**Computer Networks Assignment 3: Implement 1-persistent, non-persistent and p-persistent CSMA techniques**

**Name: Sayantan Biswas**

**Class: BCSE 3rd year 1st sem**

**Roll: 001910501057**

**Group: A2**

**Problem Statement:** In this assignment, you have to implement 1-persistent, non-persistent and p-persistent CSMA techniques. 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). Plot the comparison graphs for throughput and forwarding delay by varying p. State your observations on the impact of performance of different CSMA techniques.

DESIGN: The code has been written in python language and designed using simpy framework.

This entire assignment has been designed using the following features of simpy :

1) To create a simulated network environment having a single receiver,

2) A channel and

3) Option to create multiple stations which can act as senders to the receiver.

There is no actual transfer of packet, but whenever a packet is required to be sent, the channel is kept busy by a particular station having an unique id.

SimPy is a process-based discrete-event simulation framework based on standard Python. Its event dispatcher is based on Python's generators and can also be used for asynchronous networking or to implement multi-agent systems.

***STRUCTURE DIAGRAM:***

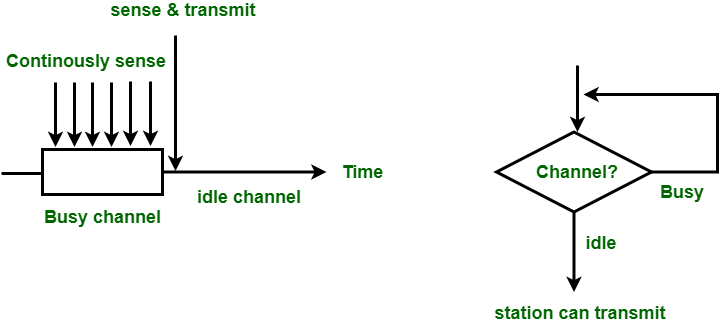
****

IMPLEMENTATION:

CSMA persistent methods:

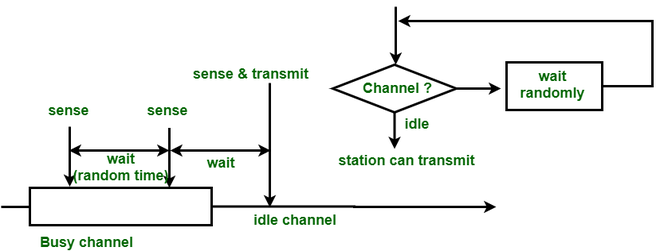
1) ***1 persistent:***

In 1-persistent CSMA, the station continuously senses the channel to check its state i.e. idle or busy so that it can transfer data or not. In case when the channel is busy, the station will wait for the channel to become idle. When station found idle channel, it transmits the frame to the channel without any delay. It transmits the frame with probability 1. Due to probability 1, it is called 1-persistent CSMA.



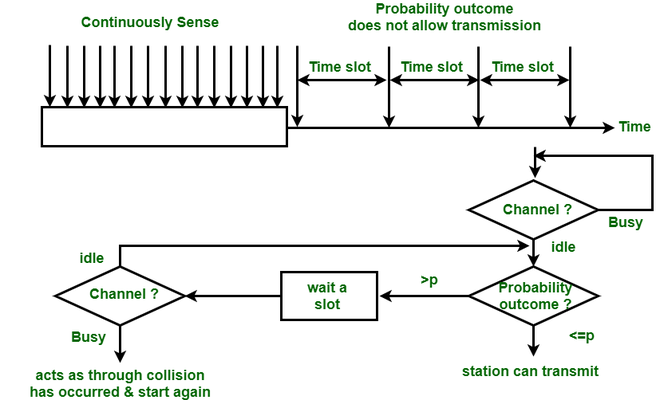
2) ***non-persistent:***

In this method, the station that has frames to send, only that station senses for the channel. In case of an idle channel, it will send frame immediately to that channel. In case when the channel is found busy, it will wait for the random time and again sense for the state of the station whether idle or busy. In this method, the station does not immediately sense for the channel for only the purpose of capturing it when it detects the end of the previous transmission. The main advantage of using this method is that it reduces the chances of collision. The problem with this is that it reduces the efficiency of the network.



3) ***p persistent:***

This is the method that is used when channel has time-slots and that time-slot duration is equal to or greater than the maximum propagation delay time. When the station is ready to send the frames, it will sense the channel. If the channel found to be busy, the channel will wait for the next slot. If the channel found to be idle, it transmits the frame with probability p, thus for the left probability i.e. q which is equal to 1-p the station will wait for the beginning of the next time slot. In case, when the next slot is also found idle it will transmit or wait again with the probabilities p and q. This process is repeated until either the frame gets transmitted or another station has started transmitting.



**channel class :-** It is a discrete event used to establish a connection between

sender and the receiver. It can have two states (True : Busy, False : Idle) , depending on which a sender can acquire it for transferring packets to the receiver. It is also responsible for maintaining the count of successful counts and collision counts. The receiver is present as part of the channel, since the main objective of this assignment is to study the effectiveness of CSMA protocols, we can avoid creating a real receiver process and use the channel instead. Instead of transferring real packets we can simulate packet transfer using the channel.

