Name : Arkadeep De

Roll No. : 001910501069

Group : A3

Assignment : 3

Third year first semester.

Problem statement : 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.

Introduction

Objective of this assignment is to implement 3 CSMA techniques that are used in the MAC layer, namely one-persistent, non-persistent and p-persistent CSMA. They are used to ensure proper communication of data from multiple senders through a single shared channel. The CSMA techniques makes sure that the sender senses the channel before sending any data and effectively wait for some time if the channel is busy. This results and lesser collision and higher throughput of the entire channel. Here in this assignment we have implemented all the 3 techniques and rigorously tested them to create a detailed comparative study of their performance metrics.

Design

In this assignment the shared channel has been simulated using a contiguous memory block protected with a lock. The memory unit denotes the bandwidth of the channel. The channel has been wrapped into a class with certain necessary attributes - the total size of the channel, a list of nodes it's connected to, a lock to prevent data race during concurrency along with the memory-block itself which allows random access. Some implementation specific attributes were required, which includes a queue for denoting the current positions of all the propagating packets in the channel, a thread which is responsible for continuously carry out a 'slide' operation in the channel which simulates the propagation of data. This simulation can be imagined like a conveyor belt which keeps moving continuously and carries any object which is placed on it. Just that this conveyor belt carries object bidirectionally.

The 'object' mentioned above is nothing but the data packet from an individual sender. The nodes or machines connected to the channel is again wrapped up to a class whose instances would be different nodes. This Node class has several attributes besides majorly consisting of 2 primary threads - a receiving thread and a sending thread. The sending thread is responsible for sending any data when it has to send. This where the CSMA techniques comes into play - so we have 3 different sender algorithms or 3 different types of nodes (each with a separate sender algorithm). The sender thread senses the channel, receives collision information and subsequently resends data. The receiver thread receives all the data whose header address matches with the address of the node.

Now the important attributes of a node includes - the location of the node (w.r.t the channel). In this case the index of the shared array ; a reference to the shared channel ; size of frames ; position of the current file pointer from where it is reading ; a backoff counter which counts the number of failed attempts for the current data packet (this is required to customise the waiting time of the node), and the path to the data file.

So finally the design includes a channel shared by many instances of the Node class where each node has a specific location defined by the index of array it is placed on. Each node has a sending and receiving thread and the sending thread senses the channel and sends data according to the CSMA technique it's acquainted with. The data is then broadcasted in the Chanel by the slider thread, and during this broadcasting whenever the data of 2 different sender collides, a special data packet is again broadcasted from the point of collision denoting the energy of the channel has been peaked. The senders whose packets have collided, sensed this energy rise and understands that it has to resend the packet.

Now what each element of the channel array is made up of ? The channel here is an array of tuples. Each of these tuples consists of 4 values (can be thought of a data frame) - the actual data, the address of sender, the address of receiver and an integer which denotes the mode of the Chanel - 0 for idle, 1 for busy and 2 for collision.

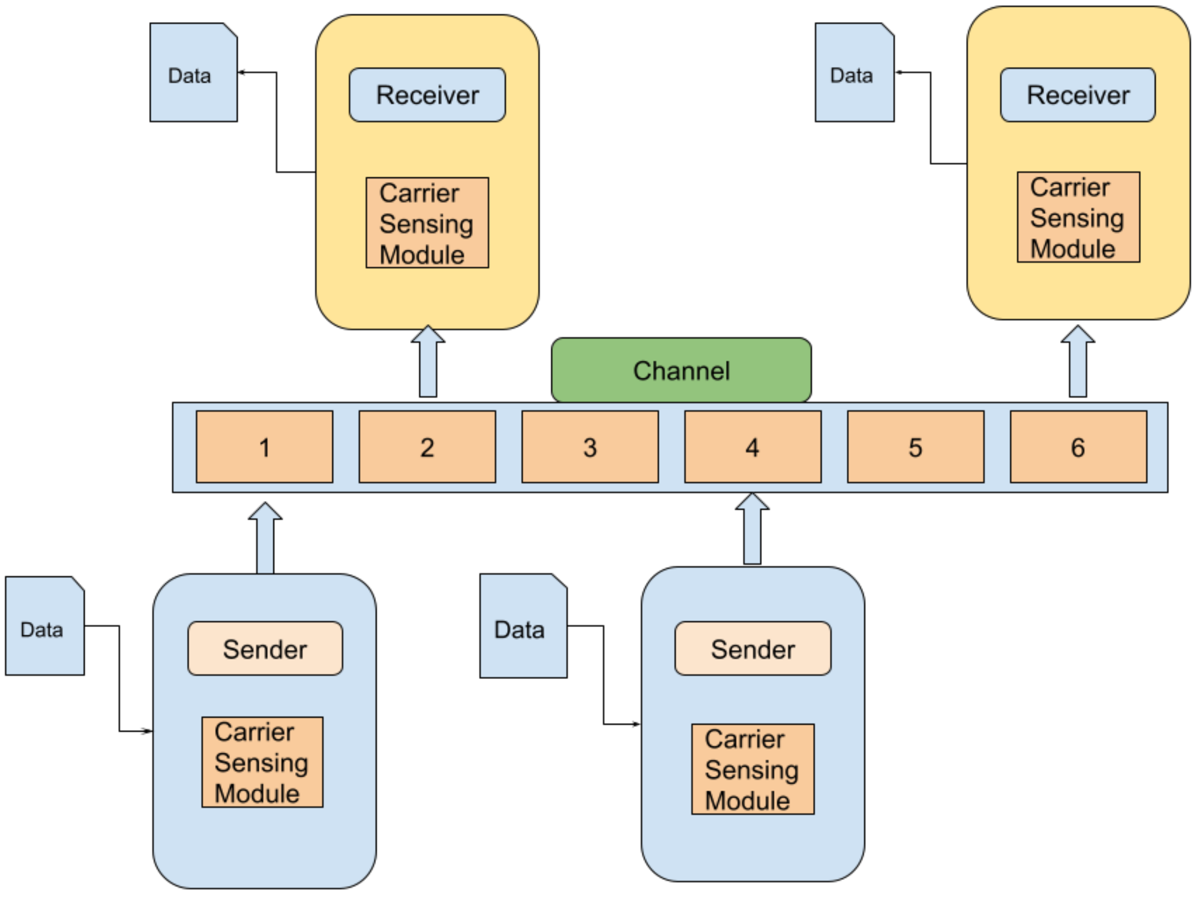
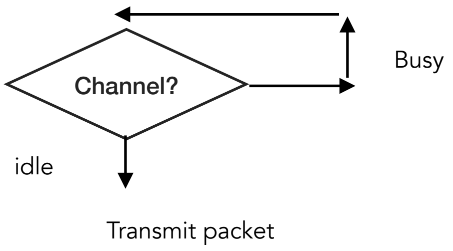


