ECE 358

Winter 2020

Project 2: CSMA/CD Performance Evaluation

Prepared By:

Chandrakiran Subramaniam (20649890)

Teresa Cao (20650034)

Table of Contents

1. PROJECT AND CODE DESCRIPTION

INTRODUCTION:

The protocol (CSMA/CD) is implemented using discrete event simulation which considers the (whole) system as a discrete sequence of events. The simulation clock (T=1000) keeps track of the current simulation time. The packet generation at the nodes and transmission time of the packets are all tracked by their respective timer variable. At every event, the time value is updated by adding the appropriate delay. The simulation clock helps in regulating the total simulation time of the algorithm. The simulation (persistent and non-persistent) has been implemented using Python.

CODE ANALYSIS:

1. **class Packet** (Packet object):

```
#packet object in packet queue
class Packet:
        def __init__(self, time):
                self.packetTime = time
```

   This class acts as an object for our Packet queue. The constructor initializes the packet time to the value passed in the time parameter.

2. **def get_exp_rv** (Random Variable Generator):

```
#random variable generator
def get_exp_rv(rate):
    return (-1/float(rate))*math.log(1-random.random())
```

This method returns an exponentially distributed random number using a random number. This method is used to generate a random packet time.

3. **def setup_buffers** (An array of nodes/hosts for our LAN):

```
#generate an array of nodes where each node has a deque of packets
def setup_buffers(N, A, T):
        nodeBuffers = []
        for i in range(N):
                nodeBuffers.append(collections.deque())

                #generate random packet time
                packetTime = get_exp_rv(A)
                while packetTime <= T:
                        #create packet with random time
                        packet = Packet(packetTime)
                        #add packet into the queue
                        nodeBuffers[i].append(packet)
                        packetTime += get_exp_rv(A)
        return nodeBuffers
```

This method returns an array of nodes where each node has a deque of packets. Note that the last packet (for each node) will have an assigned time less than or equal to our simulation time (packetTime <=T).

4. **def get_earliest_transmitter** (Returns nodeID having the smallest time in packet queue):

```
#return node index which has the smallest packet time to start transmitting
def get_earliest_transmitter(buffers):
        transmitterID = -1

        for i in range(len(buffers)):
                if len(buffers[i]) == 0:
                        continue
                if transmitterID == -1:
                        transmitterID = i
                elif buffers[i][0].packetTime < buffers[transmitterID][0].packetTime:
                        transmitterID = i

        return transmitterID
```

This method returns the nodeID of the node having the smallest packet time (using linear search). In other words, when our simulation begins, this node will start transmitting packets first.

5. **def get_exp_backoff** (Returns a backoff time when a collision happens)

```
#return backoff time when a collision happens
def get_exp_backoff(numCollisions, R):
        #used formula from manual
        k = random.randint(0,2**numCollisions - 1)
        return k* 512/float(R)
```

This method, when called, returns a backoff time for when a collision happens. The formula used is taken from the lab manual. Note that the return value is in bit time.

6. **def delay_packets** (Delays/reschedules packets' packetTime):

```
#return delay time to input time
def delay_packets(packetQueue, delayUntil):
        for packet in packetQueue:
                #if there are packets schedule to start in the middle of transmitting a packet before them
                if packet.packetTime < delayUntil:
                        #reschedule them to start after the transmission is finished
                        packet.packetTime = delayUntil
                else:
                        break
```

This method is called when there are packets scheduled to start in the middle of a packet transmission. These packets are rescheduled/delayed (given by delayUntil) to start transmission after the ongoing transmission has finished.

