COMPUTER NETWORKS LAB

REPORT ASSIGNMENT-3

NAME: SOHAM LAHIRI

CLASS: BCSE UG-III 5TH SEMESTER

ROLL NO: 002210501107

GROUP: A3

SUBMISSION DATE: 28/10/2024

Problem Statement: Design and implement medium access control mechanisms within a simulated network environment using IEEE 802 standards. Implement p-persistent CSMA technique with collision detection.

In this assignment, you have to implement a p-persistent CSMA technique with collision detection. Measure the performance parameters like throughput (i.e., average amount of data bits successfully transmitted per unit time) and forwarding delay (i.e., average end-to-end delay, including the queuing delay and the transmission delay) experienced by the CSMA frames (IEEE 802.3). Consider adding a Collision Detection module at the sender side. This will randomly inject collisions in the system.

Test the above schemes for the following cases (not limited to).
i) Plot the comparison graphs for throughput and forwarding delay by varying p.
ii) Compare efficiency of the above approach by varying p.
State your observations on the impact of performance of CSMA-CD in different scenarios.

DESIGN

Theory:

The **p-persistent Carrier Sense Multiple Access (CSMA)** is a network protocol used to manage how data packets are transmitted in a shared network, particularly in scenarios where multiple devices or nodes might try to send data simultaneously. This protocol is commonly used in local area networks (LANs), especially in wireless networks where collision handling and network efficiency are crucial. It operates by sensing the carrier (checking if the channel is free) and taking transmission actions based on the probability ppp.

Overview of CSMA

Carrier Sense Multiple Access (CSMA) protocols require each device in a network to listen to the channel for any ongoing transmission. If the channel is busy, the device will defer its transmission until the channel becomes idle. Once idle, the CSMA protocol uses different strategies to decide when a device should begin transmission, minimizing the likelihood of collisions, which occur when two or more devices transmit simultaneously.

Key Features of p-persistent CSMA

In **p-persistent CSMA**, each device that wants to send data follows these steps:

1. **Carrier Sensing**: The device senses the channel to determine if it is free (idle) or busy.
2. **Probability-based Transmission**:
    - If the channel is free, the device transmits with a probability ppp.
    - With a probability (1−p)(1 - p)(1−p), the device waits for the next time slot and repeats the process until it can transmit or the channel becomes busy.
3. **Collision Detection**: While transmitting, the device monitors for any collision.
    - If a collision is detected, the device stops transmission immediately and performs a backoff process before attempting to resend.

This approach balances efficient channel use and minimizes collision chances, especially useful in high-traffic networks.

p-persistent CSMA with Collision Detection:

Steps:

1. **Sense the Channel**: The device checks if the channel is idle.
    - If busy, it waits until the channel is free.
2. **Probability of Transmission**:
    - Once the channel is free, the device transmits with probability ppp.
    - If it does not transmit (with probability 1−p1 - p1−p), it defers to the next time slot.

3. **Collision Handling**:
    - If the device transmits and detects a collision, it ceases transmission.
    - It then applies a backoff algorithm (like binary exponential backoff) and retries after a random delay.
4. **Repeat Process**: The device follows this process continuously, which helps in evenly distributing the attempts to access the channel among all devices, reducing the likelihood of collisions.

Parameters of p-persistent CSMA

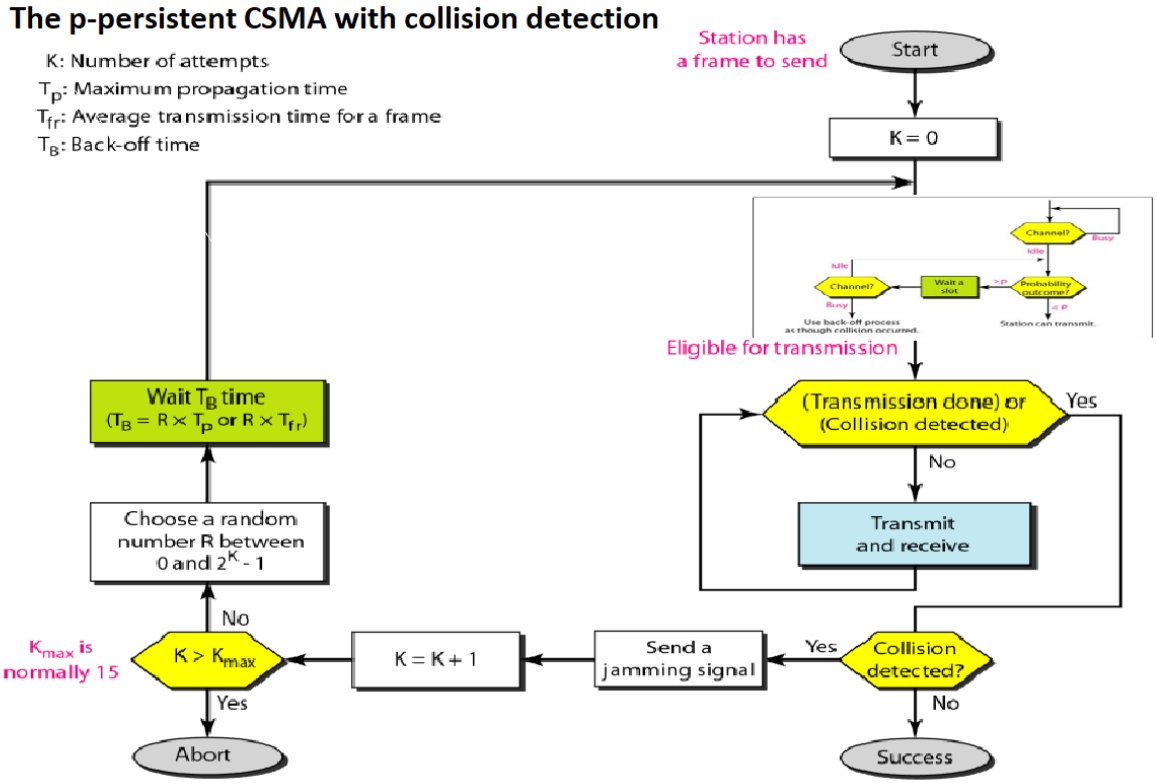
- **Probability p**: Determines the likelihood that a device will transmit when the channel is free. A higher p increases the chance of quicker transmission but also raises the risk of collisions.
- **Time Slot**: Devices defer to the next available time slot if they do not transmit.

