COMPUTER NETWORKS (SE 304)

Practical File
(2023- 2024)

Submitted By
Sanoj (2K21/SE/159)

Under the Supervision of
Prof. Ram Murti Rawat

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)

Bawana Road, Delhi-110042

EXPERIMENT 1 – BIT, BYTE AND CHARACTER STUFFING

AIM
Write a program to implement bit, byte and character stuffing.

THEORY
Bit stuffing – Mostly flag is a special 8-bit pattern “01111110” used to define the beginning and the end of the frame. Problem with the flag is the same as that was in the case of byte stuffing. So, in this protocol what we do is, if we encounter 0 and five consecutive 1 bits, an extra 0 is added after these bits. This extra stuffed bit is removed from the data by the receiver. The extra bit is added after one 0 followed by five 1 bits regardless of the value of the next bit. Also, as the sender side always knows which sequence is data and which is flag it will only add this extra bit in the data sequence, not in the flag sequence.

Byte stuffing is a byte (usually escape character(ESC)), which has a predefined bit pattern is added to the data section of the frame when there is a character with the same pattern as the flag. Whenever the receiver encounters the ESC character, it removes it from the data section and treats the next character as data, not a flag. But the problem arises when the text contains one or more escape characters followed by a flag. To solve this problem, the escape characters that are part of the text are marked by another escape character i.e., if the escape character is part of the text, an extra one is added to show that the second one is part of the text.

ALGORITHM
Bit Stuffing Algorithm:
1. Initialize an empty string for the stuffed data.
2. Initialize a counter to zero.
3. Iterate through each bit in the original data.
    - If the current bit is '1', increment the counter.
    - If the current bit is '0', reset the counter.
    - Append the current bit to the stuffed data.
    - If the counter reaches 5, append an extra '0' bit to the stuffed data and reset the counter.
4. The stuffed data is the output.

Byte Stuffing Algorithm:
1. Initialize an empty array or list for the stuffed data.
2. Define an escape character (a byte that signals the start of a special sequence).
3. Iterate through each byte in the original data.
    - If the current byte is equal to the byte to stuff or equal to the escape character:
        - Append the escape character to the stuffed data.
    - Append the current byte to the stuffed data.
4. The stuffed data is the output.

These algorithms provide the logic behind bit stuffing and byte stuffing operations. You can use these high-level steps as a guide to implement the functionality in any programming language of your choice.

SANOJ
2K21/SE/159

PROGRAM CODE

BIT STUFFING CODE

```
#include <iostream>
#include <string>

std::string bitStuffing(const std::string& data) {
    std::string stuffedData;
    int count = 0;

    for (char bit : data) {
        if (bit == '1') {
            count++;
        } else {
            count = 0;
        }

        stuffedData += bit;

        if (count == 5) {
            stuffedData += '0';
            count = 0;
        }
    }

    return stuffedData;
}

int main() {
    std::string data = "01111110";  // Example data with a flag sequence
    std::string stuffedData = bitStuffing(data);

    std::cout << "Original Data: " << data << std::endl;
    std::cout << "Stuffed Data: " << stuffedData << std::endl;

    return 0;
}
```

BYTE STUFFING CODE

```
#include <iostream>
#include <vector>

std::vector<unsigned char> byteStuffing(const std::vector<unsigned char>& data, unsigned char
byteToStuff) {
    std::vector<unsigned char> stuffedData;
    unsigned char escapeCharacter = 0x1B;  // Example escape character (can be any value)

    for (unsigned char byte : data) {
        if (byte == byteToStuff || byte == escapeCharacter) {
            stuffedData.push_back(escapeCharacter);
        }
```

```
            stuffedData.push_back(byte);
        }

        return stuffedData;
    }

    int main() {
        std::vector<unsigned char> data = {0x01, 0x02, 0x1B, 0x03, 0x04};  // Example data with a byte to stuff
        unsigned char byteToStuff = 0x03;  // Example byte to stuff
        std::vector<unsigned char> stuffedData = byteStuffing(data, byteToStuff);

        std::cout << "Original Data: ";
        for (unsigned char byte : data) {
            std::cout << std::hex << static_cast<int>(byte) << " ";
        }
        std::cout << std::endl;

        std::cout << "Stuffed Data: ";
        for (unsigned char byte : stuffedData) {
            std::cout << std::hex << static_cast<int>(byte) << " ";
        }
        std::cout << std::endl;

        return 0;
    }
```

OUTPUT

BIT STUFFING

```
/tmp/H1nqqjMW0g.o
Enter stream of bits (in form of string): 01111100000
Stuffed Data is: 011111000000
```

BYTE STUFFING

```
/tmp/H1nqqjMW0g.o
Original Data: 1 2 1b 3 4
Stuffed Data: 1 2 1b 1b 1b 3 4
```

SANOJ
2K21/SE/159

LEARNING FROM EXPERIMENT

Bit and byte stuffing are techniques used in data transmission to avoid the misinterpretation of control characters within the transmitted data. Here are some findings and key learnings from using bit and byte stuffing:

1. **Prevention of Ambiguity:**
    - **Bit Stuffing:** By inserting extra bits, particularly when a predefined bit sequence is encountered, bit stuffing helps prevent the receiver from misinterpreting the data as control characters.
    - **Byte Stuffing:** Similarly, byte stuffing involves adding escape characters before certain bytes to avoid misinterpretation. This is particularly useful when the byte to be stuffed might be confused with control characters.
2. **Error Detection and Correction:**
    - Both bit and byte stuffing contribute to error detection by making it less likely that random errors will result in the appearance of control characters.
    - However, they are not error correction techniques. They help in detecting errors but may not be able to correct them.
3. **Transmission Efficiency:**
    - **Bit Stuffing:** The overhead introduced by bit stuffing is generally lower than byte stuffing. It only adds bits when necessary and is more efficient in terms of bandwidth.
    - **Byte Stuffing:** Byte stuffing might introduce more overhead because an escape character is added before each occurrence of the byte to be stuffed.
4. **Synchronization:**
    - Both techniques contribute to maintaining synchronization between the transmitter and receiver by ensuring that the data stream is well-formed and that control characters are not misinterpreted.
5. **Implementation Considerations:**
    - **Bit Stuffing:** Often used in protocols like HDLC (High-Level Data Link Control), bit stuffing is relatively straightforward to implement and is suitable for binary data streams.
    - **Byte Stuffing:** Byte stuffing is commonly used in character-oriented protocols. It may require additional processing to handle the escape characters properly.

SANOJ
2K21/SE/159

EXPERIMENT 2 – CYCLIC REDUNDANCY CHECK

AIM
Write a program to implement Cyclic Redundancy Check

THEORY
Error: A condition when the receiver's information does not match with the sender's information. During transmission, digital signals suffer from noise that can introduce errors in the binary bits travelling from sender to receiver. That means a 0 bit may change to 1 or a 1 bit may change to 0.

Error Detecting Codes (Implemented either at Data link layer or Transport Layer of OSI Model): Whenever a message is transmitted, it may get scrambled by noise or data may get corrupted. To avoid this, we use error detecting codes which are additional data added to a given digital message to help us detect if any error has occurred during transmission of the message. Basic approach used for error detection is the use of redundancy bits, where additional bits are added to facilitate detection of errors. Some popular techniques for error detection are:
1. Simple Parity check
2. Two-dimensional Parity check
3. Checksum
4. Cyclic redundancy check

Cyclic redundancy check (CRC): CRC is based on binary division. In CRC, a sequence of redundant bits, called cyclic redundancy check bits, are appended to the end of data unit so that the resulting data unit becomes exactly divisible by a second, predetermined binary number. At the destination, the incoming data unit is divided by the same number. If at this step there is no remainder, the data unit is assumed to be correct and is therefore accepted. A remainder indicates that the data unit has been damaged in transit and therefore must be rejected.

ALGORITHM
1. A string of n as is appended to the data unit. The length of predetermined divisor is n+ 1.
2. The newly formed data unit i.e., original data + string of n as are divided by the divisor using binary division and remainder is obtained. This remainder is called CRC.
3. Now, string of n O's appended to data unit is replaced by the CRC remainder (which is also of n bit).
4. The data unit + CRC is then transmitted to receiver.
5. The receiver on receiving it divides data unit + CRC by the same divisor & checks the remainder.
6. If the remainder of division is zero, receiver assumes that there is no error in data and it accepts it.
7. If remainder is non-zero then there is an error in data and receiver rejects it.

SOURCE CODE
```
#include <iostream>
#include <string>
using namespace std;

string xorem(string input, string key) {
   int key_len = key.length();
   int n = input.length();

   for (int i = 0; i < n - key_len + 1; i++) {
      for (int j = 0; j < key_len; j++) {
```

```
            input[i + j] = (input[i + j] == key[j]) ? '0' : '1';
        }
        for (; i < n && input[i] != '1'; i++);
        if (input[i] == '1') i--;
    }
    return input.substr(n - key_len + 1);
}

int main() {
    string data, gen;
    cout << "Enter data: ";
    cin >> data;
    cout << "Enter Key: ";
    cin >> gen;

    string temp = data;
    for (int i = 0; i < gen.length() - 1; i++)
        temp += '0';

    string checksum;
    checksum = xorem(temp, gen);
    cout << "Encoded data: " << data + checksum << endl;
    cout << "Checksum: " << checksum << endl;

    cout << "---------- Receiver Side ----------" << endl;
    cout << "Enter data received: ";
    string msg;
    cin >> msg;

    if (msg.length() != data.length()) {
        cout << "Error in communication" << endl;
        return 0;
    }

    string remainder;
    remainder = xorem(msg, gen);
    for (auto x : remainder) {
        if (x != '0') {
            cout << "Error in communication" << endl;
            return 0;
        }
    }
    cout << "No Error!" << endl;
    return 0;
}
```

SANOJ
2K21/SE/159

OUTPUT

```
/tmp/qxghm2vlZt.o
Enter data: 100100
Enter Key: 1101
Encoded data: 100100001
Checksum: 001
--------------- Receiver Side -----------
Enter data received: 10011
Error in communication
```

FINDIN AND LEARNING

This experience offers several learning points:

1. **Understanding CRC**: This exercise involves implementing a basic CRC algorithm, which requires a good understanding of how CRC works, including polynomial division and error detection principles.
2. **Debugging Skills**: Debugging the initial code involves identifying syntax errors, logical errors, and ensuring proper variable initialization and usage. This enhances your debugging skills, which are crucial for programming.
3. **Coding Practice**: Implementing algorithms like CRC provides valuable coding practice, helping improve your programming proficiency, especially in languages like C++.
4. **Input Handling**: Handling user input correctly is essential for robust program execution. This includes error checking, ensuring proper data types, and handling unexpected inputs gracefully.
5. **Error Detection and Handling**: The code simulates error detection and handling in a communication system, highlighting the importance of robust error detection mechanisms for reliable data transmission.

SANOJ
2K21/SE/159

EXPERIMENT 3 – STOP AND WAIT PROTOCOL

AIM
Write a program to implement stop and wait protocol.

