|  |
| --- |
| <!--from netmiko import ConnectHandler |
|  |  |
|  |  |
|  | ip=input('Enter IP address: ') |
|  | username=input('Enter username: ') |
|  | password=input('Enter password: ') |
|  | port=input('Enter port number: ') |
|  |  |
|  | MT = { |
|  | 'device\_type': 'mikrotik\_routeros', |
|  | 'host': ip, |
|  | 'port': port, |
|  | 'username': username, |
|  | 'password': password |
|  | } |
|  |  |
|  | ssh = ConnectHandler(\*\*MT) |
|  | print('Connected to {}.'.format(ip)) |
|  |  |
|  |  |
|  | print(""" |
|  | Commands: |
|  | 1. Change Router's Identity |
|  | 2. Control (Lan) interface up/downs |
|  | 3. Control and change the ports of services (Telnet, ssh , web) |
|  | 4. Disconnect |
|  | """) |
|  |  |
|  | while(True): |
|  | choice=input('Enter command number:') |
|  |  |
|  | if choice=="1": |
|  | name=input("Enter new name: ") |
|  | output = ssh.send\_command(f'system identity set name={name}') |
|  | output = ssh.send\_command('system identity print') |
|  |  |
|  | elif choice=="2": |
|  | name=input("Enter LAN interface name: ") |
|  | choicee=input("up(u) or down(d)?: ") |
|  | if ed=='u': |
|  | output = ssh.send\_command(f'ip link set dev {name} up') |
|  | if ed=='d': |
|  | output = ssh.send\_command(f'ip link set dev {name} down') |
|  | else: print('Invalid input!') |
|  |  |
|  | elif choice=="3": |
|  | choicee=input("control port(1) or change port(2)?: ") |
|  | serv=input(""" choose one: |
|  | 1.telnet |
|  | 2.ssh |
|  | 3.www |
|  | """) |
|  | if choicee=="1": |
|  | ed=input("enable(e) or disable(d)?: ") |
|  | if ed=='e': |
|  | output = ssh.send\_command(f'ip service enable {serv}') |
|  | if ed=='d': |
|  | output = ssh.send\_command(f'ip service disable {serv}') |
|  | else: print('Invalid input!') |
|  | if choicee=="2": |
|  | serv\_port=input(f"Specify port number for service {serv}: ") |
|  | output = ssh.send\_command(f'ip service set {serv} port={serv\_port}') |
|  | else: print('Invalid input!') |
|  |  |
|  | elif choice=="4": |
|  | break |
|  |  |
|  | else: print('Invalid input!') |
|  | print(output)--> |
|  | <!--import threading |
|  | import socket |
|  | alias = input('Choose an alias >>> ') |
|  | client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) |
|  | client.connect(('127.0.0.1', 59000)) |
|  |  |
|  |  |
|  | def client\_receive(): |
|  | while True: |
|  | try: |
|  | message = client.recv(1024).decode('utf-8') |
|  | if message == "alias?": |
|  | client.send(alias.encode('utf-8')) |
|  | else: |
|  | print(message) |
|  | except: |
|  | print('Error!') |
|  | client.close() |
|  | break |
|  |  |
|  |  |
|  | def client\_send(): |
|  | while True: |
|  | message = f'{alias}: {input("")}' |
|  | client.send(message.encode('utf-8')) |
|  |  |
|  |  |
|  | receive\_thread = threading.Thread(target=client\_receive) |
|  | receive\_thread.start() |
|  |  |
|  | send\_thread = threading.Thread(target=client\_send) |
|  | send\_thread.start() |
|  | Server |
|  | 'Chat Room Connection - Client-To-Client' |
|  | import threading |
|  | import socket |
|  | host = '10.33.114.242' |
|  | port = 5555 |
|  | server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) |
|  | server.bind((host, port)) |
|  | server.listen(1) |
|  | clients = [] |
|  | aliases = [] |
|  |  |
|  |  |
|  | def broadcast(sender, message): |
|  | for client in clients: |
|  | if(client != sender): |
|  | client.send(message) |
|  |  |
|  | # Function to handle clients'connections |
|  |  |
|  |  |
|  | def handle\_client(client): |
|  | while True: |
|  | try: |
|  | message = client.recv(1024) |
|  | print(message) |
|  | broadcast(client, message) |
|  | except: |
|  | index = clients.index(client) |
|  | clients.remove(client) |
|  | client.close() |
|  | alias = aliases[index] |
|  | broadcast(server, f'{alias} has left the chat room!'.encode('utf-8')) |
|  | aliases.remove(alias) |
|  | break |
|  | # Main function to receive the clients connection |
|  |  |
|  |  |
|  | def receive(): |
|  | while True: |
|  | print('Server is running and listening ...') |
|  | client, address = server.accept() |
|  | print(f'connection is established with {str(address)}') |
|  | client.send('alias?'.encode('utf-8')) |
|  | alias = client.recv(1024) |
|  | aliases.append(alias) |
|  | clients.append(client) |
