

2) Write a Program for error detection technique using CRC Algorithm.

```
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 crc_divide(dividend, divisor):
    pick = len(divisor)
    temp = dividend[0:pick]

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

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

    return temp

def crc_encode(data, divisor):
    appended_data = data + '0' * (len(divisor) - 1)
    remainder = crc_divide(appended_data, divisor)
    codeword = data + remainder
    return codeword

def crc_decode(received_data, divisor):
    remainder = crc_divide(received_data, divisor)
    return remainder

def main():
    data = input("Enter the data to be sent: ")
    divisor = input("Enter the CRC divisor: ")

    encoded_data = crc_encode(data, divisor)
    print("Encoded data (Codeword):", encoded_data)

    received_data = input("Enter the received data: ")
    decoded_remainder = crc_decode(received_data, divisor)
```

```

if '1' in decoded_remainder:
    print("Error detected in received data.")
else:

    print("No errors detected in received data.")

if __name__ == "__main__":
    main()

```

3) Write a program to implement error correction technique using Hamming code.

```

def hamming_encode(data):

    # Calculate the number of parity bits needed

    r = 0

    while 2**r < len(data) + r + 1:
        r += 1

    # Create the encoded data array with space for parity bits

    encoded_data = [0] * (len(data) + r)

    # Fill in the data bits

    j = 0

    for i in range(1, len(encoded_data) + 1):
        if i & (i - 1) == 0:
            continue # Skip parity bits
        encoded_data[i - 1] = int(data[j])
        j += 1

    # Calculate parity bits

    for i in range(r):
        parity_index = 2**i - 1
        parity_value = 0

        for j in range(parity_index, len(encoded_data), 2 * parity_index + 2):
            parity_value ^= encoded_data[j]

        encoded_data[parity_index] = parity_value

```

```
return ".join(map(str, encoded_data))

def hamming_correct(encoded_data):
    # Calculate the number of parity bits needed
    r = 0
    while 2**r < len(encoded_data):
        r += 1

    # Initialize the error syndrome
    error_syndrome = 0

    # Calculate the error syndrome
    for i in range(r):
        parity_index = 2**i - 1
        parity_value = 0
        for j in range(parity_index, len(encoded_data), 2 * parity_index + 2):
            parity_value ^= int(encoded_data[j])
        error_syndrome |= (parity_value << i)

    # If the error syndrome is not zero, correct the error
    if error_syndrome != 0:
        print("Error detected at position:", error_syndrome)
        encoded_data = list(encoded_data)
        error_position = error_syndrome - 1
        encoded_data[error_position] = str(int(not int(encoded_data[error_position])))
        print("Corrected data:", ".join(encoded_data))
    else:
        print("No errors detected. Data:", encoded_data)

def main():
```

```

data = input("Enter the 4-bit data to be sent: ")

# Encode the data
encoded_data = hamming_encode(data)
print("Encoded data:", encoded_data)

# Introduce an error for demonstration purposes
error_position = 5
encoded_data = list(encoded_data)
encoded_data[error_position - 1] = str(int(not int(encoded_data[error_position - 1])))
print("Received data with error:", ''.join(encoded_data))

# Correct the error
hamming_correct(''.join(encoded_data))

if __name__ == "__main__":
    main()

```

4) Write a program to implement TCP Client Server Programming.

```

import socket

def client_program():
    host = socket.gethostname() # as both code is running on same pc
    port = 5000 # socket server port number

    client_socket = socket.socket() # instantiate
    client_socket.connect((host, port)) # connect to the server

    message = input(" -> ") # take input

    while message.lower().strip() != 'bye':
        client_socket.send(message.encode()) # send message
        data = client_socket.recv(1024).decode() # receive response

        print('Received from server: ' + data) # show in terminal

```

```

        message = input(" -> ") # again take input

        client_socket.close() # close the connection

if __name__ == '__main__':
    client_program()

import socket

def server_program():
    # get the hostname
    host = socket.gethostname()
    port = 5000 # initiate port no above 1024

    server_socket = socket.socket() # get instance
    # look closely. The bind() function takes tuple as argument
    server_socket.bind((host, port)) # bind host address and port together

    # configure how many client the server can listen simultaneously
    server_socket.listen(2)
    conn, address = server_socket.accept() # accept new connection
    print("Connection from: " + str(address))
    while True:
        # receive data stream. it won't accept data packet greater than 1024 bytes
        data = conn.recv(1024).decode()
        if not data:
            # if data is not received break
            break
        print("from connected user: " + str(data))
        data = input(' -> ')
        conn.send(data.encode()) # send data to the client

    conn.close() # close the connection

if __name__ == '__main__':
    server_program()

