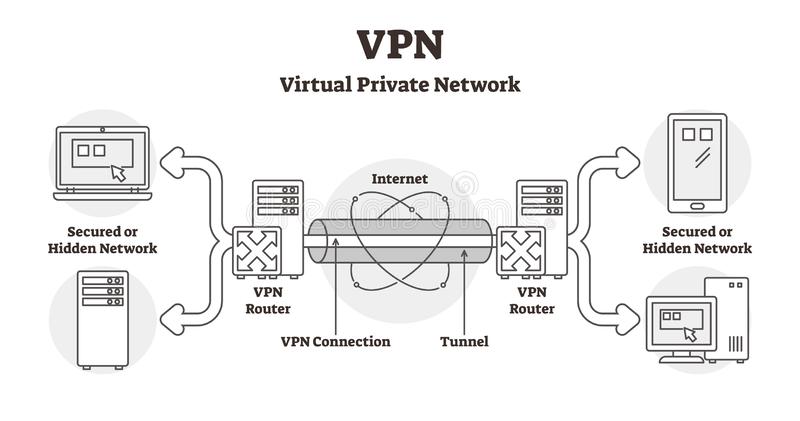
No entanto, nem todos provedores de internet possibilitam o uso de IPs públicos para todos os seus clientes ou então eles podem cobram por esse serviço. A maioria dos provedores usam o CGNAT (*Carrier-Grade Network Address Translation*) como uma solução para lidar com a escassez de endereços IPv4 e fornece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

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.