Fig : architecture summary of the implementation of CSMA techniques.

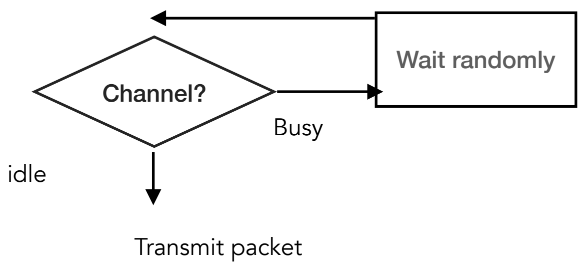
Implementation

The implementation of the above design has been done in python 3. Some relevant code snippets and pseudo codes for certain algorithms are given below. Before that, a brief explanation of each of the 3 CSMA techniques is given below.

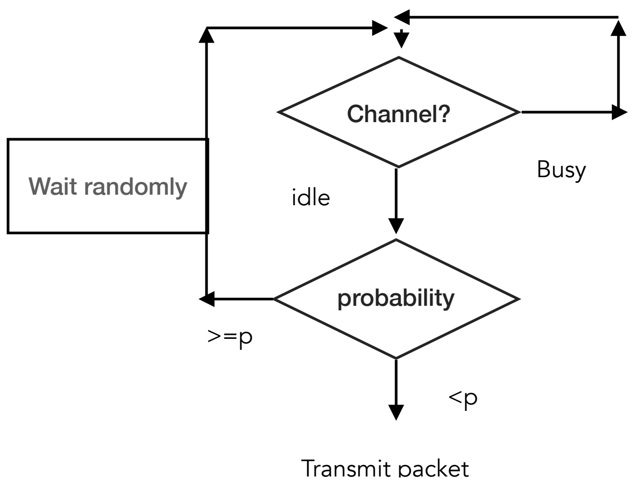
1. **1-persistent CSMA :** In 1-persistent CSMA, station continuously senses channel to check its state i.e. idle or busy so that it can transfer data. In case when channel is busy, station will wait for channel to become idle. When station finds an idle channel, it transmits frame to channel without any delay with probability 1. Due to probability 1, it is called 1-persistent CSMA. The problem with this method is that there is a huge chance of collision, as two or more stations can find channel in idle state and transmit frames at the same time. At the time when a collision occurs station has to wait for random time for channel to be idle and to start all again.



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 :** p-persistent CSMA is used when a channel has time-slots and that time-slot duration is equal to or greater than maximum propagation delay time for that channel. When station is ready to send frames, it will sense channel. If channel found to be busy, station will wait for next time-slot. But if channel is found to be idle, station transmits frame immediately with a probability p. The station thus waits for left probability i.e. q which is equal to 1-p, for beginning of next time-slot. If the next time-slot is also found idle, station transmits or waits again with probabilities p and q. This process repeats until either frame gets transmitted or another station starts transmitting.



