INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal - 500 043, Hyderabad, Telangana

COURSE CONTENT

COMPUTER NETWORKS LABORATORY								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
AITC12	**Core**	**L**	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
		0	0	2	1	30	70	100
Contact Classes: Nil	**Tutorial Classes: Nil**	**Practical Classes: 45**				**Total Classes: 45**		
Prerequisite: There are no prerequisites to take this course.								

I. COURSE OVERVIEW:

The main emphasis of this course is on the organization and management of local area networks (LANs) wide area networks (WANs). The course includes learning about computer network organization and implementation, obtaining a theoretical understanding of data communication and computer networks. Topics include layered network architectures, addressing, naming, forwarding, routing, communication reliability, the client-server model, and web and email protocols. The applications of this course are to design, implement and maintain a basic computer networks.

II. COURSE OBJECTIVES:

The students will try to learn:

I. How computer network hardware and software can operate.

II. Investigate and understand the fundamental concepts of computer networking.

III. The data transmission through protocols across the network in wired and wireless using routing algorithms.

III. COURSE OUTCOMES:

After successful completion of the course, students should be able to:

CO1	Outline the basic concepts of data communications including the key aspects of networking and their interrelationship, packet, circuit and cell switching as internal and external operations, physical structures, types, models, and internetworking
CO2	Make use of different types of of bit errors and the concept of bit redundancy for error detection and error correction.
CO3	Identify the suitable design parameters and algorithms for assuring quality of service and internetworking in various internet protocols
CO4	Interpret transport protocols (TCP,UDP) for measuring the network performance.
CO5	Illustrate the various protocols (FTP, SMTP, TELNET, EMAIL, and WWW) and standards (DNS) in data communications among network.
CO6	Compare various networking models (OSI, TCP/IP) in terms of design parameters and communication modes.

COMPUTER NETWORKS LABORATORY

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

Getting Started Exercises

Introduction:

The main emphasis of this course is on the organization and management of local area networks (LANs) wide area networks (WANs). The course includes learning about computer network organization and implementation, obtaining a theoretical understanding of data communication and computer networks. Topics include layered network architectures, addressing, naming, forwarding, routing, communication reliability, the client-server model, and web and email protocols. The applications of this course are to design, implement and maintain a basic computer networks.

1. Bit Stuffing

1.1 Bit Stuffing in Computer Network

The data link layer is responsible for something called Framing, which is the division of stream of bits from network layer into manageable units (called frames). Frames could be of fixed size or variable size. In variable-size framing, we need a way to define the end of the frame and the beginning of the next frame.

Bit stuffing is the insertion of non-information bits into data. Note that stuffed bits should not be confused with overhead bits. Overhead bits are non-data bits that are necessary for transmission (usually as part of headers, checksums etc.).

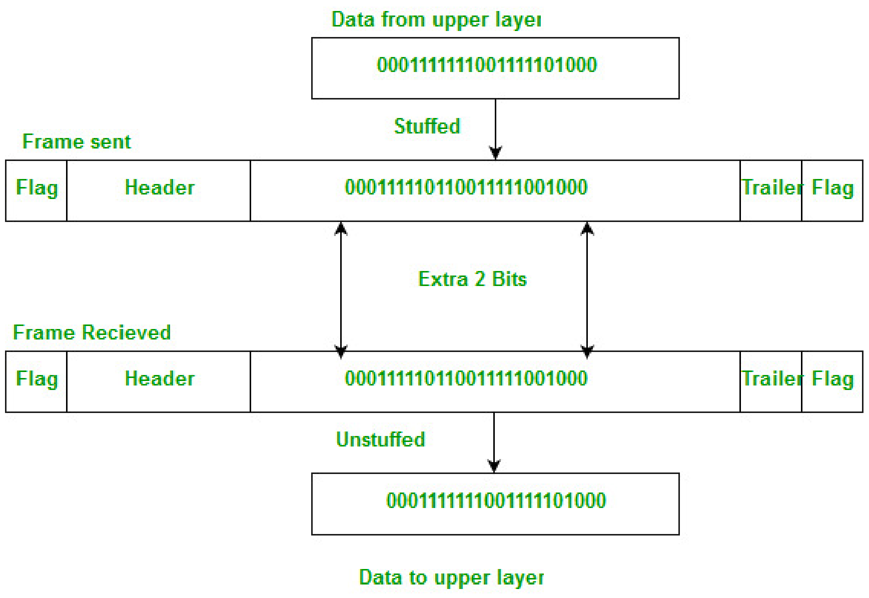


Fig-1: (Bit Stuffing)

1.2 Bit Stuffing in python

Implement the data link layer framing methods such as bit stuffing by using a python code.

Hint:

1. Initialize a variable to count consecutive 1s.
2. Iterate through the data and append each bit to the result.
3. Check for 5 consecutive 1s and stuff a zero after them.
4. Add start and end delimiters to the data.
5. For bit unstuffing, remove the start and end delimiters.
6. Detect 5 consecutive 1s followed by a stuffed zero and remove the stuffed zero.

```
def bit_stuffing(data):
    stuffed_data = "01111110"  # Start delimiter
    #write your code here
def bit_unstuffing(stuffed_data):
    data = ""
    count = 0
    # Remove start delimiter
    stuffed_data = stuffed_data[8:]
    # Remove end delimiter
    stuffed_data = stuffed_data[:-8]
    #write your code here
# Example usage:
original_data = "1101111111111111110111"
stuffed_data = bit_stuffing(original_data)
unstuffed_data = bit_unstuffing(stuffed_data)

print(f"Original Data: {original_data}")
print(f"Stuffed Data: {stuffed_data}")
print(f"Unstuffed Data: {unstuffed_data}")
```

Try:
Change the stuffed data and original data and check for output.

2. Character stuffing

2.1 Character stuffing in computer Network

Character stuffing is also known as byte stuffing or character-oriented framing and is same as that of bit stuffing but byte stuffing operates on bytes whereas bit stuffing operates on bits. In byte stuffing, special byte that is basically known as ESC (Escape Character) that has predefined pattern is generally added to data section of the data stream or frame when there is message or character that has same pattern as that of flag byte.

But receiver removes this ESC and keeps data part that causes some problems or issues. In simple words, we can say that character stuffing is addition of 1 additional byte if there is presence of ESC or flag in text.

