Desenvolvimento de aplicações com sockets

Princípios básicos relacionados ao desenvolvimento de aplicações de redes de computadores utilizando socket.

Prof. Fabio Henrique Silva

Propósito

O entendimento do desenvolvimento de socket, como parte do desenvolvimento de aplicações para redes de computadores, possibilita aos profissionais de Tecnologia da Informação (TI) uma melhor compreensão de como as aplicações de rede trocam informações entre si.

Objetivos

- Reconhecer os fundamentos básicos em torno da programação de sockets.
- Descrever a estrutura de um socket com UDP.
- Descrever a estrutura de um socket com TCP.
- Descrever os aspectos relacionados ao desenvolvimento de aplicações seguras com socket.

Introdução

As aplicações para os computadores evoluíram ao longo do tempo. Nos primeiros sistemas computacionais as aplicações eram executadas em um computador e, normalmente, não precisavam acessar recursos externos ou comunicar com outras aplicações.

Conforme as redes de computadores tornaram-se mais presentes no cotidiano das empresas e das pessoas, as aplicações deixaram de ser executadas localmente, no próprio computador, e passaram a trocar informações através da rede.

Com o surgimento e crescimento exponencial da Internet, a troca de informações entre as aplicações tornou-se essencial. Hoje, quando utilizamos uma aplicação, as informações não estão apenas na aplicação instalada no seu dispositivo, mas podem estar espalhadas em diversos dispositivos na Internet, tornando essencial a troca de dados através da rede.

Para que possamos entender como essa comunicação ocorre, nós exploraremos o desenvolvimento de aplicações por meio de sockets, empregando a linguagem Python. Vamos iniciar entendendo o que é um socket, os tipos existentes e como eles permitem a troca de informações entre programas da camada de aplicação.

Depois, passaremos pelas implementações de socket com os protocolos TCP e UDP e, por fim, discutiremos alguns pontos importantes para o desenvolvimento de aplicações seguras.

Antes de continuarmos, assista ao vídeo a seguir.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Protocolos e a camada de aplicação

Confira a arquitetura, elementos e serviços da camada de aplicação em que os desenvolvedores atuam, desenvolvendo a interface e os protocolos da camada de aplicação.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Arquitetura em camadas

Um dispositivo de rede, como um computador, notebook, celular, entre outros, não tem utilização prática se não for possível utilizar algum tipo de serviço. Os serviços são oferecidos pelo que chamamos aplicações de rede. Para que essas aplicações de rede sejam funcionais, elas trocam mensagens, que costumamos chamar de pacotes que, geralmente, não são maiores do que alguns milhares de bytes.

As aplicações de rede são executadas na camada de aplicação da arquitetura Internet ou arquitetura TCP/IP, por meio de processos que são executados nessa camada. Para fins de estudo, vamos considerar a arquitetura Internet com 5 camadas:

- Físico
- Enlace
- Rede
- Transporte
- Aplicação

O modelo OSI possui sete camadas e também é bastante conhecido. Listando do topo até a base, temos as camadas (de): aplicação, apresentação, sessão, transporte, rede, enlace, física.

Em uma arquitetura em camadas, temos os seguintes elementos:

Serviços

Família de funções ou serviços. Cada camada requisita os serviços da camada imediatamente inferior a ela e oferece serviços para a camada imediatamente acima. Os serviços definem a comunicação vertical, com a camada superior consumindo serviços da camada inferior, até chegar na camada física.

Protocolos

Regras para a execução dos serviços, que ocorre entre máquinas. A camada "x" em uma máquina se comunica com a camada "x" da outra máquina, orientada por uma série de regras e convenções estabelecidas.

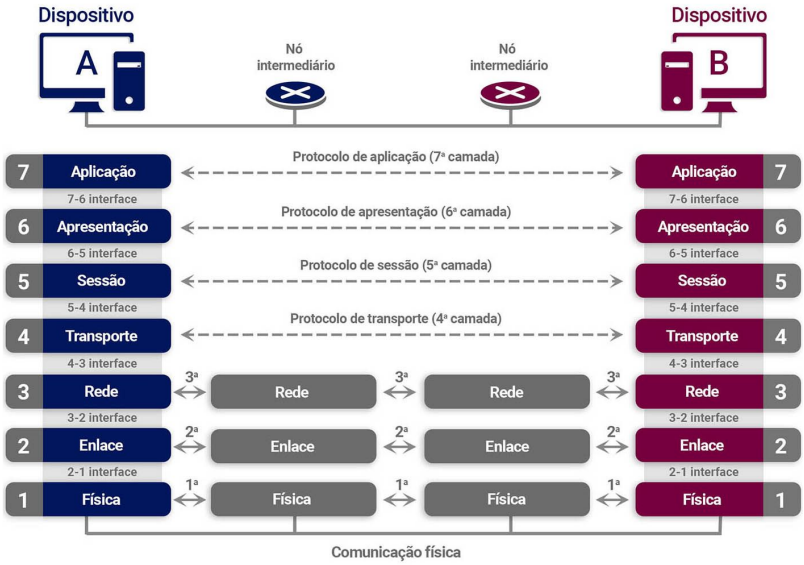
Interfaces

Ponto de comunicação entre cada par de camadas adjacentes. Cada interface define as informações e serviços que uma camada deve fornecer para a camada superior. Em cada máquina (ou seja, em uma mesma pilha de camadas), existe um ponto de interface.

As interfaces estão entre as camadas, veja:

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Interação entre camadas no modelo OSI.

Apesar de apresentarmos o conceito dos serviços, protocolos e interfaces no modelo OSI, ele é válido também para a arquitetura TCP/IP, na qual vamos concentrar nosso estudo.

Serviços da camada de aplicação

A camada de aplicação tem por finalidade oferecer aos usuários da rede acesso aos serviços de rede, como transferência de arquivos, troca de mensagens eletrônicas, chats, jogos, entre diversos outros. Os serviços de rede serão implementados nos processos executados na camada de aplicação.

Podemos afirmar que é na camada de aplicação que os desenvolvedores de aplicações de rede atuarão. Estes serão responsáveis por definir que tipo e como esse serviço será oferecido para os usuários, ou seja, ele terá que desenvolver como será a interface da aplicação e como as aplicações trocarão informações. Podemos dizer então que o desenvolvedor criará um protocolo da camada de aplicação.

Reflexão

Vale a pena ressaltar, principalmente para aqueles que já atuam como desenvolvedores de aplicação e que devem estar se perguntando: “Ora, eu desenvolvo aplicações, mas nunca criei um protocolo!” Você está certo. Normalmente, você cria uma aplicação Web, que utiliza o protocolo HTTP (hypertext transfer protocol) para realizar a troca de informações e todo o processo de troca de informações fica mascarado, porque o navegador é que se encarregará de fazer isso.

Mas você, que quer se tornar um excelente profissional de TI, precisa entender o que ocorre por trás disso tudo. Como efetivamente ocorre essa comunicação entre as aplicações? É o que vamos saber agora!

Interface socket e comunicação das aplicações

Entenda o conceito de socket e como uma interface entre a camada de aplicação e a camada de transporte é implementada por meio da API do sistema operacional.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

A API (Application Programming Interface) socket

Para que as aplicações possam efetivamente trocar informações, as mensagens que são geradas em um host de origem precisam ser entregues ao host de destino, obviamente. Essa tarefa pode parecer simples, mas não é!

Para que os dados possam percorrer as redes até alcançarem o seu destino, diversos serviços precisam ser executados, como: definir o endereço, definir o caminho, verificar erros que possam ter ocorrido, entre outros.

E é agora que entra a magia da arquitetura de camadas.

Como você viu, uma camada utiliza os serviços da camada imediatamente inferior. Então, a camada de aplicação consumirá os serviços da camada de transporte; esta consumirá os serviços da camada de rede que, por sua vez, consumirá os serviços da camada de enlace e, por fim, consumirá os serviços da camada física.

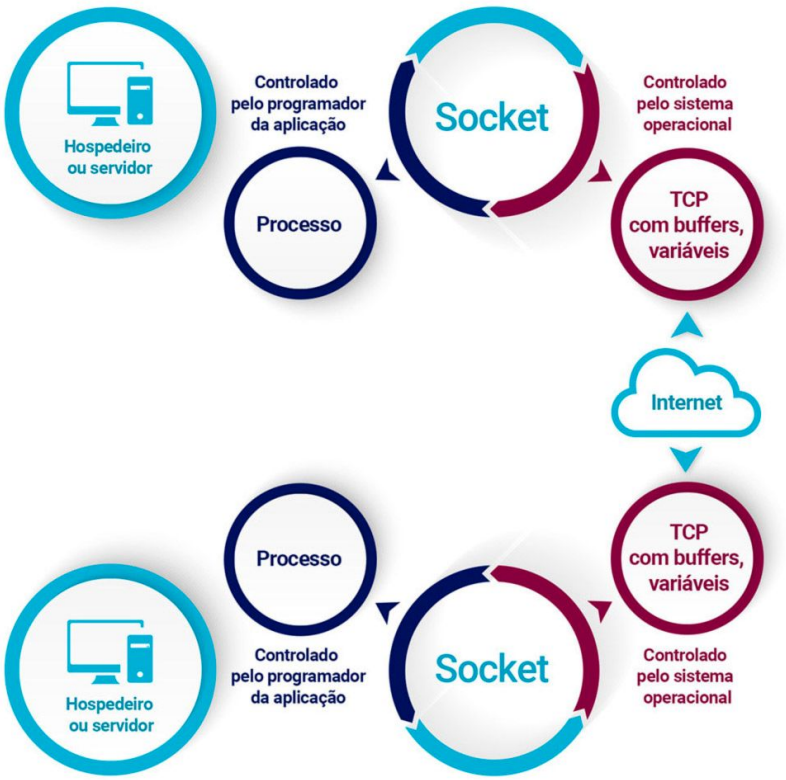
E como a camada de aplicação consome os serviços da camada de transporte?