```

5) Write a program to implement UDP Client Server Programming.

```

import socket

msgFromClient      = "Hello UDP Server"

```

```
bytesToSend      = str.encode(msgFromClient)

serverAddressPort = ("127.0.0.1", 20001)

bufferSize       = 1024

# Create a UDP socket at client side

UDPClientSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

# Send to server using created UDP socket

UDPClientSocket.sendto(bytesToSend, serverAddressPort)

msgFromServer = UDPClientSocket.recvfrom(bufferSize)

msg = "Message from Server {}".format(msgFromServer[0])

print(msg)
```

```
import socket

localIP      = "127.0.0.1"

localPort    = 20001

bufferSize   = 1024

msgFromServer      = "Hello UDP Client"

bytesToSend      = str.encode(msgFromServer)

# Create a datagram socket
```

```

UDPServerSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

# Bind to address and ip
UDPServerSocket.bind((localIP, localPort))

print("UDP server up and listening")

# Listen for incoming datagrams
while(True):
    bytesAddressPair = UDPServerSocket.recvfrom(bufferSize)
    message = bytesAddressPair[0]
    address = bytesAddressPair[1]
    clientMsg = "Message from Client:{}".format(message)
    clientIP = "Client IP Address:{}".format(address)
    print(clientMsg)
    print(clientIP)

    # Sending a reply to client
    UDPServerSocket.sendto(bytesToSend, address)

```

6) Write a program to implement Stop and Wait Protocol at data link layer.

```

import socket

def start_client():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_address = ('localhost', 12345)

    sequence_number = 0

    while True:
        message = "Frame {}".format(sequence_number)

```

```

        client_socket.sendto(message.encode('utf-8'), server_address)
        print("Sent data to server: {}".format(message))

        # Wait for acknowledgment
        acknowledgment, _ = client_socket.recvfrom(1024)
        print("Received ACK from server: {}".format(acknowledgment.decode('utf-
8')))

        sequence_number = 1 - sequence_number # Alternating between 0 and 1

if __name__ == "__main__":
    start_client()

import socket
import time

def start_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_address = ('localhost', 12345)
    server_socket.bind(server_address)

    print("Server listening on {}:{}".format(*server_address))

    while True:
        data, client_address = server_socket.recvfrom(1024)
        print("Received data from {}: {}".format(client_address, data.decode('utf-
8')))

        # Simulate processing delay
        time.sleep(1)

        # Send acknowledgment back to the client
        acknowledgment = "ACK"
        server_socket.sendto(acknowledgment.encode('utf-8'), client_address)
        print("Sent ACK to {}: {}".format(client_address, acknowledgment))

if __name__ == "__main__":
    start_server()

```

7) Write a program to Sliding Window protocol, at data link layer.

```
import socket
```

```

import time

def start_client():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_address = ('localhost', 12345)

    window_size = 3
    base = 0
    next_sequence_number = 0

    while True:
        if next_sequence_number < base + window_size:
            message = "{}:Frame{}".format(next_sequence_number,
next_sequence_number)
            client_socket.sendto(message.encode('utf-8'), server_address)
            print("Sent data to server: {}".format(message))
            next_sequence_number += 1

        try:
            client_socket.settimeout(2) # Timeout for receiving ACK
            acknowledgment, _ = client_socket.recvfrom(1024)
            received_ack_number = int(acknowledgment.decode('utf-8').split(":")[0])
            print("Received ACK from server: {}".format(received_ack_number))
            if received_ack_number >= base:
                base = received_ack_number + 1
        except socket.timeout:
            print("Timeout occurred. Resending unacknowledged frames.")
            next_sequence_number = base

if __name__ == "__main__":
    start_client()

```

```

import socket
import time
import random

def start_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_address = ('localhost', 12345)
    server_socket.bind(server_address)

    print("Server listening on {}:{}".format(*server_address))

    while True:
        data, client_address = server_socket.recvfrom(1024)
        sequence_number = int(data.decode('utf-8').split(":")[0])
        message = data.decode('utf-8').split(":")[1]