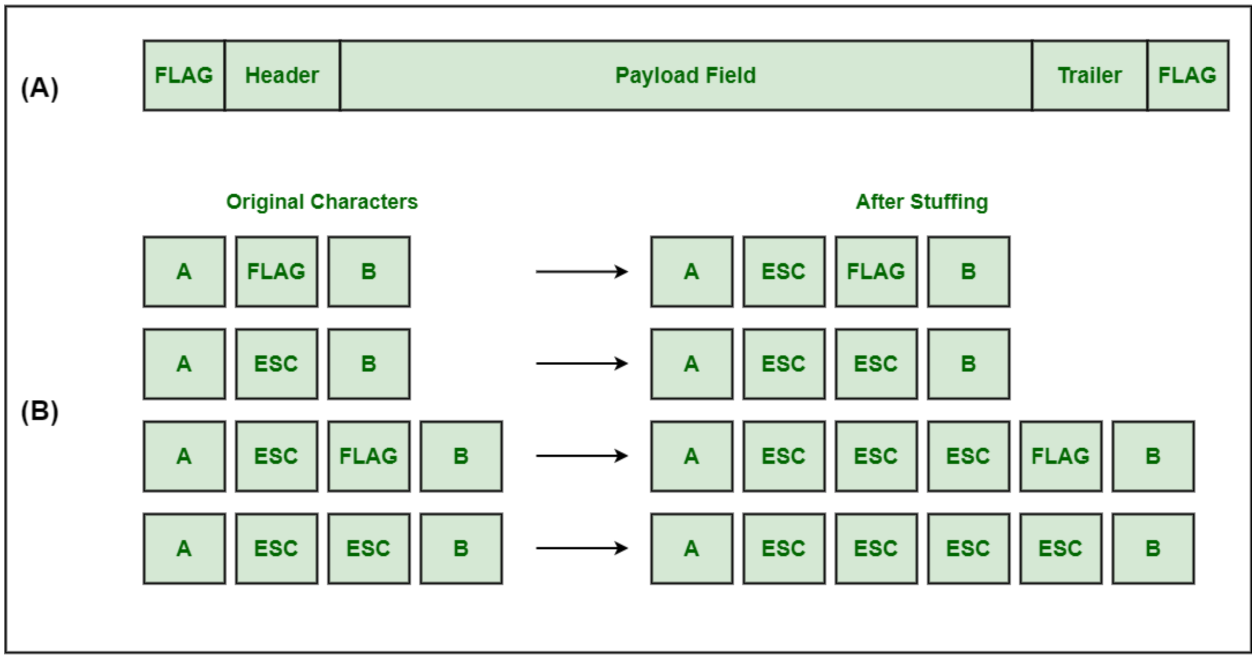


Fig 2. CHARACTER STUFFING

(a) A frame delimited by flag bytes.

(b) Four examples of byte sequences before and after byte stuffing

2.2 Character stuffing in Python

Implement the data link layer framing method such as character stuffing using a python code

.**Hint:**

1. Use a FLAG sequence to denote the start and end of the frame.
2. Choose an ESCAPE_CHAR sequence that cannot be confused with the FLAG sequence.
3. Iterate through the original data, and whenever you encounter a certain number of consecutive '1's (in this case, 5), insert the ESCAPE_CHAR before the next '1'.

```
def character_stuffing(data):
    FLAG = '01111110'
    ESCAPE_CHAR = '11110'
    #Write your code

# Example usage:
   original_data = '0111101111101111011110'

stuffed_data = character_stuffing(original_data)

print(f'Original Data: {original_data}')
print(f'Stuffed Data: {stuffed_data}')
```

Try: Use Escape_char ='10111001' and Flag='11001110' and observe output.

2.3 Character destuffing in Python

Write a program called to implement the data link layer framing method such as character stuffing using python code.

Hint:

1. Iterate through the stuffed data, and whenever you encounter five consecutives '1's, skip the next '0' as it is the ESCAPE_CHAR.

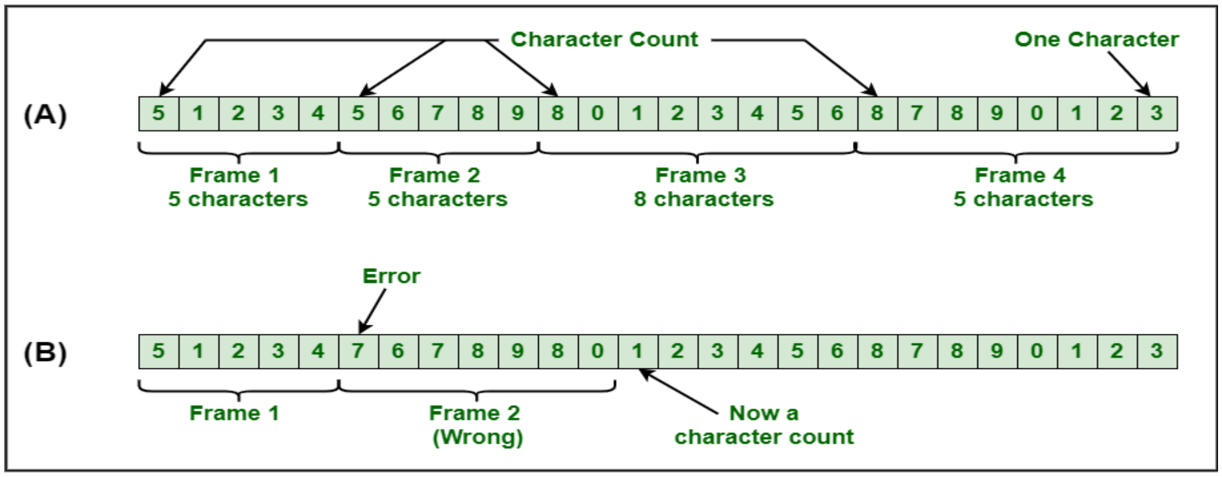
```
def character_destuffing(stuffed_data):
    FLAG = '01111110'
    ESCAPE_CHAR = '11110'
    #write your code

# Example usage:
destuffed_data = character_destuffing(stuffed_data)
print(f'Destuffed Data: {destuffed_data}')
```

Try:
Change Escape char and flag check the change in results.

3. Data Count

3.1 Character Count

This method is rarely used and is generally required to count total number of characters that are present in frame. This is be done by using field in header. Character count method ensures data link layer at the receiver or destination about total number of characters that follow, and about where the frame ends. There is disadvantage also of using this metchod i.e., if anyhow character count is disturbed or distorted by an



A Character Stream
(A) Without Errors
(B) With one Error

error occurring during transmission, then destination or receiver might lose synchronization. The destination or receiver might also be not able to locate or identify beginning of next frame.

3.2 Basic Character Count

Develop a program to implement data link layer framing method data count using a python code.

Hint:

1. Iterate through each character in the text, use `isalpha()` to check if it's an alphabet character, and update the count in a dictionary.

```
def count_characters(text):
    character_count = {}
    #write your code

# Example usage:
text = "Hello, World!"
result = count_characters(text)
print(result)
```

Try:
Try without using isaplha() function.