> Por meio da interface entre a camada de aplicação e de transporte! E esta interface tem um nome: socket!

Um **socket** é a interface entre a camada de aplicação e a de transporte dentro de um host que é implementada como uma API (Application Programming Interface – Interface de Programação de Aplicação) do sistema operacional. O socket permite que o processo de origem possa entregar dados ao processo de destino.

Um processo é semelhante a uma casa e, seu socket, à porta da casa. Quando um processo deseja enviar uma mensagem para um processo em outro host, ele empurra a mensagem pela porta (socket). Ao chegar ao host de destino, a mensagem passa pela porta (socket) do processo receptor, que então executa alguma ação sobre a mensagem.

A imagem a seguir ilustra esse conceito e nos permite perceber que os processos da camada de aplicação são controlados pelo desenvolvedor da aplicação, enquanto os protocolos TCP/IP são controlados pelo sistema operacional. Vejamos:



Comunicação entre processos.

Serviços da camada de transporte

Para que os serviços de rede possam funcionar corretamente, é necessário adicionar dois serviços essenciais aos aplicativos dos hosts (usuários finais):

Multiplexação

Os pacotes que viajam entre dois hosts precisam ser rotulados para que os pacotes do serviço Web possam ser distinguidos dos pacotes de e-mail, e para que ambos possam ser separados de quaisquer outras conversas de rede nas quais a máquina esteja envolvida.

Transporte confiável

Todos os dados enviados em um host de origem devem chegar íntegros no host de destino. Os pacotes perdidos precisam ser retransmitidos até que cheguem inteiramente. Os pacotes que chegam fora de ordem precisam ser remontados na ordem correta, e os pacotes duplicados precisam ser descartados para que nenhuma informação no fluxo de dados seja repetida.

Os serviços citados são oferecidos pela camada de transporte e, na Internet, os dois principais protocolos são:

UDP (User Datagram Protocol)

Resolve apenas o primeiro dos dois problemas descritos. Ele fornece números de porta, para que os pacotes destinados a diferentes serviços em uma única máquina possam ser demultiplexados, ou seja, possam ser entregues para o processo de destino correto. Porém, o UDP não oferece suporte à perda, duplicação e ordenação de pacotes.

TCP (Transmission Control Protocol)

Resolve ambos os problemas. Oferece números de porta, fluxos de dados ordenados e confiáveis que ocultam dos aplicativos o fato de que o fluxo contínuo de dados foi dividido em pacotes e remontado na outra extremidade.

Multiplexação dos fluxos de rede

A **multiplexação** é uma técnica que permite que várias conversas compartilhem um mesmo meio. A **porta de origem** identifica o processo ou programa específico que enviou o pacote da máquina de origem, e a **porta de destino** especifica o aplicativo no endereço IP de destino ao qual a comunicação deve ser entregue.

Ao tomar decisões sobre a definição de números de porta, a IANA (*Internet Assigned Numbers Authority*) considera que eles se enquadram em três intervalos, aplicando-se aos números de porta UDP e TCP:

Portas bem conhecidas (0–1023)

Serviços mais importantes e amplamente usados. Em muitos sistemas operacionais, os programas de usuário normais não podem funcionar nessas portas.

Portas registradas (1024–49151)

Serviços não tratados como especiais pelos sistemas operacionais – qualquer usuário pode escrever um programa que utilize a porta 5432 e fingir ser um banco de dados PostgreSQL, por exemplo. No entanto, essas portas podem ser registradas pela IANA para serviços específicos, e a IANA recomenda que você evite usá-los para qualquer coisa que não seja o serviço atribuído.

Portas dinâmicas (49152–65535)

Serviços gratuitos para qualquer uso. Constituem o pool no qual os sistemas operacionais modernos se baseiam para gerar números de porta arbitrários quando um cliente não se importa com a porta atribuída para sua conexão de saída.

Uma cópia atualizada contendo as portas lógicas é mantida pela IANA em seu site na Web.

Endereçamento do socket

Confira como as aplicações são endereçadas por meio da combinação IP, Porta e, assim, como identificar os processos de forma exclusiva.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Então, para que tudo funcione a contento, é necessário endereçar as aplicações para que seja possível realizar a troca de mensagens.

> Para que essa comunicação possa ocorrer, é necessário definir um endereço para o socket combinando um endereço IP (Internet Protocol) e um número de porta.

Assim, o endereço socket no cliente define o processo cliente de forma exclusiva, da mesma forma que o endereço socket no servidor estabelece o processo servidor de modo exclusivo.

O protocolo de camada de transporte precisa de um par de endereços socket: o endereço socket no cliente e o endereço socket no servidor. Essas informações fazem parte do cabeçalho IP e do cabeçalho do protocolo de camada de transporte. O cabeçalho IP contém os endereços IP; o cabeçalho UDP ou TCP contém os números das portas.

No protocolo UDP, que não é orientado à conexão, o processo emissor definirá o endereço do socket de destino que consiste no endereço IP e no número de porta do destino . Convém ressaltar que também será enviado o endereço do socket de origem, para que seja possível o processo de destino responder ao processo originário.

Já o TCP é um protocolo orientado à conexão, ou seja, eles precisam primeiramente se apresentar e estabelecer uma conexão, para então começar a enviar dados. Um lado dessa conexão está ligado ao socket cliente e o outro está ligado a um socket servidor. Ao criar a conexão, associamos a ela o endereço de socket (endereço IP e número de porta) do cliente e do servidor . Uma vez estabelecida a conexão, quando um lado quer enviar dados para o outro, basta deixá-los na conexão TCP por meio do socket.

As primitivas de sockets

Confira as primitivas do socket Berkeley e o fluxo de funcionamento para um socket TCP.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Berkeley Socket

Após as aplicações serem endereçadas, podemos partir efetivamente para a comunicação entre os processos da camada de aplicação. Para que essa comunicação ocorra, são utilizadas as chamadas “primitivas de sockets”.

Vejamos a seguir as primitivas de sockets aplicadas no UNIX de Berkeley para o TCP, amplamente utilizadas em programação para a Internet e seus significados:

SOCKET
Cria um novo ponto final de comunicação.

BIND
Anexa um endereço local a um soquete.

LISTEN
Anuncia a disposição para aceitar conexões; mostra o tamanho da fila.

ACCEPT
Bloqueia o responsável pela chamada até uma tentativa de conexão ser recebida.

CONNECT
Tenta estabelecer uma conexão ativamente.

SEND
Envia alguns dados por meio da conexão.

RECEIVE
Recebe alguns dados da conexão.

CLOSE
Encerra a conexão.

Considerando que a arquitetura da aplicação seja do tipo cliente servidor, vamos analisar a sequência que será executada em cada lado do programa.

Lado servidor

Nesse lado, a primeira primitiva a ser invocada é a SOCKET, para criar o socket que será utilizado nos processos seguintes.

Após criar o socket, o programador precisa vincular o socket à porta, utilizando a primitiva BIND. Vinculada a porta, agora o socket precisa ficar disponível para aceitar conexões, isso é feito chamando a primitiva LISTEN. A primitiva ACCEPT será a próxima a ser executada, e que bloqueará o programa esperando chegar um pedido de conexão do lado cliente.

Nesse momento, se você executar o comando netstat para visualizar as conexões de rede, aparecerá algo semelhante à linha a seguir, para o programa que você executou:

> TCP IP_LOCAL:PORTA_LOCAL IP_DESTINO:PORTA_DESTINO LISTENING

Perceba que nessa linha será apresentado o endereço do socket de origem e o endereço do socket de destino, informando que está no modo LISTENING, ou seja, o soquete está apto a receber pedidos de conexão.

Após o cliente enviar o pedido de conexão e esta for estabelecida, agora entrará a lógica do programa propriamente dita, ocorrendo a troca de informações com as primitivas SEND e RECEIVE.

Como tipicamente em uma aplicação cliente-servidor é o cliente que envia a requisição, pode-se imaginar que a primeira primitiva do lado servidor será a RECEIVE, para que seja possível receber a requisição, tratá-la e enviar uma mensagem de resposta por meio da primitiva SEND.