7. **def main** (CSMA/CD simulation body):

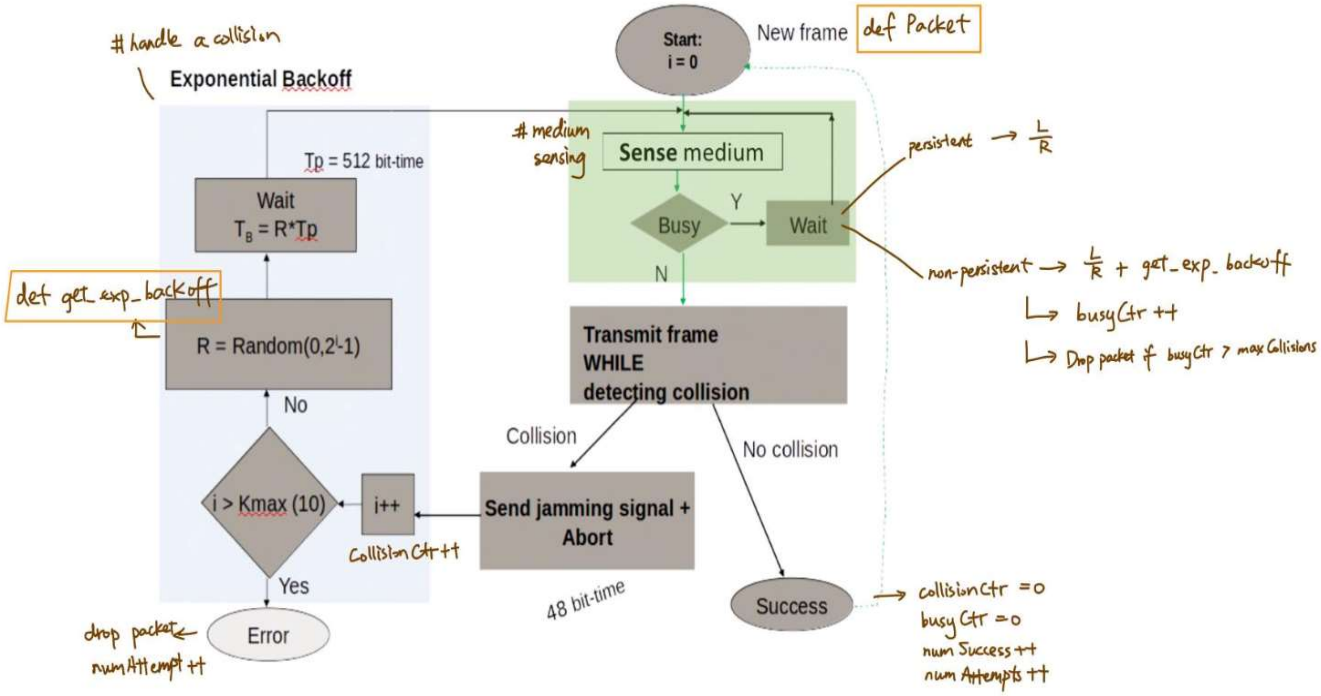
```
#main function with all necessary handling of transmitting
def main(T,N,A,R,L,D,S,persistent):
        maxCollisions = 10

        #setup nodes
        buffers = setup_buffers(N, A, T)
        collisionCtrs = [0] * N
        busyCtrs = [0] * N

        #setup overall attempts
        numAttempts = 0
        numSuccesses = 0
```

The main method is the heart of our CSMA/CD simulation. The CSMA/CD algorithm is realized in this part using the helper functions/methods discussed above. The above code snippet shows *only* the variable initialization. The code that follows will be explored in depth in the design section of this report.

2. SYSTEM DESIGN AND IMPLEMENTATION

SYSTEM DESIGN:

The CSMA/CD protocol design is as follows (flowchart has been edited to address code variables):



The simulator progresses in a sequential manner, following the above flowchart, using one thread (single-threaded) for both persistent and non-persistent detection. Our simulation is programmed to process and transmit packets of all nodes. Console outputs include network statistics like efficiency and throughput. Inputs to the simulator are as follows:

```
T = 1000
D = 10
C = 3*pow(10,8)
S = (2/float(3))*C

N = 20
for N in range(20, 101, 20):
    for A in [7,10,20]:
        R = 1*pow(10,6)
        L = 1500
        main(T,N,A,R,L,D,S,True)

for N in range(20, 101, 20):
    for A in [7,10,20]:
        R = 1*pow(10,6)
        L = 1500
        main(T,N,A,R,L,D,S,False)
```

The CSMA/CD protocol will be simulated in two modes (boolean *persistent* parameter in main). In the persistent mode (*persistent* = *True*), all users listen to the line prior to transmitting. If traffic is sensed, they wait. When the line becomes free, packets are transmitted immediately. In the non-persistent mode (*persistent* = *False*), a user is ready to send data senses the line and commences transmission if free. If the line is busy, the user does not continue sensing, but backs off for a random time before sensing it again, and so on.