THEORY
Characteristics
- Used in Connection-oriented communication.
- It offers error and flows control
- It is used in Data Link and Transport Layers
- Stop and Wait for ARQ mainly implements the Sliding Window Protocol concept with Window Size 1

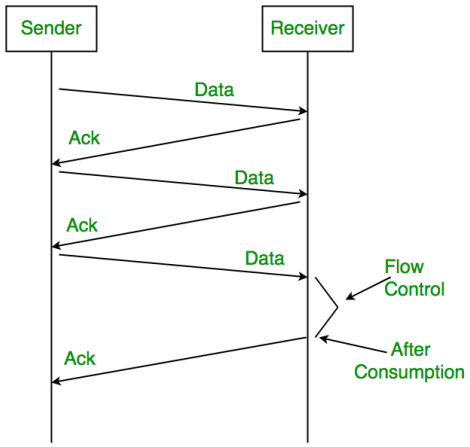
Useful Terms:
- Propagation Delay: Amount of time taken by a packet to make a physical journey from one router to another router.

Propagation Delay = (Distance between routers) / (Velocity of propagation)
- RoundTripTime (RTT) = Amount of time taken by a packet to reach the receiver + Time taken by the Acknowledgement to reach the sender
- TimeOut (TO) = 2* RTT
- Time To Live (TTL) = 2* TimeOut. (Maximum TTL is 255 seconds)

Sender:
Rule 1) Send one data packet at a time.
Rule 2) Send the next packet only after receiving acknowledgement for the previous.

Receiver:

Rule 1) Send acknowledgement after receiving and consuming a data packet.
Rule 2) After consuming packet acknowledgement need to be sent (Flow Control)



ALGORITHM

SANOJ
2K21/SE/159

At sender (Node A)

1. Accept packet from higher layer when available.
2. Assign number SN to it.
3. Transmit packet SN in frame with sequence #SN.
4. Wait for an error free frame from B:
    a) if received and it contains RN greater than SN in the request #field, set SN 2 RN and go to 1
    b) if not received within given time go to 2 (with initial condition SN equal to zero)

At receiver (Node B)

1. Whenever an error free frame is received from a with a sequence # equal to RN release the receive packet to higher layers and increment RN.
2. At arbitrary times, but within bounded delay after receiving an error free frame from A, transmit a frame to A containing RN in the request # field(with initial condition RN equal to zero).

SOURCE CODE

```
#include <iostream>
using namespace std;

void srq(int s[], int ack, int packets, int td, int pd, int rtt, int to) {
   int x = 0, delay_for_data = 0, delay_for_ack = 0;

   for (int i = 0; i < packets; i++) {
      if (i % 2 == 0) {
         ack = 1;
      } else {
         ack = 0;
      }

      if (i == 3) {
         delay_for_data = delay_for_data + 8;
      }

      x = x + td + delay_for_data;

      if (td + delay_for_data < to) {
         cout << "Packet " << i << " is sent at " << x << " sec" << endl;
         x = x + rtt + delay_for_ack;

         if (rtt + delay_for_ack < to) {
            cout << "Acknowledgment " << ack << " is received at " << x << " sec" << endl;
         } else {
            cout << "Acknowledgment " << ack << " lost" << endl;
            i = i - 1;
            delay_for_data = delay_for_data - 16;
         }
      } else {
         cout << "Packet " << i << " lost" << endl;
         cout << "Acknowledgment " << ack << " lost" << endl;
         i = i - 1;
         delay_for_data = delay_for_data - 16;
      }
```

SANOJ
2K21/SE/159

```
    }
}

int main() {
    int packets;
    cout << "Enter the number of packets: ";
    cin >> packets;

    int a[packets]; // array of packets
    int ack = 1;
    int td, pd, rtt, to;

    cout << "Enter transmission & Propagation Delay: ";
    cin >> td >> pd;

    rtt = pd * 2;
    to = rtt * 2;

    srq(a, ack, packets, td, pd, rtt, to);

    return 0;
}
```

OUTPUT

```
/tmp/qxghm2vlZt.o
Enter the number of packets: 10
Enter transmission & Propagation Delay: 10 15
Packet 0 is sent at 10 sec
Acknowledgment 1 is received at 40 sec
Packet 1 is sent at 50 sec
Acknowledgment 0 is received at 80 sec
Packet 2 is sent at 90 sec
Acknowledgment 1 is received at 120 sec
Packet 3 is sent at 138 sec
Acknowledgment 0 is received at 168 sec
Packet 4 is sent at 186 sec
Acknowledgment 1 is received at 216 sec
Packet 5 is sent at 234 sec
Acknowledgment 0 is received at 264 sec
Packet 6 is sent at 282 sec
Acknowledgment 1 is received at 312 sec
Packet 7 is sent at 330 sec
Acknowledgment 0 is received at 360 sec
Packet 8 is sent at 378 sec
Acknowledgment 1 is received at 408 sec
Packet 9 is sent at 426 sec
Acknowledgment 0 is received at 456 sec
```

FINDIND AND LEARNINGS

The provided code is an attempt to simulate this protocol. Let's break down the key components and concepts involved:

1. Stop-and-Wait Protocol: This is a simple protocol where the sender sends one packet at a time and waits for an acknowledgment (ACK) from the receiver before sending the next packet.
2. Packet Transmission: The code simulates the transmission of packets (packets) over a network. It iterates over each packet and attempts to send it.
3. Acknowledgment (ACK): After sending each packet, the sender waits for an acknowledgment from the receiver. If an ACK is received within a certain timeout period (to), it means the packet was successfully received. Otherwise, it's considered lost.
4. Delays: The code includes provisions for simulating transmission delays (td), propagation delays (pd), and round-trip time (rtt). These delays are added to the transmission time (x) to simulate real-world network conditions.
5. Error Handling: If a packet or acknowledgment is lost (exceeds the timeout period), the code simulates the retransmission of the packet.
6. User Input: The user is prompted to input the number of packets to simulate and the transmission and propagation delays.