Ao terminar a execução do programa, será chamada a primitiva CLOSE.

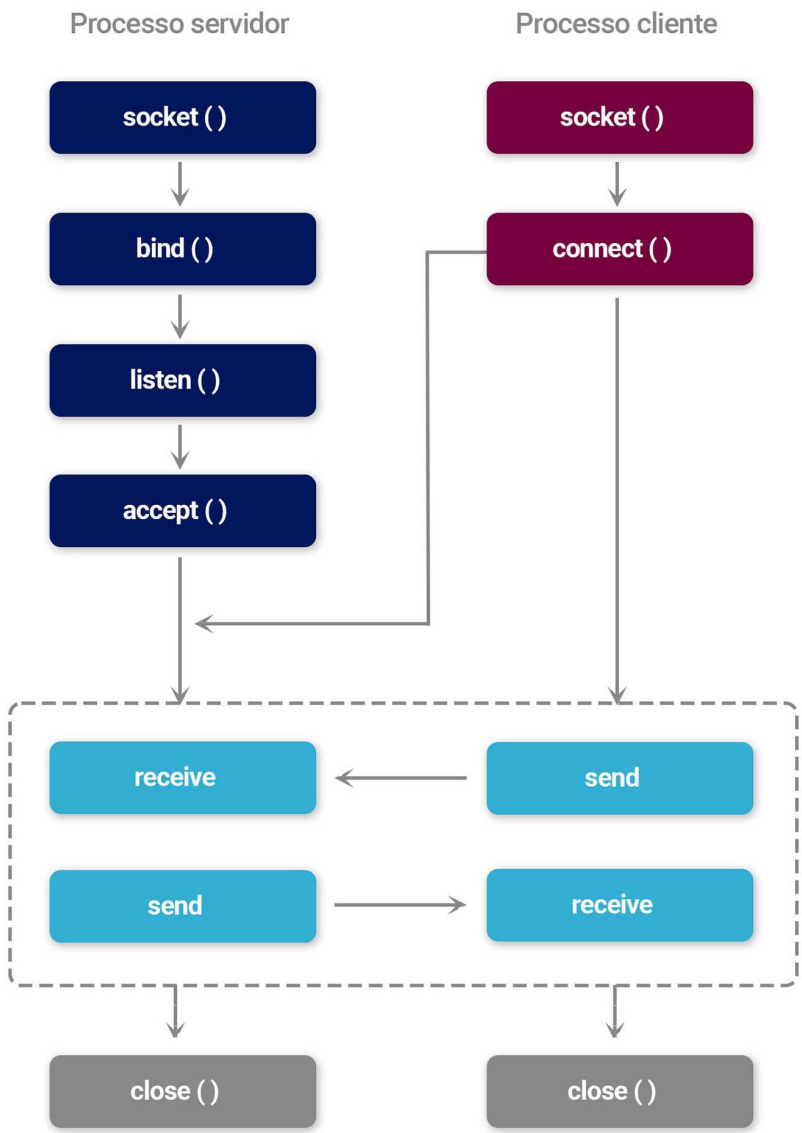
Lado cliente

Esse lado da comunicação será mais simples. A primeira primitiva a ser chamada é a SOCKET, cria-se o socket do lado cliente e depois é chamada a primitiva CONNECT para estabelecer conexão com o lado servidor.

Após a conexão ser estabelecida, serão enviados e recebidos dados entre o cliente e o servidor para que os serviços possam ser executados.

Em uma aplicação típica, o cliente enviará uma requisição ao servidor utilizando a primitiva SEND e receberá a resposta com a primitiva RECEIVE.

Veja o processo de comunicação entre os dois processos:



Fluxo de comunicação entre cliente e servidor.

Verificando o aprendizado

Questão 1

A programação para Internet envolve, dentre outros aspectos, a programação de socket.

Podemos dizer que o endereço de um **socket** é a combinação entre

A

endereço IP e nenhum número de porta.

B

nenhum endereço IP e número de porta.

C

endereço IP e número de porta.

D

nenhum endereço IP e dois números de porta.

E

dois endereços IP e nenhum número de porta.

A alternativa C está correta.

No UDP, sem conexão, o processo emissor inclui no pacote um endereço de destino que consiste no endereço IP e no número de porta do destino, tupla . No TCP, ao criar a conexão, associamos a ela o endereço de socket (endereço IP e número de porta) do cliente e do servidor, tupla .

Questão 2

Ao tomar decisões sobre a definição de números de porta, a IANA (Internet Assigned Numbers Authority) considera que eles se enquadram em três intervalos, aplicando-se aos números de porta UDP e TCP.

Dentre as opções a seguir, o número de porta que corresponde ao intervalo de portas registradas é

A

521.

B

49966.

C

62012.

D

20.

E

3289.

A alternativa E está correta.

As portas bem conhecidas variam entre 0 e 1023; as portas registradas variam entre 1024 e 49151; as portas gratuitas variam entre 49152 e 65535.

Desenvolvendo sockets UDP em Python

Todos os códigos utilizados neste conteúdo são desenvolvidos na linguagem Python 3, e executados em interpretador de linhas de comandos (também chamado de terminal ou prompt).

Python é uma linguagem de programação interpretada, orientada a objetos e de alto nível com semântica dinâmica. Suas estruturas de dados incorporadas de alto nível, combinadas com tipagem dinâmica e ligação dinâmica, o tornam muito atraente para o desenvolvimento rápido de aplicativos, bem como para uso como script ou linguagem de ligação para conectar componentes existentes (PYTHON, 2022).

A instalação do ambiente está fora do escopo deste conteúdo, mas é importante que você instale o ambiente, caso ainda não tenha.

A biblioteca padrão socket do Python fornece uma interface baseada em objetos para as chamadas de baixo nível do sistema operacional, que normalmente são usadas para realizar tarefas de rede em sistemas operacionais compatíveis com **POSIX**. Ou seja, trata-se de uma linguagem de nível superior que nos permite fazer chamadas de sistema operacional de baixo nível quando precisamos dela.

> POSIX
>
> O padrão POSIX foi criado com o intuito de padronizar as interfaces dos sistemas operacionais baseados no UNIX, permitindo compatibilidade entre eles. Outros sistemas operacionais podem aderir o POSIX, garantindo interoperabilidade

O código mostrado a seguir, adaptado de GitHub (2022), apresenta um servidor e um cliente UDP simples. A diferenciação na execução do código cliente ou código servidor está no momento que você executa o comando para rodar o programa.

Para executar como servidor, você executará o comando:

```
$ python udp_local.py servidor
```

Para executar como cliente, você executará o comando:

```
$ python udp_local.py cliente
```

Na função main do programa, será verificada qual opção foi escolhida e executada a função servidor ou cliente do código.

Considere que o código foi salvo com o nome “udp_local.py” para que você possa acompanhar as execuções posteriores. Vejamos um exemplo no código do **cliente**:

```
python
```

```
#!/usr/bin/env python3

import argparse, socket
from datetime import datetime

MAX_BYTES = 65535

def servidor(porta):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # Cria um socket simples
    sock.bind(('127.0.0.1', porta)) # Solicita um endereço de rede UDP
    print('Servidor >> Escutando no IP e porta {}'.format(sock.getsockname()))
    while True: # Executa repetidamente recvfrom()
        data, address = sock.recvfrom(MAX_BYTES) # Recebe mensagens ate 65.535 bytes;
retorna o endereço do cliente e conteúdo do datagrama no formato de bytes
        text = data.decode('ascii')
        print('Servidor >> O cliente no IP e porta {} enviou a mensagem {!
r}'.format(address, text))
        text = 'Mensagem para o cliente: O dado enviado possui comprimento de {}
bytes'.format(len(data))
        data = text.encode('ascii')
        sock.sendto(data, address) # Datagrama de resposta enviado ao cliente

def cliente(porta):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    text = 'Mensagem para o servidor: Hora atual {}'.format(datetime.now())
    data = text.encode('ascii')
    sock.sendto(data, ('127.0.0.1', porta)) # Possui uma mensagem e um endereço de destino
    print('Cliente >> O sistema operacional do cliente informou o IP e porta
{}'.format(sock.getsockname())) # O SO atribui um IP e porta, na saída da chamada
getsockname()
    data, address = sock.recvfrom(MAX_BYTES)
    text = data.decode('ascii')
    print('Cliente >> O servidor {} respondeu {!r}'.format(address, text))

if __name__ == '__main__':
    choices = {'cliente': cliente, 'servidor': servidor}
    parser = argparse.ArgumentParser(description='Enviar e receber UDP localmente')
    parser.add_argument('regra', choices=choices, help='Qual regra sera desempenhada.')
    parser.add_argument('-p', metavar='PORTA', type=int, default=1060, help='Porta UDP
(padrao: 1060)')
    args = parser.parse_args()
    function = choices[args.regra]
    function(args.p)
```

O código faz apenas uma chamada da Biblioteca Padrão do Python, para a função **socket.socket()**, e todas as outras chamadas são para os métodos do objeto socket que ele retorna.