IMPLEMENTATION:

The simulation is implemented in the following phases:
Phase 1: ***Recognize the transmitter node and detect for collision in the remaining nodes.***
Phase 2: ***A collision has occurred! How do we handle it?***
Phase 3: ***No collision has occurred! How do we handle it?***
Phase 4: ***Calculate desired network statistics***

As per the manual, some assumptions have been made for this simulation. It is worth to note them:

1. You are not required to implement the function of the jamming signal in the case of a collision.
2. Once a collision happens, all the nodes will discover it immediately.
3. All the nodes are equally spaced on the bus/channel, i.e., the distance between any two adjacent nodes is the same.
4. The random wait time before sensing the bus/channel in the non-persistent CSMA/CD protocol is the same as the wait time for the exponential backoff.
5. You may use any of the following programming languages: C/C++, Python, Java.

Let us now walk through each phase of the implementation (concluded by code snippets of our simulator). Important variables are italicized followed by a short description.

Phase 1: ***Recognize the transmitter node and detect for collision in the remaining nodes.***

- *transmitterID:* The transmitting node ID/position
- *busyCtrs:* Non-persistent sensing counter
- *firstBitTime:* Time for the first bit of packet from transmitter to arrive at any given node

The *transmitterID* is set by calling the helper method get_earliest_transmitter() to get the node index that gets to transmit first. Keep note of the non-persistent sensing counter *busyCtrs*. Once we get our transmitter ID, we enter the collision detection loop. In this loop, the time needed for the first bit from the transmitter to arrive at that respective node is calculated (via *firstBitTime*). But what causes collision? If the node starts to transmit before the first bit arrives, from the transmitter (*buffers[nodeID][0].packetTime* <= *firstBitTime*), there is a collision!

```
while True:
        #get the node index that gets to transmit first
        transmitterID = get_earliest_transmitter(buffers)
        #if the node is not found stop the simulation
        if transmitterID == -1:
                break
        #if the simulation time reaches T, stop the simulation
        elif buffers[transmitterID][0].packetTime > T:
                break
        #reset when a node becomes the transmitter
        if not persistent:
                busyCtrs[transmitterID] = 0

        maxPropDelay = -1
        #detect collision loop
        for nodeID in range(len(buffers)):
                #skip the transmitting node as it can't collide with itself
                if nodeID == transmitterID:
                        continue
                #skip the node if all the packets on this node is transmitted/dropped
                elif len(buffers[nodeID]) == 0:
                        continue
                #calculate the time need for the first bit from the transmitter to arrive at this node
                propDelay = abs(nodeID - transmitterID) * D / float(S)
                firstBitTime = buffers[transmitterID][0].packetTime + propDelay

                #if the node starts before the first bit arrives, there is a collision
                if buffers[nodeID][0].packetTime <= firstBitTime:
                        #the maximum time needed before the transmitter can successfully transmit the packet
                        maxPropDelay = max(maxPropDelay, propDelay)
```

Phase 2: ***A collision has occurred! How do we handle it?***

- *maxCollisions:* A cap on the maximum number of collisions (10)
- *delayUntil:* Holds the return value of our back off method *get_exp_backoff()*
- *collisionCtrs[ID]:* Holds the number of collisions at given node/transmitter ID

A collision is detected. Let us address the nodes in our network (call it the node domain). We loop through all our nodes in the network and increment the collision counter for the nodes where a collision is experienced. Our non-persistent sensing counter is reset. We also need to check if the packet, for that given node, has exceeded our *maxCollisions*. If it has, we drop the packet and reset the collision counter on that node. If not, we follow the algorithm and perform an exponential back off (*delayUntil*). The helper method *delay_packets()* is called to delay all remaining packets in the queue by the back off time. We should also increment our attempts counter.