3.3 Case-Insensitive Character Count

Write a python program to implement data link layer framing method data count.

Hint:

1. Convert each character to lowercase (or uppercase) before updating the count to make the counting case insensitive.

```
def count_characters_case_insensitive(text):
    character_count = {}
    #write your code

# Example usage:
text = "Hello, World!"
result = count_characters_case_insensitive(text)
print(result)
```

Try:
Use the function. upper () or lower() and convert them.

3.4 Word Character Count

Develop a python program to implement data link layer framing method data count.

Hint:

1. Use `isalnum()` to check if a character is an alphanumeric character (letter or digit).

```
def count_word_characters(text):
    character_count = {}
    #write your code
```

```
# Example usage:
text = "Hello, 123 World!"
result = count_word_characters(text)
print(result)
```

Try:
Iterate loop and eliminate special characters and count the character.

3.5 Character Frequency Percentage

Generate a python code to implement data link layer framing method data count.

Hint:

1. Calculate the total number of characters in the text and then calculate the percentage frequency of each character.

```
def character_frequency_percentage(text):
    total_characters = len(text)
    #write your code

# Example usage:
text = "Hello, World!"
result = character_frequency_percentage(text)
print(result)
```

Try:
Use dictionary and store the elements and count the frequency.

4. CRC polynomials

4.1 Cyclic Redundancy Check

CRC or Cyclic Redundancy Check is a method of detecting accidental changes/errors in the communication channel.

CRC uses Generator Polynomial which is available on both sender and receiver side. An example generator polynomial is of the form like x3 + x + 1. This generator polynomial represents key 1011. Another example is x2 + 1 that represents key 101.

n : Number of bits in data to be sent from sender side.

k : Number of bits in the key obtained from generator polynomial.

4.2 CRC at server side

Generate a python code to implement on a data set characters the three CRC polynomials-CRC 12 , CRC-16 , CRC CCIP.

Hint:

1. Perform modulo-2 division again and if the remainder is 0, then there are no errors.

```
# First of all import the socket library
import socket

def xor(a, b):
       # initialize result
       result = []

       # Traverse all bits, if bits are
       # same, then XOR is 0, else 1
       #write your code

# Performs Modulo-2 division
def mod2div(divident, divisor):

       # Number of bits to be XORed at a time.
       pick = len(divisor)

       # Slicing the divident to appropriate
       # length for particular step
       #write Your code

       # For the last n bits, we have to carry it out
       # normally as increased value of pick will cause
       # Index Out of Bounds.

# Function used at the receiver side to decode
# data received by sender
   def decodeData(data, key):

       #write your code

# Creating Socket
s = socket.socket()
print("Socket successfully created")

# reserve a port on your computer in our
# case it is 12345 but it can be anything
port = 12345

s.bind(('', port))
print("socket binded to %s" % (port))
# put the socket into listening mode
s.listen(5)
print("socket is listening")

while True:
       # Establish connection with client.
       c, addr = s.accept()
       print('Got connection from', addr)

       # Get data from client
       data = c.recv(1024)

       print("Received encoded data in binary format :", data.decode())

       if not data:
              break
       key = "1001"

       ans = decodeData(data, key)
       print("Remainder after decoding is->"+ans)

       # If remainder is all zeros then no error occured
       temp = "0" * (len(key) - 1)
       if ans == temp:
              c.sendto(("THANK you Data ->"+data.decode() +
                           " Received No error FOUND").encode(), ('127.0.0.1',
12345))
       else:
              c.sendto(("Error in data").encode(), ('127.0.0.1', 12345))

       c.close()
```

Try:
Change the Ip to 127.2.2.1 and port to 8080.

4.3 CRC at client side

Hint:

1. The binary data is first augmented by adding k-1 zeros in the end of the data
2. Use modulo-2 binary division to divide binary data by the key and store remainder of division.
3. Append the remainder at the end of the data to form the encoded data and send the same

```
# Import socket module
import socket

def xor(a, b):

    # initialize result
    result = []

    # Traverse all bits if bits are
    # same, then XOR is 0, else 1
    # Write your code
# Performs Modulo-2 division
def mod2div(divident, divisor):

    # Number of bits to be XORed at a time.
    pick = len(divisor)

    # Slicing the divident to appropriate
    # length for step
    #Write your code

    # For the last n bits, we have to carry it out
    # normally as increased value of pick will cause
    # Index Out of Bounds.
    #write your code

# Function used at the sender side to encode
# data by appending remainder of modular division
# at the end of data.
def encodeData(data, key):

    #write your code

# Create a socket object
s = socket.socket()

# Define the port on which you want to connect
port = 12345

# connect to the server on local computer
s.connect(('127.0.0.1', port))

# Send data to server 'Hello world'

## s.sendall('Hello World')

input_string = input("Enter data you want to send->")
#print("Enter data you want to send->")
#input_string = input()
#print(input_string)
#s.sendall(input_string)
data =(''.join(format(ord(x), 'b') for x in input_string))
print("Entered data in binary format :",data)
key = "1001"

ans = encodeData(data,key)
print("Encoded data to be sent to server in binary format :",ans)
s.sendto(ans.encode(),('127.0.0.1', 12345))
```

```
 # receive data from the server
print("Received feedback from server :",s.recv(1024).decode())

# close the connection
s.close()
```

Try:

Check at every IP and ports and listen to data being sent.

5. Shortest path

5.1 shortest path in graph

Shortest path algorithms are a family of algorithms designed to solve the shortest path problem. The shortest path problem is something most people have some intuitive familiarity with: given two points, A and B, what is the shortest path between them? In computer science, however, the shortest path problem can take different forms and so different algorithms are needed to be able to solve them all.

For simplicity and generality, shortest path algorithms typically operate on some input graph, \(G\). This graph is made up of a set of vertices, \(V\), and edges, \(E\), that connect them. If the edges have weights, the graph is called a weighted graph. Sometimes these edges are bidirectional, and the graph is called undirected. Sometimes there can even be cycles in the graph. Each of these subtle differences are what makes one algorithm work better than another for certain graph type.

5.2 Shortest path by Dijkstra's algorithm

Create a python program to implement Dijkstra's algorithm to compute the shortest path through a given path.

