CN PRAC SUGGESSTIONS

#1) WAP in Python to find out the ip address of a local machine or any other machine

```
import socket

def get_local_ip():
    try:
        hostname = socket.gethostname()
        ip_address = socket.gethostbyname(hostname)
        return ip_address
    except socket.error as e:
        return "Error: {}".format(e)

def get_ip_address(domain):
    try:
        ip_address = socket.gethostbyname(domain)
        return ip_address
    except socket.error as e:
        return "Error: {}".format(e)

local_ip = get_local_ip()
print("Local IP Address:", local_ip)

domain = input("Enter the domain name (e.g., www.google.com): ")

domain_ip = get_ip_address(domain)
print("IP Address of", domain + ":", domain_ip)
```

#----------#

#2) Sliding Window Protocol

"""Write a program to simulate the Sliding Window Protocol in a network transmission. Implement the following features:

1. Prompt the user to enter the window size.

2. Prompt the user to enter the number of frames to transmit.

3. Allow the user to enter the frames.

4. Simulate the sending of frames in windows, where the sender sends a window of frames and waits for an acknowledgment before sending the next window of frames.

5. Print the frames being sent and when the acknowledgment for those frames is received.

"""

```
def main():
    w = int(input("Enter window size: "))

    f = int(input("Enter number of frames to transmit: "))

    frames = []
    print(f"Enter {f} frames: ")
    for i in range(f):
        frames.append(int(input()))

    print("\nWith sliding window protocol, the frames will be sent in the following way (assuming no corruption of frames):")

    i = 0
    while i < f:
        print(f"\nSending frames: {frames[i:i+w]}")
        if (i + w) <= f:
            print("Acknowledgement of above frames sent is received by sender\n")
        else:
            print("Acknowledgement of above frames sent is received by sender\n")
```

```
        i += w

if __name__ == "__main__":
    main()

#-------------------------------------------------------------------------------------------------------------------------#

#3)Write a server program to implement selective repeat ARQ.

import socket
import threading
import time
import random

# Constants
WINDOW_SIZE = 4
PACKET_SIZE = 1024
TIMEOUT = 2  # seconds
MAX_SEQ = 8

class SelectiveRepeatServer:
    def __init__(self, host='localhost', port=12345):
        self.host = host
        self.port = port
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.server_socket.bind((self.host, self.port))
        self.buffer = [None] * MAX_SEQ
        self.ack_received = [False] * MAX_SEQ
        self.expected_seq = 0
```

```
def simulate_packet_loss(self):
    return random.random() < 0.1

def receive_packet(self):
    while True:
        try:
            packet, client_addr = self.server_socket.recvfrom(PACKET_SIZE)
            seq_num, data = self.unpack_packet(packet)

            if self.simulate_packet_loss():
                print(f"Simulated loss of packet {seq_num}")
                continue

            if seq_num in range(self.expected_seq, self.expected_seq + WINDOW_SIZE):
                print(f"Received packet {seq_num}")
                self.buffer[seq_num % MAX_SEQ] = data
                self.ack_received[seq_num % MAX_SEQ] = True
                ack_packet = self.pack_packet(seq_num, b'ACK')
                self.server_socket.sendto(ack_packet, client_addr)

                while self.ack_received[self.expected_seq % MAX_SEQ]:
                    self.ack_received[self.expected_seq % MAX_SEQ] = False
                    self.expected_seq += 1

        except socket.timeout:
            print("Timeout, waiting for packets...")

def pack_packet(self, seq_num, data):
    return seq_num.to_bytes(1, 'big') + data

def unpack_packet(self, packet):
```

```
        seq_num = int.from_bytes(packet[0:1], 'big')
        data = packet[1:]
        return seq_num, data

    def start(self):
        print(f"Server started at {self.host}:{self.port}")
        self.server_socket.settimeout(TIMEOUT)
        threading.Thread(target=self.receive_packet).start()

if __name__ == "__main__":
    server = SelectiveRepeatServer()
    server.start()
#-----------------------------------------------------------------------------------------------------------#
```