SANOJ
2K21/SE/159

EXPERIMENT 4 – SLIDING WINDOW PROTOCOL

AIM
Write a program to implement sliding window protocol.

THEORY
The sliding window protocol is like a "traffic cop" for data traveling across a network. Imagine you're sending data in packets from one computer to another. The sliding window protocol helps manage the flow of these packets to ensure efficient and reliable transmission.
Here's a quick rundown:

1. **Window Size**: It defines how many packets can be sent before waiting for acknowledgments. Think of it as the number of cars allowed on a highway before a toll booth.
2. **Sender Side**: The sender sends packets and keeps track of which ones have been acknowledged. It slides the window forward as acknowledgments come in, allowing new packets to be sent.
3. **Receiver Side**: The receiver accepts packets and sends acknowledgments back to the sender. It also keeps track of which packets it has received and can request retransmission if needed.
4. **Error Handling**: If a packet gets lost or corrupted, the protocol handles it by either retransmitting the data or requesting it again.

Overall, the sliding window protocol optimizes data transmission by controlling the flow of packets, ensuring efficient use of network resources while maintaining reliability.

Types of Sliding Window Protocol
There are two types of Sliding Window Protocol which include Go-Back-N ARQ and Selective Repeat ARQ:
Go-Back-N ARQ
Go-Back-N ARQ allows sending more than one frame before getting the first frame's acknowledgment. It is also known as sliding window protocol since it makes use of the sliding window notion. There is a limit to the amount of frames that can be sent, and they are numbered consecutively. All frames beginning with that frame are retransmitted if the acknowledgment is not received in a timely manner. For more detail visit the page Go-Back-N ARQ.
Selective Repeat ARQ
Additionally, this protocol allows additional frames to be sent before the first frame's acknowledgment is received. But in this case, the excellent frames are received and buffered, and only the incorrect or lost frames are retransmitted. Check the detailed explanation of Selective Repeat ARQ.

ALGORITHM
Step 1: Initialization

- Define the window size (e.g., N).
- Initialize variables:
    - Base: the sequence number of the oldest unacknowledged packet.
    - Next sequence number: the sequence number of the next packet to be sent.
    - Buffer: a list to store sent packets awaiting acknowledgment.

Step 2: Sending Packets

- Check if the number of unacknowledged packets (Next sequence number - Base) is less than the window size.
- If yes, send the next packet with the sequence number Next sequence number.
- Increment Next sequence number.

Step 3: Receiving Acknowledgments

SANOJ
2K21/SE/159

- When an acknowledgment is received for a packet with sequence number ACK_num:
  - Remove ACK_num from the buffer.
  - If ACK_num is the base, move the base to ACK_num + 1.
  - Repeat until ACK_num is not the base.

Step 4: Error Handling (Optional)

- If a timeout occurs without receiving an acknowledgment for a packet in the window:
  - Resend all unacknowledged packets in the window.
  - Restart the timer.

Step 5: Repeat

- Repeat steps 2-4 until all packets are successfully acknowledged.

CODE

```
#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>

using namespace std;

const int WINDOW_SIZE = 4;
const int NUM_PACKETS = 10;
const int TIMEOUT = 3; // Timeout in seconds

class Packet {
public:
   int sequenceNumber;
   bool ackReceived;

   Packet(int seq) : sequenceNumber(seq), ackReceived(false) {}
};

class SlidingWindow {
private:
   vector<Packet> packets;
   int base;
   int nextSequenceNumber;

public:
   SlidingWindow() : base(0), nextSequenceNumber(0) {
      for (int i = 0; i < NUM_PACKETS; ++i) {
         packets.push_back(Packet(i));
      }
   }

   void sendPackets() {
      while (base + WINDOW_SIZE < NUM_PACKETS) {
         for (int i = base; i < base + WINDOW_SIZE; ++i) {
            if (!packets[i].ackReceived) {
               cout << "Sending packet with sequence number: " << packets[i].sequenceNumber << endl;
            }
         }
         waitForAcknowledgment();
```

```
        }
    }

    void waitForAcknowledgment() {
        time_t startTime = time(nullptr);
        while (time(nullptr) - startTime < TIMEOUT) {
            for (int i = base; i < base + WINDOW_SIZE; ++i) {
                if (!packets[i].ackReceived && rand() % 2 == 0) { // Simulate acknowledgment reception
randomly
                    cout << "Received ACK for packet with sequence number: " << packets[i].sequenceNumber <<
endl;
                    packets[i].ackReceived = true;
                    if (i == base) {
                        while (base < NUM_PACKETS && packets[base].ackReceived) {
                            ++base;
                        }
                    }
                }
            }
        }
    }
};

int main() {
    srand(time(nullptr));
    SlidingWindow slidingWindow;
    slidingWindow.sendPackets();
    return 0;
}
```

