CSMA/CD Performance Evaluation

ECE358 S20, Lab 2

7/7/2020

Arjun Bawa
20711916
(no partner)

Contents

Simulator Internals

I used a class-based approach to encapsulate the logic associated with different segments of the simulator. Common tasks that each component of the simulator needs to do at some point are generating an exponential random number and calculating exponential back-off time. I made two helper functions for this purpose.

```
# Exponential random number generator
def expn_random(rate):
    return (-(1/rate) * log(1 - random.uniform(0, 1)))
# Calculate backoff time based on transmission attempts and data rate
def expn_backoff_time(tries, rate):
    return random.uniform(0,(2**tries)-1)*512/rate
```

Packet class

```
class Packet:
    # class representing packets in each node.
    # keeps track of arrival time, number of collisions,
    # and how often the bus is sensed to be busy
    def __init__(self, time):
        self.time = time
        self.collisions = 0
        self.bus_busy_measure = 0

    def __lt__(self,other):
        return self.time < other.time
```

A **Packet** in my simulator is just a wrapper object to hold some variables associated with each packet. A **Node** keeps a queue of Packet objects, each with a certain arrival time. The number of collisions each Packet encounters is initialized to 0 and the number of times each Packet detects the **Bus** as being busy is initialized to 0 (this counter is used in the non-persistent mode of simulation).

Node class

Constructor

```
# class representing a node in the LAN
def __init__(self,_id,sim_time,arrival_rate,data_rate=1E6,max_trans_retries=10):
      self.data_rate = data_rate
      self.arrival_rate = arrival_rate
      self.sim_time = sim_time
      self.max_trans_retries = max_trans_retries
      self.queue = deque()
      self.id = _id
      self.gen_arrivals()
```

The **Node** class is responsible for the behaviors of a Node in the network. Each node is assigned an *id* to keep track of its position on the **Bus**. Each Node holds a queue of Packet objects, as well as some other parameters used for computations on the packets in the queue. The constructor initializes the Node's queue with packets.

gen_arrivals

```
# Create arrival events (based on technique from Lab 1)
def gen_arrivals(self):
      t = expn_random(self.arrival_rate)
      while t < self.sim_time:
            self.queue.append(Packet(t))
            t += expn_random(self.arrival_rate)
```

This class method generates Packets and stores them in the Node's queue using the same technique to generate packets as was used in Lab 1.

backoff

```
# Applies expn backoff delay given that a node's
# collision counter is within threshold
def backoff(self):
      pkt = self.queue[0]
      pkt.collisions+=1
      if pkt.collisions > self.max_trans_retries:
            self.drop_packet()
      else:
            pkt.time += expn_backoff_time(pkt.collisions, self.data_rate)
```

When a Node needs to undergo an exponential back-off delay this method takes care of counting the collision as well as rescheduling the packet to be transmitted using exponential back-off (and dropping the packet if the number of collisions experienced exceeds the threshold).

drop_packet

```
# Removes first packet from node's queue and
# updates arrival time of next packet
def drop_packet(self):
      if len(self.queue)<=0: return
      d = self.queue.popleft()
      if len(self.queue)>0:
            self.queue[0].time = max(d.time, self.queue[0].time)
```

When a Node needs to drop a packet from the front of its queue, this method takes care of popping the packet from the queue and updating the next packet's arrival time accordingly. This happens if a successful transmission occurs, or if the number of collisions experienced by a packet reaches the threshold.

Bus class

Constructor

```
# class representing the shared bus in the LAN
def __init__(self, persistent, num_nodes, arrival_rate, sim_time,
     data_rate=1E6, node_distance=10, prop_speed=2E8, max_trans_retries=10,
     packet_len=1500):
        self.persistent = persistent
        self.num_nodes = num_nodes
        self.arrival_rate = arrival_rate
        self.data_rate = data_rate
        self.sim_time = sim_time
        self.node_list = []
        self.current_transmitter = None
        self.current_transmitter_pkt = None
        self.max_trans_retries = max_trans_retries
        self.prop_delay = node_distance/prop_speed
        self.trans_time = packet_len/data_rate
        self.transmitted_packets = 0
        self.successes = 0
        self.init_nodes()
```

The **Bus** class represents the shared channel between all Nodes in the network. Most parameters are unchanging values accessed by different methods in the class but some are of particular interest.