#4) Write a server program in python to create server side socket and use it to receive a client side request, process it to generate response in form of current date and time where server program should be running and sent response back to the client

```
#Server Side

import socket
import datetime

server_ip = '127.0.0.1'
server_port = 12345

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_socket.bind((server_ip, server_port))

server_socket.listen()
```

```
print("Server is listening on", server_ip, "port", server_port)

while True:
    client_socket, client_address = server_socket.accept()

    print("Client connected from", client_address)
    request = client_socket.recv(1024).decode()

    if request == "GET /date-time":
        current_datetime = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

        response = current_datetime.encode()
        client_socket.sendall(response)

    client_socket.close()

# Client Side

import socket
server_ip = '127.0.0.1'
server_port = 12345

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

client_socket.connect((server_ip, server_port))

request = "GET /date-time"
client_socket.sendall(request.encode())

response = client_socket.recv(1024)
```

```
print("Current Date and Time:", response.decode())


client_socket.close()

#-------------------------------------------------------------------------------------------------------------------#


#5) Given an IP Address and the required subnet. WAP to Find out the subnet mask and subnetwork address of each subnet

import ipaddress

def ipv4_subnet(ip_address, subnet):
    try:
        network = ipaddress.ip_interface(f"{ip_address}/{subnet}")
    except (ValueError,ipaddress.AddressValueError) as e:
        raise ValueError(f"Invalid Ip address or subnet: {e}")
    subnet_mask = str(network.netmask)
    subnetwork_address = str(network.network)
    return subnet_mask,subnetwork_address
ip_address = str(input("Enter the IPv4 address :- "))#(e.g., 192.168.1.0)
subnet = int(input("Enter the required subnet :- "))#(e.g., 24)
try:
    subnet_mask, subnetwork_address = ipv4_subnet(ip_address, subnet)
    print(f"Subnet Mask : {subnet_mask}")
    print(f"Subnetwork Address : {subnetwork_address}")
except ValueError as e:
    print(f"Error : {e}")



#-------------------------------------------------------------------------------------------------------------------#


# 6) Write client-server interaction programs to implement multicasting
```

#Server Side

```
import socket
import struct
import time

multicast_group = '224.3.29.71'
server_port = 10000

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

ttl = struct.pack('b', 1)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, ttl)

message = "Hello, multicast world!"

try:
    print("Sending message:", message)
    sent = sock.sendto(message.encode(), (multicast_group, server_port))

finally:
    sock.close()
```

Client Side

```
import socket
import struct

multicast_group = '224.3.29.71'
server_port = 10000
```

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

sock.bind(('', server_port))

group = socket.inet_aton(multicast_group)
mreq = struct.pack('4sL', group, socket.INADDR_ANY)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)

while True:
    print("\nWaiting to receive message...")
    data, address = sock.recvfrom(1024)

    print("Received", len(data), "bytes from", address)
    print("Message:", data.decode())



#-------------------------------------------------------------------------------------------------------------------------#

#7) Given an IP Address, WAP to find out its class network id and host id
def ipv4_info(ip_address):
    octets = ip_address.split('.')
    octets = [int(octet) for octet in octets]
    if octets[0] >= 1 and octets[0] <= 126:
        ip_class = 'A'
        network_id = octets[:1]
        host_id = octets[1:]
    elif octets[0] >= 128 and octets[0] <= 191:
        ip_class = 'B'
        network_id = octets[:2]
        host_id = octets[2:]
    elif octets[0] >= 192 and octets[0] <= 223:
```

```
        ip_class = 'C'
        network_id = octets[:3]
        host_id = octets[3:]
    elif octets[0] >= 224 and octets[0] <= 239:
        ip_class = 'D'
        network_id = None
        host_id = None
    else:
        ip_class = 'Unknown'
        network_id = None
        host_id = None

    return ip_class, network_id, host_id
ip_address = str(input("Enter the IPv4 address :- "))#(e.g., 192.168.1.10)
ip_class, network_id, host_id = ipv4_info(ip_address)
print("IPv4 Class:", ip_class)
print("Network ID:", '.'.join(str(octet) for octet in network_id))
print("Host ID:", '.'.join(str(octet) for octet in host_id))

#-----------------------------------------------------------------------------------------------------#

#8) Write a program to create client and server side sockets and use them to send a string in lowercase from client to the server, the server then converts the string to uppercase and return back to client.