|  | print(f'The alias of this client is {alias}'.encode('utf-8')) |
|  | broadcast(server, f'{alias} has connected to the chat room'.encode('utf-8')) |
|  | client.send('you are now connected!'.encode('utf-8')) |
|  | thread = threading.Thread(target=handle\_client, args=(client,)) |
|  | thread.start() |
|  |  |
|  |  |
|  | if \_name\_ == "\_main\_": |
|  | receive()--> |
|  | <!--How does TCP provide a reliable connection? |
|  | • Every TCP packet is given a sequence number so that the system on the receiving end can put them back together in the right order and can also notice missing packets in the sequence and ask that they be retransmitted. |
|  |  |
|  | • Instead of using sequential integers (1, 2, 3...) to sequence packets, TCP uses a counter that counts the number of bytes transmitted. A 1,024-byte packet with a sequence number of 7,200, for example, would be followed by a packet with a sequence number of 8,224. This means that a busy network stack does not have to remember how it broke up a data stream into packets. If asked for a retransmission, it can break up the stream into new packets some other way (which might let it fit more data into a packet if more bytes are now waiting for transmission), and the receiver can still put the packets back together. |
|  |  |
|  | • The initial sequence number, in good TCP implementations, is chosen randomly so that villains cannot assume that every connection starts at byte zero. Predictable sequence numbers unfortunately make it easier to craft forged packets that might interrupt a conversation by looking like they are a legitimate part of its data. |
|  |  |
|  | • Rather than running very slowly in lock step by needing every packet to be acknowledged before it sends the next one, TCP sends whole bursts of packets at a time before expecting a response. The amount of data that a sender is willing to have on the wire at any given moment is called the size of the TCP window. |
|  |  |
|  | • The TCP implementation on the receiving end can regulate the window size of the transmitting end and thus slow or pause the connection. This is called flow control. This lets a receiver forbid the transmission of additional packets in cases where its input buffer is full, and it would have to discard more data anyway even if it were to arrive. |
|  |  |
|  | • Finally, if TCP believes that packets are being dropped, it assumes that the network is becoming congested and reduces how much data it sends every second. This can be something of a disaster on wireless networks and other media where packets are lost simply because of noise. It can also ruin connections that are running fine until a router reboots and the endpoints cannot talk for, say, 20 seconds. By the time the network comes back up, the two TCP peers will have decided that the network is extraordinarily overloaded with traffic, and upon reestablishing. |
|  |  |
|  | The design of TCP involves many other nuances and details beyond the behaviors just described, but ideally this description gives you a good feel for how it will work—even though, you will remember, all that your application will see is a stream of data, with the actual packets and sequence numbers cleverly hidden away by your operating system network stack. |
|  |  |
|  | 1. 3-way handshake |
|  | 2.send packets of data: The first computer sends a packet with data and a sequence number. The second computer acknowledges it by setting the ACK bit and increasing the acknowledgement number by the length of the received data. |
|  | 3.3. Close connection: FIN in the segment is set to 1 and other computer replies with ack and another FIN |
|  | 4. Detecting lost packets: using a timeout option sender starts a timer and puts the packet in retransmission queue |
|  | 5. Handling out of order packets- when receiver sees higher sequence number it’s obvious that some packets are missing and receiver must deal with out of order packets, by reassembling them with sequence number. |
|  | 5)Compare the VPNs (protocols) and the functionality in network programming viewpoint |
|  | ¬ PPTP -- Point-to-Point Tunneling Protocol |
|  | ¬ L2TP -- Layer 2 Tunneling Protocol |
|  | ¬ IPsec -- Internet Protocol Security |
|  | ¬ OpenVPN Protocol |
|  | ¬ SSL and TLS |
|  | ¬ SSTP |
|  | Point-to-Point Tunneling Protocol (PPTP) |