Benefits and Drawbacks

- **Benefits**:
    - Balances channel utilization and minimizes collision chances.
    - Reduces idle time by allowing controlled attempts based on probability.
- **Drawbacks**:
    - Collisions can still occur, especially when many devices attempt to transmit simultaneously.
    - Requires fine-tuning of the probability p to optimize performance.

This protocol is particularly effective in networks with moderate to high traffic loads, as it can manage access dynamically based on probability, preventing network overload and efficiently handling contention.

Flow Diagram:



Implementation:

Sender Program:

```
#!/usr/bin/env python3
import sys
import time
import random
import threading
from packetManager import Packet
import datetime


class Sender:
    def __init__(self, id: int, filename: str, senderToChannel, channelToSender,
method: int, senderCount:int) -> None:
        self.start = 0
        self.seq = 0
        self.packetCount = 0
        self.collisionCount = 0
        self.senderCount = senderCount
        self.busy = False
        self.id = id
        self.filename = filename
        self.senderToChannel = senderToChannel
        self.channelToSender = channelToSender
        self.method = method
        self.packet_type = {'data': 0, 'ack': 1}
        self.dest = self.id

    def read_file(self, filename: str):
        try:
            fd = open(filename, "r", encoding='utf-8')
        except FileNotFoundError as err:
            current_time = datetime.datetime.now()
            print(f"\n [{current_time.strftime('%d/%m/%Y %H:%M:%S')}] ERROR: {err}
File {filename} not found!")
            sys.exit(f"File {filename} Not Found!")
        return fd

    def p_persistent(self, packet):
        while True:
            if not self.busy:
                x = random.random()
```

```
            p = 1/self.senderCount
            if x <= p:
                f = self.read_file("./logs/collide.txt")
                collision = f.read()
                f.close()
                if collision == '1':
                    self.collisionCount += 1
                    current_time = datetime.datetime.now()
                    with open("logs/log.txt", "a+", encoding="utf-8") as fp:
                        fp.write(f"[{current_time.strftime('%d/%m/%Y %H:%M:%S')}]
Sender {self.id+1} encounters COLLISION.")
                    time.sleep(0.1) # wait after collision
                else:
                    current_time = datetime.datetime.now()
                    with open("logs/log.txt", "a+", encoding="utf-8") as fp:
                        fp.write(f"[{current_time.strftime('%d/%m/%Y %H:%M:%S')}]
Sender {self.id+1} sent Packet {self.packetCount+1} to Channel\n")
                    f = open('logs/collide.txt', "w", encoding='utf-8')
                    f.write(str(1))
                    f.close()
                    time.sleep(0.1) # vulnerable time
                    f = open('logs/collide.txt', "w",  encoding='utf-8')
                    f.write(str(0))
                    f.close()
                    self.senderToChannel.send(packet)
                    time.sleep(1) # propagation time
                    break
            else:
                current_time = datetime.datetime.now()
                with open("logs/log.txt", "a+", encoding="utf-8") as fp:
                    fp.write(f"[{current_time.strftime('%d/%m/%Y %H:%M:%S')}] Sender
{self.id+1} is WAITING, wait period:0.25secs.\n")
                time.sleep(0.25) # wait a certain time
        else:
            current_time = datetime.datetime.now()
            with open("logs/log.txt", "a+", encoding="utf-8") as fp:
                fp.write(f"[{current_time.strftime('%d/%m/%Y %H:%M:%S')}] Sender
{self.id+1} finds Channel is BUSY.")
            time.sleep(0.5)
            continue

    def carrier_sense(self):
        while True:
            if self.channelToSender.recv() == '1':
```

```
            self.busy = True
        else:
            self.busy = False

    def transfer_data(self):
        current_time = datetime.datetime.now()
        with open("logs/log.txt", "a+", encoding="utf-8") as fp:
            fp.write(f"\n[{current_time.strftime('%d/%m/%Y %H:%M:%S')}] Sender
{self.id+1} Starts sending to Receiver {self.dest+1}.\n")
        self.start = time.time()
        fd = self.read_file(self.filename)
        data = fd.read(46)
        self.seq = 0
        while data:
            packet = Packet(self.packet_type['data'], self.seq, data, self.id,
self.dest).generate_packet()
            if self.method == 1:
                self.one_persistent(packet)
            elif self.method == 2:
                self.non_persistent(packet)
            else:
                self.p_persistent(packet)
            self.packetCount += 1
            data = fd.read(46)
            if len(data) == 0:
                break
            if len(data) < 46:
                l = len(data)
                for _ in range(46-l): data += ' '
        fd.close()
        current_time = datetime.datetime.now()
        with open("logs/log.txt", "a+", encoding="utf-8") as fp:
            fp.write(f"\n\n**********[{current_time.strftime('%d/%m/%Y %H:%M:%S')}]
Sender {self.id+1} FINISHES sending data*********\n\n")
        with open('logs/analysis.txt', 'a+', encoding='utf-8') as fp:
            fp.write(f"\n\n+---------- {current_time} SENDER-{self.id+1} STATS ----------+"
+ '\n' + \
                     "[*]\tTotal packets: {}".format(self.packetCount) + '\n' + \
                     "[*]\tTotal Delay: {} secs".format(round(time.time() - self.start, 2)) +
'\n' + \
                     "[*]\tTotal collisions: {}".format(self.collisionCount) + '\n' + \
                     "[*]\tThroughput:
{}".format(round(self.packetCount/(self.packetCount + self.collisionCount), 3)) + '\n'
+ \
```

```
                "+-------------------------------------------------+\n\n")

    def init_sender(self):
        sender_thread = threading.Thread(name="sender_thread",
target=self.transfer_data)
        sensing_thread = threading.Thread(name="sensing_thread",
target=self.carrier_sense)
        sender_thread.start()
        sensing_thread.start()
        sender_thread.join()
        sensing_thread.join()
```