Now the code snippets of the sender side algorithms of 3 different CSMA techniques are provided below :

1. 1 persistent CSMA

```
 def send_data(self):
        print("sender started")
        s = len(self.data)

        while self.current_pointer<=s-self.frame_size:
            packet  = self.data[self.current_pointer:self.current_pointer+self.frame_size]
            print(packet)
# tuple (1, data, senders' loc, receiver's loc)
            while self.resend_flag:
                if self.resource.channel[self.location][0]==0:
                    frame=(1, packet, self.location, self.recv_loc)
                    time.sleep((random.randint(0,self.back_off))*0.5)
                    self.resource.put_packet(frame)
                    self.resend_flag=True
                    self.back_off+=1
                else:
                    time.sleep(0.1)

            self.current_pointer+= self.frame_size
            self.resend_flag=True
            self.back_off=0
```

2. Non persistent CSMA

```
def send_data(self):
        print("sender started")
        s = len(self.data)
        self.time_taken = time.time()
        while self.current_pointer <= s - self.frame_size:
            packet = self.data[self.current_pointer:self.current_pointer + self.frame_size]
            print(packet)
```

```
  # tuple (1, data, senders' loc, receiver's loc)
            while self.resend_flag:
                if self.resource.channel[self.location][0] == 0:
                    frame = (1, packet, self.location, self.recv_loc)
                    time.sleep((random.randint(0, self.back_off)) * 1.5)
                    self.resource.put_packet(frame)
                    self.resend_flag = True
                    self.back_off += 1
                else:
                    time.sleep(random.random()*1.5)
            self.current_pointer += self.frame_size
            self.resend_flag = True
            self.back_off = 0

            time.sleep(0.1)
```

3. P - persistent CSMA

```
def send_data(self):
        print("sender started")
        s = len(self.data)
        self.time_taken = time.time()
        while self.current_pointer <= s - self.frame_size:
            packet = self.data[self.current_pointer:self.current_pointer + self.frame_size]
            print(packet)
            # tuple (1, data, senders' loc, receiver's loc)
            while self.resend_flag:
                if self.resource.channel[self.location][0] == 0:
                    frame = (1, packet, self.location, self.recv_loc)
                    if random.random() > self.P:
                        time.sleep((random.randint(0, self.back_off)) * 1.5)
                    else:
                        self.resource.put_packet(frame)
                        self.resend_flag = True
                        self.back_off += 1
                else:
                    time.sleep(0.1)
            self.current_pointer += self.frame_size
            self.resend_flag = True
            self.back_off = 0

            time.sleep(0.1)
```

Back off time

The backoff time refers to the count of unsuccessful attempts to send a particular packet. With each collision for a specific data packet, this count incremented by 1 and the random waiting time for which the sender waits before sending the packet depends on this back-off time. This done to decrease the chances of collision. The back-off count decides the upper bound of the waiting time. For the first attempt this wait is 0. So the waiting time is also 0. As attempts increase the upper bound increases and the waiting time also increases, overall delaying the sending of the packet.

The data broadcasting in channel

As we have already discussed earlier that the channel is a memory block, where each element is represented by a tuple of 4 values. No what exactly happens when a sender, or more than 1 sender puts their data into this channel ? The sender puts their data exactly on the array where array index is the location of that sender. Now we push this occupied element as the current data positions into our channel queue. Now channel takes up the lock and carries out the broadcasting operation which is basically a breadth first traversal across the channel. For a single slide call, the data packets move 1 step ahead from their previous positions.

During this broadcasting if there is any collision, it gets detected by a condition that the data has not reached the receiver and has no void space to traverse through. This means that it has faced some other data packets in the channel and hence a high energy signal starts getting propagated from the index of collision and it reaches the respective senders.

The implementation of the sliding algorithm is given below.

```
def slide(self):
        print("sliding")
        with self.lock:
            # level order traversal in the channel
            n=self.q.qsize()

             while n>0:
                i = self.q.get()
                if i-1>=0:
                    #check if empty
                    if self.channel[i-1][2]==0:
                        self.channel[i-1] = self.channel[i]
                        self.q.put(i-1)
                    elif self.channel[i-1][2]!=self.channel[i][2]:
                        #collision occurred
                         self.table[self.channel[i][2]].resend_flag = True
                         self.table[self.channel[i-1][2]].resend_flag = True
                         self.channel = [(0,0,0,0)]*self.size
                         self.q = queue.Queue()
                         break

                if i+1<self.size:
                    #check if empty
                    if self.channel[i+1][2]==0:
                        self.q.put(i+1)
                        self.channel[i+1]=self.channel[i]
                    elif self.channel[i+1][2]!=self.channel[i][2]:
                        #collision occurred
                        self.table[self.channel[i][2]].resend_flag = True
                        self.table[self.channel[i+1][2]].resend_flag = True
                        self.channel = [(0,0,0,0)]*self.size
                        self.q = queue.Queue()
                        break

                n=n-1
```

A visual representation of the data propagation and collision detection is shown below.