Hints:

1. Initialize Distance and Previous
2. Set Initial Values
3. Explore Neighbors
4. Mark Visited Nodes
5. Select Next Node
6. Repeat Until Completion
7. Construct Shortest Path

```
# Python program for Dijkstra's single
# source shortest path algorithm. The program is
# for adjacency matrix representation of the graph
class Graph():

      def __init__(self, vertices):
            self.V = vertices
            self.graph = [[0 for column in range(vertices)]
                              for row in range(vertices)]

      def printSolution(self, dist):
            print("Vertex \t Distance from Source")
            #write your code

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minDistance(self, dist, sptSet):

            # Initialize minimum distance for next node
            min = 1e7

            # Search not nearest vertex not in the
            # shortest path tree
            #write your code

    # Function that implements Dijkstra's single source
    # shortest path algorithm for a graph represented
    # using adjacency matrix representation
    def dijkstra(self, src):

            dist = [1e7] * self.V
            dist[src] = 0
            sptSet = [False] * self.V
            # Pick the minimum distance vertex from
            # the set of vertices not yet processed.
            # u is always equal to src in first iteration
            #write your code

# Driver program
g = Graph(9)
g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
            [4, 0, 8, 0, 0, 0, 0, 11, 0],
            [0, 8, 0, 7, 0, 4, 0, 0, 2],
            [0, 0, 7, 0, 9, 14, 0, 0, 0],
            [0, 0, 0, 9, 0, 10, 0, 0, 0],
            [0, 0, 4, 14, 10, 0, 2, 0, 0],
            [0, 0, 0, 0, 0, 2, 0, 1, 6],
            [8, 11, 0, 0, 0, 0, 1, 0, 7],
            [0, 0, 2, 0, 0, 0, 6, 7, 0]
            ]

g.dijkstra(0)
```

Try:

Use two lists one for visited and other for not visited and then measure the distance.

5.3 Basic Floyd-Warshall Implementation

Modify a python program to implement Floyd-War shall algorithm to compute the shortest path through a given path.

Hints:

1. Use a 2D matrix to represent distances between all pairs of nodes.
2. Initialize the matrix with direct edges and update it iteratively using the Floyd-Warshall algorithm.

```
def floyd_warshall(graph):
    nodes = list(graph.keys())
    distance = {i: {j: float('inf') for j in nodes} for i in nodes}

    # Initialize distances with direct edges
    #write Your code

     # Floyd-Warshall algorithm
     #write your code

# Example usage:
graph = {
    'A': {'B': 2, 'C': 6},
    'B': {'A': 2, 'C': 3},
    'C': {'A': 6, 'B': 3}
}

result = floyd_warshall(graph)
print("Shortest distances between all pairs:")
for node in result:
    print(f"{node}: {result[node]}")
```

Try:

Use while loop inside the floyd-warshall algorithm and observe the changes.

5.4 Path Reconstruction

Insert a python code to implement Floyd-War shall algorithm to compute the shortest path through a given path.

Hints:

1. Maintain an additional matrix to track the next node in the shortest path for each pair of nodes.
2. Use the matrix to reconstruct the shortest path between any two nodes.

```
def floyd_warshall_with_path(graph):
    nodes = list(graph.keys())
    distance = {i: {j: float('inf') for j in nodes} for i in nodes}
    next_node = {i: {j: None for j in nodes} for i in nodes}

    # Initialize distances with direct edges
    #write your code

    # Floyd-Warshall algorithm
    #write your code

def reconstruct_path(start, end, next_node):
    #write form

# Example usage:
```

```
graph = {
    'A': {'B': 2, 'C': 6},
    'B': {'A': 2, 'C': 3},
    'C': {'A': 6, 'B': 3}
}

distances, next_node = floyd_warshall_with_path(graph)
start_node, end_node = 'A', 'C'
shortest_path = reconstruct_path(start_node, end_node, next_node)
print(f"Shortest path from {start_node} to {end_node}: {shortest_path}")
```

Try:
Use linked list for storing the path of the graph.

6. Subnet of graph

6.1 Distance Vector Routing (DVR) Protocol

A distance-vector routing (DVR) protocol requires that a router inform its neighbors of topology changes periodically. Historically known as the old ARPANET routing algorithm (or known as Bellman-Ford algorithm).

Bellman Ford Basics – Each router maintains a Distance Vector table containing the distance between itself and ALL possible destination nodes. Distances, based on a chosen metric, are computed using information from the neighbors' distance vectors.

Information kept by DV router -

- Each router has an ID
- Associated with each link connected to a router,
  there is a link cost (static or dynamic).
- Intermediate hops

Distance Vector Table Initialization -

- Distance to itself = 0
- Distance to ALL other routers = infinity number.

6.2 Distance Vector Algorithm

1. A router transmits its distance vector to each of its neighbors in a routing packet.
2. Each router receives and saves the most recently received distance vector from each of its neighbors.
3. A router recalculates its distance vector when:
    - It receives a distance vector from a neighbor containing different information than before.
    - It discovers that a link to a neighbor has gone down.

The DV calculation is based on minimizing the cost to each destination.

Generate a python code to take an example subnet graph with weights indicating delay between nodes. Now obtain routing table art each node using distance vector routing algorithm.

Hint:

1. Initialize distances from the source to all vertices as infinity and set the distance to the source vertex itself as 0.
2. Iterate through all edges multiple times (V-1 times, where V is the number of vertices). In each iteration, relax all the edges.
3. For each edge (u, v), if the distance to vertex u plus the weight of the edge (u, v) is less than the current distance to vertex v, update the distance to v.
4. After V-1 iterations, check for negative cycles. If any distance can still be reduced, then there is a negative cycle in the graph.

```
# Python3 program for Bellman-Ford's single source
# shortest path algorithm.
# Class to represent a graph
class Graph:

	def __init__(self, vertices):
		self.V = vertices # No. of vertices
		self.graph = []

	# function to add an edge to graph
	def addEdge(self, u, v, w):
		self.graph.append([u, v, w])

	# utility function used to print the solution
	def printArr(self, dist):
		print("Vertex Distance from Source")
		#write Your code

	# The main function that finds shortest distances from src to
	# all other vertices using Bellman-Ford algorithm. The function
	# also detects negative weight cycle
	def BellmanFord(self, src):

		#write Your code

g = Graph(5)
g.addEdge(0, 1, -1)
g.addEdge(0, 2, 4)
g.addEdge(1, 2, 3)
g.addEdge(1, 3, 2)
g.addEdge(1, 4, 2)
g.addEdge(3, 2, 5)
g.addEdge(3, 1, 1)
g.addEdge(4, 3, -3)

# Print the solution
g.BellmanFord(0)
```

Try:

Change the graph source and add more edges to the graph.

6.3 Subnet Calculator

Develop a Python program that takes an IP address and subnet mask as input and calculates the network address, broadcast address, and available host addresses for the given subnet.