Explanation:

This code implements a network packet sender with a p-persistent Carrier Sense Multiple Access (CSMA) approach. Here's a breakdown of the main components and functionality in the simplified version focusing on p-persistent:

1. **Class Initialization:**
   - The Sender class initializes essential variables like id, filename, and communication channels (senderToChannel and channelToSender) to connect the sender to the network. The method parameter specifies the type of CSMA protocol to use (1 for p-persistent here), and senderCount is used to calculate the probability p in the p-persistent approach.
2. **File Handling:**
   - read_file opens the specified file for reading. It logs an error and exits if the file isn't found, ensuring the program doesn't proceed with invalid data.
3. **P-Persistent CSMA:**
   - The p_persistent function contains the core logic for the p-persistent CSMA protocol. When the channel is not busy, a random probability check determines if the sender attempts to send a packet. If the probability is satisfied, the sender checks for any collisions by reading a collide.txt log. If no collision is detected, the sender proceeds to send the packet to the channel and enters a vulnerable period.
   - Collision detection is managed by checking the collide.txt file, which is updated to 1 on collision and reset to 0 afterward.
   - If a collision occurs, it's logged, and the sender waits briefly before retrying. Otherwise, if the probability condition is not met, the sender waits a fixed interval (0.25 seconds) before reattempting.
4. **Carrier Sensing:**
   - carrier_sense listens to the channel to check whether it's busy by reading from channelToSender. When it receives a signal, it updates the busy status of the channel accordingly.
5. **Data Transfer:**
   - transfer_data initiates the process of reading data from the file, segmenting it into packets, and transmitting each packet using the p_persistent protocol.
   - It logs the start and completion of data transmission, and it gathers statistics on the total packets sent, collisions encountered, total delay, and throughput, writing this information into an analysis log.
6. **Initialization and Execution:**
   - init_sender starts two threads: one for sending data packets and another for continuously sensing the channel's status. Both threads run concurrently, allowing real-time channel sensing while transmitting data.

Receiver Program:

```
#!/usr/bin/env python3
import sys
import datetime
from packetManager import Packet

class Receiver:
    def __init__(self, id: int, channelToReceiver) -> None:
        self.seq = 0 #must be synced with sender seq
        self.id = id
        self.sender_dict = {} # key value pair of sender_id:outfile_path
        self.channelToReceiver = channelToReceiver

    def write_file(self, filename: str):
        try:
            fd = open(filename, "a+")
        except FileNotFoundError as err:
            current_time = datetime.datetime.now()
            print(f"\n [{current_time.strftime('%d/%m/%Y %H:%M:%S')}] ERROR: {err}
File {filename} not found!")
            sys.exit(f"File {filename} Not Found!")
        return fd

    def init_receiver(self):
        while True:
            packet = self.channelToReceiver.recv()
            sender = packet.get_src()
            if sender not in self.sender_dict.keys():
                self.sender_dict[sender] = "./logs/output/output" + str(sender+1) + '.txt'

            outfile = self.sender_dict[sender]
            fd = self.write_file(outfile)
            datastr = packet.get_data()
            fd.write(datastr)
            fd.close()
            current_time = datetime.datetime.now()
            with open("logs/log.txt", "a+") as f:
                f.write(f"\n[{current_time.strftime('%d/%m/%Y %H:%M:%S')}] Receiver-
{self.id+1} received Packet SUCCESSFULLY!\n")
```

Explanation:

This Receiver class in Python is designed to receive packets from a communication channel, save the data to output files, and log the process. Here's a breakdown of the components and functionality:

1. Class Initialization:

1. __init__ Method:
    - Takes parameters id (unique receiver ID) and channelToReceiver (communication channel where packets arrive).
    - Initializes:
        - seq: Sequence number initialized to 0, expected to match the sender's sequence number for synchronization.
        - sender_dict: Dictionary to keep track of sender IDs and their corresponding output file paths.
        - channelToReceiver: Channel from which the receiver will receive packets.

2.Methods:

1. write_file Method:
    - Opens a specified file in append mode (a+) to allow continuous writing without overwriting existing data.
    - Handles FileNotFoundError by logging the error message and exiting if the specified file does not exist.
2. init_receiver Method:
    - Infinite loop to continually receive packets.
    - Each time a packet is received, it:
        - Extracts the sender's ID from the packet.
        - Checks if the sender's ID is already in sender_dict:
            - If not, it adds an entry mapping the sender ID to a unique output file path.
        - Writes the packet's data to the associated file for that sender.
        - Logs the reception of the packet to a log file, logs/log.txt, with a timestamp to monitor when each packet is successfully received.

3.Workflow:

When the receiver is running, it will:

- Continuously check channelToReceiver for incoming packets.
- Write each packet's data to a designated output file based on the sender's ID.
- Log every successful packet reception in logs/log.txt.