Código do socket para o lado servidor

Confira o trecho do código referente ao socket UDP para o lado servidor e a função de cada linha de comando.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Vamos agora analisar o código do lado servidor!

O código-fonte do **servidor** funciona da seguinte maneira:

- Inicialmente é criado um socket simples com a chamada **socket.socket(socket.AF_INET, socket.SOCK_DGRAM)** e retorna o objeto sock que será utilizado para realizar as ações posteriores. Até agora, o socket não está vinculado a um endereço IP ou número de porta, não está conectado a nada e, por isso, vai gerar uma exceção se você tentar usá-lo para se comunicar. Como parâmetros, o socket é configurado como sendo da família **AF_INET**, a família de protocolos da Internet, e é do tipo de datagrama **SOCK_DGRAM**, indicando que usará o protocolo UDP.
- Em seguida, o comando **sock.bind(('127.0.0.1', porta))** vincula o endereço IP e porta ao socket criado. Como você pode perceber, é uma tupla Python simples, combinando um endereço IP do tipo string (**str**) e um número de porta UDP do tipo inteiro (**int**). Perceba que nesse exemplo estamos utilizando o endereço de loopback 127.0.0.1, o que permite e obriga você a executar o programa na mesma máquina. Caso você queira executar o cliente em uma máquina diferente do servidor, você precisa colocar o endereço correto.

Importante ressaltar que essa etapa pode falhar se outro programa já estiver usando essa porta UDP e o script do servidor não puder obtê-la. Por exemplo, se você já estiver executando o servidor, e tentar executar outra instância ao mesmo tempo, o programa retornará com uma mensagem do tipo:

```
python

$ python udp_local.py servidor
Traceback (most recent call last):
...
OSError: [Errno 98] Endereço já em uso
```

Vejamos mais detalhes:

- A linha seguinte (print('Servidor >> Escutando no IP e porta {}'.format(sock.getsockname()))) não é obrigatória, mas serve para exibir uma mensagem no terminal do servidor indicando que o servidor está em execução.
- Uma vez que o socket foi vinculado com sucesso, o próximo passo é o servidor receber as requisições dos clientes. Para isso, o servidor entra em loop infinito (while True:) e executa repetidamente o comando sock.recvfrom(MAX_BYTES), estando pronto para receber mensagens de até 65.535 bytes, o maior comprimento possível de um datagrama UDP.
- Quando o datagrama chega, o comando recvfrom() retorna o endereço do cliente que enviou um datagrama (address), bem como o conteúdo (data) do datagrama no formato de bytes.
- Como o Python é capaz de traduzir bytes diretamente em strings com o comando data.decode('ascii'), o servidor apresenta uma mensagem no console por meio do comando print('Servidor >> O cliente no IP e porta {} enviou a mensagem {!r}'.format(address, text).
- Por fim, o servidor envia uma mensagem para o cliente com o comando sock.sendto(data, address). Address é o endereço do cliente e data é o conteúdo que foi preparado na linha anterior.

Importante ressaltar que esse exemplo é simples, ele apenas recebe uma mensagem do cliente e envia uma mensagem de resposta informando o comprimento, em bytes da mensagem. Em um programa mais complexo, entre o comando **recvfrom()** e **sendto()** que estará a lógica do programa.

Exemplo

Suponha que você está desenvolvendo um programa para receber mensagens de requisição HTTP. Após receber a mensagem, seu programa verificará qual foi o método HTTP utilizado (GET, POST, HEAD...) e executará o trecho de código definido para o método.

Pode parecer algo muito complexo, mas não é tanto quanto você imagina!

Código do socket UDP para o lado cliente

Confira o trecho do código referente ao socket UDP para o lado cliente e a função de cada linha de comando.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Agora que já vimos o código do servidor, vamos executá-lo. Primeiramente, inicie o **servidor** digitando o seguinte comando no seu prompt:

```
python

$ python udp_local.py servidor
```

A execução do programa exibirá a seguinte saída na tela:

```
python

Servidor >> Escutando no IP e porta ('127.0.0.1', 1060)
```

A partir de agora, o servidor aguardará alguma mensagem que for enviada pelo cliente.

O código do **cliente**, já apresentado, será examinado agora:

- O primeiro passo é criar o socket com o comando: sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM). O objeto sock que foi retornado será utilizado para as ações de envio e recebimento das mensagens.
- A linha text = 'Mensagem para o servidor: Hora atual {}'.format(datetime.now()) preparou uma mensagem a ser enviada ao servido e armazenada na variável text.
- A linha data = text.encode('ascii') codificou os dados para envio e armazenou na variável data.
- O comando sock.sendto(data, ('127.0.0.1', porta)) envia a mensagem para o servidor, sendo data a mensagem a ser enviada e a tupla ('127.0.0.1', porta) informa o endereço e porta do servidor. O sendto() contém tudo o que é necessário para enviar um datagrama para o servidor.
- Perceba que no cliente não existe o bind(), porque o sistema operacional atribui automaticamente um endereço IP e número de porta, no lado do cliente. Na linha seguinte é exibido no terminal do cliente o endereço e porta utilizada, na saída da chamada para sock.getsockname().

- Semelhante ao que ocorreu no servidor, a linha data, address = sock.recvfrom(MAX_BYTES) recebe os dados enviados pelo programa servidor, armazenando o conteúdo na variável data e o endereço na variável address.
- As duas linhas subsequentes decodificam os dados e exibem no terminal do cliente a mensagem que foi recebida.

Vamos executar o cliente e ver o que acontece!

Enquanto o servidor ainda estiver em execução, abra outra janela de comando em seu sistema, e execute o cliente duas vezes seguidas, veja:

```
python

$ python udp_local.py cliente
Cliente >> O sistema operacional do cliente informou o IP e porta ('0.0.0.0', 46056)
Cliente >> O servidor ('127.0.0.1', 1060) respondeu 'Mensagem para o cliente: O dado
enviado possui comprimento de 63 bytes'

$ python udp_local.py cliente
Cliente >> O sistema operacional do cliente informou o IP e porta ('0.0.0.0', 39288)
Cliente >> O servidor ('127.0.0.1', 1060) respondeu 'Mensagem para o cliente: O dado
enviado possui comprimento de 63 bytes'
```

A janela do servidor relatará cada conexão atendida:

```
python

Servidor >> O cliente no IP e porta ('127.0.0.1', 46056) enviou a mensagem 'Mensagem para
o servidor: Hora atual 2022-11-15 23:08:56.025842'

Servidor >> O cliente no IP e porta ('127.0.0.1', 39288) enviou a mensagem 'Mensagem para
o servidor: Hora atual 2022-11-15 23:08:56.025842'
```

De forma semelhante ao que foi comentado sobre a lógica do programa no servidor, no cliente teremos que acrescentar a lógica antes da função **sendto()** e após o **recvfrom()**. Antes do **sendto()** estará a preparação da requisição a ser enviada ao servidor e após o **recvfrom()** estará o tratamento da mensagem de resposta.

Exemplo

Se for um cliente para o protocolo HTTP, o programa terá que verificar o código de resposta e, de acordo com o código, tomar a ação necessária. Se recebeu o código 200 ok, enviar o conteúdo para o interpretador HTML; se for o famoso 404 Page Not Found, exibir uma mensagem na tela indicando que o objeto solicitado não foi encontrado.

Agora é sua vez! Modifique o código visto e exercite criando outras lógicas, por exemplo, implementando um jogo da velha remoto!

Comunicação utilizando socket UDP

Confira todos os passos que são necessários para a execução do socket UDP e o que ocorre em cada etapa.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

A linha a seguir cria um socket:

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

O socket é do tipo **SOCK_DGRAM**, o que significa que é um socket

A

FTP.

B

IP.

C

TCP.

D

UDP.

E

STP.

A alternativa D está correta.

O socket do tipo **SOCK_DGRAM** pertence à família **AF_INET** de protocolos da Internet e indica/significa que é um socket UDP (em vez de um socket TCP).

Questão 2

A aplicação prática de sockets com UDP permite verificar o comportamento das primitivas de sockets.

Qual é o comando que solicita um endereço de rede UDP, formando uma tupla Python simples, combinando um endereço IP do tipo string (**str**) e um número de porta UDP do tipo inteiro (**int**)?

A

socket()

B

bind()

C

connect()

D

receive()

E

send()

A alternativa B está correta.

O comando **bind()** é responsável em anexar um endereço local a um soquete, isto é, consegue vincular um endereço IP e porta ao socket criado.

Sockets TCP em Python

Neste vídeo apresentaremos os tipos de sockets TCP e suas principais características.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O protocolo UDP é um protocolo não orientado à conexão e sem confiabilidade, portanto, não é adequado para aplicações que precisam de confiabilidade, como transferência de arquivos, acesso a páginas Web, entre outros.

Dica

Para os casos em que a aplicação depende de confiabilidade, o mais recomendável é que você utilize o protocolo TCP, utilizando um socket do tipo stream. O TCP implementa recursos que garantem que os dados enviados serão entregues, em ordem, sem duplicação, ou seja, realiza a entrega confiável!