Now let us address our transmitter. We also need to delay our transmitter packet times. This delay is given by the sum of the current packet time + a back off time + propagation delay. Let's not forget to increment our attempts counter.

```
#handle a collision
if maxPropDelay != -1:
        for nodeID in range(len(buffers)):
                if nodeID == transmitterID:
                        continue
                elif len(buffers[nodeID]) == 0:
                        continue
                propDelay = abs(nodeID - transmitterID) * D / float(S)
                firstBitTime = buffers[transmitterID][0].packetTime + propDelay
                if buffers[nodeID][0].packetTime <= firstBitTime:
                        #increase the number of collisions on this node
                        collisionCtrs[nodeID] += 1
                        #reset the non persistent sensing counter
                        if not persistent:
                                busyCtrs[nodeID] = 0
                        #drop the packet if it has collided too many times
                        if collisionCtrs[nodeID] > maxCollisions:
                                buffers[nodeID].popleft()
                                #reset collision counter on this node
                                collisionCtrs[nodeID] = 0
                        else:
                                #backoff after a collision (other nodes)
                                delayUntil = buffers[nodeID][0].packetTime + get_exp_backoff(collisionCtrs[nodeID], R)
                                delay_packets(buffers[nodeID], delayUntil)
                        numAttempts += 1
        # backoff after a collision (transmitter)
        transmitterDelayUntil = buffers[transmitterID][0].packetTime + get_exp_backoff(collisionCtrs[transmitterID], R) + maxPropDelay
        delay_packets(buffers[transmitterID], transmitterDelayUntil)
        numAttempts += 1
```

Phase 3: ***No collision has occurred! How do we handle it?***

- *lastBitTime:* Time for the last bit of packet from transmitter to arrive at any given node
- *nonPersistentWaitime:* Back off time calculated in the non-persistent mode
- *lastBitSentTime:* Time (at transmitter side) when last bit of packet is on the link

A collision has not been detected. We perform medium sensing and perform actions based on the operation mode, persistent or non-persistent. Note that we need to calculate the time taken for the last bit to arrive at given nodes in the network to ensure successful transmission of a packet.

Let's address the nodes in our network (the node domain) via a for loop. The *lastBitTime* is calculated and we check if the channel is busy. If it is busy and persistent, we defer the packet times of that node by *lastBitTime*. If the channel is busy and non-persistent, we first increment our *busyCtrs* counter. We drop the packet if it exceeds our *maxCollisions* and reset *busyCtrs*. If *maxCollisions* is not exceeded, we proceed to calculate our non-persistent backoff (*nonPersistentWaitime*) and delay all packets of that node by this back off time.

We increment our attempts and success counter after the above steps. A packet has successfully been transmitted. So, we now calculate the last bit sent time (different from *lastBitTime* as *propDelay* is not included) and delay our remaining transmitter packets' times by this amount. The transmitted packet is popped, and collision counter is reset on the transmitter.

```
else:
        #medium sensing
        for nodeID in range(len(buffers)):
                if nodeID == transmitterID:
                        continue
                elif len(buffers[nodeID]) == 0:
                        continue
                propDelay = abs(nodeID - transmitterID) * D / float(S)
                firstBitTime = buffers[transmitterID][0].packetTime + propDelay
                lastBitTime = firstBitTime + L/float(R)
                #check if the channel is busy
                if firstBitTime <= buffers[nodeID][0].packetTime <= lastBitTime:
                        #defer after busy sense (persistent)
                        if persistent:
                                delay_packets(buffers[nodeID], lastBitTime)
                        else:
                                #defer after busy sense(non persistent)
                                while firstBitTime <= buffers[nodeID][0].packetTime <= lastBitTime:
                                        busyCtrs[nodeID] += 1
                                        #drop the packet if
                                        if busyCtrs[nodeID] > maxCollisions:
                                                buffers[nodeID].popleft()
                                                #reset when it has already pass the medium sensing part
                                                busyCtrs[nodeID] = 0
                                                #if we popped the last packet
                                                if len(buffers[nodeID]) == 0:
                                                        break
                                        else:
                                                #delay exponential backoff
                                                nonPersistentWaitime = buffers[nodeID][0].packetTime + get_exp_backoff(busyCtrs[nodeID], R)
                                                delay_packets(buffers[nodeID], nonPersistentWaitime)

        numAttempts += 1
        numSuccesses += 1

        #delay packets in transmitter queue
        lastBitSentTime = buffers[transmitterID][0].packetTime + L/float(R)
        delay_packets(buffers[transmitterID], lastBitSentTime)
        #reset collision counter on transmitter
        collisionCtrs[transmitterID] = 0
        #remove the transmitted packet
        buffers[transmitterID].popleft()
```