```

```

# Simulate packet loss
if random.random() < 0.8: # 80% probability of receiving the packet
    print("Received data from {}: {}".format(client_address, message))

    # Simulate processing delay
    time.sleep(1)

    acknowledgment = "{}:ACK".format(sequence_number)
    server_socket.sendto(acknowledgment.encode('utf-8'), client_address)
    print("Sent ACK to {}: {}".format(client_address, acknowledgment))
else:
    print("Packet loss simulated. Discarding packet from client: {}".
          format(client_address))

if __name__ == "__main__":
    start_server()

```

8) Write a program to implement Dijkstra Shortest path routing protocol.

```

import sys

class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.graph = [[0] * vertices for _ in range(vertices)]

    def add_edge(self, u, v, weight):
        self.graph[u][v] = weight
        self.graph[v][u] = weight

    def dijkstra(self, start):
        distance = [sys.maxsize] * self.vertices
        distance[start] = 0
        visited = [False] * self.vertices

        for _ in range(self.vertices):
            min_distance = sys.maxsize
            min_index = 0

            for v in range(self.vertices):
                if distance[v] < min_distance and not visited[v]:
                    min_distance = distance[v]
                    min_index = v

            visited[min_index] = True

```

```

        for v in range(self.vertices):
            if (
                self.graph[min_index][v] > 0
                and not visited[v]
                and distance[v] > distance[min_index] +
self.graph[min_index][v]
            ):
                distance[v] = distance[min_index] + self.graph[min_index][v]

        self.print_solution(distance)

def print_solution(self, distance):
    print("Vertex \t Distance from Source")
    for node in range(self.vertices):
        print(f"{node}\t {distance[node]}")

# Example usage:
if __name__ == "__main__":
    graph = Graph(6)
    graph.add_edge(0, 1, 4)
    graph.add_edge(0, 2, 2)
    graph.add_edge(1, 2, 5)
    graph.add_edge(1, 3, 10)
    graph.add_edge(2, 3, 3)
    graph.add_edge(2, 4, 2)
    graph.add_edge(3, 4, 7)
    graph.add_edge(4, 5, 1)

    start_vertex = 0
    print(f"Shortest paths from vertex {start_vertex} using Dijkstra's algorithm:")
    graph.dijkstra(start_vertex)

```

9) Write a program to implement Distance Vector Routing.

```

import sys

class DistanceVectorRouter:
    def __init__(self, nodes):
        self.nodes = nodes
        self.routing_table = {node: {dest: sys.maxsize for dest in range(nodes)}}
for node in range(nodes):

    def add_link(self, node1, node2, cost):
        self.routing_table[node1][node2] = cost
        self.routing_table[node2][node1] = cost

```

```

def update_routing_table(self):
    for node in range(self.nodes):
        for dest in range(self.nodes):
            if dest != node:
                for neighbor in range(self.nodes):
                    if (
                        self.routing_table[node][neighbor] +
self.routing_table[neighbor][dest]
                        < self.routing_table[node][dest]
                    ):
                        self.routing_table[node][dest] =
self.routing_table[node][neighbor] + self.routing_table[neighbor][dest]

def print_routing_table(self):
    print("Routing Table:")
    for node in range(self.nodes):
        print(f"Node {node}: {self.routing_table[node]}")

def main():
    # Create a router with 5 nodes
    router = DistanceVectorRouter(5)

    # Add links with associated costs
    router.add_link(0, 1, 1)
    router.add_link(0, 2, 3)
    router.add_link(1, 3, 4)
    router.add_link(2, 3, 2)
    router.add_link(2, 4, 5)
    router.add_link(3, 4, 1)

    # Initial routing table
    router.print_routing_table()

    # Simulate updates until convergence
    for _ in range(3):
        router.update_routing_table()
        router.print_routing_table()

if __name__ == "__main__":
    main()

```

10) Write a program to implement link state routing

```

import sys

class LinkStateRouter:
    def __init__(self, nodes):
        self.nodes = nodes