Assim, o desenvolvedor não precisa se preocupar com esses detalhes. O processo envia os dados através do socket e o protocolo TCP se encarrega de todo o trabalho de garantir a confiabilidade.

Agora vem uma seguinte pergunta em mente:

> E se a minha aplicação requerer confiabilidade e eu quiser usar o UDP por ser um protocolo mais simples e rápido? É possível?

Sim, você pode utilizar o UDP, entretanto, você, desenvolvedor, vai ter que implementar a confiabilidade no nível da camada de aplicação, ou seja, você implementará a confiabilidade na própria aplicação que está desenvolvendo.

Características do socket TCP

As principais características do socket TCP estão relacionadas a ser orientado à conexão e tratar os dados como um fluxo de bytes (stream de bytes).

Por ser orientado à conexão, a troca de dados entre o cliente e o servidor não pode ser realizada de forma direta.

Será necessário passar pela fase de estabelecimento de conexão, quando o cliente enviará um pedido de abertura de conexão e o servidor aceitará, ou não, o pedido de conexão.

Quando o cliente envia o pedido de conexão, o TCP realizará o processo de abertura de conexão de 3 vias, conhecido como *Three Way Handshake*, e cuja apresentação do funcionamento está fora do escopo deste estudo.

Com relação ao envio ser como um fluxo de bytes, a troca de dados deverá ser um pouco mais cuidadosa, porque o conteúdo enviado pode ser uma mensagem inteira da camada de aplicação ou apenas um trecho de alguns bytes apenas.

Diferenças entre o socket TCP e UDP

Confira todos os passos do socket TCP que são necessários para a execução e o que ocorre em cada etapa, diferenciando com o funcionamento do socket UDP.

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

Para podermos fazer uma comparação entre o código utilizado no socket TCP e no UDP, vamos analisar o código em seguida. Considere que o código possui o nome de arquivo "tcp_exemplo.py".

De forma semelhante ao apresentado no código para UDP, o mesmo código tem a porção cliente e servidor. A escolha de qual trecho será executado está na hora da chamada ao código, especificando cliente ou servidor na linha de comando a ser executada.

Para executar como servidor, utilize:

```
python
$ tcp_exemplo.py servidor "Endereço IP"
```

Para executar como cliente, utilize:

```
python
$ tcp_exemplo.py cliente "Endereço IP”
```

Analise a seguir o codigo: **tcp_exemplo.py**

python

```
#!/usr/bin/env python3
# Cliente e servidor TCP simples que enviam e recebem 16 octetos

import argparse, socket

def recvall(sock, length):
    data = b''
    while len(data) < length:
        more = sock.recv(length - len(data))
        if not more:
            raise EOFError('esperava %d bytes, mas recebeu apenas %d bytes antes do
fechamento do socket' % (length, len(data)))
        data += more
    return data

def servidor(interface, port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # SO_REUSEADDR permite que
um socket seja vinculado à força a uma porta em uso por outro socket
    sock.bind((interface, port)) # Reivindica uma porta especifica
    sock.listen(1) # O programa decide virar um servidor
    print('Escutando em', sock.getsockname())
    while True:
        print('Esperando para aceitar uma nova conexao')
        sc, sockname = sock.accept() # Aguarda pedidos de conexões
        print('Aceito uma conexao de ', sockname)
        print('  Nome do socket:', sc.getsockname()) # Visualizar qual porta TCP o socket
está usando no local
        print('  Par do socket:', sc.getpeername()) # Visualizar o endereço do cliente ao
qual um soquete conectado está vinculado
        message = recvall(sc, 16)
        print('  Recebendo mensagem do tipo 16 octetos:', repr(message))
        sc.sendall(b'Fechado, cliente') # Envia os dados. Obs.: Atencao ao conteudo
        sc.close()
        print('  Resposta enviada, socket fechado')

def cliente(host, port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((host, port)) # Inicia o three-way handshake entre o cliente e o servidor
    print('O cliente recebeu um nome de socket', sock.getsockname())
    sock.sendall(b'Enviei, servidor') # Envia os dados. Obs.: Atencao ao conteudo
    reply = recvall(sock, 16)
    print('O servidor diz', repr(reply))
    sock.close()

if __name__ == '__main__':
    choices = {'cliente': cliente, 'servidor': servidor}
    parser = argparse.ArgumentParser(description='Enviar e receber usando TCP')
    parser.add_argument('regra', choices=choices, help='Qual regra sera desempenhada.')
    parser.add_argument('host', help='interface o servidor escuta; host o cliente envia')
    parser.add_argument('-p', metavar='PORT', type=int, default=1060, help='Porta TCP
(padrao: 1060)')
    args = parser.parse_args()
    function = choices[args.regra]
    function(args.host, args.p)
```

Vamos verificar diferenças desse código para o código UDP iniciando pelo lado cliente.

A primeira diferença é a chamada **sock.connect((host, port))** que inicia o three-way handshake (handshake de três vias) entre o cliente e o servidor, para que eles estejam prontos para se comunicar. Perceba nessa

chamada que estão indicados os parâmetros host, correspondente ao endereço de destino, e port, indicando a porta de destino. Após o estabelecimento da conexão, o cliente pode enviar os dados para o servidor.

Caso não seja possível estabelecer a conexão, será gerada uma mensagem de erro, como pode ser verificado executando o cliente quando o servidor não estiver em execução.

```
python

ConnectionRefusedError: [WinError 10061] Nenhuma conexão pôde ser feita porque a máquina
de destino as recusou ativamente
```

A segunda diferença é que o cliente TCP tende a ser mais simples do que o cliente UDP. Como ele não precisa fazer nenhuma provisão para pacotes perdidos, devido às garantias que o TCP fornece, o cliente pode enviar dados sem parar para verificar se a extremidade remota os recebe, e receber sem necessidade de considerar a possibilidade de retransmitir sua solicitação. O cliente pode ter certeza de que o TCP executará qualquer retransmissão necessária para obter seus dados.

No código apresentado para o socket UDP foram utilizados os comandos:

sendto()	recvfrom()
Significa “enviar este datagrama”.	Significa “receber um datagrama”.

Cada datagrama chega ou não como uma unidade de dados independente. Um aplicativo nunca verá datagramas UDP enviados pela metade ou recebidos pela metade. Já o TCP pode dividir seu fluxo de dados em pacotes de vários tamanhos diferentes durante a transmissão, e reagrupá-los gradualmente no receptor. Embora isso seja improvável de acontecer com as pequenas mensagens de 16 octetos do nosso exemplo, o código precisa estar preparado para essa possibilidade.

Vamos analisar o que ocorre quando **send()** é executado utilizando o socket TCP. A pilha de protocolos de rede do seu sistema operacional enfrentará uma das três situações:

1. Os dados podem ser imediatamente aceitos pela pilha de rede do sistema local, e **send()** retorna imediatamente o comprimento da string de dados como valor de retorno, pois toda a string está sendo transmitida.
2. Se a placa de rede estiver ocupada e o buffer de dados de saída para o socket estiver cheio, o comportamento padrão de **send()** será bloquear, pausando seu programa até que os dados possam ser aceitos para transmissão.
3. Se o buffer de saída estiver quase cheio, parte dos dados que você está tentando enviar podem ser imediatamente enfileirados, mas o restante do bloco de dados terá que esperar. Nesse caso, o **send()** é concluído imediatamente e retorna o número de bytes aceitos desde o início de sua string de dados, deixando de fora o restante dos dados não processados. Devido a essa possibilidade, se você estiver programando um código desse tipo, deverá sempre verificar o valor de retorno, e colocar uma chamada send() dentro de um loop que, no caso de uma transmissão parcial, continuará tentando enviar os dados restantes até que toda a cadeia de bytes seja enviada.
4. Para evitar esse problema, no nosso código de exemplo, fazemos isso de modo mais amigável utilizando o método **sendall()**, que se encarrega de realizar esse tratamento.

A implementação no sistema operacional do **recv()** usa uma lógica muito próxima da utilizada no envio:

- Se nenhum dado estiver disponível, recv() é bloqueado e seu programa pausa até que os dados cheguem.
- computer
  Se muitos dados já estão disponíveis no buffer de entrada, você recebe tantos bytes quantos você deu permissão a recv() para entregar.

- computer
  Se o buffer contiver apenas alguns dados em espera, mas não tanto quanto você deu permissão a recv() para retornar, você receberá imediatamente o que estiver lá, mesmo que não seja tanto quanto você solicitou.

A chamada **recv()** deve estar dentro de um loop. O sistema operacional não pode adivinhar quando os dados recebidos podem finalmente ser somados ao que seu programa considera como sendo uma mensagem completa, e assim ele fornece todos os dados que puder o mais rápido possível.