Logging

- The log file records each step with timestamps to trace the packet flow, which is especially useful for debugging or analyzing transmission efficiency and errors.

P-persistent CSMA Program:

```
#!/usr/bin/env python3
import threading
import time
import random

class bcolors:
    HEADER = '\033[106m'
    OKBLUE = '\033[94m'
    OKCYAN = '\033[96m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'

    @classmethod
    def return_color(cls, style, fg, bg, text):
        format = ';'.join([str(style), str(fg), str(bg)])
        s1 = '\x1b[%sm%s \x1b[0m' % (format, text)
        return s1

total_frames = 0
frametime = 0.2
inbetween_frametime = 0.05
nframes = 0
frame_attempts = 0
#backoff_period = 0

def channel(lock: threading.Lock, total_frames_to_send: int) -> None:
    global total_frames
    global frame_attempts
    being_used = 0
    total = 0
    while total_frames < total_frames_to_send:
        if lock.locked():
            being_used += 1
        total += 1
        #print(f"{bcolors.return_color(0, 32, 40, f'[i] INFO: Used:{being_used},
Total:{total}')}")
    try:
        print(f"{bcolors.return_color(0, 0, 0, f'[i] INFO: Channel Utilisation:
{being_used/total}')}")
        print(f"{bcolors.return_color(0, 0, 0, f'[i] INFO: Channel Idle: {(total-
```

```
being_used)/total}')}")
        print(f"{bcolors.return_color(0, 0, 0, f'[i] INFO: Efficiency:
{total_frames/frame_attempts}')}")
    except ZeroDivisionError:
        print(f"{bcolors.return_color(0, 0, 0, '[-] ERROR: Division by zero is not
possible!')}")

class PPersistentCsma(threading.Thread):
    def __init__(self, lock: threading.Lock, index: int) -> None:
        super().__init__()
        self.index = index
        self.lock = lock

    def run(self):
        global total_frames
        global nframes
        global frame_attempts
        global probability
        global backoff_period
        random.seed(time.time())
        cnt = 1
        while cnt <= nframes:
            print(f"{bcolors.return_color(0, 0, 0, f'[i] ATTEMPT: Frame {cnt}, Station
{self.index}')}")
            while self.lock.locked():
                print(f"{bcolors.return_color(0, 0, 0, f'[-] CARRIER SENSE: Frame {cnt},
Station {self.index}')}")

            x = random.random()
            while x > probability:
                print(f"{bcolors.return_color(0, 0, 0, f'[-] WAITING: Frame {cnt}, Station
{self.index}, Period: {backoff_period}, x: {x}')}")
                time.sleep(backoff_period)
                while self.lock.locked():
                    print(f"{bcolors.return_color(0, 0, 0, f'[-] FAILED: Frame {cnt}, Station
{self.index}')}")
                frame_attempts += 1
                x = random.random()

            self.lock.acquire()
            #critical section
            time.sleep(frametime)
            total_frames += 1
            frame_attempts += 1
```

```
            print(f"{bcolors.return_color(0, 0, 0, f'[+] SUCCESS: Frame {cnt}, Station
{self.index}')}")
            self.lock.release()
            time.sleep(inbetween_frametime)
            cnt += 1

def main():
    global nframes
    global backoff_period
    global probability
    nstations = int(input(f"{bcolors.OKBLUE}[*] PROMPT: Number of
stations:{bcolors.ENDC}"))
    nframes = int(input(f"{bcolors.OKBLUE}[*] PROMPT: Number of frames to send
per station:{bcolors.ENDC}"))
    backoff_period = round(random.uniform(0.3, 0.5), 2)
    probability = 1/nstations

    lock = threading.Lock()
    station_list = [PPersistentCsma(lock, i+1) for i in range(nstations)]
    channel_thread = threading.Thread(target=channel, args=[lock,
nstations*nframes])
    channel_thread.start()
    for station in station_list:
        station.start()
    for station in station_list:
        station.join()
    channel_thread.join()

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

Explanation:

This Python script simulates a p-persistent CSMA (Carrier Sense Multiple Access) protocol, which is commonly used in network communication to manage how multiple devices (stations) access a shared channel without collision. The code uses threading to represent multiple stations trying to send frames over the channel.

Key Components:

1. **Color Output Class (bcolors):**
    - Provides ANSI escape sequences to print colored text in the console.
    - Includes a method, return_color, which formats text with custom color codes for easy interpretation of logs.

2. **Global Variables:**
    - total_frames, nframes, and frame_attempts track the number of frames successfully sent, frames each station should send, and total attempts to send frames.
    - frametime and inbetween_frametime set delays for sending frames and pauses between frames.
    - probability and backoff_period define the probability threshold for a station to access the channel and the delay before reattempting if the channel is busy.

3. **channel Function:**
    - Simulates the channel's status and utilization over time.
    - Checks if the lock (channel) is busy and counts usage (being_used).
    - At the end, calculates and prints:
        - Channel Utilization (time spent being used vs total time).
        - Idle Time (when the channel was free).
        - Efficiency (success rate of frames sent over attempts).

4. **Station Thread Class (P-Persistent Csma):**
    - Each instance represents a station trying to send frames.
    - Each station continuously checks if the channel (lock) is free:
        - If busy, the station waits (carrier sense).
        - If free, the station generates a random number (x) to decide whether to attempt to send the frame based on probability.
    - If x > probability, the station waits for the backoff_period, and the process repeats.
    - If successful, it enters the critical section (acquires the lock, sends the frame, and releases the lock).

5. **Main Program (main Function):**
    - Takes user input for the number of stations and frames.
    - Sets backoff_period and probability based on the number of stations.
    - Starts a channel-monitoring thread and individual threads for each station.
    - Waits for all station threads and the channel thread to finish.

Workflow

1. The program initializes the channel lock and starts n stations and a channel monitoring thread.
2. Each station tries to send frames, waiting if the channel is busy and reattempting with a certain probability.
3. The channel thread reports the channel's usage statistics at the end, while each station prints logs of each attempt and success/failure.

This simulation illustrates how p-persistent CSMA works, where each station probabilistically accesses a shared medium and logs performance metrics to evaluate protocol efficiency.

Packet Manager Program:

```
#!/usr/bin/env python3
import validate
from crypto.Util.number import long_to_bytes

class Packet:
    def __init__(self, _type, seq, segment_data, src, dest) -> None:
        self.type = _type
        self.seq = seq
        self.segment_data = segment_data
        self.src = src
        self.dest = dest
        self.packet = ''

    def generate_packet(self):
        preamble = '01'*28
        sfd = '10101011'
        src_addr = '{0:048b}'.format(int(self.src))
        dest_addr = '{0:048b}'.format(int(self.dest))
        seqbits = '{0:08b}'.format(self.seq)
        length = '{0:08b}'.format(len(self.segment_data))
        data = ''
        for i in range(len(self.segment_data)):
            character = self.segment_data[i]
            data += '{0:08b}'.format(ord(character))
        packet = preamble + sfd + dest_addr + src_addr + seqbits + length + data
        checksum = validate.get_checksum(packet)
        packet += checksum
        self.packet = packet
        return self

    def __str__(self) -> str:
        return str(self.packet)

    def get_datalen(self) -> int:
        return len(self.segment_data)

    def get_type(self) -> int:
        return self.type

    def get_seqno(self) -> int:
        seqbits = self.packet[160:168]
        return int(seqbits, 2)
```

```
def get_src(self) -> int:
    return int(self.packet[112:160], 2)

def get_dest(self) -> int:
    return int(self.packet[64:112], 2)

def get_data(self) -> str:
    datastr = ""
    databits = self.packet[176:544]
    datastr = long_to_bytes(int(databits, 2)).decode('utf-8')
    return datastr

def validate_packet(self) -> bool:
    return validate.validate_checksum(self.packet)
```

Explanation:

This Python code defines a Packet class for constructing and handling network packets with custom binary formatting. The Packet class provides methods to build packets, extract data, and verify integrity using a checksum function. Here's a detailed breakdown of each part:

Key Components:

1. **Attributes of Packet Class:**
    - type: Represents the packet type (could indicate whether it's a data or acknowledgment packet).
    - seq: Sequence number of the packet.
    - segment_data: The actual data being transmitted in the packet.
    - src and dest: Source and destination addresses, respectively.
    - packet: A binary string representation of the fully constructed packet, initialized as an empty string.
2. **Packet Construction (generate_packet):**
    - Preamble: A fixed bit pattern '01' * 28, a common feature in network packets to help synchronize the receiver.
    - SFD (Start Frame Delimiter): '10101011', a fixed bit sequence marking the start of the packet.
    - Source and Destination Addresses: Converted to 48-bit binary strings.
    - Sequence Number and Length: Represented as 8-bit binary strings.
    - Data: Each character in segment_data is converted to an 8-bit binary ASCII representation and concatenated.
    - Checksum: Generated using an external validate module (assumed to provide checksum calculation and validation methods) and appended to the end of the packet.
    - Final Packet: Concatenates all parts and stores it in self.packet.
3. **String Representation (__str__):**
    - Converts the binary packet into a readable string format, enabling easy printing of the entire packet structure.
4. **Packet Information Extraction:**
    - get_datalen: Returns the length of segment_data.
    - get_type, get_seqno, get_src, get_dest: Extracts the packet type, sequence number, source, and destination addresses from self.packet. Binary segments are converted back to integers where necessary.
    - get_data: Extracts the payload data from the packet binary. It:
        - Reads the binary data segment,
        - Converts it into an integer, then to bytes, and finally decodes it to a UTF-8 string for easy access.

5. **Packet Validation (validate_packet):**
    - Calls validate.validate_checksum, an external function that checks if the packet's checksum matches the calculated value, ensuring data integrity.

Summary

The Packet class is designed for creating, managing, and verifying custom network packets in binary format. It encodes attributes like addresses and sequence numbers into binary strings, attaches a checksum for error checking, and provides methods for extracting and verifying packet data. This structure is typical in network communication, where data integrity and structure are paramount.

Validate Program:

```
#!/usr/bin/env python3
def get_checksum(segment_data: str) -> str:
    total = 0
    chunks = [segment_data[i:i+32] for i in range(0, len(segment_data), 32)]
    for chunk in chunks:
        total += int(chunk, 2)
        if total >= 4294967295:
            total -= 4294967295
    checksum = 4294967295 - total
    return '{0:032b}'.format(checksum)