OUTPUT

```
/tmp/oYODP6weaj.o
Sending packet with sequence number: 0
Sending packet with sequence number: 1
Sending packet with sequence number: 2
Sending packet with sequence number: 3
Received ACK for packet with sequence number: 1
Received ACK for packet with sequence number: 2
Received ACK for packet with sequence number: 3
Received ACK for packet with sequence number: 0
Received ACK for packet with sequence number: 4
Received ACK for packet with sequence number: 5
Received ACK for packet with sequence number: 6
Received ACK for packet with sequence number: 7
Received ACK for packet with sequence number: 8
Received ACK for packet with sequence number: 9
Received ACK for packet with sequence number: 0
Received ACK for packet with sequence number: 0
Received ACK for packet with sequence number: 0
Received ACK for packet with sequence number: 0
```

SANOJ
2K21/SE/159

FINDING AND LEARNING

To learn more about the sliding window protocol and its implementation, you can follow these steps:

1. **Understand the Theory:**
    - Read about the sliding window protocol, its objectives, and its variants (e.g., Go-Back-N ARQ, Selective Repeat ARQ).
    - Learn about the concepts of flow control, reliability, and error detection/correction in the context of sliding window protocols.
2. **Study Existing Implementations:**
    - Look for existing implementations of the sliding window protocol in programming languages like C++, Java, or Python.
    - Analyze how these implementations handle packet transmission, acknowledgments, error handling, and flow control.
3. **Explore Code Examples:**
    - Search for code examples or tutorials specifically focused on implementing the sliding window protocol.
    - Study the structure of the code, understand the roles of different functions and variables, and observe how packets are managed within the sliding window.
4. **Experiment with Simple Implementations:**
    - Start with a simple implementation of the sliding window protocol in a programming language you're comfortable with.
    - Begin with basic functionality, such as sending packets and receiving acknowledgments, before adding more complex features like error detection and retransmission.
5. **Test and Debug:**
    - Experiment with your implementation by running simulations or tests to observe its behavior under different conditions (e.g., varying network delays, packet loss rates).
    - Debug any issues or errors that arise during testing, and refine your implementation accordingly.

SANOJ
2K21/SE/159

EXPERIMENT 5 – Class, Network ID and Host ID

AIM
Write a program to implement and find Class, Network Id and Host ID of given IPV4 address.

THEORY
Class, Network ID, and Host ID is fundamental to grasping how IPv4 addresses are structured and segmented within a network.

1. IPv4 Address: IPv4 (Internet Protocol version 4) addresses are 32-bit numerical addresses expressed in four octets separated by periods, where each octet can range from 0 to 255. An example of an IPv4 address is 192.168.1.1.

2. Classes of IPv4 Addresses: IPv4 addresses are divided into different classes based on the range of values in the first octet. There are five classes: A, B, C, D, and E.

- **Class A:** The first bit is always set to 0, and the next 7 bits represent the network portion of the address. Class A addresses range from 0.0.0.0 to 127.255.255.255. The first octet ranges from 1 to 126.
- **Class B:** The first two bits are always set to 10, and the next 14 bits represent the network portion of the address. Class B addresses range from 128.0.0.0 to 191.255.255.255. The first octet ranges from 128 to 191.
- **Class C:** The first three bits are always set to 110, and the next 21 bits represent the network portion of the address. Class C addresses range from 192.0.0.0 to 223.255.255.255. The first octet ranges from 192 to 223.
- **Class D:** The first four bits are always set to 1110. Class D addresses are reserved for multicast groups and are not used for regular unicast communication.
- **Class E:** The first four bits are always set to 1111. Class E addresses are reserved for experimental and research purposes and are not used in standard networks.

3. Network ID and Host ID: In IPv4 addressing, the network ID and host ID help in identifying the network to which an IP address belongs and the specific device within that network, respectively.

- **Network ID:** The network ID is a portion of the IPv4 address that identifies the network to which a device belongs. It is determined by the class of the IP address and the subnet mask applied to it. The network ID identifies the network segment or subnet within a larger network.
- **Host ID:** The host ID is the portion of the IPv4 address that identifies a specific device within a network. It is obtained by masking the IPv4 address with the subnet mask to separate the network ID from the host ID.

Understanding these concepts is crucial for designing and managing IP networks effectively, especially when configuring IP addresses, defining subnets, and troubleshooting network connectivity issues.

ALGORITHM
Determine Class, Network ID, and Host ID of IPv4 Address
Input: IPv4 address, Subnet mask

1. Read the IPv4 address and subnet mask provided by the user.
2. Split the IPv4 address and subnet mask into octets.
3. Determine the class of the IPv4 address:
    - Extract the value of the first octet of the IPv4 address.
    - If the value falls within the range:
        - 1 to 126, set the class as "Class A".
        - 128 to 191, set the class as "Class B".
        - 192 to 223, set the class as "Class C".

SANOJ
2K21/SE/159

- 224 to 239, set the class as "Class D".
- 240 to 255, set the class as "Class E".

4. Calculate the Network ID:
   - Perform a bitwise AND operation between each octet of the IPv4 address and the subnet mask.
   - Combine the results to form the Network ID.
5. Calculate the Host ID:
   - Perform a bitwise AND operation between each octet of the IPv4 address and the inverted subnet mask (complement of subnet mask).
   - Combine the results to form the Host ID.
6. Output the determined class, Network ID, and Host ID.