Atenção

A biblioteca padrão do Python não inclui um equivalente de sendall() para o método recv(). Um provável motivo é devido às mensagens de tamanho fixo serem incomuns hoje em dia. A maioria dos programas do mundo real executa o loop recv() de modo mais complicado do que o do nosso exemplo, pois geralmente é preciso ler ou processar parte da mensagem antes de "adivinhar" quanto mais está por vir. Para simplificar o nosso exemplo, foi definida uma função recvall() que implementa o loop e está sendo utilizada no lado cliente e lado servidor.

Detalhamento da aplicação prática de sockets com TCP

Neste vídeo apresentaremos os tipos de sockets de fluxo. Confira!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Existem dois tipos diferentes de sockets de fluxo (stream sockets):

Socket de escuta

Tipo de socket no qual os servidores **disponibilizam uma porta** para conexões de entrada.

Socket conectado

Tipo de socket que representa uma conversa que um servidor **está tendo com um cliente específico.**

Vamos seguir as etapas na listagem do programa descrito no código **tcp_exemplo.py** para ver a ordem em que as operações de soquete ocorrem.

Primeiro, o servidor executa **bind()** para reivindicar uma porta específica. Isso ainda não decide se o programa será um cliente ou servidor. O verdadeiro momento de decisão vem com a próxima chamada de método, quando o servidor anuncia que deseja usar o socket para **listen()**.

Após a chamada de **listen()**, o socket agora pode ser usado apenas para receber conexões de entrada por meio de seu método **accept()**, e cada uma dessas chamadas espera por um novo cliente para se conectar e, em seguida, retorna um socket totalmente novo que rege a nova conversa que acabou de começar com eles.

O método **getsockname()** funciona bem em sockets de escuta e conectados e, em ambos os casos, permite que você descubra qual porta TCP local o socket está usando. Para saber o endereço do cliente ao qual um soquete conectado está vinculado, você pode executar o método **getpeername()** a qualquer momento ou pode armazenar o nome do soquete que é retornado como o segundo valor de retorno de **accept()**. Ao executar esse servidor, você verá que ambos os valores fornecem o mesmo endereço:

```
python

$ tcp_exemplo.py servidor "127.0.0.1"
Escutando em ('127.0.0.1', 1060)
Esperando para aceitar uma nova conexao
Aceito uma conexao de ('127.0.0.1', 57099)
  Nome do socket: ('127.0.0.1', 1060)
  Par do socket: ('127.0.0.1', 57099)
  Recebendo mensagem do tipo 16 octetos: b'Enviei, servidor'
  Resposta enviada, socket fechado
Esperando para aceitar uma nova conexao
```

Quando o cliente realiza uma conexão com o servidor, produz a seguinte saída:

```
python

$ C:\tcp_exemplo.py cliente "127.0.0.1"
O cliente recebeu um nome de socket ('127.0.0.1', 57099)
O servidor diz b'Fechado, cliente'
```

A chamada **recv()** retorna dados à medida que se tornam disponíveis, e **sendall()** é a melhor maneira de enviar um bloco inteiro de dados quando você deseja garantir que tudo seja transmitido.

Um argumento de número inteiro foi fornecido para **listen()** quando ele foi chamado no socket do servidor. Esse número indica quantas conexões em espera, que ainda não tiveram sockets criados para elas por chamadas **accept()**, devem ser empilhadas antes que o sistema operacional comece a ignorar novas conexões e adiar quaisquer handshakes de três vias adicionais. O valor 1 utilizado aqui neste exemplo significa o suporte a apenas um cliente conectando por vez.

Depois que o cliente e o servidor dizem tudo o que eles precisavam dizer, eles fecham a extremidade do socket com **close()**, que diz para o sistema operacional transmitir quaisquer dados restantes no buffer de saída e, em seguida, concluir a sessão TCP com o procedimento de desligamento do pacote FIN.

Endereço já em uso

Há um último detalhe no código, que é o fato de o servidor definir a opção de **SO_REUSEADDR** antes de tentar vincular a uma porta. Para verificar as consequências de não definir essa opção, comente a linha, acrescentando o caractere **#** no início dela, ficando do seguinte modo:

```
python

#sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

Faça o seguinte: abra uma janela de prompt e execute o servidor; mantenha essa janela aberta em execução, e abra outra janela de prompt e, ao invés de executar o cliente, execute o servidor. Você receberá um erro:

```
python

OSError: [WinError 10048] Normalmente é permitida apenas uma utilização de cada endereço
de soquete (protocolo/endereço de rede/porta)
```

A opção **SO_REUSEADDR** permite que um socket seja vinculado à força a uma porta em uso por outro socket. Se você continuar tentando executar o servidor sem a opção **SO_REUSEADDR**, o endereço não ficará disponível novamente até vários minutos após a última conexão do cliente. Isso acontece porque, mesmo depois que a pilha de rede envia o último pacote desligando o socket (FIN), esse pacote final pode se perder e precisar ser retransmitido algumas vezes antes que a outra extremidade finalmente o receba.

Resumindo

A solução adotada então é manter um registro da conexão TCP por até quatro minutos em um estado de espera. O RFC nomeia esses estados como CLOSE-WAIT e TIME-WAIT.

Verificando o aprendizado

Questão 1

O protocolo TCP é um protocolo da camada de transporte orientado a fluxos de bits. Existem dois tipos diferentes de sockets de fluxo. O socket de fluxo no qual os servidores disponibilizam uma porta para conexões de entrada é denominado

A

socket conectado.

B

socket de fluxo.

C

stream socket.

D

socket de escuta.

E

socket de datagrama.

A alternativa D está correta.

O tipo socket de escuta é o socket que necessita de uma porta para que ocorram as conexões de entrada. O outro tipo é socket conectado que é realizado por meio de uma conversa entre servidor e um cliente específico.

Questão 2

Em um código que implementa o socket TCP, qual é o nome do método responsável por iniciar o 3-way handshake?

A

connect()

B

send()

C

recv()

D

sendall()

E

listen()

A alternativa A está correta.

Uma conexão TCP é estabelecida da seguinte maneira: o cliente primeiro envia um segmento TCP especial; o servidor responde com um segundo segmento TCP especial e, por fim, o cliente responde novamente com um terceiro segmento especial. Como três segmentos são enviados entre dois hospedeiros, esse procedimento de estabelecimento de conexão é muitas vezes denominado apresentação de três vias (3-way handshake).

Clientes promíscuos e respostas indesejadas

Neste vídeo apresentaremos o código UDP inseguro e como pode ser resolvido. Veja mais!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Exemplo de código UDP inseguro

Considere o código de socket UDP “udp_local.py”, veja:

```
python
```

```
#!/usr/bin/env python3

import argparse, socket
from datetime import datetime

MAX_BYTES = 65535

def servidor(porta):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # Cria um socket simples
    sock.bind(('127.0.0.1', porta)) # Solicita um endereço de rede UDP
    print('Servidor >> Escutando no IP e porta {}'.format(sock.getsockname()))
    while True: # Executa repetidamente recvfrom()
        data, address = sock.recvfrom(MAX_BYTES) # Recebe mensagens ate 65.535 bytes;
retorna o endereço do cliente e conteúdo do datagrama no formato de bytes
        text = data.decode('ascii')
        print('Servidor >> O cliente no IP e porta {} enviou a mensagem {!
r}'.format(address, text))
        text = 'Mensagem para o cliente: O dado enviado possui comprimento de {}
bytes'.format(len(data))
        data = text.encode('ascii')
        sock.sendto(data, address) # Datagrama de resposta enviado ao cliente

def cliente(porta):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    text = 'Mensagem para o servidor: Hora atual {}'.format(datetime.now())
    data = text.encode('ascii')
    sock.sendto(data, ('127.0.0.1', porta)) # Possui uma mensagem e um endereço de destino
    print('Cliente >> O sistema operacional do cliente informou o IP e porta
{}'.format(sock.getsockname())) # O SO atribui um IP e porta, na saída da chamada
getsockname()
    data, address = sock.recvfrom(MAX_BYTES)
    text = data.decode('ascii')
    print('Cliente >> O servidor {} respondeu {!r}'.format(address, text))

if __name__ == '__main__':
    choices = {'cliente': cliente, 'servidor': servidor}
    parser = argparse.ArgumentParser(description='Enviar e receber UDP localmente')
    parser.add_argument('regra', choices=choices, help='Qual regra sera desempenhada.')
    parser.add_argument('-p', metavar='PORTA', type=int, default=1060, help='Porta UDP
(padrao: 1060)')
    args = parser.parse_args()
    function = choices[args.regra]
    function(args.p)
```

Podemos dizer que o procedimento **cliente** é "**perigoso**".

A função **recvfrom()** nunca verifica o endereço de origem (IP + porta) do datagrama que recebe, para verificar se, de fato, é uma resposta do servidor.

Você pode ver esse problema em funcionamento atrasando a resposta do servidor e vendo se outra pessoa pode enviar uma resposta na qual esse cliente ingênuo confiará.

Inicie um novo servidor, mas logo após o seu início, suspenda-o, conforme exemplo mostrado na execução a seguir.

- No Windows, usando PowerShell: pressione a tecla “Pause”.
- No Mac OS X e Linux: pressione a combinação de teclas “Ctrl+Z”.

Vejamos:

```
python

$ python udp_local.py server
Servidor >> Escutando no IP e porta ('127.0.0.1', 1060)
^Z
[1] + 9370 suspended python udp_local.py servidor
```

Execute, agora, o cliente, veja que ele enviará seu datagrama e depois travará, esperando receber uma resposta.

Atenção

Anote o número da porta informada.

No caso do exemplo mostrado a seguir, a porta é a de número 39692. Vejamos:

```
python

$ python udp_local.py cliente
Cliente >> O sistema operacional do cliente informou o IP e porta ('0.0.0.0', 39692)
```

Suponha que você seja um invasor, e deseja forjar uma resposta do servidor enviando seu datagrama antes que o servidor tenha a chance de enviar sua própria resposta.

Como o cliente disse ao sistema operacional que está disposto a receber qualquer datagrama, ele deve confiar que sua resposta falsa de fato se originou no servidor.

Você pode enviar esse pacote usando uma sessão rápida no prompt do Python. Abra um novo prompt, e digite as linhas de comando apresentadas abaixo.

No exemplo mostrado, a porta é a de número 39692, você inserirá o número da porta dentro da tupla, conforme o número da porta que você anotou quando fez a execução do cliente. No nosso caso, será a 39692.