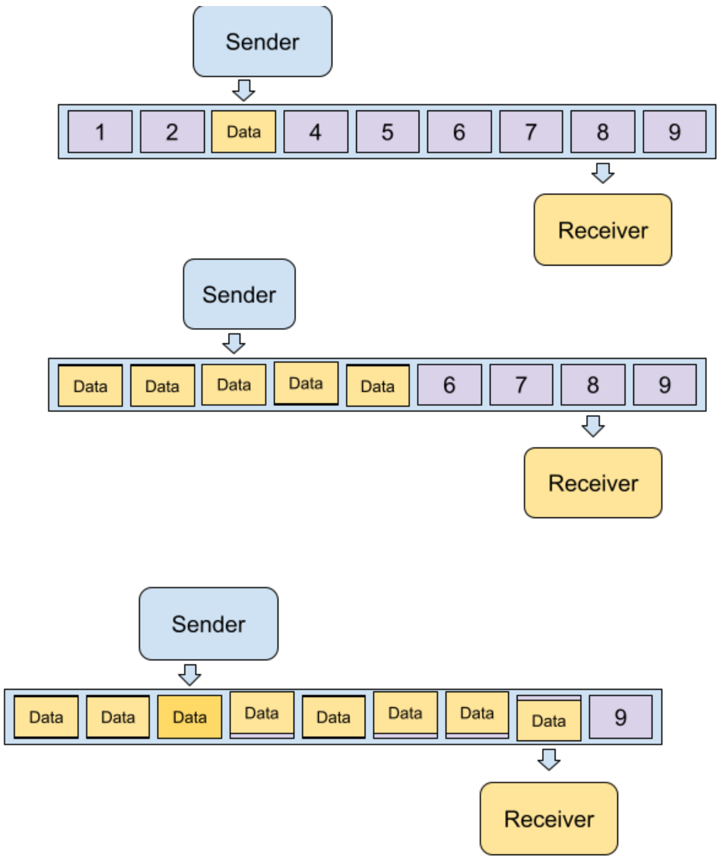


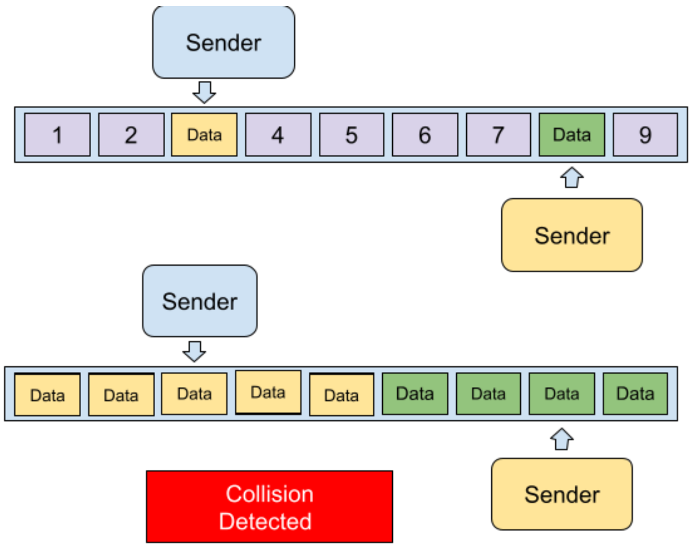
Fig : channel scenario where data is correctly received

Input output :

The overall simulation does not expect any input from user. It starts executing with the execution of the program. The main has ‘n’ number of nodes which can both behave as sender and receiver. The nodes are initialised with its location on the channel, file to read data from, state of sender or receiver, the address of receiver to which it will send data to and the frame characteristics.

The channel is initialised with its total size. The list of nodes gets added to the shared resource when individual nodes are initialised. As soon as the nodes are added, the atomic sliding operation starts executing.

The output of this program is the performance metrics - individual times, total execution time, and number of collisions. As a peripheral effect we get a new written file from each receiver which is used to verify if the data has been actually received correctly.