- The *persistent* parameter specifies whether the simulation is 1-persistent (True) or non-persistent (False)
- The *node_list* is the list of Nodes the Bus manages on its channel
- *self.current_transmitter* is essentially a pointer to the currently transmitting node on the Bus. It is set in the *find_next_transmitter* method of the Bus
- *self.current_transmitter_pkt* is a pointer to the packet being transmitted by the currently transmitting node. When the transmitting packet is popped off the transmitting node's queue, its no longer accessible by indexing into the queue of *current_transmitter* from *node_list*, so it is important to have a reference to it

After all the class variables are initialized, a queue of Nodes are generated for the Bus by calling *init_nodes*.

init_nodes

```
# generate and initialize nodes with their packet queues
def init_nodes(self):
      for i in range(self.num_nodes):
            n = Node(i, self.sim_time, self.arrival_rate)
            self.node_list.append(n)
```

collision

```
# determine if a collision will occur if the current
# transmitting node transmits
def collision(self):
      collision = False
      for i,n in enumerate(self.node_list):
            if n is self.current_transmitter: continue
            if len(n.queue)<=0: continue
            pkt = n.queue[0]
            pd = abs(self.current_transmitter.id-i)*self.prop_delay
            if pkt.time <= self.current_transmitter_pkt.time+pd:
                self.transmitted_packets+=1
                collision = True
                n.backoff()
      # if any node experienced a collision that means they tried
      # to transmit while the bus was busy. Must apply expn backoff
      # for the current transmitting node to retry transmission
      if collision:
            self.transmitted_packets+=1
            self.current_transmitter.backoff()
      return collision
```

This method is meant to determine the occurrence of a collision when a node attempts to transmit. First it iterates through all the nodes in *node_list* that aren't the current transmitter node. In each iteration it calculates the time node *n* will see the bus as busy and checks if *n* is scheduled to transmit before that time (leading to a collision). When a collision occurs, *n* undergoes exponential back-off to reschedule its transmission (or drop its packet). If any collisions occurred when *current_transmitter* was transmitting, that transmission was lost and must be retried after telling *current_transmitter* to undergo exponential back-off.

find_next_transmitter

```
# decides next transmitting node by comparing
# arrival times of the first packets in each
# node's queue. Node with earliest arrival time
# is the current transmitting node
def find_next_transmitter(self):
      self.current_transmitter = None
      t = self.sim_time + 1
      for n in self.node_list:
           if len(n.queue)>0 and n.queue[0].time < min(t, self.sim_time):
                t = n.queue[0].time
                self.current_transmitter = n
                self.current_transmitter_pkt = n.queue[0]
      return self.current_transmitter
```

This method sets the *current_transmitter* pointer, deciding which node in the network is to transmit next. It compares the arrival times of the first packets of each Node's queue and selects the node with the minimum packet arrival time. It also sets the *current_transmitter_pkt* pointer for use in other class methods.

update_packets

```
# update packet times for all nodes
def update_packets(self):
      for i,n in enumerate(self.node_list):
            if len(n.queue) <= 0: continue
            pkt = n.queue[0]
            pd = abs(self.current_transmitter.id-i)*self.prop_delay
            start_busy = self.current_transmitter_pkt.time + pd
            end_busy = start_busy + self.trans_time
            if self.persistent: # 1-persistent case
                # if a node tries to transmit while bus is busy,
                # reschedule the transmission time to the end of the busy period
                if start_busy <= pkt.time < end_busy:
                    pkt.time = end_busy
            else: # non-persistent case
                # while the bus is detected as busy
                while start_busy <= pkt.time < end_busy:
                    # try to reschedule the node's transmission by delaying
                    # it by an exponential backoff. The backoff is determined
                    # by how many attempts were made to transmit (up till the
                    # max threshold of transmission attempts)
                    if pkt.bus_busy_measure < self.max_trans_retries:
                        pkt.bus_busy_measure=+1
                    pkt.time+=expn_backoff_time(pkt.bus_busy_measure,
                                                            self.data_rate)
                # when the bus is no longer detected to be busy, reset the
                # counter for transmission attempts
                pkt.bus_busy_measure=0
```

This method iterates through each Node on the Bus and updates their packet's times according to some rules depending on whether the simulation is 1-persistent or non-persistent. First in each iteration, the method finds the interval of time that *n* sees the bus as busy according to its distance away from *current_transmitter* and the *current_transmitter_pkt* that's being transmitted.