```
python

$ python
Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 bit (AMD64)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import socket
>>> sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
>>> sock.sendto('FAKE'.encode('ascii'), ('127.0.0.1', 39692))
```

Veja que o prompt do cliente encerrará a janela, e informará a resposta enviada por terceiros (que você acabou de simular), como sendo a resposta pela qual estava esperando.

```
python
Cliente >> O servidor ('127.0.0.1', 50339) respondeu 'FAKE'
```

Veja a diferença do endereço dos servidores:

- "Falso" ('127.0.0.1', 50339).
- "Verdadeiro" ('127.0.0.1', 1060).

Descongele o servidor (digite **fg** no prompt Linux, ou pressione a tecla “enter” no PowerShell), e deixe-o continuar em execução. Agora, ele verá o pacote do cliente enfileirado e esperando por sua resposta, e enviará sua resposta para o socket do cliente que nesse momento se encontra fechado.

Você pôde ver que o cliente é vulnerável a qualquer usuário que possa endereçar um pacote UDP para ele.

Exemplo

Um remetente sem privilégios, operando completamente dentro das regras, envia um pacote com um endereço de retorno legítimo, e tem os seus dados aceitos. Um cliente de rede de escuta que aceita ou registra cada pacote que vê, sem levar em conta se o pacote está endereçado corretamente, é conhecido tecnicamente como um cliente promíscuo.

Soluções para esse problema incluem o uso de criptografia, o uso de protocolos que façam uso de um identificador exclusivo na solicitação que é repetida na resposta, a verificação do endereço do pacote de resposta com o endereço para o qual você enviou, o uso de funções específicas para que estabeleçam uma conexão fim a fim, dentre outras medidas possíveis.

Problemas na transmissão por meio de redes

Confiabilidade na comunicação com socket UDP

Nos exemplos vistos até o momento, tanto o cliente quanto o servidor estavam sendo executados na mesma máquina, conversando por meio da interface de loopback. Como não estavam usando uma placa de rede física, sujeita a falhas de sinalização, não conseguimos ver nenhuma das inconveniências do UDP, como pacotes que foram perdidos “no meio do caminho”.

Curiosidade

O protocolo IP provê um serviço de melhor esforço. Significa que ele faz seu melhor esforço para transportar um datagrama do remetente ao receptor o mais rapidamente possível, sem garantias sobre o atraso fim a fim ou sobre o limite de perdas de pacotes.

A perda pode ser eliminada com o envio dos pacotes por TCP em vez de por UDP. O TCP oferece transferência de dados confiável, porém os mecanismos de retransmissão são muitas vezes considerados

inaceitáveis para aplicações multimídia em tempo real (tipicamente áudio e vídeo), pois aumentam o atraso fim a fim. Além disso, após a perda de pacote, a taxa de transmissão no remetente pode ser reduzida a uma taxa mais baixa do que a de reprodução no receptor, por causa do controle de congestionamento do TCP. Resumindo, vejamos as diferenças:

TCP	UDP
Pacotes por TCP são usados tipicamente para dados enviados em rajada (ou seja, de modo intermitente dentro de um intervalo de tempo).	Pacotes por UDP são usados tipicamente para dados enviados em fluxo contínuo, ou streaming (ou seja, mantendo a taxa relativamente constante dentro de um intervalo de tempo).

Vamos ver o quanto a complexidade do código aumenta quando os pacotes puderem realmente ser perdidos, olhando o exemplo do código a seguir, adaptado de GitHub (2022). Chamamos esse arquivo de "**udp_remote.py**".

python

```
#!/usr/bin/env python3

import argparse, random, socket, sys

MAX_BYTES = 65535

def servidor(interface, port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((interface, port))
    print('Servidor >> Escutando no IP e porta', sock.getsockname())
    while True:
        data, address = sock.recvfrom(MAX_BYTES)
        if random.random() < 0.5: # Sorteio aleatório para decidir se essa solicitação
será atendida
            print('Servidor >> Fingindo descartar pacote de {}'.format(address))
            continue
        text = data.decode('ascii')
        print('Servidor >> O cliente no IP e porta {} enviou a mensagem {!
r}'.format(address, text))
        message = 'Mensagem para o cliente: O dado enviado possui comprimento de {}
bytes'.format(len(data))
        sock.sendto(message.encode('ascii'), address)

def cliente(hostname, port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.connect((hostname, port))
    print('Cliente >> Nome do socket do cliente: {}'.format(sock.getsockname()))

    delay = 0.1  # Tempo de espera inicial em segundos
    text = 'Mensagem para o servidor: Estou enviando uma mensagem'
    data = text.encode('ascii')
    while True:
        sock.send(data)
        print('Cliente >> Aguardando {} segundos para enviar a resposta'.format(delay))
        sock.settimeout(delay) # Define o tempo em que o cliente espera pela resposta do
servidor
        try:
            data = sock.recv(MAX_BYTES)
        except socket.timeout as exc: # Chamada seja interrompida com uma exceção
socket.timeout quando uma chamada tiver esperado por muito tempo
            delay *= 2  # Backoff exponencial: Espere ainda mais pelo próximo pedido
            if delay > 2.0: # Desiste depois de ter feito tentativas suficientes
                raise RuntimeError('Cliente >> Eu acho que o servidor caiu') from exc
        else:
            break   # terminamos e podemos parar de repetir

    print('Cliente >> O servidor respondeu {!r}'.format(data.decode('ascii')))

if __name__ == '__main__':
    choices = {'cliente': cliente, 'servidor': servidor}
    parser = argparse.ArgumentParser(description='Enviar e receber UDP, fingindo que os
pacotes são frequentemente descartados')
    parser.add_argument('regra', choices=choices, help='Qual regra sera desempenhada.')
    parser.add_argument('host', help='interface o servidor escuta; host o cliente envia')
    parser.add_argument('-p', metavar='PORTA', type=int, default=1060, help='Porta UDP
(padrao: 1060)')
    args = parser.parse_args()
    function = choices[args.regra]
    function(args.host, args.p)
```

Em vez de sempre responder às solicitações dos clientes, esse servidor escolherá, de **modo aleatório**, responder apenas metade das solicitações dos clientes. Esse comportamento permitirá que você veja rapidamente como criar confiabilidade no código do cliente, sem ter que aguardar um pacote ser descartado em um cenário de rede real.

Simulando a perda de dados

Diferentemente dos exemplos anteriores, nos quais explicitamos ao servidor que recebesse apenas os pacotes oriundos da mesma máquina por meio da interface 127.0.0.1, no código **udp_remote.py**, podemos especificar o endereço IP 0.0.0.0, que significa “qualquer interface local”.