```

```

        self.link_state_database = {node: {dest: sys.maxsize for dest in
range(nodes)} for node in range(nodes)}

    def add_link(self, node1, node2, cost):
        self.link_state_database[node1][node2] = cost
        self.link_state_database[node2][node1] = cost

    def dijkstra(self, start):
        distance = {node: sys.maxsize for node in range(self.nodes)}
        distance[start] = 0
        visited = set()

        while len(visited) < self.nodes:
            current = min((node for node in range(self.nodes) if node not in
visited), key=lambda x: distance[x])
            visited.add(current)

            for neighbor in range(self.nodes):
                if neighbor not in visited and
self.link_state_database[current][neighbor] > 0:
                    new_distance = distance[current] +
self.link_state_database[current][neighbor]
                    if new_distance < distance[neighbor]:
                        distance[neighbor] = new_distance

        return distance

    def print_link_state_database(self):
        print("Link State Database:")
        for node in range(self.nodes):
            print(f"Node {node}: {self.link_state_database[node]}")

def main():
    # Create a router with 5 nodes
    router = LinkStateRouter(5)

    # Add links with associated costs
    router.add_link(0, 1, 1)
    router.add_link(0, 2, 3)
    router.add_link(1, 3, 4)
    router.add_link(2, 3, 2)
    router.add_link(2, 4, 5)
    router.add_link(3, 4, 1)

    # Initial link state database
    router.print_link_state_database()

    # Calculate shortest paths using Dijkstra's algorithm
    for node in range(5):

```

```

shortest_paths = router.dijkstra(node)
print(f"Shortest paths from Node {node}: {shortest_paths}")

if __name__ == "__main__":
    main()

```

- 11) Write a code to display the class of IP address, network mask and generate the subnet IP address based on the IP address entered from the keyboard.

```

def get_ip_class(ip_address):
    first_octet = int(ip_address.split('.')[0])
    if 1 <= first_octet <= 126:
        return 'A'
    elif 128 <= first_octet <= 191:
        return 'B'
    elif 192 <= first_octet <= 223:
        return 'C'
    elif 224 <= first_octet <= 239:
        return 'D (Multicast)'
    elif 240 <= first_octet <= 255:
        return 'E (Reserved)'
    else:
        return 'Invalid IP address'

def generate_subnet(ip_address, subnet_mask):
    ip_parts = list(map(int, ip_address.split('.')))
    mask_parts = list(map(int, subnet_mask.split('.')))

    subnet = []
    for i in range(4):
        subnet.append(ip_parts[i] & mask_parts[i])

    return '.'.join(map(str, subnet))

```

```

def main():

    ip_address = input("Enter the IP address: ")

    ip_class = get_ip_class(ip_address)
    print(f"The IP address {ip_address} belongs to Class {ip_class}")

    subnet_mask = input("Enter the subnet mask: ")

    subnet_ip = generate_subnet(ip_address, subnet_mask)
    print(f"The subnet IP address is: {subnet_ip}")

if __name__ == "__main__":
    main()

```

12) Write a program to transfer the file operation using TCP.

```

import socket

def receive_file(server_address, filename):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect(server_address)

    try:
        client_socket.send(filename.encode('utf-8'))

        with open('received_' + filename, 'wb') as file:
            data = client_socket.recv(1024)
            while data:
                file.write(data)
                data = client_socket.recv(1024)
            print("File received successfully.")

    finally:
        client_socket.close()

def main():
    server_address = ('localhost', 12345)
    filename = input("Enter the filename to request from the server: ")

    receive_file(server_address, filename)

if __name__ == "__main__":

```

```
main()

import socket

def send_file(client_socket, filename):
    try:
        with open(filename, 'rb') as file:
            data = file.read(1024)
            while data:
                client_socket.send(data)
                data = file.read(1024)
            print("File sent successfully.")
    except FileNotFoundError:
        print("File not found.")

def start_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = ('localhost', 12345)
    server_socket.bind(server_address)

    print("Server listening on {}:{}".format(*server_address))
    server_socket.listen(1)

    while True:
        print("Waiting for a connection...")
        client_socket, client_address = server_socket.accept()
        print("Accepted connection from {}:{}".format(*client_address))

        try:
            filename = client_socket.recv(1024).decode('utf-8')
            print("Received request for file: {}".format(filename))
            send_file(client_socket, filename)
        finally:
            client_socket.close()

if __name__ == "__main__":
    start_server()
```