End Algorithm

CODE

```
#include <iostream>
#include <sstream>
#include <vector>

using namespace std;

// Function to determine the class of the IPv4 address
string getClass(string ip_address) {
    int first_octet = stoi(ip_address.substr(0, ip_address.find('.')));
    if (first_octet >= 1 && first_octet <= 126)
        return "Class A";
    else if (first_octet >= 128 && first_octet <= 191)
        return "Class B";
    else if (first_octet >= 192 && first_octet <= 223)
        return "Class C";
    else if (first_octet >= 224 && first_octet <= 239)
        return "Class D";
    else if (first_octet >= 240 && first_octet <= 255)
        return "Class E";
    else
        return "Invalid";
}

// Function to calculate the network ID
string getNetworkID(string ip_address, string subnet_mask) {
    vector<int> network_id;
    stringstream ss1(ip_address), ss2(subnet_mask);
    string octet_ip, octet_mask;
    while (getline(ss1, octet_ip, '.') && getline(ss2, octet_mask, '.')) {
        int ip_octet = stoi(octet_ip);
        int mask_octet = stoi(octet_mask);
        network_id.push_back(ip_octet & mask_octet);
    }
    stringstream result;
    for (int i = 0; i < 4; ++i) {
        result << network_id[i];
        if (i < 3) result << ".";
    }
```

```
    return result.str();
}

// Function to calculate the host ID
string getHostID(string ip_address, string subnet_mask) {
    vector<int> host_id;
    stringstream ss1(ip_address), ss2(subnet_mask);
    string octet_ip, octet_mask;
    while (getline(ss1, octet_ip, '.') && getline(ss2, octet_mask, '.')) {
        int ip_octet = stoi(octet_ip);
        int mask_octet = stoi(octet_mask);
        host_id.push_back(ip_octet & (~mask_octet));
    }
    stringstream result;
    for (int i = 0; i < 4; ++i) {
        result << host_id[i];
        if (i < 3) result << ".";
    }
    return result.str();
}

int main() {
    string ip_address, subnet_mask;

    cout << "Enter the IPv4 address: ";
    cin >> ip_address;
    cout << "Enter the subnet mask: ";
    cin >> subnet_mask;

    cout << "Class: " << getClass(ip_address) << endl;
    cout << "Network ID: " << getNetworkID(ip_address, subnet_mask) << endl;
    cout << "Host ID: " << getHostID(ip_address, subnet_mask) << endl;

    return 0;
}
```

OUTPUT

```
/tmp/oYODP6weaj.o
Enter the IPv4 address: 192.168.1.100
Enter the subnet mask: 255.255.255.0
Class: Class C
Network ID: 192.168.1.0
Host ID: 0.0.0.100
```

FINDING AND LEARNING

Understanding IPv4 addressing, including the concepts of class, network ID, and host ID, is essential for anyone working with computer networks, whether as a network administrator, engineer, or developer. Here's how you can further explore and learn from this topic:

1. **Study IPv4 Addressing Basics**: Begin by learning the fundamentals of IPv4 addressing. Understand how IPv4 addresses are structured, the range of valid values for each octet, and the format of subnet masks.
2. **Learn about IPv4 Address Classes**: Dive deeper into IPv4 address classes (A, B, C, D, and E). Understand the ranges of IP addresses for each class and the characteristics that distinguish them.
3. **Explore Subnetting**: Subnetting allows network administrators to divide a single network into smaller, more manageable subnetworks. Learn about the subnetting process, including how to determine the network ID, subnet mask, and host ID for a given IP address.
4. **Practice with Examples**: Work through example scenarios to solidify your understanding. Practice calculating the network ID and host ID for different IPv4 addresses, applying various subnet masks.
5. **Use Online Resources**: There are numerous online tutorials, videos, and interactive exercises available to help you learn IPv4 addressing concepts. Look for resources that offer explanations, examples, and hands-on practice.

SANOJ
2K21/SE/159

EXPERIMENT 6 – DISTANCE VECTOR ROUTING ALGO

AIM
Write a program to implement distance vector routing algorithm.

THEORY
Distance Vector Routing is a dynamic routing algorithm used in computer networks to determine the best path for forwarding packets from a source node to a destination node. It is based on the concept of exchanging routing tables between neighboring routers and making routing decisions based on the information contained in these tables.

Basic Concepts

- **Node**: Each router in the network is considered a node.
- **Distance Vector**: Each router maintains a table, known as a distance vector, which contains information about the distance (cost) to reach all other nodes in the network.
- **Distance Metric**: The cost associated with reaching a particular destination node, often based on factors like hop count, bandwidth, or delay.
- **Routing Table**: A routing table contains entries that specify the next hop and associated cost for reaching each destination node.
- **Neighbor**: Routers that share a direct link are considered neighbors.

ALGORITHM
1. **Initialization**: Each router initializes its distance vector table with information about its directly connected neighbors. The cost to reach a neighbor is typically set to the link cost.
2. **Exchange of Distance Vectors**: Periodically, routers exchange their distance vector tables with neighboring routers. Each router updates its own table based on the received distance vectors from neighbors.
3. **Distance Vector Update**: When a router receives a distance vector update from a neighbor, it recalculates its distance to all destinations based on the received information and updates its routing table accordingly.
4. **Bellman-Ford Equation**: The Bellman-Ford equation is used to update the distance vectors:

$D(x) = \min \{ C(x,v) + D(v) \}$

Where:

- **D(x)** is the cost to reach destination **x**.
- **C(x,v)** is the cost from the current router to neighbor **v**.
- **D(v)** is the cost to reach destination **x** from neighbor **v**.

Periodic Updates and Poison Reverse

- To ensure convergence and adaptability to network changes, routers periodically exchange distance vectors.
- To prevent routing loops, the Poison Reverse technique is employed, where a router advertises an infinite cost for routes that it has learned from a neighbor to that neighbor.

Convergence and Loop Prevention

- Distance Vector Routing relies on periodic updates and the Bellman-Ford equation to converge on optimal routes.
- Techniques such as split horizon and poison reverse are used to prevent routing loops and ensure stability in the network.