- If in 1-persistent mode, the method reschedules *n*'s first packet's transmission to the end of the *current_transmitter*'s transmission if *n*'s first packet's transmission time is set to a time during the busy interval

- If in non-persistent mode, the method reschedules *n*'s first packet's transmission time by applying exponential back-off while the bus is detected to be busy. It keeps applying exponential back-off until the bus is no longer busy (relative to the new time of the packet). The exponential back-off time keeps increasing according to how many times the bus is detected to be busy, up till the max threshold. When the bus is no longer detected to be busy, the counter for measuring if the bus is busy is reset

transmitted

```
# call when packet successfully transmitted
def transmitted(self):
      self.transmitted_packets+=1
      self.successes+=1
      self.current_transmitter.drop_packet()
      self.update_packets()
```

When a packet is successfully transmitted, this function serves as a macro to update the *transmitted_packets* counter, *successes* counter, drop the transmitted packet from *current_transmitter* and update the times on each Node's packets.

Simulation functions

sim

```
def sim(persistence,n, a, R, L, sim_time):
    # set up the nodes and bus
    bus = Bus(persistence, n, a, sim_time)
    # while there exists a next transmitting node
    while bus.find_next_transmitter():
        # check if a collision will occur
        c = bus.collision()
        # if no collision occurs, we have a
        # successful transmission
        if not c:
            bus.transmitted()
    eff = bus.successes/bus.transmitted_packets
    thru = bus.successes*L / (sim_time * R)
    return {"efficiency":eff,"throughput":thru}
```

This function carries out the simulation, instantiating a Bus object and calling methods from it. This function takes the results of the simulation (stored in member variables in the Bus object) and computes efficiency and throughput from it, returning it as a *dict* type.

question

```
def question(q_num,sim_time):
    N = [20*i for i in range(1,6)]
    A = [7, 10, 20]
    R = 1E6 # in bps
    L = 1500 #bits
    f = ""
    p = True
    if q_num ==1:
        print('1-persistent')
        f="Q1-"+str(sim_time)+".csv"
        p = True
    elif q_num == 2:
        print('non-persistent')
        f="Q2-"+str(sim_time)+".csv"
        p = False
    with open(f, "w", newline='') as f:
        w = csv.writer(f, delimiter=",")
        w.writerow(["num_nodes", "arrival_rate", "efficiency", "throughput"])
        for n in N:
            for a in A:
                print('N:',n,', A:',a)
                r = sim(p,n,a,R,L,sim_time)
                print("efficiency",r['efficiency'])
                print("throughput",r['throughput'])
                w.writerow([n,a,r['efficiency'],r['throughput']])
```

This function is just meant to set up the input to each iteration of the simulation with different parameters. It also formats and writes the results to an appropriately named csv file on each call of *sim*.

Question 1 Results

The following table shows the aggregated results from running the 1-persistent simulation for simulation times of 1000 and 2000 (the simulator also computes simulation results for simulation time of 3000, but those results have been left out from here for brevity). Please see *aggregated-data.xlsx* for the full table, the *Q1-xxxx.csv* files and *command-line-output.txt*.

Table 1: 1-persistent simulation results

num_nodes	arrival_rate	efficiency		throughput	
		sim_time = 1000	sim_time = 2000	sim_time = 1000	sim_time = 2000
20	7	0.955753058	0.954609707	0.210021	0.21019575
40	7	0.798816147	0.800280038	0.42105	0.42009
60	7	0.575283931	0.573297724	0.629268	0.6306285
80	7	0.326445853	0.326391221	0.822717	0.823605
100	7	0.189495892	0.190448857	0.8952615	0.894852
20	10	0.900757067	0.902629519	0.3000075	0.300162
40	10	0.611949738	0.608812442	0.597993	0.60055725
60	10	0.272840303	0.273806293	0.8564835	0.855678
80	10	0.189970902	0.190020192	0.908385	0.908493
100	10	0.166965596	0.166898617	0.915315	0.91535025
20	20	0.61735401	0.622111434	0.6006765	0.5995305
40	20	0.286629058	0.287105082	0.9048855	0.90519
60	20	0.224393631	0.224217405	0.903468	0.903399
80	20	0.189932205	0.189711754	0.908637	0.90865575
100	20	0.16700303	0.166754274	0.915291	0.91530975