def validate_checksum(segment_data: str) -> bool:
    total = 0
    chunks = [segment_data[i:i+32] for i in range(0, len(segment_data), 32)]
    for chunk in chunks:
        total += int(chunk, 2)
        if total >= 4294967295:
            total -= 4294967295
    return True if total == 0 else False
```

Explanation:

This code defines two functions, get_checksum and validate_checksum, which are used for calculating and verifying the checksum of a binary data string. These functions ensure data integrity by detecting errors in transmission. Here's how each function works:

1. get_checksum

- **Purpose**: Calculates the checksum for a given binary string (representing a data segment).
- **Process**:
    1. **Divide the Data**: The input segment_data is split into 32-bit chunks.
    2. **Sum the Chunks**: Each chunk is converted from binary to an integer and added to a total.
    3. **Handle Overflow**: If total exceeds 4294967295 (32-bit max value), the overflow is handled by subtracting 4294967295 from total. This ensures the result stays within 32 bits.
    4. **Calculate the Checksum**: The checksum is computed by taking 4294967295 - total, which produces the binary complement. This binary string of the checksum is returned in 32-bit format.
- **Return**: The function returns a 32-bit binary string representing the checksum.

2. validate_checksum

- **Purpose**: Verifies that the checksum of a given binary data segment is correct.
- **Process**:
    1. **Divide and Sum**: Similar to get_checksum, it splits segment_data into 32-bit chunks, converts each to an integer, and accumulates them in total.
    2. **Handle Overflow**: Any overflow beyond 4294967295 is managed by subtracting 4294967295.
    3. **Validation Check**: The data is considered valid if the total equals 0 after summing all chunks. This indicates that the data, combined with its checksum, is error-free.
- **Return**: Returns True if the checksum is valid, indicating data integrity, or False if not.

Summary

- **get_checksum** creates a checksum to detect errors in segment_data.
- **validate_checksum** verifies whether a segment (with its checksum appended) is intact and error-free

Output:

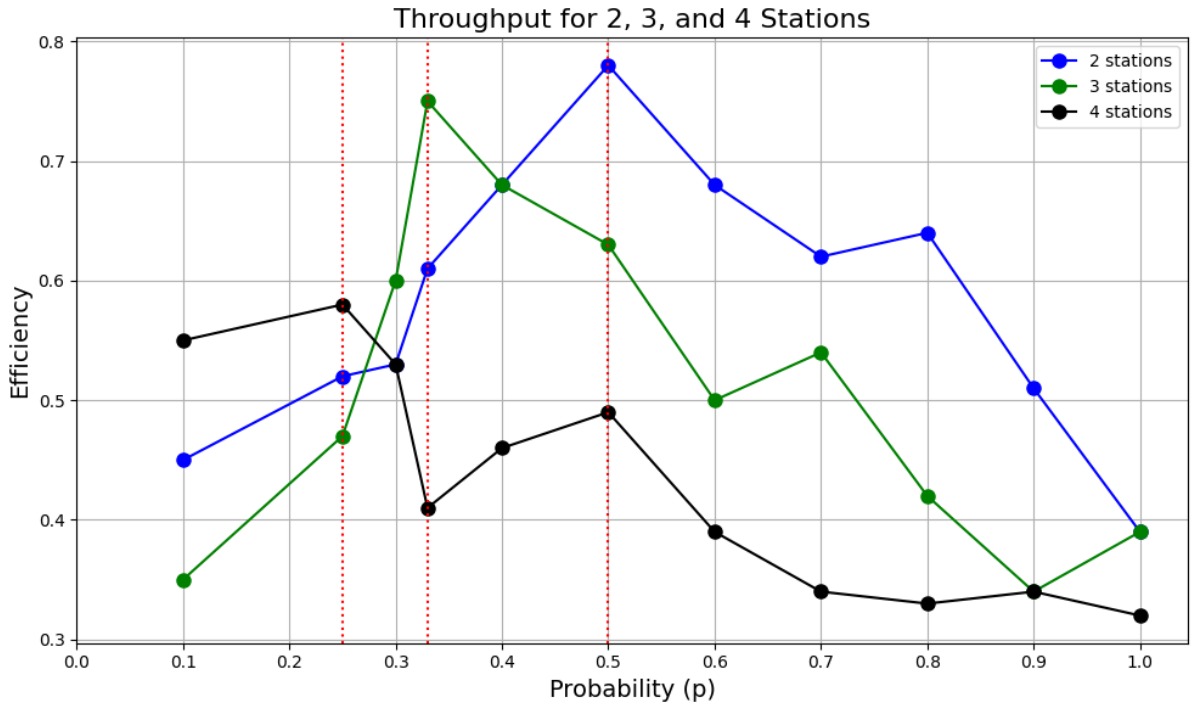
```
[*] PROMPT: Number of stations:2
[*] PROMPT: Number of frames to send per station:2
[i] ATTEMPT: Frame 1, Station 1
[-] WAITING: Frame 1, Station 1, Period: 0.34, x: 0.8450252641980329
[i] ATTEMPT: Frame 1, Station 2
[-] WAITING: Frame 1, Station 2, Period: 0.34, x: 0.8181155006927213
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
```

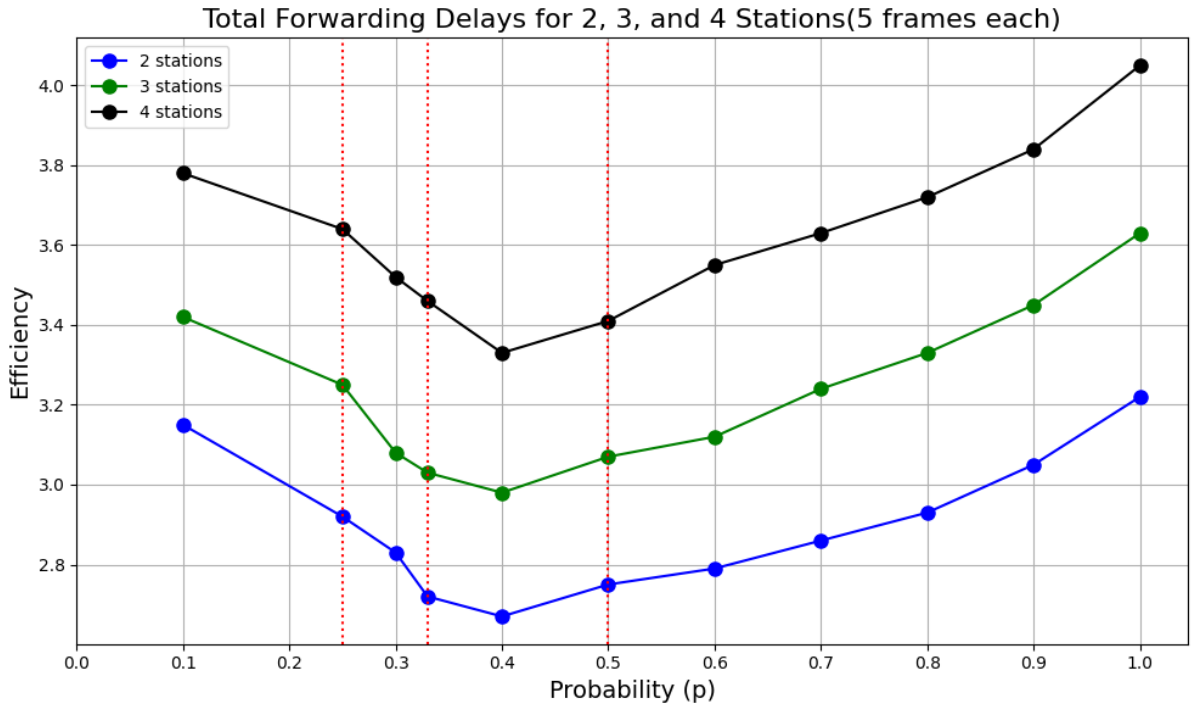
```
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[-] FAILED: Frame 1, Station 2
[+] SUCCESS: Frame 1, Station 1
[i] ATTEMPT: Frame 2, Station 1
[-] CARRIER SENSE: Frame 2, Station 1
[-] CARRIER SENSE: Frame 2, Station 1
[-] CARRIER SENSE: Frame 2, Station 1
[-] CARRIER SENSE: Frame 2, Station 1
[-] CARRIER SENSE: Frame 2, Station 1
[-] CARRIER SENSE: Frame 2, Station 1
[-] CARRIER SENSE: Frame 2, Station 1
[-] CARRIER SENSE: Frame 2, Station 1
[-] CARRIER SENSE: Frame 2, Station 1
[-] CARRIER SENSE: Frame 2, Station 1
[-] CARRIER SENSE: Frame 2, Station 1
[-] CARRIER SENSE: Frame 2, Station 1
[-] CARRIER SENSE: Frame 2, Station 1
[-] CARRIER SENSE: Frame 2, Station 1
[+] SUCCESS: Frame 1, Station 2
[i] ATTEMPT: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
```