#Server Side

import socket

server_ip = '127.0.0.1'
server_port = 12345
```

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_socket.bind((server_ip, server_port))

server_socket.listen()

print("Server is listening on", server_ip, "port", server_port)

while True:
    client_socket, client_address = server_socket.accept()

    print("Client connected from", client_address)

    lowercase_string = client_socket.recv(1024).decode()

    uppercase_string = lowercase_string.upper()

    client_socket.sendall(uppercase_string.encode())

    client_socket.close()

# Client Side

import socket

server_ip = '127.0.0.1'
server_port = 12345

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

client_socket.connect((server_ip, server_port))
```

```
lowercase_string = str(input("Enter a String :- "))
lowercase_string = lowercase_string.lower()
client_socket.sendall(lowercase_string.encode())

uppercase_string = client_socket.recv(1024).decode()

print("Uppercase String from Server:", uppercase_string)

client_socket.close()

#-----------------------------------------------------------------------------------------------------------------------#

#9) Write a program to implement Stop and Wait ARQ Protocol.(Using Java)
"""
#Server Side

import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.util.Random;

public class StopAndWaitServer {
    private static final int PACKET_SIZE = 1024;
    private static final int TIMEOUT = 2000; // milliseconds

    private DatagramSocket socket;
    private int expectedSeqNum = 0;

    public StopAndWaitServer(int port) throws Exception {
        socket = new DatagramSocket(port);
```

```
        socket.setSoTimeout(TIMEOUT);
    }

    private boolean simulatePacketLoss() {
        return new Random().nextDouble() < 0.1;
    }

    private void receivePacket() throws Exception {
        byte[] buffer = new byte[PACKET_SIZE];
        while (true) {
            DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
            try {
                socket.receive(packet);
                int seqNum = buffer[0];
                byte[] data = new byte[packet.getLength() - 1];
                System.arraycopy(buffer, 1, data, 0, data.length);

                if (simulatePacketLoss()) {
                    System.out.println("Simulated loss of packet " + seqNum);
                    continue;
                }

                if (seqNum == expectedSeqNum) {
                    System.out.println("Received packet " + seqNum + ": " + new String(data));
                    expectedSeqNum++;
                    byte[] ackData = new byte[] { (byte) seqNum, 'A', 'C', 'K' };
                    DatagramPacket ackPacket = new DatagramPacket(ackData, ackData.length,
packet.getAddress(), packet.getPort());
                    socket.send(ackPacket);
                } else {
                    System.out.println("Received out-of-order packet " + seqNum + ", expected " +
expectedSeqNum);
```

```
            }
        } catch (Exception e) {
            System.out.println("Timeout, waiting for packets...");
        }
    }
  }

  public static void main(String[] args) throws Exception {
    int port = Integer.parseInt(args.length > 0 ? args[0] : "12345");
    StopAndWaitServer server = new StopAndWaitServer(port);
    System.out.println("Server started at port " + port);
    server.receivePacket();
  }
}

#Client Side

import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.util.Scanner;

public class StopAndWaitClient {
  private static final int PACKET_SIZE = 1024;
  private static final int TIMEOUT = 2000; // milliseconds

  private DatagramSocket socket;
  private InetAddress serverAddress;
  private int serverPort;
  private int seqNum = 0;
```

```
    public StopAndWaitClient(String host, int port) throws Exception {
        socket = new DatagramSocket();
        socket.setSoTimeout(TIMEOUT);
        serverAddress = InetAddress.getByName(host);
        serverPort = port;
    }

    private void sendPacket(String message) throws Exception {
        byte[] data = message.getBytes();
        byte[] packetData = new byte[data.length + 1];
        packetData[0] = (byte) seqNum;
        System.arraycopy(data, 0, packetData, 1, data.length);

        DatagramPacket packet = new DatagramPacket(packetData, packetData.length, serverAddress,
serverPort);
        socket.send(packet);

        try {
            byte[] buffer = new byte[PACKET_SIZE];
            DatagramPacket ackPacket = new DatagramPacket(buffer, buffer.length);
            socket.receive(ackPacket);

            int ackSeqNum = buffer[0];
            if (ackSeqNum == seqNum && new String(buffer, 1, 3).equals("ACK")) {
                System.out.println("Received ACK for packet " + seqNum);
                seqNum++;
            } else {
                System.out.println("Received incorrect ACK: " + ackSeqNum);