Test Cases

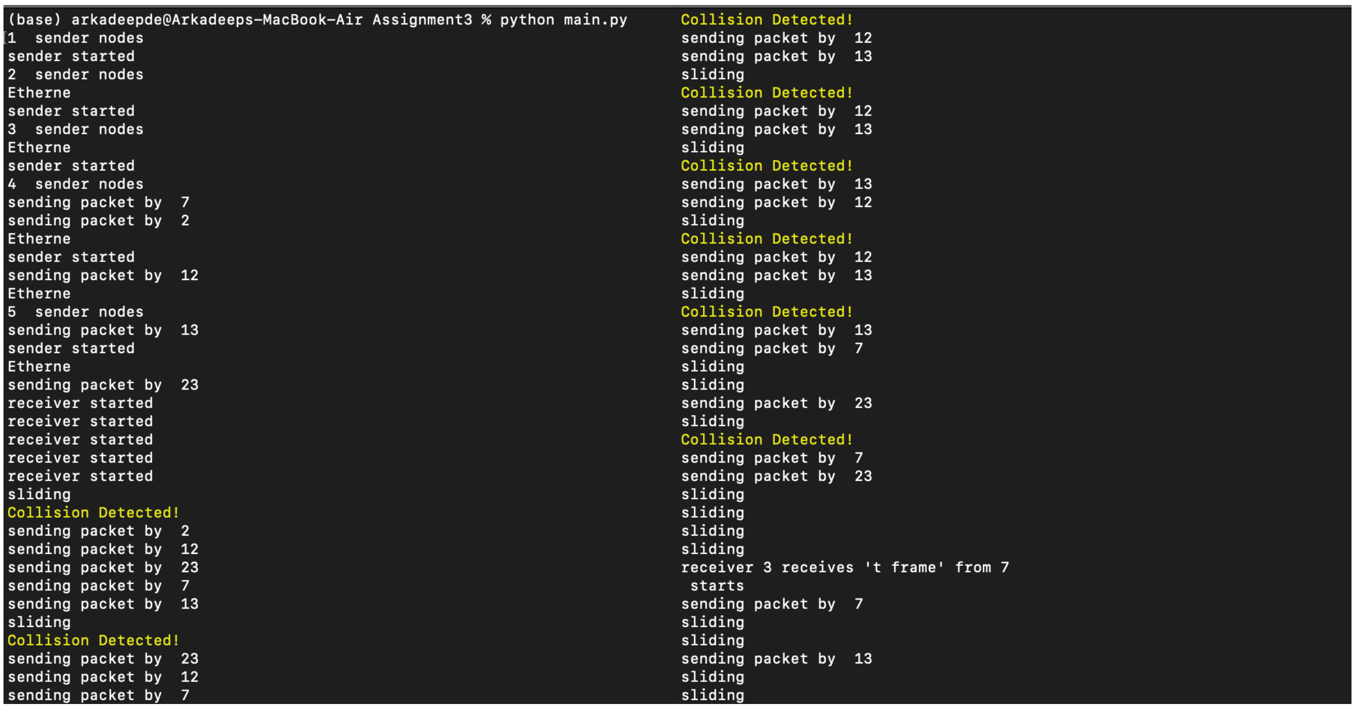
We did not have any specific test cases to try this simulation so what we decided was to try the channel on various attributes that we have already considered - like the size of the channel and location of the nodes, the number of across connections, the amount of waiting time, the size of data packet and many other minor variations. What was found - all of these factors have their own impact on the simulation but its very difficult of define a crisp relation on how they vary each other.

A few obvious observations include - if we increase number of sender-receiver couples keeping the channel size intact, number of collisions and time taken increases.

If the channel size is increased the total execution time definitely increases.
If the distance between sender receiver couples increase in total, the average completion time also increases.

Also as a vast observation, if the number of intersecting sender receiver pair increases, the collisions are bound to increase. Intersecting sender receiver pair is like (sender at 1 , sender at 7, receiver at 9, receiver at 15). So a lot of things depend on how the nodes are oriented on the channel. To nullify this bias we have taken all the comparable metrics in a fixed standard situation for all the 3 techniques.

Fig : A glimpse of the channel broadcasting and the output of our program



```
(base) arkadeepde@Arkadeeps-MacBook-Air Assignment3 % python main.py
1  sender nodes
sender started
2  sender nodes
Etherne
sender started
3  sender nodes
Etherne
sender started
4  sender nodes
sending packet by  7
sending packet by  2
Etherne
sender started
sending packet by  12
Etherne
5  sender nodes
sending packet by  13
sender started
Etherne
sending packet by  23
receiver started
receiver started
receiver started
receiver started
receiver started
sliding
Collision Detected!
sending packet by  2
sending packet by  12
sending packet by  23
sending packet by  7
sending packet by  13
sliding
Collision Detected!
sending packet by  23
sending packet by  12
sending packet by  7
```

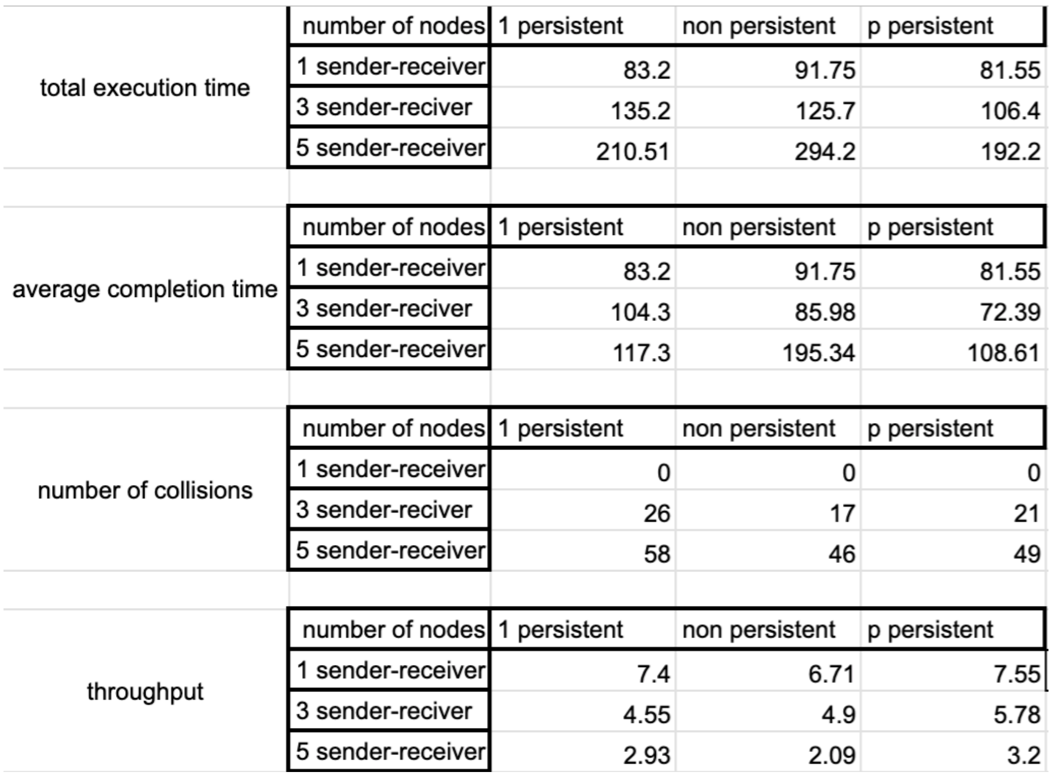
```
Collision Detected!
sending packet by  12
sending packet by  13
sliding
Collision Detected!
sending packet by  12
sending packet by  13
sliding
Collision Detected!
sending packet by  13
sending packet by  12
sliding
Collision Detected!
sending packet by  12
sending packet by  13
sliding
Collision Detected!
sending packet by  13
sending packet by  7
sliding
sliding
sending packet by  23
sliding
Collision Detected!
sending packet by  7
sending packet by  23
sliding
sliding
sliding
sliding
receiver 3 receives 't frame' from 7
 starts
sending packet by  7
sliding
sliding
sending packet by  13
sliding
sliding
```