```
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[-] CARRIER SENSE: Frame 2, Station 2
[+] SUCCESS: Frame 2, Station 1
[+] SUCCESS: Frame 2, Station 2
[i] INFO: Channel Utilisation: 0.5984153585743
[i] INFO: Channel Idle: 0.4015846414257
[i] INFO: Efficiency: 0.6666666666666666
```

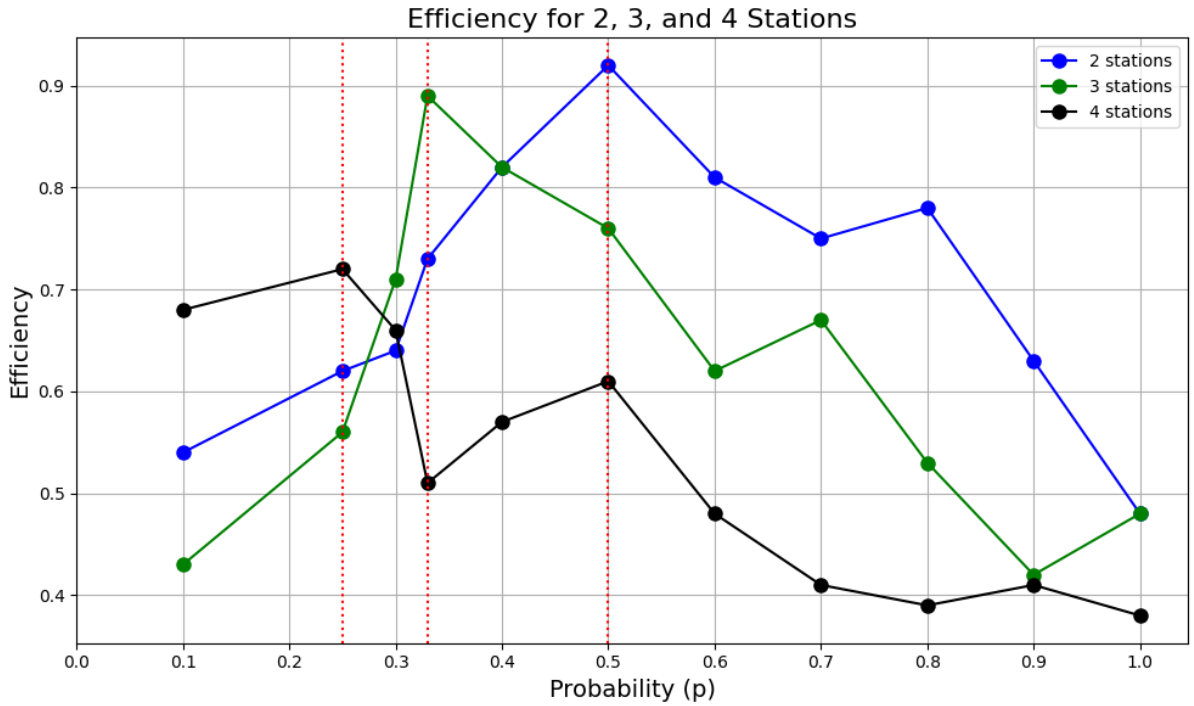
Graphs

i)Throughput and forwarding delay





ii)Efficiency of the approach



Observations

Throughput Analysis:

1. **General Behavior**:
    - Throughput varies with the probability p, rising initially at lower values of p, reaching a peak, and then decreasing as p nears 1.
    - For each station count (2, 3, and 4), the throughput curve displays a single peak, indicating an optimal p value that minimizes idle and collision times.
2. **Influence of Station Count**:
    - As the station count increases, the maximum achievable throughput decreases.
    - For 2 stations (blue line): Throughput is the highest across the p range, achieving peak efficiency at a higher value than configurations with more stations.
    - For 3 stations (green line): The throughput peak is slightly reduced compared to the 2-station configuration, suggesting increased contention and collision frequency.
    - For 4 stations (black line): The peak throughput is lowest, with a more pronounced drop at higher probabilities due to increased collisions in a more congested network.
3. **Impact of Probability**:
    - **Lower probabilities (approximately p=0.1−0.3p = 0.1 - 0.3p=0.1−0.3)**: Throughput increases as p rises since each station has a low transmission chance, minimizing idle time and collision frequency.
    - **Middle probability range (around p=0.3−0.5p = 0.3 - 0.5p=0.3−0.5)**: Throughput reaches its peak as the system balances collision avoidance with channel utilization.
    - **Higher probabilities (above p=0.5p = 0.5p=0.5)**: Throughput declines due to increased collision frequency, as stations are more likely to transmit simultaneously, leading to delays and reduced throughput.
4. **Interpretation**:
    - This trend aligns with the expected behavior of p-persistent CSMA/CD, where an optimal probability enables maximum throughput by balancing idle and collision times.
    - An increased station count leads to a reduced peak throughput and faster throughput decline as p increases due to heightened collision frequency.

Forwarding Delay Analysis:

1. **General Trend**:
    - Forwarding delay follows a U-shaped curve, decreasing initially as p increases to an optimal point, then rising again as p nears 1.
    - At low probabilities, forwarding delay is high due to extended idle times. As p increases, idle times shorten, minimizing forwarding delay until reaching an optimal level. At high probabilities, increased collisions drive up delay.
2. **Effect of Station Count**:
    - The station count has a direct effect on average forwarding delay.

- For 2 stations (blue line): The delay is lowest across the probability range due to fewer collisions.
- For 3 stations (green line): Delays are moderately higher, especially at high probabilities, as collision likelihood grows.
- For 4 stations (black line): Delay is highest, particularly at higher probabilities, owing to increased contention and collisions.

3. **Probability Impact**:
   - **Low probability (p<0.3p < 0.3p<0.3)**: High forwarding delay arises due to idle times, as stations are less likely to attempt transmission.
   - **Optimal probability range (approximately p=0.3−0.5p = 0.3 - 0.5p=0.3−0.5)**: Delay is minimized as idle time reduces and network efficiency improves with fewer collisions.