Hint:
1. Validate the input to ensure that the entered IP address and subnet mask are in a valid format.
2. Use the bitwise AND operation to extract the network address by combining the IP address and subnet mask.
3. Use the bitwise OR operation to calculate the broadcast address by combining the network address with the inverted subnet mask.
4. Subtract 2 from the total number of addresses in the subnet (network and broadcast) to get the count of available host addresses.
5. Print the calculated network address, broadcast address, and the range of available host addresses.

```
def subnet_calculator(ip_address, subnet_mask):
    # Input validation
    #use try except
    #write your code

    # Extract network address
    network_address = [ip & mask for ip, mask in zip(ip_parts, subnet_parts)]

    # Calculate broadcast address
    # Calculate available host addresses
    #write your code

    # Output results
    print("Network Address:", '.'.join(map(str, network_address)))
    print("Broadcast Address:", '.'.join(map(str, broadcast_address)))
    print("Available Host Addresses:", available_host_addresses)

# Example usage:
subnet_calculator("192.168.1.10", "255.255.255.0")
```

Try:
Use the libraries in python and then find out the subnet details.

7. Broadcast Tree

7.1 Broadcast Tree in computer networks

A broadcast tree refers to a network topology designed to efficiently forward broadcast or multicast traffic to all nodes in a network. Broadcast and multicast are communication paradigms used to send data to multiple recipients simultaneously.

Brief overview of how a broadcast tree works:

Root Node: The broadcast tree has a root node from which the broadcast or multicast traffic originates. This node initiates the transmission of the message.

Tree Structure: The network is organized in a tree-like structure, where the root node is at the top, and branches extend down to the leaf nodes. Each branch represents a path for the broadcast or multicast traffic to follow.
Optimized Paths: The tree is designed to ensure that the broadcast or multicast traffic reaches every node in the network while minimizing redundancy. This optimization is achieved by selecting the most efficient paths for the transmission.
Tree Construction: Broadcast trees can be constructed using various algorithms, such as the Minimum Spanning Tree (MST) algorithm or the Reverse Path Forwarding (RPF) algorithm. These algorithms help create a tree that spans the entire network with minimum cost.
Switches/Routers Configuration: Network devices such as switches or routers are configured to forward broadcast or multicast traffic based on the established broadcast tree. This ensures that the traffic follows the optimized paths through the network.
Efficiency: By using a broadcast tree, the network reduces unnecessary traffic and prevents loops that could occur in more straightforward broadcast scenarios. This improves the efficiency of the network and reduces the risk of congestion.
Broadcast trees are particularly useful in scenarios where efficient distribution of broadcast or multicast traffic is essential, such as in video streaming applications, online gaming, or other scenarios where simultaneous communication to multiple nodes is required.

7.2 Basic Broadcast Tree Construction

Modify a Python program for Basic Broadcast Tree construction.

Hint:

1. Create a class to represent nodes in the tree, where each node has a name and a list of children.
2. Connect nodes to form a tree structure where the root is the node initiating the broadcast.

```
class Node:
    def __init__(self, name):
        self.name = name
        self.children = []

def build_broadcast_tree():
    # Create nodes
    # Connect nodes to form a broadcast tree
    # Return the root of the broadcast tree
    #write your code here

# Example usage:
root_node = build_broadcast_tree()
```

Try:
Build your Tree with linked list.

7.3 Broadcast Message Propagation

Create a Python program for Broadcast Message Propagation.

Hint:

1. Implement a recursive function that starts from the root node and propagates the broadcast message to all its children.

```
def broadcast_message(node, message):
    #write your code

# Example usage:
broadcast_message(root_node, "Broadcast: Hello, nodes!")
```

Try:

Implement the recursive function using for loop.

7.4 Dynamic Broadcast Tree

Modify the below code to insert a Python Code for Dynamic Broadcast Tree.

Hint:

1. Extend the node class to allow dynamic addition of children, making it easier to adapt the broadcast tree structure during runtime.

```
class DynamicNode:
    def __init__(self, name):
        #write your code

# Example usage:
dynamic_root = DynamicNode('A')
dynamic_root.add_child(DynamicNode('B'))
dynamic_root.children[0].add_child(DynamicNode('C'))
```

Try:

Use linked list for the above example.

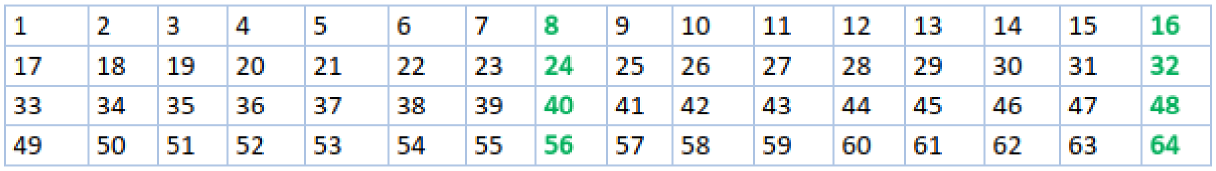
8. Encryption

8.1 DES Algorithm

Data Encryption Standard (DES) is a block cipher with a 56-bit key length that has played a significant role in data security. Data encryption standard (DES) has been found vulnerable to very powerful attacks therefore, the popularity of DES has been found slightly on the decline. DES is a block cipher and encrypts data in blocks of size of 64 bits each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is 56 bits.

The basic idea is shown below:

We have mentioned that DES uses a 56-bit key. Actually, The initial key consists of 64 bits. However, before the DES process even starts, every 8th bit of the key is discarded to produce a 56-bit key. That is bit positions 8, 16, 24, 32, 40, 48, 56, and 64 are discarded.



1	2	3	4	5	6	7	**8**	9	10	11	12	13	14	15	**16**
17	18	19	20	21	22	23	**24**	25	26	27	28	29	30	31	**32**
33	34	35	36	37	38	39	**40**	41	42	43	44	45	46	47	**48**
49	50	51	52	53	54	55	**56**	57	58	59	60	61	62	63	**64**

Figure - discording of every 8th bit of original key

Fig-4

Thus, the discarding of every 8th bit of the key produces a 56-bit key from the original 64-bit key.

DES is based on the two fundamental attributes of cryptography: substitution (also called confusion) and transposition (also called diffusion). DES consists of 16 steps, each of which is called a round. Each round performs the steps of substitution and transposition. Let us now discuss the broad-level steps in DES.

- In the first step, the 64-bit plain text block is handed over to an initial Permutation (IP) function.
- The initial permutation is performed on plain text.
- Next, the initial permutation (IP) produces two halves of the permuted block; saying Left Plain Text (LPT) and Right Plain Text (RPT).
- Now each LPT and RPT go through 16 rounds of the encryption process.
- In the end, LPT and RPT are rejoined and a Final Permutation (FP) is performed on the combined block
- The result of this process produces 64-bit ciphertext.

8.2 Key Generation

Extend a python program to implement Key Generation in DES algorithm.

Hint:

1. Implement the key permutation function for generating round keys.
2. Understand the left shifts and permutations involved in key generation.
3. Create a function to generate the 16 subkeys used in the encryption process.

```
def generate_keys(key):
    # Perform key permutation and generate 16 subkeys
    # Write your code
    return round_keys
```

Try:

Use for loop instead of while loop.