Phase 4: ***Calculate desired network statistics***

- *efficiency:* Efficiency of the protocol
- *throughput:* Throughput of the protocol

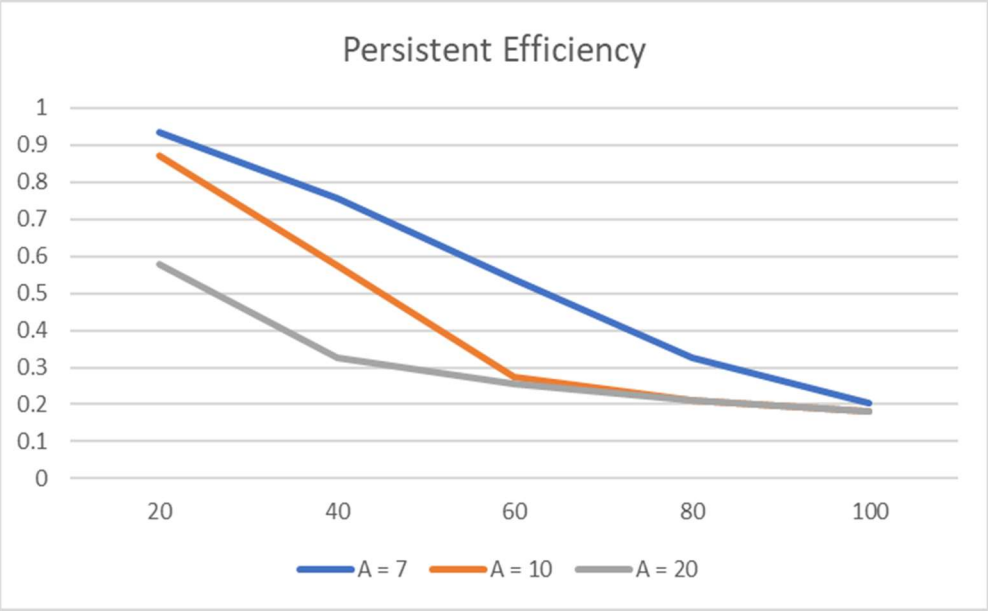
Note that if the simulation time reaches T (1000 seconds), we exit our while loop. That is, we end our simulation. The network statistics are then calculated and displayed to the console.

```
efficiency = numSuccesses / float(numAttempts)
throughput = numSuccesses*L / float(T)*pow(10,-6)

print("Number of Nodes: ", N,  "A = ", A, "persistent: ", persistent)
print("Efficiency= ", efficiency)
print("Throughput= ", throughput)
```

3. SIMULATION RESULTS: Persistent CSMA/CD

EFFICIENCY:

Efficiency			
N	**A = 7**	**A = 10**	**A = 20**
20	0.93350681	0.87021131	0.57858964
40	0.7546114	0.57497576	0.32496166
60	0.53577398	0.27524707	0.25421617
80	0.32735365	0.20999205	0.2111692
100	0.20256775	0.18173425	0.18184348