Results

As we have already mentioned the performance metrics that we have used to compare the 3 CSMA techniques - the total channel time, the average completion time per sender and the number of collisions along with efficiency and throughput . We have taken the readings of all these metrics separately for the 3 CSMA methods in a standard situation (same for the 3 methods) to compare their overall utilisation. The table given below shows the numeric readings of all the atoned metrics. All the times mentioned are in seconds.

Obviously for 1 pair of sender-receiver the readings replicate the ideal situation with 0 number of collisions. With respect to this ideal situation we have calculated the efficiency and the throughput.

The visual representation of these metrics are also provided in terms of charts so that trend if present could be easily



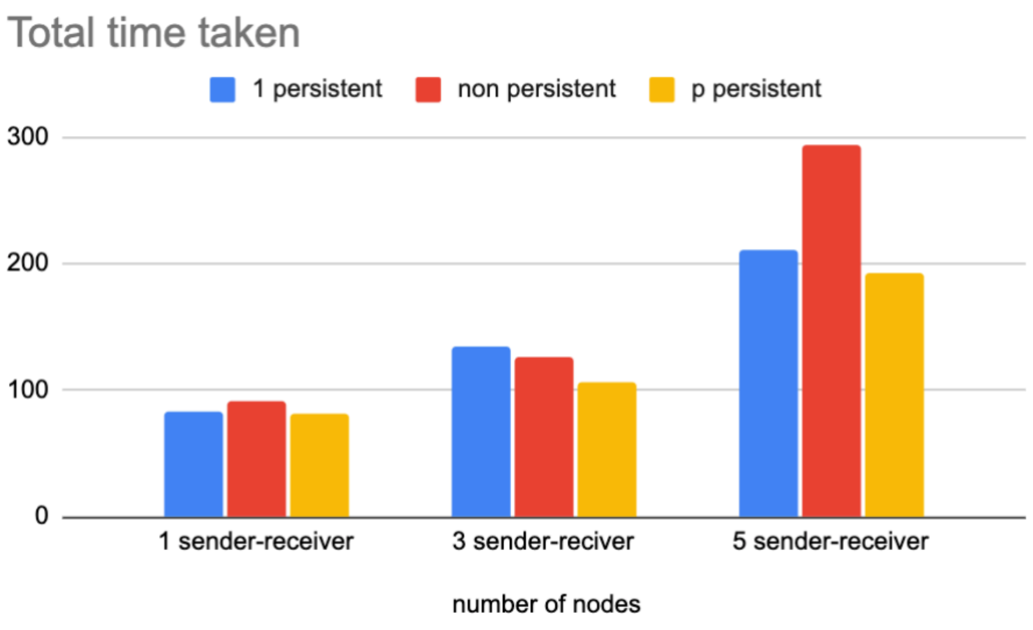
	number of nodes	1 persistent	non persistent	p persistent
total execution time	1 sender-receiver	83.2	91.75	81.55
	3 sender-reciver	135.2	125.7	106.4
	5 sender-receiver	210.51	294.2	192.2