8.3 Initial and Final Permutation

Illustrate a python program to implement Initial and Final Permutation in DES algorithm.

Hint:

1. Implement the initial and final permutation using predefined matrices.
2. Learn how to manipulate bits to perform the permutations.

```
def initial_permutation(data):
    # Perform the initial permutation
    # Write your code
    return permuted_data

def final_permutation(data):
    # Perform the final permutation
    # Write your code
    return permuted_data
```

Try:

Take 16 bit keys instead of 8 bit.

8.4 F-function (Feistel Function)

Implement a python program to create F-function (Feistel Function) in DES algorithm.

Hint:

1. Implement the expansion and permutation steps within the F-function.
2. Understand the substitution step using S-boxes.
3. Implement XOR operations and the final permutation (P-box).

```
def feistel_function(right_half, round_key):
    # Expansion and permutation
    # Write your code

    # XOR with round key
    # Write your code

    # Substitution using S-boxes
    # Write your code

    # Permutation (P-box)
    # Write your code

    return result
```

Try:

Use xor operator instead of performing the manipulations.

8.5 DES Encryption

Develop a python code to implement DES algorithm using Generate keys.

Hint:

Implement the initial permutation and split the input into two halves.

1. Iterate through 16 rounds, applying the F-function and swapping halves.
2. Perform the final permutation before returning the encrypted data.

```
def des_encrypt(data, key):
    # Initial permutation and split
    # Write your code
    # Generate round keys
    round_keys = generate_keys(key)

    # 16 rounds of encryption
    for round_num in range(16):
        # Write your code
    # Final permutation
    # Write your code

    return encrypted_data
```

Try:
Use while loop for 16 rounds of encryption.

9. Decryption

9.1 DES Decryption algorithm

The steps involved in the steps for data decryption are:

1. The order of the 16 48-bit keys is reversed such that key 16 becomes key 1, and so on.

2. The steps for encryption are applied to the ciphertext.

9.2 DES Decryption Function

Generate a python program to implement to break the above DES coding using the cipher keys.

Hint:
1. Use a library like PyCryptodome to simplify DES encryption and decryption.
2. Ensure that the key used for decryption is the same as the one used for encryption.

```
from Crypto.Cipher import DES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad

def des_decrypt(ciphertext, key):
    #write your code

# Example usage:
key = get_random_bytes(8)
plaintext = b'This is a secret'
cipher = DES.new(key, DES.MODE_ECB)
ciphertext = cipher.encrypt(pad(plaintext, DES.block_size))
decrypted_text = des_decrypt(ciphertext, key)
print(f"Original: {plaintext}")
print(f"Ciphertext: {ciphertext}")
print(f"Decrypted: {decrypted_text.decode('utf-8')}")
```

Try:
Change the text to "Hi Hello" and key to 16 bits.

9.3 File Decryption

Write a python program to implement File Decryption by DES coding.

Hint:
1. Read the encrypted data from the input file and use the `Crypto.Cipher.DES` module for decryption.
2. Write the decrypted data back to a new file.

```
from Crypto.Cipher import DES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import unpad

def des_decrypt_file(input_file, output_file, key):
    #write your code

# Example usage:
key = get_random_bytes(8)
input_filename = 'encrypted_file.txt'
output_filename = 'decrypted_file.txt'

# Assume 'encrypted_file.txt' contains the encrypted data
des_decrypt_file(input_filename, output_filename, key)
```

Try:
Change the encrypted file and observe the results.

10. RSA Encryption and Decryption

10.1 RSA Algorithm

RSA algorithm is an asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e. Public Key and Private Key. As the name describes that the Public Key is given to everyone and the Private key is kept private.

An example of asymmetric cryptography:

A client (for example browser) sends its public key to the server and requests some data.

The server encrypts the data using the client's public key and sends the encrypted data.

The client receives this data and decrypts it.

Since this is asymmetric, nobody else except the browser can decrypt the data even if a third party has the public key of the browser.

The idea! The idea of RSA is based on the fact that it is difficult to factorize a large integer. The public key consists of two numbers where one number is a multiplication of two large prime numbers. And private key is also derived from the same two prime numbers. So if somebody can factorize the large number, the private key is compromised. Therefore encryption strength totally lies on the key size and if we double or triple the key size, the strength of encryption increases exponentially. RSA keys can be typically 1024 or 2048 bits long, but experts believe that 1024-bit keys could be broken in the near future. But till now it seems to be an infeasible task.

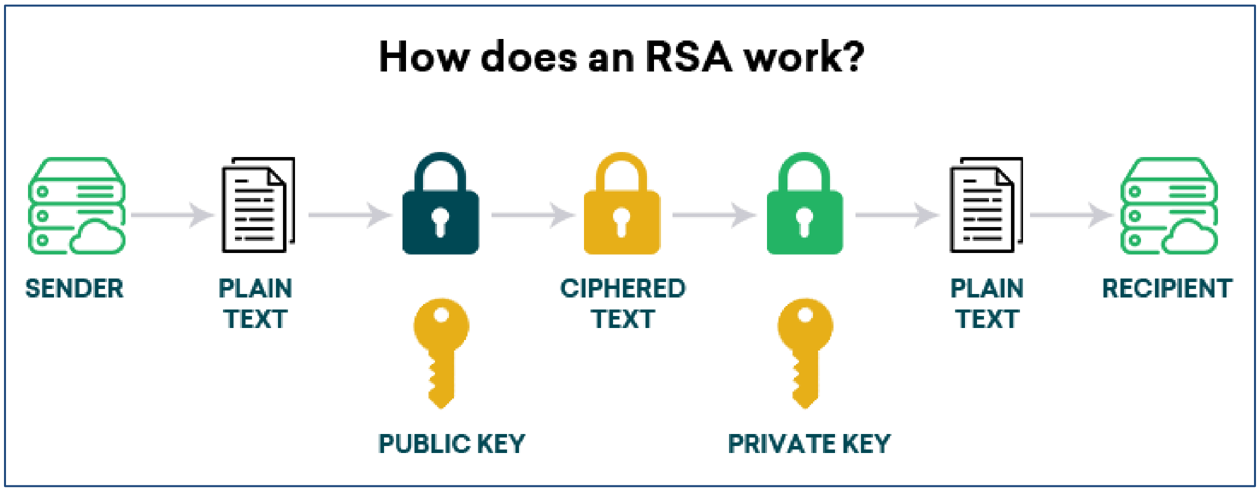


Fig-4(Working of RSA Algorithm)

10.2 Key Generation

Implement a python code to include Key Generation in RSA using Public and private keys.