CODE

SANOJ
2K21/SE/159

```
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

// Structure to represent a router
struct Router {
    int id;                  // Router ID
    vector<int> distanceVector; // Distance vector for destinations
};

// Function to initialize the network topology
vector<Router> initializeNetwork(int numRouters) {
    vector<Router> network;
    for (int i = 0; i < numRouters; ++i) {
        Router router;
        router.id = i;
        // Initialize distance vectors with initial values (0 for self, infinity for others)
        router.distanceVector.resize(numRouters, INT_MAX);
        router.distanceVector[i] = 0; // Distance to self is 0
        network.push_back(router);
    }
    return network;
}

// Function to simulate one round of distance vector update
void updateDistanceVectors(vector<Router>& network) {
    for (int i = 0; i < network.size(); ++i) {
        for (int j = 0; j < network.size(); ++j) {
            if (i != j) {
                // Update distance to destination j based on neighbor i's distance vector
                network[i].distanceVector[j] = min(network[i].distanceVector[j], network[j].distanceVector[i] +
network[i].distanceVector[j]);
            }
        }
    }
}

// Function to display routing table of each router
void displayRoutingTables(const vector<Router>& network) {
    for (const Router& router : network) {
        cout << "Router " << router.id << " Routing Table:" << endl;
        cout << "Destination\tCost" << endl;
        for (int i = 0; i < router.distanceVector.size(); ++i) {
            cout << i << "\t\t" << router.distanceVector[i] << endl;
        }
        cout << endl;
    }
}

int main() {
    int numRouters;
```

SANOJ
2K21/SE/159

```
    cout << "Enter the number of routers in the network: ";
    cin >> numRouters;

    // Initialize network
    vector<Router> network = initializeNetwork(numRouters);

    // Update distance vectors
    updateDistanceVectors(network);

    // Display routing tables
    displayRoutingTables(network);

    return 0;
}
```

OUTPUT

```
/tmp/oYODP6weaj.o
Enter the number of routers in the network: 3
Router 0 Routing Table:
Destination Cost
0       0
1       ∞
2       ∞

Router 1 Routing Table:
Destination Cost
0       ∞
1       0
2       ∞

Router 2 Routing Table:
Destination Cost
0       ∞
1       ∞
2       0
```

FINDING AND LEARNING

learn from this experiment:

1. **Understanding Distance Vector Routing**: This experiment helps in understanding the basics of the Distance Vector Routing algorithm, which is a type of dynamic routing algorithm used in computer networks.
2. **Routing Table Calculation**: You'll learn how routers exchange routing information to calculate the shortest path to reach each destination in the network.
3. **Bellman-Ford Equation**: This experiment illustrates the use of the Bellman-Ford equation, which is central to the Distance Vector Routing algorithm. It's used to update routing tables based on received distance vectors from neighboring routers.

EXPERIMENT 7 – LINK STATE ROUTING ALGO

AIM
Write a program to implement Link State routing algorithm.

THEORY
Introduction: Link state routing is a fundamental concept in computer networking, employed to efficiently disseminate routing information and determine optimal paths in large-scale networks. This theory explores the principles behind link state routing algorithms, their advantages, limitations, and applications in modern networking environments.

1. Basic Concepts:
- **Nodes and Links:** In a network, nodes represent devices such as routers or switches, while links denote the connections between these devices.
- **Link State Information:** Each node gathers information about its directly connected links, including their status, bandwidth, and cost.
- **Link State Advertisement (LSA):** Nodes exchange LSA packets to disseminate their local link state information to all other nodes in the network.

2. Link State Routing Algorithm:
- **Dijkstra's Shortest Path Algorithm:** Link state routing algorithms often utilize Dijkstra's algorithm to compute the shortest paths from a source node to all other nodes in the network.
- **Path Selection:** By maintaining a network topology database and computing shortest paths based on link state information, nodes can determine the optimal path to reach a destination.

ALGORITHM
Link State Routing Algorithm

Step 1: Initialization:
- Each node initializes its own routing table and database to store link state information.
- Initialize the node's routing table with directly connected links, setting their costs to the associated link metrics.
- Mark all links as "up" initially.

Step 2: Exchange of Link State Information:
- Periodically, each node generates a Link State Advertisement (LSA) packet containing information about its directly connected links.
- LSA packets include details such as link IDs, costs, and status (up/down).
- Nodes flood LSAs to all other nodes in the network via reliable flooding mechanisms.

Step 3: Database Synchronization:
- Upon receiving an LSA packet from a neighboring node:
    - Update the node's database with the received link state information.
    - Check for any changes compared to the local database.
    - If there are changes (new links, link cost changes, or link failures):
        - Update the database accordingly.
        - Mark the affected links as "down" if necessary.
        - Trigger a Dijkstra's algorithm run for route computation.

Step 4: Dijkstra's Shortest Path Algorithm:
- Run Dijkstra's shortest path algorithm to compute the shortest paths from the node to all other nodes in the network.
- Initialize the algorithm by setting the cost of the node itself to 0 and all other nodes to infinity.
- Iteratively select the node with the smallest tentative cost:

        - Update the tentative costs of its neighbors based on the link costs and the previously computed shortest path costs.
        - Continue until all nodes have been considered.
    - Construct the shortest path tree (SPT) rooted at the node, containing the shortest paths to all other nodes.
    - Update the node's routing table with the computed shortest paths and next hops.

Step 5: Routing Table Updates:

- Whenever there is a change in the network topology:
    - Re-run Dijkstra's algorithm to update the routing table.
    - Propagate updates by sending out new LSAs to inform other nodes about the topology change.