                sendPacket(message);
            }
        } catch (Exception e) {
```

```
            System.out.println("Timeout, resending packet " + seqNum);
            sendPacket(message);
        }
    }

    public static void main(String[] args) throws Exception {
        String host = args.length > 0 ? args[0] : "localhost";
        int port = Integer.parseInt(args.length > 1 ? args[1] : "12345");
        StopAndWaitClient client = new StopAndWaitClient(host, port);

        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter messages to send to the server (type 'exit' to quit):");
        while (true) {
            String message = scanner.nextLine();
            if (message.equalsIgnoreCase("exit")) {
                break;
            }
            client.sendPacket(message);
        }
        scanner.close();
    }
}
"""

#-----------------------------------------------------------------------------------------------------#

#10)Write a program to implement the following. Input an IP address in either dotted decimal or binary format & print the class of the IP Address as well as the corresponding network address.

binary_ip = input("Enter the binary IP address: ")
octets = binary_ip.split(".")
```

```
decimal_ip = ""
for octet in octets:
    decimal_ip += str(int(octet, 2)) + "."
print("The decimal IP address is:", decimal_ip)
ip_address=decimal_ip
def ip_info(ip_address):
    octets = ip_address.split('.')
    octets = [int(octet) for octet in octets if octet]
    if octets[0] >= 1 and octets[0] <= 126:
        ip_class = 'A'
        network_id = octets[:1]
    elif octets[0] >= 128 and octets[0] <= 191:
        ip_class = 'B'
        network_id = octets[:2]
    elif octets[0] >= 192 and octets[0] <= 223:
        ip_class = 'C'
        network_id = octets[:3]
    elif octets[0] >= 224 and octets[0] <= 239:
        ip_class = 'D'
        network_id = None
    else:
        ip_class = 'Unknown'
        network_id = None
    return ip_class, network_id
ip_class, network_id = ip_info(ip_address)
print("IPv4 Class:", ip_class)
print("Network ID:", '.'.join(str(octet) for octet in network_id))

#-----------------------------------------------------------------------------------------------------------------------#

#11) Write a program to create client & Server Side Sockets & use them to implement Echo Server
```

```
#Server Side

import socket

server_ip = '127.0.0.1'
server_port = 12345

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_socket.bind((server_ip, server_port))

server_socket.listen()

print("Echo Server is listening on", server_ip, "port", server_port)

while True:
    client_socket, client_address = server_socket.accept()

    print("Client connected from", client_address)

    message = client_socket.recv(1024).decode()

    client_socket.sendall(message.encode())

    client_socket.close()

# Client Side

import socket
```

```
server_ip = '127.0.0.1'
server_port = 12345

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

client_socket.connect((server_ip, server_port))

message = "Hello, Echo Server!"
client_socket.sendall(message.encode())

response = client_socket.recv(1024).decode()

print("Server Response:", response)

client_socket.close()

#-----------------------------------------------------------------------------------------------------------------------#

#12) Write a program to implement error detection at Datalink layer using CRC

def xor(a, b):
    result = []
    for i in range(1, len(b)):
        if a[i] == b[i]:
            result.append('0')
        else:
            result.append('1')
    return ''.join(result)

def mod2div(dividend, divisor):
    pick = len(divisor)
```

```
    tmp = dividend[0 : pick]

    while pick < len(dividend):
        if tmp[0] == '1':
            tmp = xor(divisor, tmp) + dividend[pick]
        else:
            tmp = xor('0'*pick, tmp) + dividend[pick]
        pick += 1

    if tmp[0] == '1':
        tmp = xor(divisor, tmp)
    else:
        tmp = xor('0'*pick, tmp)

    return tmp

def encodeData(data, key):
    l_key = len(key)
    appended_data = data + '0'*(l_key-1)
    remainder = mod2div(appended_data, key)
    codeword = data + remainder
    return codeword

def decodeData(data, key):
    remainder = mod2div(data, key)
    return remainder

data = '11010011101100'
key = '1011'

print("Data:", data)
```

```
print("Key:", key)


encoded_data = encodeData(data, key)
print("Encoded Data:", encoded_data)


remainder = decodeData(encoded_data, key)
print("Remainder after decoding:", remainder)


if '1' in remainder:
    print("Error detected in the received data")
else:
    print("No error detected in the received data")


#-----------------------------------------------------------------------------------------------------------------------#
```