Hint:

1. Utilize a cryptography library, such as PyCryptodome, to generate RSA key pairs.
2. The key length (2048 bits in this example) determines the strength of the keys.

```
from Crypto.PublicKey import RSA

def generate_key_pair():
    #write your code

# Example usage:
public_key, private_key = generate_key_pair()
print("Public Key:")
print(public_key.decode('utf-8'))
print("Private Key:")
print(private_key.decode('utf-8'))
```

Try:

Use the private key in another format other than utf-8.

10.3 Encryption

Create a python program to implement Encryption in RSA using public key.

Hint:

1. Use the public key to initialize the RSA object.

2. PKCS#1 OAEP is a commonly used padding scheme for RSA encryption.
3. The `encrypt` method converts the message to bytes and encrypts it.

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
def encrypt_message(message, public_key):
    #write your code

# Example usage:
message = "Hello, RSA!"
public_key = "<Public key goes here>"
encrypted_message = encrypt_message(message, public_key)
print("Encrypted Message:")
print(encrypted_message.hex())
```

Try:
Generate your own public key and length greater than 16 bits.

10.4 Decryption

Create a python program to implement Decryption in RSA using Private key.

Hint:
1. Use the private key to initialize the RSA object.
2. The `decrypt` method decrypts the ciphertext and converts it back to a string.

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

def decrypt_message(ciphertext, private_key):
    #write your code

# Example usage:
ciphertext = b"<Encrypted message goes here>"
private_key = "<Private key goes here>"
decrypted_message = decrypt_message(ciphertext, private_key)
print("Decrypted Message:")
print(decrypted_message)
```

Try:
Give cipher text encrypted with 64 bits and give the generate the same private key.

11. Packets

11.1 packet loss

Packet Loss Definition
Packets are small units of data transmitted over a network from a particular source to a destination. Packet loss occurs when a network packet fails to reach its expected destination, resulting in information loss.

Causes of packet loss

Packet loss across network connections is common and may be caused by any of the following reasons:

- **Network Congestion**: One of the primary reasons for packet loss and implies a situation where network traffic is at its peak. With excessive traffic, it becomes critical for each packet to wait until its delivery. However, packets might also get discarded if the network connection reaches its maximum capacity and can't accommodate any more packets.
- **Network Hardware Bottlenecks**: Faulty network hardware or components, such as routers and network switches, may drastically impede network traffic speed. Usually, growing companies with expanding scale witness packet loss due to connectivity issues and experience lag because of outdated hardware. It becomes imperative to update existing hardware to effectively handle growing throughput.
- **Software Bugs**: In the absence of exhaustive software testing or unchecked updates, bugs might prevail and disrupt network performance. A system reboot, requisite software update, or a patch can help fix such bugs.
- **Security Threats and Attack**: A security breach is also a possibility to tamper with a network connection, resulting in packet loss. Some prevalent security attacks include packet drop attacks or denial-of-service (DoS) attacks.

Effects of packet loss

In today's hyper-connected world, businesses strive for an effective network that supports seamless interactions and business operations. Any instance of packet loss hampers the overall experience of communicating via email, accessing business-critical applications, or any other operational task.

Packet loss eventually hinders network performance and overall business in the following ways:

- **Low Network Throughput:** In case of packet loss, some data never reaches the intended destination, lowering the throughput of a network connection.
- **High Latency:** Additional time for retransmitting the lost packets to ensure complete information transmission increases network latency.
- **Poor Application Experience:** Packet loss impacts business applications and disrupts the end-user experience. Mission-critical applications and apps based on real-time packet processing witness drastic impacts of packet loss.
- **Increased Costs:** Packet loss may also add to operational expenses as businesses need to spend on additional IT tools to overcome the network lag. Organizations also face productivity challenges due to the delay in tasks.

11.2 Simple Packet Drop Simulation

Modify a python program for Simple Packet Drop Simulation. (`probability`).

Hint:

1. Use the `random` module to generate a random number between 0 and 1. If the generated number is less than the specified probability, simulate a packet drop.

```
import random
def simulate_packet_drop(probability):
    #write your code

# Example usage:
simulate_packet_drop(0.2)  # Simulate a 20% probability of packet drop
```

Try:

Change the probability to 40% and observe the change in packet drop.

11.3 Packet Drop with Threshold

Generate a python program for Packet Drop with Threshold for probability.

Hint:

1. Introduce an additional threshold probability. If a packet drop occurs, check if another random number is less than the threshold to decide whether to drop the packet.

```
import random
def simulate_packet_drop_with_threshold(probability, threshold):
    #write your code

# Example usage:
simulate_packet_drop_with_threshold(0.3, 0.1)  # Simulate a 30% probability of
packet drop with a 10% threshold
```

Try:

Decrease the probability of packet drop to 15% and increase the threshold to 20%.

11.4 Adaptive Packet Drop Simulation

Write a python program for Adaptive Packet Drop Simulation.

Hint:

1. Create a class `AdaptivePacketDropper` with methods to simulate packet drops. Adjust the drop probability based on both increase and decrease rates.

```
import random

class AdaptivePacketDropper:
    def __init__(self, initial_probability, increase_rate, decrease_rate):
        #write your code

    def simulate_packet_drop(self):
        #write your code

# Example usage:
dropper = AdaptivePacketDropper(0.1, 0.02, 0.01)
for _ in range(10):
    dropper.simulate_packet_drop()
```

Try:

Use while loop instead of for loop.

11.5 Network Packet Drop Simulator

Implement a python program for Network Packet Drop Simulator by creating a class.

Hint:

1. Create a class `NetworkPacketDropSimulator` that simulates packet transmission and acknowledgment. Keep track of the network capacity and the current load.

```
import random

class NetworkPacketDropSimulator:
    def __init__(self, network_capacity):
        self.network_capacity = network_capacity
        self.current_load = 0

    def send_packet(self):
        #Write your code

    def receive_acknowledgment(self):
        #Write your code

# Example usage:
simulator = NetworkPacketDropSimulator(5)
simulator.send_packet()
simulator.send_packet()
simulator.receive_acknowledgment()
simulator.send_packet()
```

Try: Include the network capacity to maximum and then check for the simulation.

12. UDP AND TCP

12.1 Basic UDP Echo Server

Insert a python program code for UDP echo server side.

Hint:

1. Use ‘socket.AF_INET’ for IPv4 and ‘socket.SOCK_DGRA’ for UDP.
2. Bind the server socket to a specific address and port.
3. Use a loop to continuously listen for incoming messages.
4. Receive data and client address using ‘recvfrom’.
5. Print the received message and send it back to the client using ‘sendto’.

```
import socket

def udp_server(port):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    #Write your code

# Example usage:
udp_server(5555)
```

Try:
Change the port to universal 8080.