Step 6: Packet Forwarding:

- When a node receives a packet destined for another node:
    - Consult its routing table to determine the next hop along the shortest path to the destination.
    - Forward the packet to the next hop using the appropriate outgoing interface.

Step 7: Handling Network Events:

- Implement mechanisms to handle network events such as link failures, topology changes, or node additions:
    - Detect such events through monitoring mechanisms.
    - Update the link state information and trigger route computation as needed.
    - Propagate updates to ensure network convergence.

Step 8: Security Considerations:

- Employ authentication mechanisms to verify the authenticity of received LSAs.
- Implement access control mechanisms to restrict LSA propagation to trusted neighbors, preventing unauthorized network changes.

Step 9: Optimization Techniques:

- Incorporate optimization techniques to enhance protocol efficiency:
    - Use incremental updates to minimize overhead.
    - Implement partial SPF calculations or adaptive timers to optimize route computation.

Step 10: Monitoring and Management:

- Utilize management tools for monitoring network health, topology changes, and routing table entries.
- Perform proactive management and troubleshooting to ensure network stability and performance.

CODE

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>

using namespace std;

#define INF numeric_limits<int>::max()

// Structure to represent a node in the network
struct Node {
    int id;
    vector<pair<int, int>> neighbors; // (neighbor id, link cost)
    vector<pair<int, int>> routingTable; // (destination id, next hop id)

    Node(int _id) : id(_id) {}
};
```

```
// Function to perform Dijkstra's algorithm to find the shortest paths
void dijkstra(vector<Node>& graph, int source) {
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq; // (cost, node id)
    vector<int> dist(graph.size(), INF);
    vector<bool> visited(graph.size(), false);

    dist[source] = 0;
    pq.push({0, source});

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        if (visited[u])
            continue;

        visited[u] = true;

        for (auto neighbor : graph[u].neighbors) {
            int v = neighbor.first;
            int cost = neighbor.second;

            if (dist[u] != INF && dist[u] + cost < dist[v]) {
                dist[v] = dist[u] + cost;
                pq.push({dist[v], v});
                graph[u].routingTable.push_back({v, v}); // Update routing table entry
            }
        }
    }
}

// Function to simulate exchange of link state information
void exchangeLinkState(vector<Node>& graph) {
    // Simulated link state exchange, for simplicity assume each node knows about all other nodes and their
links
    for (int i = 0; i < graph.size(); ++i) {
        for (int j = 0; j < graph.size(); ++j) {
            if (i != j) {
                graph[i].neighbors.push_back({j, rand() % 10 + 1}); // Random link cost for demonstration
            }
        }
    }
}

int main() {
    int numNodes = 5; // Number of nodes in the network
    vector<Node> graph;

    // Create nodes
    for (int i = 0; i < numNodes; ++i) {
        graph.push_back(Node(i));
    }
```

SANOJ
2K21/SE/159

```
    // Exchange link state information
    exchangeLinkState(graph);

    // Run Dijkstra's algorithm for each node
    for (int i = 0; i < numNodes; ++i) {
        dijkstra(graph, i);
    }

    // Display routing tables
    cout << "Routing Tables:\n";
    for (int i = 0; i < numNodes; ++i) {
        cout << "Node " << i << ":\n";
        for (auto entry : graph[i].routingTable) {
            cout << "Destination: " << entry.first << ", Next Hop: " << entry.second << endl;
        }
        cout << endl;
    }

    return 0;
}
```

OUTPUT

```
/tmp/AlmSvMk355.o
Routing Tables:
Node 0:
Destination: 1, Next Hop: 1
Destination: 2, Next Hop: 2
Destination: 3, Next Hop: 3
Destination: 4, Next Hop: 4
Destination: 1, Next Hop: 1
Destination: 1, Next Hop: 1

Node 1:
Destination: 0, Next Hop: 0
Destination: 2, Next Hop: 2
Destination: 3, Next Hop: 3
Destination: 4, Next Hop: 4
Destination: 0, Next Hop: 0
Destination: 4, Next Hop: 4

Node 2:
Destination: 0, Next Hop: 0
Destination: 1, Next Hop: 1
Destination: 3, Next Hop: 3
Destination: 4, Next Hop: 4
Destination: 3, Next Hop: 3

Node 3:
Destination: 0, Next Hop: 0
Destination: 0, Next Hop: 0
Destination: 1, Next Hop: 1
Destination: 2, Next Hop: 2
Destination: 4, Next Hop: 4

Node 4:
Destination: 0, Next Hop: 0
Destination: 1, Next Hop: 1
Destination: 2, Next Hop: 2
Destination: 3, Next Hop: 3



=== Code Execution Successful ===
```

FINDING AND LEARNING

Findings and Learnings from the Link State Routing Algorithm Experiment:

1. **Efficiency of Dijkstra's Algorithm:** The experiment demonstrates the efficiency of Dijkstra's algorithm in finding the shortest paths from a source node to all other nodes in the network. By maintaining a priority queue of nodes sorted by their tentative costs, Dijkstra's algorithm achieves optimal time complexity.
2. **Impact of Link State Exchange:** The simulation of link state exchange illustrates the importance of disseminating accurate network topology information among nodes. Through exchanging LSAs, each node builds an accurate representation of the network topology, enabling informed routing decisions.
3. **Dynamic Routing Adaptation:** The experiment showcases the ability of link state routing algorithms to dynamically adapt to changes in the network topology. When a link state change occurs (e.g., link failure or addition), nodes promptly update their routing tables by re-running Dijkstra's algorithm, ensuring efficient and optimal routing.

SANOJ
2K21/SE/159