CN QUESTIONS

1. **Various Transmission Media:**

   - **Guided Media:**

     - **Twisted Pair Cables:** Consists of pairs of wires twisted together to reduce electromagnetic interference. Commonly used in local area networks (LANs), especially with Ethernet cables (e.g., Cat5e, Cat6).

     - **Coaxial Cables:** Composed of a central conductor, insulating layer, metallic shield, and outer insulating layer. Used for cable TV and older Ethernet networks.

     - **Fiber Optic Cables:** Uses light to transmit data. Consists of a core, cladding, and protective outer layer. Offers high bandwidth, long-distance transmission, and resistance to electromagnetic interference.

   - **Unguided Media:**

     - **Radio Waves:** Used for wireless communication over long distances, such as Wi-Fi, cellular networks, and satellite communication.

     - **Microwaves:** Used for point-to-point communication links, satellite communication, and radar.

     - **Infrared:** Used for short-range communication in devices like remote controls and some wireless peripherals.

2. **Utility of a Network Interface Card (NIC):**

A NIC is a critical hardware component that enables a computer or other device to connect to a network. It converts data from the computer into a format suitable for transmission over the network and manages the physical layer of the network stack. NICs can be for wired connections (Ethernet) or wireless connections (Wi-Fi). They handle error checking and frame formatting, and each NIC has a unique MAC address that identifies the device on the network.

3. **Difference Between Logical and Physical Address:**

- **Logical Address:** An IP address assigned to each device on a network to facilitate routing and communication. It can change if the device connects to different networks (dynamic IP) or remain fixed (static IP). Logical addresses are used by the network layer (Layer 3) of the OSI model.

- **Physical Address:** A MAC address, a unique identifier assigned to the network interface card (NIC) of a device. It is used for data link layer (Layer 2) communications within the same network segment and is fixed to the hardware.

4. **Comparison of Network Topologies:**

- **Star Topology:** All devices are connected to a central hub or switch. If one device fails, it doesn't affect others, but if the hub fails, the entire network goes down. It is easy to add or remove devices.

- **Bus Topology:** All devices share a single communication line or backbone. If the backbone fails, the network goes down. It is cost-effective for small networks but not suitable for large or heavily loaded networks due to potential data collisions.

- **Ring Topology:** Devices are connected in a circular manner. Each device has exactly two neighbors. Data travels in one direction, reducing collisions, but if one device or the connection fails, the entire network is affected unless there is a redundant path.

- **Mesh Topology:** Each device is connected to every other device. This provides high redundancy and reliability. It is expensive and complex to install and maintain but offers excellent fault tolerance and load balancing.

5. **Telnet:**

Telnet is an application layer protocol used to provide a bidirectional interactive text-based communication facility over a network. It allows remote login and command execution on another machine as if the user were physically present at the machine. Telnet operates over TCP and is known for being insecure because it transmits data, including passwords, in plain text.

6. **Firewalls:**

Firewalls are network security devices or software that monitor and control incoming and outgoing network traffic based on predetermined security rules. They act as barriers between trusted internal networks and untrusted external networks (like the internet). Firewalls can be hardware-based, software-based, or a combination of both. They can perform various functions, including packet filtering, stateful inspection, proxying, and logging.