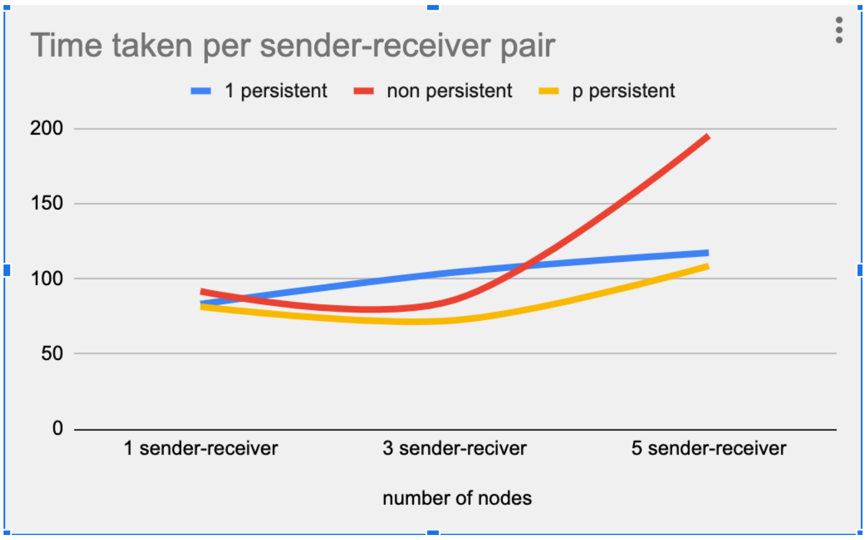
	number of nodes	1 persistent	non persistent	p persistent
average completion time	1 sender-receiver	83.2	91.75	81.55
	3 sender-reciver	104.3	85.98	72.39
	5 sender-receiver	117.3	195.34	108.61

	number of nodes	1 persistent	non persistent	p persistent
number of collisions	1 sender-receiver	0	0	0
	3 sender-reciver	26	17	21
	5 sender-receiver	58	46	49

	number of nodes	1 persistent	non persistent	p persistent
throughput	1 sender-receiver	7.4	6.71	7.55
	3 sender-reciver	4.55	4.9	5.78
	5 sender-receiver	2.93	2.09	3.2

found. We have taken the time readings for 3 different situations - in a channel size of 30, we took the readings in case of 1 sender-receiver pair, 3 pairs and 5 pairs.





We observed that the total time taken for non persistent CSMA was maximum among the three, followed by 1 persistent and then p persistent took the least time in all the cases.

Similar trend is found in the average completion time of the senders, as the non persistent took a lot of time in the long run.

Observing the number of collisions in the 3 CSMA techniques, for a single pair it is 0 as it should be. The non persistent CSMA performs best in this case. Followed by p persistent and then 1 persistent the worst performer.

Throughput was calculated as efficiency times bandwidth where the ideal time (0 collisions) was compared against actual time taken and then multiplied to the number of bits actually sent. Here we observe that 1 persistent and non persistent at times compete each other based on situation but p-persistent always has a better throughput compared to the other 2 techniques.

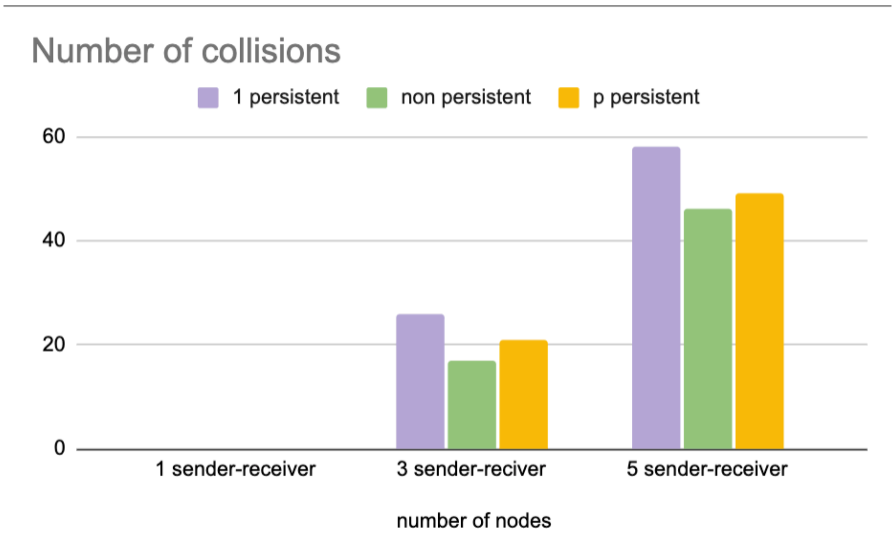
Another experiment that appeared on the way was to observe how the value of p in p-persistent actually affects the efficiency of the system.

How the value of p can eventually tilt the simulation towards a 1 persistent or a non persistent method is what the observation objected to see.

The observations from this above experiment is also provided below.