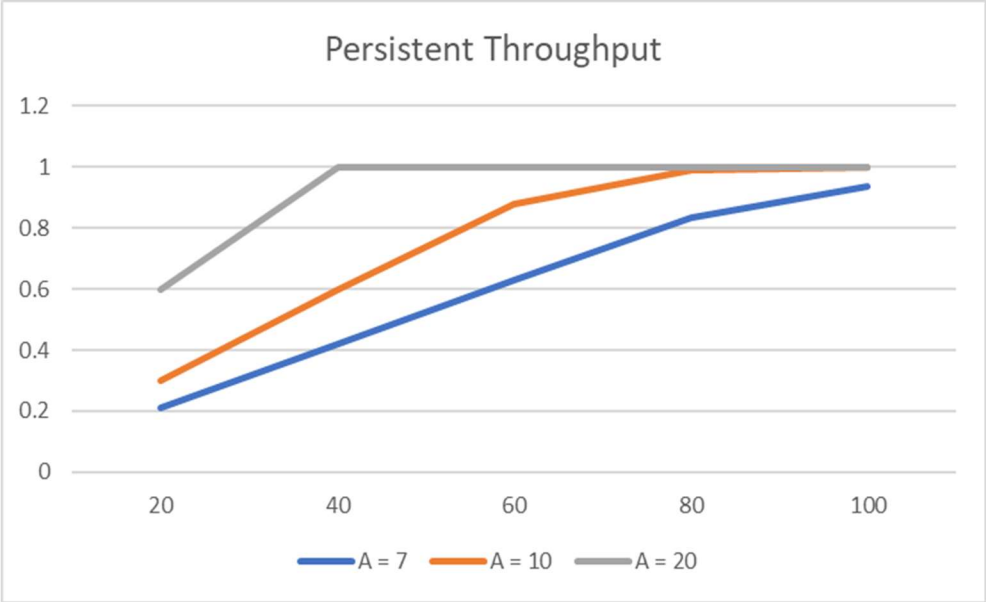
The above graph shows that for the persistent mode, CSMA/CD has a greater efficiency it is applied to fewer nodes. As the number of nodes increases, the probability of having a collision increases and the number of successful transmits decreases. When the packet rate increases the efficiency becomes lower as there are more packets scheduled to transmit and it is more likely for them to collide. But when the number of nodes is large (around 100), simulations at different packet rates come to a similar low efficiency(around 20%).

All packets have the same back off time so that even though more packets are in the queue, the numbers of successfully transmitted packets with different packet rates don't vary by very much.

THROUGHPUT:

Throughput			
N	**A = 7**	**A = 10**	**A = 20**
20	0.209871	0.3001485	0.5993775
40	0.419124	0.59868	0.9982125

60	0.6298725	0.8779425	0.999624
80	0.8344125	0.9899625	0.9998115
100	0.9347175	0.999156	0.999861

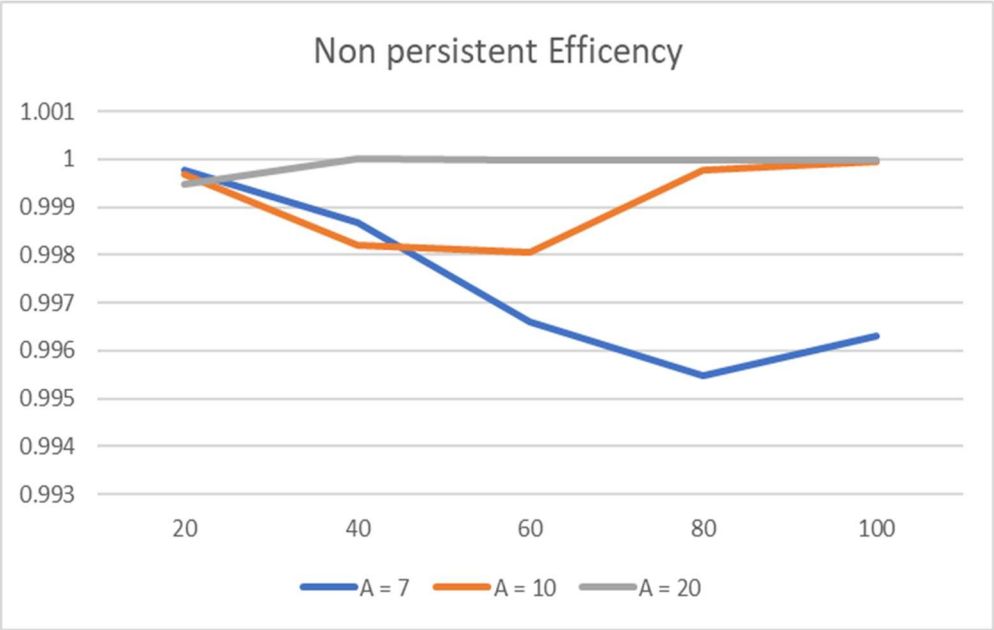


The throughput graph of the persistent CSMA/CD simulation shows that when there are more nodes, the channel is more likely to be busy for most of the time as there are more packets distributed at each node waiting for transmission. When the packet rate increases, each node is carrying more packets so that the channel is more likely to be busy for most of the time. Throughput increases when the number of nodes or packet rate increases till 1Mb/s.