```
python

$ python udp_remote.py server "0.0.0.0"
Servidor >> Escutando no IP e porta ('0.0.0.0', 1060)
```

Cada vez que uma solicitação é recebida, o servidor usará um sorteio **random()** para decidir se essa solicitação será atendida. Seja qual for a decisão tomada, uma mensagem é exibida na tela para que você possa acompanhar sua atividade.

Perda de dados na aplicação

Confira os mecanismos que podem ser utilizados para tratar a perda de dados na aplicação quando é utilizado o protocolo UDP.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Corrigindo a perda de dados

Como escrevemos um cliente UDP “real”, que tenha que lidar com pacotes que podem ser perdidos?

Primeiro, a falta de confiabilidade do UDP significa que o cliente deve estar preparado para esperar eternamente por uma resposta, ou então ser um tanto arbitrário ao decidir que esperou “muito tempo” por uma resposta, e que precisa enviar outra. Essa escolha difícil é necessária, pois geralmente não há como o cliente distinguir entre três eventos bastante diferentes:

1. A resposta está demorando muito para voltar, mas chegará em breve.
2. A resposta nunca chegará porque ela, ou a solicitação, foi perdida.
3. O servidor está fora do ar e não está respondendo a ninguém.

Um cliente UDP deve escolher um momento no qual enviará solicitações duplicadas, se estiver esperado um tempo razoável sem obter uma resposta. Em vez do sistema deixá-lo parado para sempre na chamada **recv()**, o cliente primeiro faz um **settimeout()** no socket, isto é, o cliente deseja que a chamada seja interrompida com uma exceção **socket.timeout** quando uma chamada tiver esperado por muito tempo.

O cliente do exemplo começa com um tempo de espera modesto de 0,1 segundo (um décimo de segundo). Em uma rede doméstica, onde os tempos de ping são geralmente algumas dezenas de milissegundos, isso raramente fará com que o cliente envie uma solicitação duplicada simplesmente porque a resposta demora para voltar.

Atenção

Uma característica importante desse programa é o que acontece se o tempo limite for atingido quando a técnica de backoff exponencial for utilizada, o que se torna cada vez menos frequente. Isso mantém o cliente ativo, ao mesmo tempo que possibilita a rede congestionada se recuperar lentamente, já que todos os outros clientes ativos também enviarão menos pacotes.

Embora existam algoritmos mais sofisticados para backoff exponencial, um modo simples de implementá-lo é dobrando o tempo de atraso, cada vez que uma resposta não é recebida. Observe que, se as solicitações estiverem sendo feitas para um servidor que está, por exemplo, a 200 milissegundos de distância, esse algoritmo simples sempre enviará pelo menos duas cópias de cada solicitação todas as vezes, pois nunca aprenderá que as solicitações para esse servidor sempre levam mais de 0,1 segundos.

```
python

$ python udp_remote.py cliente "127.0.0.1"
Cliente >> Nome do socket do cliente: ('127.0.0.1', 56196)
Cliente >> Aguardando 0.1 segundos para enviar a resposta
Cliente >> O servidor respondeu 'Mensagem para o cliente: O dado enviado possui
comprimento de 53 bytes'
```

Em algumas ocasiões, o cliente verá que uma ou mais de suas solicitações nunca resultaram em respostas do servidor, e terá que tentar novamente (no nosso código estamos "fingindo" haver um descarte). O exemplo a seguir mostra tentativas repetidas, que possibilitam ver o recuo exponencial acontecendo em tempo real, à medida que as mensagens na tela ficam cada vez mais lentas conforme o temporizador de atraso aumenta:

```
python

$ python udp_remote.py cliente "127.0.0.1"
Cliente >> Nome do socket do cliente: ('127.0.0.1', 64981)
Cliente >> Aguardando 0.1 segundos para enviar a resposta
Cliente >> Aguardando 0.2 segundos para enviar a resposta
Cliente >> Aguardando 0.4 segundos para enviar a resposta
Cliente >> Aguardando 0.8 segundos para enviar a resposta
Cliente >> O servidor respondeu 'Mensagem para o cliente: O dado enviado possui
comprimento de 53 bytes'
```

Você poderá ver no terminal do servidor se as requisições estão realmente sendo feitas ou se, por acaso, ocorreu um descarte de pacote, conforme a saída a seguir.

```
python

Servidor >> Fingindo descartar pacote de ('127.0.0.1', 64981)
Servidor >> Fingindo descartar pacote de ('127.0.0.1', 64981)
Servidor >> Fingindo descartar pacote de ('127.0.0.1', 64981)
Servidor >> Fingindo descartar pacote de ('127.0.0.1', 64981)
Servidor >> O cliente no IP e porta ('127.0.0.1', 64981) enviou a mensagem 'Mensagem para
o servidor: Estou enviando uma mensagem'
```

O UDP não distingue entre um servidor que está fora do ar e uma rede que está congestionada e descartando pacotes. O melhor que o cliente pode fazer é desistir depois de ter feito tentativas suficientes. Faça o teste,

suspenda o servidor (tecla “Pause” no PowerShell, ou “Ctrl+Z” no Mac OS X e Linux), e tente executar o cliente novamente.

```
python

Cliente >> Nome do socket do cliente: ('127.0.0.1', 49782)
Cliente >> Aguardando 0.1 segundos para enviar a resposta
Cliente >> Aguardando 0.2 segundos para enviar a resposta
Cliente >> Aguardando 0.4 segundos para enviar a resposta
Cliente >> Aguardando 0.8 segundos para enviar a resposta
Cliente >> Aguardando 1.6 segundos para enviar a resposta

Traceback (most recent call last):
  File "C:\udp_remote.py", line 34, in cliente
    data = sock.recv(MAX_BYTES)
TimeoutError: timed out

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
   File "C:\\udp_remote.py", line 52, in
     function(args.host, args.p)
   File "C:\udp_remote.py", line 38, in cliente
     raise RuntimeError('Cliente >> Eu acho que o servidor caiu') from exc
RuntimeError: Cliente >> Eu acho que o servidor caiu
```

Essa ação de desistir faz sentido se o seu programa estiver tentando executar alguma tarefa breve, e precisar produzir uma saída ou retornar algum tipo de resultado ao usuário.

Se você estiver escrevendo um programa daemon que roda o dia todo – como um ícone do tempo no canto da tela que exibe a temperatura e a previsão obtida de um serviço UDP remoto –, não há problema em ter um código que fica repetindo "para sempre". Nesse caso, pode-se escolher um atraso máximo, cinco minutos, e faça com que o backoff exponencial seja executado ao atingir esse período, para tentar uma atualização quando o usuário estiver na rede por cinco minutos após muito tempo desconectado. Melhor ainda, tentar adivinhar quando a rede poderá voltar.

Verificando o aprendizado

Questão 1

Suponha que um cliente está enviando dados para um servidor. O programa visto neste módulo possui a característica de, quando o tempo limite para que o cliente receba uma confirmação do servidor for esgotado, as tentativas de envio de dados a partir do cliente tornam-se cada vez menos frequentes. Qual é o nome específico que essa característica recebe?

A

Rollback

B

Aleatoriedade

C

Daemon

D

Ping

E

Backoff exponencial

A alternativa E está correta.

Quando o timer de retransmissão expira, o TCP retransmite o primeiro segmento não confirmado. Após cada expiração do tempo limite de retransmissão, a RFC 2988 recomenda dobrar o valor do tempo limite de retransmissão. Isso é chamado de backoff exponencial.

Questão 2

Na programação de socket UDP, foi visto que o procedimento cliente possui uma função específica que nunca verifica o endereço de origem do datagrama que recebe, para verificar se, de fato, é uma resposta do servidor. Dentre as opções a seguir, qual função se refere a esse fato?

A

decode()

B

sendto()

C

recvfrom()

D

encode()

E

bind()

A alternativa C está correta.

A função **recvfrom()** pode ser considerada como um cliente "perigoso", pois não realiza a verificação do endereço de origem (IP + porta) da mensagem que recebe, isto é, não verifica se é uma resposta do servidor. Dentre outras soluções para esse problema, pode-se realizar a verificação do endereço do pacote de resposta junto ao endereço para o qual você enviou.

Considerações finais

Vimos que são utilizados sockets para que os programas de aplicação possam trocar informações através da rede, e que os sockets são a interface entre a camada de aplicação e a camada de transporte, sendo implementado por meio de uma API do sistema operacional.

Apresentamos os dois tipos de sockets que podem ser utilizados: aquele que não garante a entrega das mensagens – UDP; e aquele que oferece a confiabilidade na comunicação fim a fim – TCP. Além disso mostramos exemplos de códigos em Python para implementação dos sockets.

Por fim, verificamos como podemos endereçar alguns problemas de segurança e confiabilidade quando utilizamos a comunicação via socket.

Podcast

Para encerrar, ouça um resumo sobre a importância do uso dos sockets para a comunicação dos programas da aplicação e como tratar os principais problemas que possam ocorrer.

Conteúdo interativo

Acesse a versão digital para ouvir o áudio.

Explore +

Pesquise sobre o **Guia Beej's Para Programação em Rede** e confira detalhes importantes sobre a comunicação de aplicações através de sockets.

Referências

FOROUZAN, B. A. **Comunicação de dados e redes de computadores**. 4. ed. São Paulo: McGraw-Hill, 2008.

GITHUB. **Brandon-rhodes – fopnp:** Foundations of Python Network Programming (Apress) — scripts and examples. Consultado na Internet em: 27 nov. 2022.

KUROSE, J. F.; ROSS, K. W. **Redes de computadores e a Internet**: uma abordagem top-down. 6. ed. São Paulo: Pearson, 2013.

PYTHON. **What is Python?** Executive Summary | Python.org. Consultado na Internet em: 27 nov. 2022.

RHODES, B.; GOERZEN, J. **Foundations of Python Network Programming**. Apress; 3rd ed. Edition. October 20, 2014.

TANENBAUM, A. S.; WETHERALL, D. **Redes de Computadores**. 5. ed. São Paulo: Pearson, 2011.