7. **Steps to Configure File Transfer Protocol (FTP):**

- **Install FTP Server Software:** Choose and install FTP server software, such as FileZilla Server, vsftpd, or ProFTPD, on the host machine.

- **Configure Server Settings:** Adjust server settings, including setting up the listening port (default is port 21), passive mode settings, and encryption options (e.g., FTPS for secure transmission).

- **Set Up User Accounts and Permissions:** Create user accounts and configure permissions for access to specific directories. Ensure users have the necessary read/write access based on their roles.

- **Configure Directories:** Define home directories for users and set appropriate permissions for accessing these directories.

- **Start FTP Server:** Launch the FTP server service and ensure it is running correctly.

- **Connect Using FTP Client:** Use an FTP client, such as FileZilla, WinSCP, or command-line FTP, to connect to the server using the configured user credentials.

8. **DNS (Domain Name System):**

DNS is a hierarchical and decentralized naming system for devices connected to the internet or a private network. It translates human-readable domain names (e.g., www.example.com) into numerical IP addresses (e.g., 192.0.2.1). DNS uses a distributed database maintained by a network of name servers. Key components include:

- **DNS Resolver:** A client-side service that queries DNS servers to resolve domain names.

- **DNS Server:** A server that stores DNS records and responds to queries from DNS resolvers.

- **Root Servers:** The top level of the DNS hierarchy, directing queries to the appropriate top-level domain (TLD) servers.

- **TLD Servers:** Servers that handle the top-level domains (e.g., .com, .org) and direct queries to authoritative DNS servers for specific domains.

9. **Comparison of Hub, Switch, and Router:**

- **Hub:** A simple network device that broadcasts incoming data packets to all devices connected to its ports. Hubs operate at the physical layer (Layer 1) and do not filter or manage traffic, leading to potential collisions in the network.

- **Switch:** A more advanced device that operates at the data link layer (Layer 2). It receives data packets and forwards them only to the specific device (port) that the data is intended for, based on MAC addresses. Switches reduce collisions and improve network efficiency.

- **Router:** A network device that operates at the network layer (Layer 3). It routes data packets between different networks, typically using IP addresses. Routers can connect and manage traffic between multiple networks, providing internet connectivity and supporting various protocols.

10. **Types of Cable Used in Computer Networks:**

- **Twisted Pair Cables:**

  - **Unshielded Twisted Pair (UTP):** Commonly used in Ethernet networks (e.g., Cat5e, Cat6) due to its low cost and ease of installation.

  - **Shielded Twisted Pair (STP):** Provides better protection against electromagnetic interference, used in environments with high interference.

- **Coaxial Cables:** Consists of a central conductor, insulating layer, metallic shield, and outer insulating layer. Used for cable television, broadband internet, and older Ethernet networks.

- **Fiber Optic Cables:** Uses light to transmit data, providing high bandwidth, long-distance transmission, and resistance to electromagnetic interference. Used in backbone networks, data centers, and for high-speed internet connections.

11. **Client-Server Communication Paradigm:**

In the client-server model, clients (end-user devices) request services or resources from a centralized server. The server processes these requests and provides the necessary responses. This model is widely used in networking, where the server hosts resources such as web pages, files, databases, or applications, and clients access these resources over a network. Key features include:

- **Centralization:** Servers provide centralized management and control of resources.

- **Scalability:** Servers can be scaled to handle multiple client requests simultaneously.

- **Security:** Centralized servers can implement robust security measures to protect resources.

12. **Unicast, Broadcast, & Multicast:**

- **Unicast:** A communication method where data is sent from one sender to one specific receiver. It is the most common form of communication in networks, used for one-to-one interactions such as web browsing and file transfers.

- **Broadcast:** A communication method where data is sent from one sender to all devices in a network segment. Broadcasts are used for network discovery protocols and certain types of messaging. However, excessive broadcasting can lead to network congestion.

- **Multicast:** A communication method where data is sent from one sender to a specific group of devices that have expressed interest in receiving the data. Multicast is efficient for streaming

media, video conferencing, and other applications where the same data needs to be delivered to multiple recipients simultaneously without duplicating the data for each receiver.