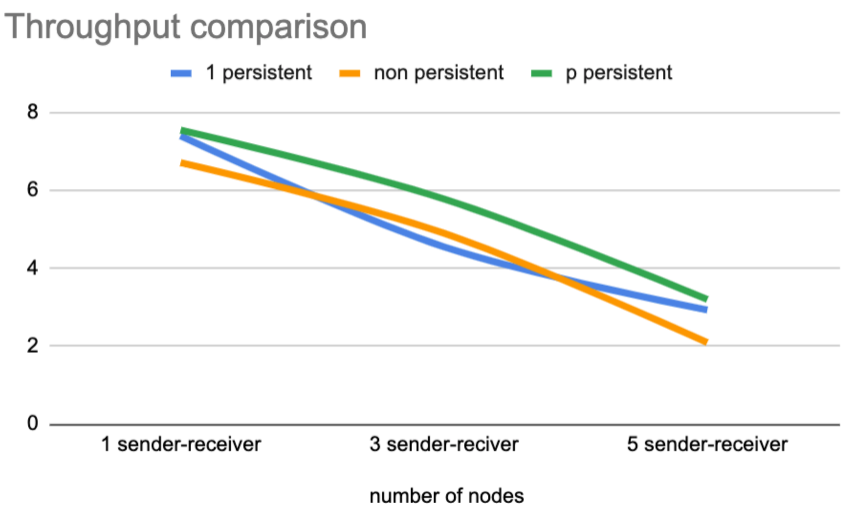
Analysis

From the observations we got there are certain things we can infer about the CSMA techniques. The non persistent CSMA takes longer time than the other two because it has a waiting slot. It senses the channel only after certain intervals and thus even if the channel becomes free between that interval the sender will not be aware of that.

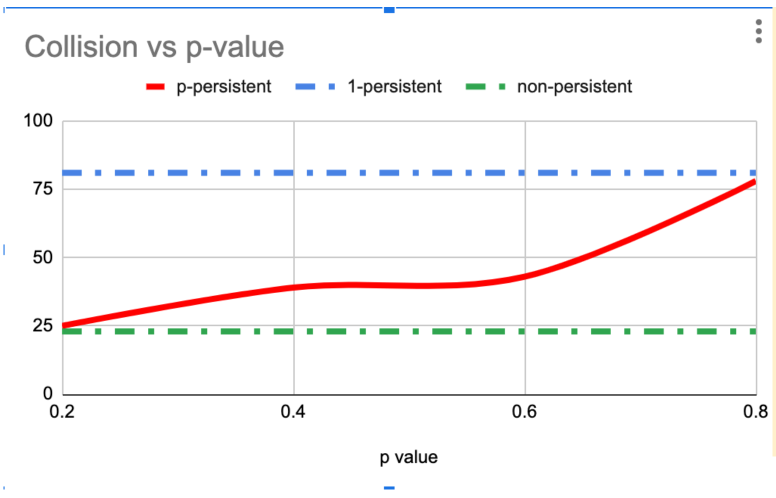


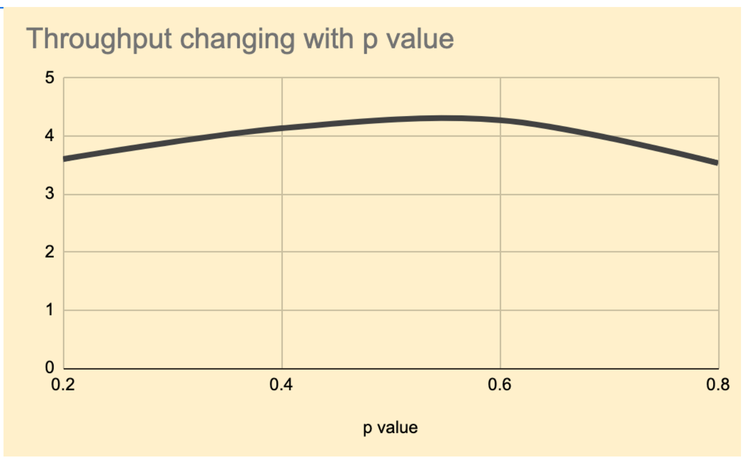
So theoretically if there is an interval where the channel is idle and none of the senders are sensing it, then that's a stale period and the time is wasted. But due to a random interval it decreases the chances of collision which we can clearly observe from the readings that we have taken. But with decreased number of collision comes a drawback of more channel time in total.

In 1 - persistent CSMA the sender continuously senses the channel and whenever found idle transmits a packet. So here multiple senders can sense the channel idle simultaneously and thus there is a huge chance of collisions. This is why the collision count is maximum in 1 persistent CSMA.



P- persistent CSMA takes the advantages from both 1 persistent and non persistent CSMA. It takes the advantage of continuously sensing the channel as in 1-persistent but whenever it senses the channel idle it sends packet with a probability p, that is not every time - which results in decreasing the number of collisions. So it has the pros of both random waiting and also decreasing





the time waste. We also have shown how changing the value of p can converge the simulation either towards 1 persistent or non persistent.

Comments

In this assignment we have compared the performance of 3 CSMA techniques through various metrics and we also analysed the reasons for which such performances are observed. Further to improve the observation and objective of the assignment we could have done some implementation related modifications to align the simulation more towards reality. For example we could utilise the SFD and PREAMBLE to their actual usages and then we could also make the senders and receivers more dynamic so that they could change their receiver within the simulation, one sender could send to multiple receivers and one receiver could receiver from multiple senders. With this improvements the scenario would replicate the real situation very closely and make the comparison closer to ground truth.