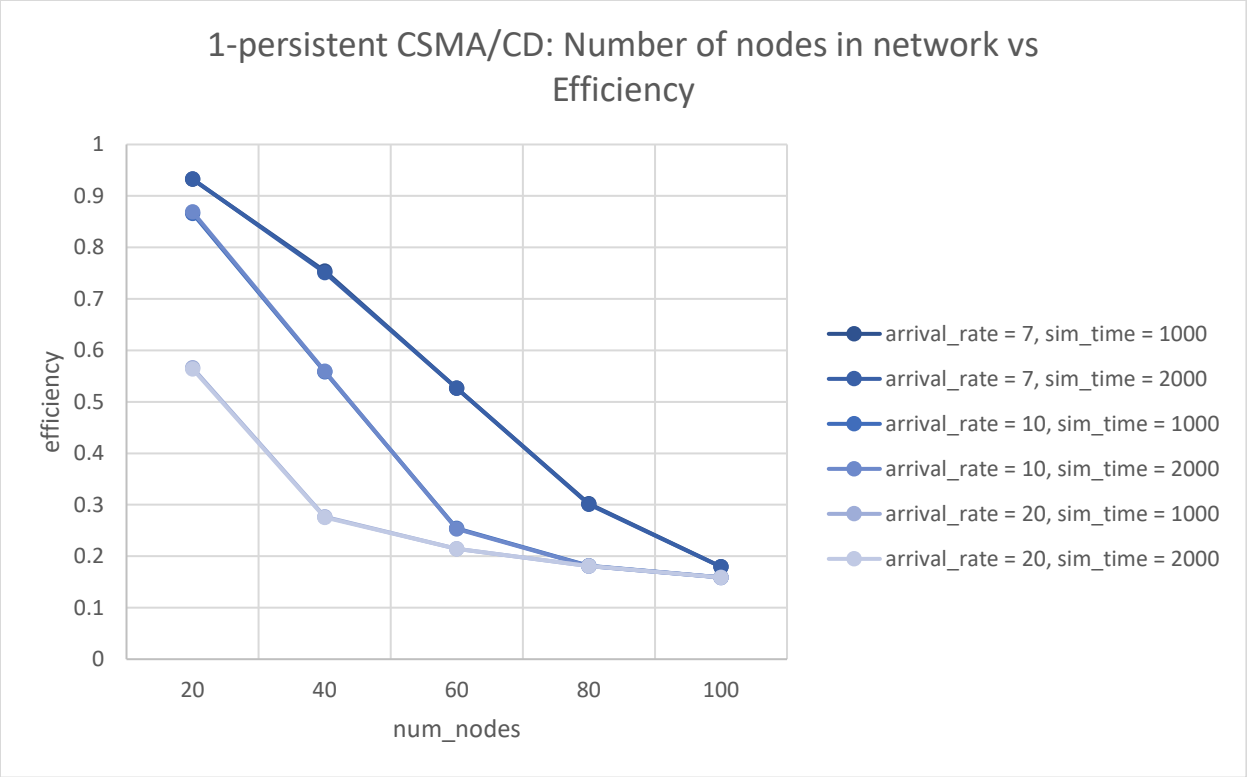

Figure 1 Graph showing relationship between number of nodes in network and efficiency computed by the simulation (for different arrival rates and total simulation times)

Firstly, in Figure 1 we see that several of the line plots are overlapping each other, namely the line plots associated with the same arrival rates. This shows that the efficiency of the simulated network doesn't vary with how long the simulation lasts. We can conclude from this that the system is stable.

We see a general negative trend for each of the line plots, meaning efficiency decreases with increasing number of nodes in the network. Also, we see that a higher arrival rate yields lower efficiency in the network. These results make sense intuitively, more nodes → higher network traffic intensity → higher queue utilization. We know that $\lambda = \frac{\rho L}{C}$ (from Lab 1), where the packet arrival rate λ is proportional to the queue utilization factor ρ and inversely proportional to the data transmission rate C.

Another way to think about it is that the efficiency is the ratio between successfully transmitted packets to the total number of transmitted packets. When there are more nodes (or larger arrival rate, which means more packets in the queue of each node), that means there is a higher probability for collision. Efficiency is negatively impacted with a higher number of collisions happening since that means extra time must be taken to re-transmit packets. So, it's reasonable to conclude that more nodes/packets in a network lead to lower efficiency.