|  | • Layer 2 remote access VPN distributed with Windows product family |
|  | ▫ Addition to Point-to-Point Protocol (PPP) |
|  | ▫ Allows multiple Layer 3 Protocols |
|  | • Uses proprietary authentication and encryption |
|  | • Limited user management and scalability |
|  | ▫ Used MPPE encryption method |
|  | Layer 2 Tunneling Protocol (L2TP) |
|  | • Layer 2 remote access VPN protocol |
|  | ▫ Combines and extends PPTP and L2F (Cisco supported protocol) |
|  | ▫ Weak authentication and encryption |
|  | ▫ Addition to Point-to-Point Protocol (PPP) |
|  | ▫ Must be combined with IPSec for enterprise-level security |
|  | Internet Protocol Security (IPSec) |
|  | • Layer 3 protocol for remote access, intranet, and extranet VPNs |
|  | ▫ Internet standard for VPNs |
|  | ▫ Provides flexible encryption and message authentication/integrity |
|  | SSL and TLS |
|  | SSL connections have HTTPS in the beginning of the URL instead of HTTP |
|  | SSL and TLS protocol is most commonly usedby online shopping websites and service providers |
|  | OpenVPN Protocol |
|  | OpenVPN is currently considered the top form of VPN tunneling protocol. That's because its encryption is especially strong. It's also adept at getting past firewalls. |
|  | SSTP |
|  | Secure Socket Tunneling Protocol is popular due to its full integration with every Microsoft operating system since Windows Vista SP 1. SSTP utilizes 2048-bit SSL/TLS certificates for authentication and 256-bit SSL keys for encryption. The biggest drawback to SSTP is that is basically a Microsoft-developed proprietary protocol and developers do not have access to the underlying code. |
|  | 3)))What TCP sockets mean? How to open socket with active/passive host? And |
|  | generally, what is the differences between active and passive sockets |
|  | TCP uses port numbers to distinguish different applications running at the same IP address, and it |
|  | follows exactly the same conventions regarding well-known and ephemeral port numbers. |
|  | Reread the section "Port Numbers" in that chapter if you want to review the details. As you saw in the |
|  | previous chapter, it takes only a single socket to speak UDP: a server can open a UDP port and |
|  | then receive datagrams from thousands of different clients. While it is certainly possible to connect() a |
|  | datagram socket to a particular peer so that the socket will always send() to only that peer and |
|  | recy()packets sent back from that peer, the idea of a connection is just a convenience. The effect of |
|  | connect() is exactly the same as your application simply deciding, on its own, to send to only one |
|  | address with sendto() calls and then ignore responses from any but that same address. |
|  | IP Address + Port Number = Socket |
|  | On the "server side"-which, by definition, is the conversation partner not doing the connect() call but |
|  | receiving the SYN packet that the connect call initiates-an incoming connection generates an even |
|  | more momentous event for a Python application: the creation of a new socket! This is because the |
|  | standard POSIX interface to TCP actually involves two completely different kinds of sockets: "passive" |
|  | listening sockets and active "connected" ones. |
|  | • The passive socket or listening socket maintains the "socket name"-the address and port number-at |
|  | which the server is ready to receive connections. No data can ever be received or sent by this kind of |
|  | socket. It does not represent any actual network conversation. Instead, it is how the server alerts the |
|  | operating system to its willingness to receive incoming connections at a given TCP port number in the |
|  | first place. |
|  | • An active, connected socket is bound to one particular remote conversation partner with a particular |
|  | IP address and port number. It can be used only for talking back and forth with that one partner, and it |
|  | can be read and written to without worrying about how the resulting data will be split up into packets. |
|  | The stream looks so much like a pipe or file that, on Unix systems, a connected TCP socket can be passed |
|  | to another program that expects to read from a normal file, and that program will never even know that |
|  | it is talking over the network. (Ctrl) - --> |