   - **High probability (p>0.5p > 0.5p>0.5)**: Delay increases with more collisions, as simultaneous transmission attempts become more frequent.
4. **Interpretation**:
   - The U-shaped delay curve is characteristic of p-persistent CSMA/CD, where an optimal p value minimizes idle and collision-induced delays.
   - Higher station counts lead to a slightly lower optimal p and steeper delay increases at high probabilities due to more frequent collisions.

Combined Throughput and Delay Observations:

1. **Relationship Between Throughput and Delay**:
   - Both throughput and forwarding delay reach their optimal points in a similar probability range (around p=0.3−0.5p = 0.3 - 0.5p=0.3−0.5), where throughput peaks, and delay is minimized.
   - Beyond this optimal p, throughput declines, and delay rises, due to increased collision likelihood when multiple stations attempt transmission.
2. **Influence of Station Count**:
   - Increased station count generally leads to reduced peak throughput and increased forwarding delay, reflecting higher contention and collision probability.
   - An optimal p value that takes station count into account is crucial for maximizing network efficiency and minimizing delays.

Efficiency Analysis:

1. **General Efficiency Trends**:
   - For all station counts, efficiency rises initially with p, peaks, and then declines as p approaches 1, demonstrating the balance between idle time and collision frequency.
2. **Probability Effects on Efficiency**:
   - **Low probability (p<0.3p < 0.3p<0.3)**: Efficiency is low due to high idle time, as stations are less likely to transmit.

- **Optimal probability range (around p=0.3−0.5p = 0.3 - 0.5p=0.3−0.5)**: Efficiency reaches its peak here, where stations use the channel effectively with minimal collisions.
- **High probability (p>0.5p > 0.5p>0.5)**: Efficiency falls as collision rates rise, leading to repeated backoff and retransmissions.

3. **Station Count and Efficiency**:
   - **2 stations (blue line)**: Achieves the highest peak efficiency, maintaining relatively high values across the probability range. Fewer collisions enhance protocol performance.
   - **3 stations (green line)**: Lower peak efficiency than 2 stations, with a greater decline at higher p values due to more frequent collisions.
   - **4 stations (black line)**: The lowest peak efficiency, with a sharper decline at high probabilities, reflecting increased contention and collision frequency in networks with more stations.
4. **Key Insights**:
   - The optimal probability range for peak efficiency shifts slightly lower as the station count grows, suggesting a lower transmission probability is preferable for maintaining efficiency in more crowded networks.
   - At very high p, all station counts show similar declines in efficiency due to the heightened likelihood of simultaneous transmissions, leading to frequent collisions and reduced transmission success.

COMMENTS:

This assignment has significantly deepened my understanding of the **p-persistent CSMA with collision detection** protocol. Through research and practical implementation, I explored how this method manages collisions in network environments, enhancing reliability and efficiency. I learned about the balance between throughput and delay and how effective collision detection optimizes data transmission.

I sincerely thank our teacher for their guidance and support, which were instrumental in helping me grasp these complex concepts and apply them practically.

4o mini

.