**packet generator class :-** It is responsible for generating packets and placing them

in a queue for transfer, by inducing a certain waiting time between the launch of two packets.

**mobile class :-** It is acting as the sender over here having multiple methods defined within it in order to transmit the packets and for carrier sensing. The carrier sensing checks whether the channel is in a busy state or not and the transmit method is targeting to send the packet as soon as the channel becomes free.

*Source Code:*

import simpy

import random

offered\_load = 0

throughput = 0

throughputA = 0

throughputB = 0

throughputC = 0

numOfMobiles = int (input ("Enter the number of mobiles : "))

print ("Types of CSMA protocol ")

print ("1. p persistent\n2. 1 persistent\n3. non persistent")

choice = int(input("Enter Choice : "))

# class slotSignal: generate slot signal to Reader and tags

Count = 0

class slotSignal:

    slotEvt = 0  # slot event

    def \_\_init\_\_(self, Tslot):

        self.env = env

        self.Tslot = Tslot

        slotSignal.slotEvt = env.event()  # slot event initialization

        # schedule process

        env.process(self.run())

    def run(self):

        while True:

            # periodic slot generation

            yield self.env.timeout(self.Tslot)

            # print("slot begins at t = %4.1f" % (self.env.now))

            slotSignal.slotEvt.succeed()  # trigger event, send slot signal

            slotSignal.slotEvt = env.event()  # slot event initialization

# Packet class

class Packet:

    def \_\_init\_\_(self, env):

        self.arvTime = env.now

# on-off traffic model with Poisson distribution

# on = 10 slots, off = random slots

class PacketGenerator:

    arrivalTime = 10000.0

    def \_\_init\_\_(self, env, Ton, Que):

        self.env = env

        self.Ton = Ton

        self.Que = Que

        # schedule process

        env.process(self.run())

    def run(self):

        global offered\_load

        yield env.timeout(random.expovariate(1.0 / PacketGenerator.arrivalTime))

        while True:

            # start ON period

            interPktTime = 1

            t = 0.0

            while t < self.Ton:

                yield self.env.timeout(interPktTime)

                self.Que.append(Packet(self.env))

                t += interPktTime

                offered\_load += 1

            # start OFF period, wait random time with Poisson distribution

            yield env.timeout(random.expovariate(1.0 / PacketGenerator.arrivalTime))

class Channel:

    # define message events

    succeedMsgEvt = 0

    failMsgEvt = 0

    noReplyMsgEvt = 0

    # collision count

    colCount = 0

    # successful read count

    readCount = 0

    # channel state

    channel = False  # True : Busy, False : Idle

    def \_\_init\_\_(self, env, tSlot):

        # initialize message event

        Channel.succeedMsgEvt = env.event()

        Channel.failMsgEvt = env.event()

        Channel.noReplyMsgEvt = env.event()

        # initialize collision count

        Channel.colCount = 0

        self.env = env

        self.tSlot = tSlot  # time slot

        # schedule process

        env.process(self.run())

    def run(self):

        global Count

        Count = 0

        while True:

            # one slot passed

            yield slotSignal.slotEvt

            # receiving the packets

            # check the collision

            tEpsilon = 0.1

            yield self.env.timeout(self.tSlot - tEpsilon)

            # send the feedback 0.1 time unit before the next slot

            if Channel.colCount == 0:

                Channel.noReplyMsgEvt.succeed(value='no\_reply')

                Channel.noReplyMsgEvt = env.event()

            if Channel.colCount == 1:

                Count = Count + 1

                Channel.channel = True

                Channel.succeedMsgEvt.succeed(value='ACK')

                Channel.succeedMsgEvt = env.event()

                # print("\nACK : success at t = %4.1f\n" % self.env.now)

            elif Channel.colCount > 1:

                Channel.failMsgEvt.succeed(value='NACK')

                Channel.failMsgEvt = env.event()

                print("\nNACK : fail at t = %4.1f\n" % int(self.env.now))

                # reset collision count

                Count = Count + Channel.colCount

                Channel.colCount = 0

class Mobile:

    mobileID = 0

    probability = 0

    if choice == 1:

        probability = 1 # probability used in p-persistent CSMA system, probability \* numOfMobiles = 1

    else:

        probability = numOfMobiles #probability used in 1 persistent CSMA system

    def \_\_init\_\_(self, env):

        # slot period

        self.slotTime = 1  # set initial slot number

        # packet queue

        self.Que = []

        PacketGenerator(env, 10, self.Que)

        # set mobile ID

        self.mID = Mobile.mobileID

        Mobile.mobileID += 1

        # mobile status

        self.status = False  # True for transmitting, False for idle

        self.state = False  # True for ready to transmit, False for not ready to transmit

        self.count = 0

        self.env = env

        # schedule process

        env.process(self.run())

    def run(self):

        while True:

            yield slotSignal.slotEvt

            # have data to transmit

            if len(self.Que) > 0:

                # carrier sensing : Idle

                if not self.carrierSense():

                    if not self.state:

                        p = random.random()  # random probability [0, 1)

                        if choice  == 3:

                            p = 0

                        if p <= Mobile.probability:  # transmit

                            yield self.env.timeout(random.randint(0, 15))  # back-off 1

                            self.state = True

                            pass

                        else:

                            yield self.env.timeout(self.slotTime)  # propagation delay

                            pass

                    else:

                        Channel.colCount += 1

                        ret = yield (Channel.noReplyMsgEvt | Channel.succeedMsgEvt | Channel.failMsgEvt)

                        values\_listed = list(ret.values())  # converted to list

                        if values\_listed[0] == 'no\_reply':  # no collision

                            yield self.env.timeout(self.slotTime)

                            pass

                        elif values\_listed[0] == 'ACK':  # no collision

                            self.status = True  # change Mobile's status to transmitting

                            self.transmit()

                            pass

                        elif values\_listed[0] == 'NACK':

                            yield self.env.timeout(random.randint(0, 15))

                            # back-off 2: random time 0~15 time slots

                            # self.state = False

                            pass

                else:  # carrier sensing : Busy

                    if self.status:  # Mobile status : transmitting

                        self.transmit()

                        if self.count == 10:

                            # print("\nACK : ID = %d, success at t = %4.1f\n" % (self.mID, self.env.now))

                            self.count = 0

                            Channel.colCount = 0

                            print("\nACK : ID = %d, success at t = %4.1f\n" % (self.mID, self.env.now))

                            self.status = False  # change Mobile's status to idle

                            self.state = False

                            yield slotSignal.slotEvt

                            Channel.channel = False  # change channel's status to Idle

                            pass

                    else:

                        pass

            else:

                yield self.env.timeout(self.slotTime)

                pass

    def carrierSense(self):

        if Channel.channel:

            return True

        else:

            return False

    def transmit(self):

        global throughput

        global throughputA

        global throughputB

        global throughputC

        print("ID = %d, trasnmitting at t = %4.1f" % (self.mID, int(self.env.now)))

        del self.Que[0]

        self.count += 1

        throughput += 1

        if self.mID == 0:

            throughputA += 1

        elif self.mID == 1:

            throughputB += 1

        elif self.mID == 2:

            throughputC += 1

env = simpy.Environment()

sim\_time = 10000

slotSig = slotSignal(1)

mSet = [Mobile(env) for i in range(numOfMobiles)]

reader = Channel(env, 1)

env.run(until=sim\_time)

print("Throughput : " + str(throughput / sim\_time)) # average amount of data bits successfully transmitted per unit time

print("offered load : " + str(offered\_load / sim\_time)) # measure of the traffic compared to the channel capacity

print ("Collision Count : " + str (Count))

***Result:***

A test has been performed for 500 mobiles(nodes,senders,stations) and **throughput**(average amount of data bits successfully transmitted per unit time), **offered load**(measure of traffic compared to channel capacity) and **collision count** were observed for each of the 3 different types of CSMA persistent methods.

Simulation time: 10000

|  |  |  |  |
| --- | --- | --- | --- |
| CSMA PROTOCOLS | THROUGHPUT | OFFERED LOAD | COLLISION COUNT |
| 1 persistent | 0.5248 | 0.5264 | 5079 |
| non persistent | 0.5060 | 0.5065 | 4717 |
| p persistent | 0.4771 | 0.4780 | 4523 |

**ANALYSIS:**

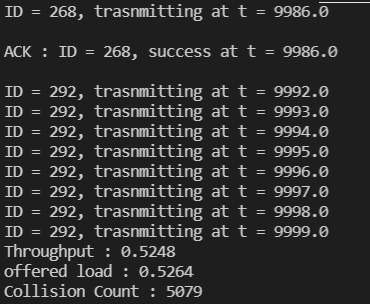
Collision Count :

1 persistent > non persistent > p persistent

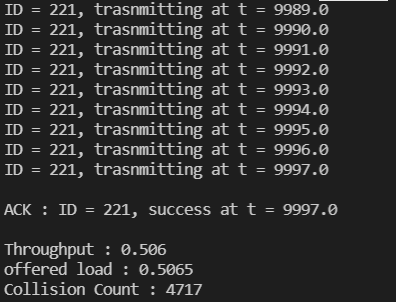
**OUTPUT:**

Output screenshots->

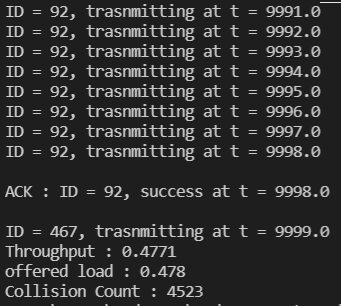
1 persistent:



non persistent:



p persistent:



------------------------