4. SIMULATION RESULTS: Non-persistent CSMA/CD

EFFICIENCY:

Efficiency			
N	**A = 7**	**A = 10**	**A = 20**
20	0.99978572	0.99969019	0.99947166
40	0.99867055	0.99821062	0.99999699
60	0.99659791	0.99805102	0.99999099
80	0.99548888	0.99978643	0.99998799
100	0.99630409	0.99994892	0.99998199



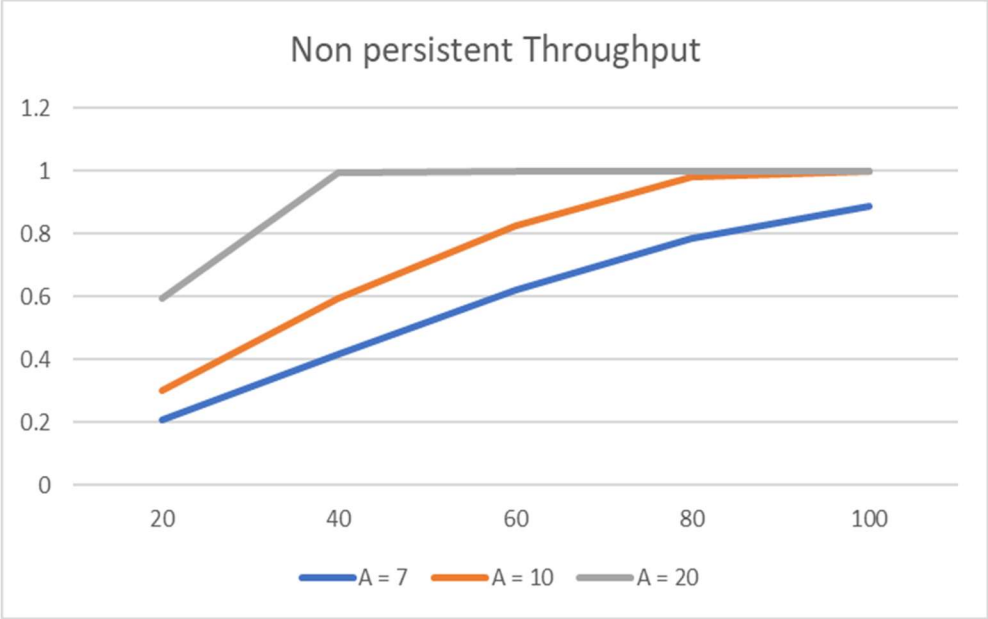
The graph above shows that the simulation in non-persistent mode has high efficiency of around 99%. The exponential back off while sensing the medium avoids more collisions and allows more packets to get transmitted upon every attempt. Each attempt of transmitting is more likely to be successful in this case. When the number of nodes increases, at high packet rates, the efficiency is not really affected.

But if the packet rate is low, its efficiency gets slightly affected by the increase in the number of nodes. More packets get dropped while sensing the medium other than getting into collisions.

THROUGHPUT:

Throughput			
N	**A = 7**	**A = 10**	**A = 20**

20	0.2099655	0.3000915	0.5958975
40	0.419163	0.595788	0.996675
60	0.621759	0.8280495	0.99999099
80	0.78516	0.983082	0.99957
100	0.8903865	0.99837	0.999702



The graph of non-persistent throughput is like the persistent throughput graph. When the number of nodes increases or the packet rate increases, the channel is more likely to be busy all the time. So that the throughput increases until it reaches the 1Mb/s.

Since the persistent and non-persistent throughput graphs are (almost) identical, we can come to the following conclusion. For low levels of traffic, persistent protocols provide best throughput. For high levels of traffic load, non-persistent protocols are by far the best!