12.2 Basic UDP Echo Client

Develop a python program for UDP echo client side.(Socket programme)

Hint:

1. Create a UDP socket on the client using `socket.AF_INET` and `socket.SOCK_DGRAM`.
2. Specify the server's address and port.
3. Send a message to the server using `sendto`.
4. Receive the response from the server using `recvfrom`.
5. Print the received message.

```
import socket

def udp_client(server_host, server_port, message):
    #Write your code

# Example usage:
udp_client('localhost', 5555, 'Hello, UDP Server!')
```

Try:
Establish a connection in between client to server.

12.3 Implementing a Simple File Transfer

Develop a python program for implementing a simple file transfer server side.

Hint:
Similar setup as the echo server, but open a file for writing.

1. Continuously receive data from the client and write it to the file.
2. Break the loop when no more data is received.
3. Close the file after the transfer is complete.

```
import socket

def udp_file_server(port, filename):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    #Write your Code

# Example usage:
udp_file_server(5555, 'received_file.txt')
```

Try:
Encrypt the received_file.txt and try to send the file.

12.4 Implementing a Simple File Transfer

Generate a python program for implementing a simple file transfer client side.

Hint:
1. Similar setup as the echo client, but open a file for reading.
2. Read the file in chunks (e.g., 1024 bytes) and send each chunk to the server.
3. Repeat until the entire file is sent.
4. Print a success message after the file is sent.

```
import socket

def udp_file_client(server_host, server_port, filename):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    #Write your code

# Example usage:
udp_file_client('localhost', 5555, 'file_to_send.txt')
```

12.5 Simple TCP Server

Insert a python program for implementing a simple TCP server side.(Socket program)

Hint:

1. Use the 'socket' module to create a TCP server socket.
2. Bind the server socket to a specific address and port.
3. Use the 'listen' method to start listening for incoming connections.
4. Accept incoming connections using the 'accept' method.
5. Communicate with the client using the returned connection socket.

```
import socket

def tcp_server(port):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #Write your code

# Example usage:
```

Try:

Use the UDP server socket instead of TCP server socket.

12.6 Simple TCP Client

Implement a python program for Generate the code on simple TCP Client side.

Hint:

1. Use the 'socket' module to create a TCP client socket.
2. Connect to the server using the 'connect' method and provide the server's address and port.
3. Send data to the server using the 'sendall' method.
4. Receive the server's response using the 'recv' method.

```
import socket

def tcp_client(server_address, server_port):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #Write your code

# Example usage:
tcp_client('localhost', 5555)
```

13. VPN AND IP

13.1 Public IP

Create a Python function named check_public_ip that retrieves your public IP address using an external service. Implement the function to make an HTTP request to the "https://api.ipify.org?format=json" endpoint, parse the JSON response, and return the public IP address.

Hint:

1. Use the 'requests' library to make an HTTP GET request to the specified URL.
2. Check if the response status code is 200 to ensure a successful request.
3. Parse the JSON response using the .json() method provided by the requests library.
4. Access the "ip" key in the JSON data to obtain the public IP address.
5. If there is an error in retrieving the IP address, print an error message along with the status code.

```
import requests

def check_public_ip():
    """Retrieves your public IP address using an external service."""

    #Write your code

# Example usage
public_ip = check_public_ip()

if public_ip:
    print(f"Your public IP address: {public_ip}")
```

Try:
Try finding the IP without using Json().

13.2 Simulate Network Traffic

Implement a Python function named generate_traffic that simulates network traffic for a specified duration.

Hint:

1. Use the random module to randomly choose a protocol, generate a packet size within the specified range, and simulate delays.
2. The time.sleep() function can be used to introduce delays between simulated network interactions.
3. Create a loop that runs for the specified duration and prints messages for each simulated network interaction.
4. Replace the print statement inside the loop with actual network interaction code if you have a specific network-related task to simulate.

```
import random
import time

def generate_traffic(duration=10, protocols=["TCP", "UDP"], packet_size_range=(100,
500)):
    """Simulates network traffic with random protocols, packet sizes, and delays."""

    #write your code

generate_traffic()
```

Try:
Change the duration, protocols and packet_size_range and observe the change in results.

14. Network Traffic Analysis

14.1 Network Basic Statistics

Develop a Python program to capture live network traffic for a specified duration and analyze basic statistics.
Hint:

1. Install the scapy library using pip install scapy.
2. Use scapy.sniff to capture live packets. Set the count parameter to the desired number of packets, and timeout to the specified duration.
3. Access the IP layer information using scapy.IP, and TCP/UDP layer information using scapy.TCP or scapy.UDP.
4. Extract source and destination IP addresses, protocols, and other relevant information from each captured packet.
5. Print the basic statistics, and demonstrate optional filtering based on specific criteria (e.g., port number).

```
import scapy.all as scapy

def capture_and_analyze(duration=10):
    """Captures live traffic for the specified duration and analyzes basic
statistics."""

    #Write your  code

capture_and_analyze()
```

Try:
Change the duration of the duration to 30.

14.2 Visualize Network Traffic

Illustrate a Python script for bar chart representing network traffic by protocol.

Hint:

1. Replace the protocol counts dictionary with your actual data, ensuring that keys are protocol names and values are corresponding packet counts.
2. Install the matplotlib library using pip install matplotlib if you haven't already.
3. Use Pl. Bar to create a bar chart. Pass protocol names as the x-axis and packet counts as the y-axis.
4. Customize the chart by setting labels, title, and adjusting the figure size for better visualization.
5. Experiment with different customization options provided by matplotlib to enhance the appearance of the chart.

```
import matplotlib.pyplot as plt

# Sample data (replace with your actual data)
protocol_counts = {"TCP": 50, "UDP": 20, "ICMP": 30}
```

```
# Create a bar chart
#Write your code
```

Try:

Use the histogram, pie chart for visualization.

V. TEXT BOOKS

1. Behrouz A. Forouzan, “Data Communications and Networking”, Tata McGraw-Hill, 5th edition, 2012
2. Andrew S. Tanenbaum, David.j.Wetherall, ”Computer Networks”, Prentice-Hall, 5th edition, 2010.

VI. REFERENCE BOOKS:

1. Douglas E. Comer, “Internetworking with TCP/IP”, Prentice-Hall, 5th Edition, 2011.
2. Peterson, Davie, Elsevier, “Computer Networks”, 5th Edition,2011
3. Comer, “Computer Networks and Internets with Internet Applications”, 4th Edition, 2004.
4. Chwan-Hwa Wu, Irwin, “Introduction to Computer Networks and Cyber Security”, CRC publications, 2014.

VII. WEB REFERENCES

1. https://www.geeksforgeeks.org/computer-network-tutorials/