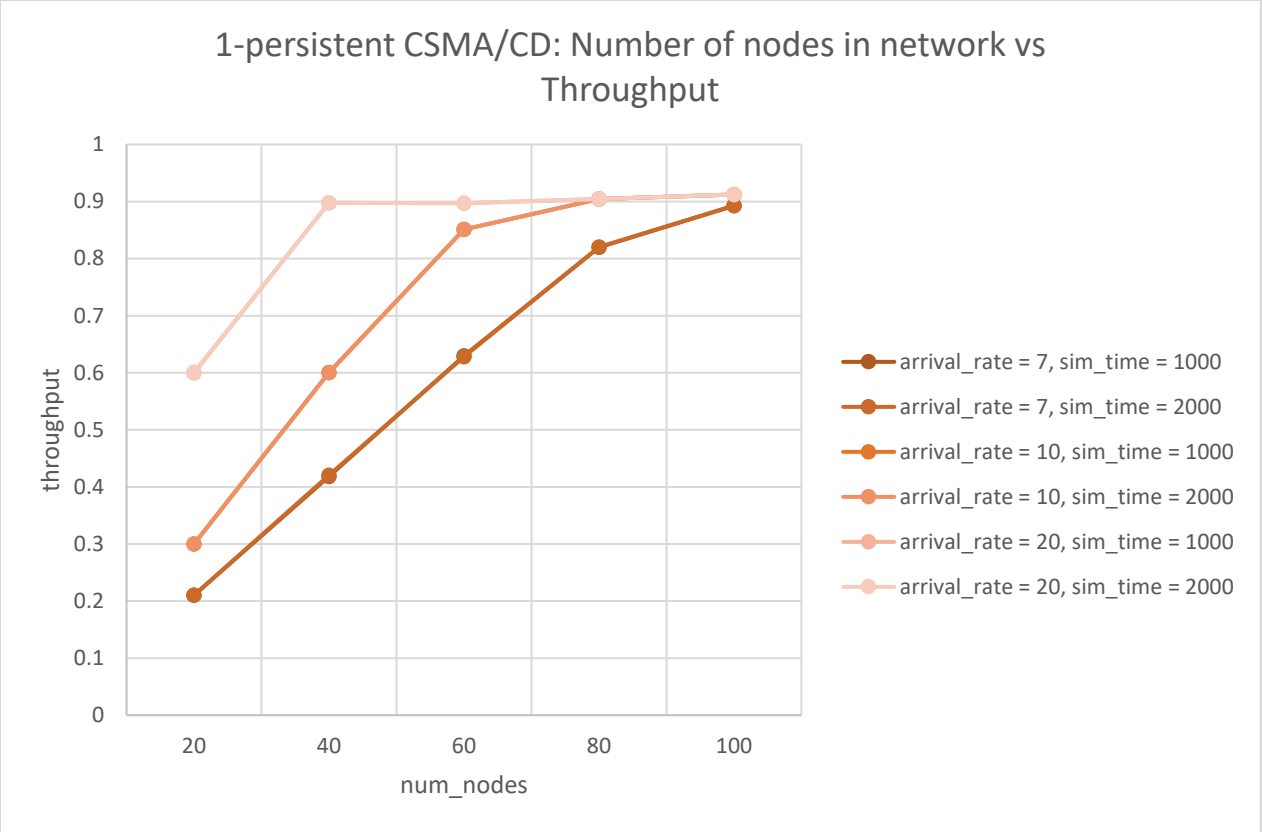

Figure 2 Graph showing relationship between number of nodes in network and throughput computed by the simulation (for different arrival rates and total simulation times)

Firstly, in Figure 2 we see that several of the line plots are overlapping each other, namely the line plots associated with the same arrival rates. This shows that the throughput of the simulated network doesn't vary with how long the simulation lasts. We can conclude from this that the system is stable.

We see a general positive trend for each of the line plots, up till a threshold of about 90% at which point they plateau. We can also see that higher arrival rates achieve larger throughput. These results intuitively make sense, higher arrival rate means more packets are being transmitted which means throughput generally increases.

Question 2 Results

The following table shows the aggregated results from running the non-persistent simulation for simulation times of 1000 and 2000 (the simulator also computes simulation results for simulation time of 3000, but those results have been left out from here for brevity). Please see *aggregated-data.xlsx* for the full table, the *Q2-xxxx.csv* files and *command-line-output.txt*.

Table 2: non-persistent simulation results

num_nodes	arrival_rate	efficiency		throughput	
		sim_time = 1000	sim_time = 2000	sim_time = 1000	sim_time = 2000
20	7	0.999771126	0.999820764	0.2096745	0.209184
40	7	0.99852136	0.998265029	0.4193595	0.41858775
60	7	0.993098575	0.993156279	0.6302715	0.63018075
80	7	0.968053629	0.967740766	0.8387115	0.8406135
100	7	0.425424895	0.42462391	0.9950265	0.99501675
20	10	0.99960994	0.999541018	0.2998365	0.30052725
40	10	0.995999214	0.995950094	0.6004695	0.5990595
60	10	0.961066509	0.960174369	0.899799	0.9003255
80	10	0.510132661	0.510842264	0.9957435	0.99575775
100	10	0.401481893	0.400364197	0.995487	0.99547725
20	20	0.99824254	0.997956413	0.598107	0.5999205
40	20	0.818675106	0.819745468	0.994872	0.99487425
60	20	0.657472173	0.657637959	0.9957015	0.995703
80	20	0.510752887	0.510067784	0.995772	0.995763
100	20	0.400182831	0.400922841	0.995472	0.9954855

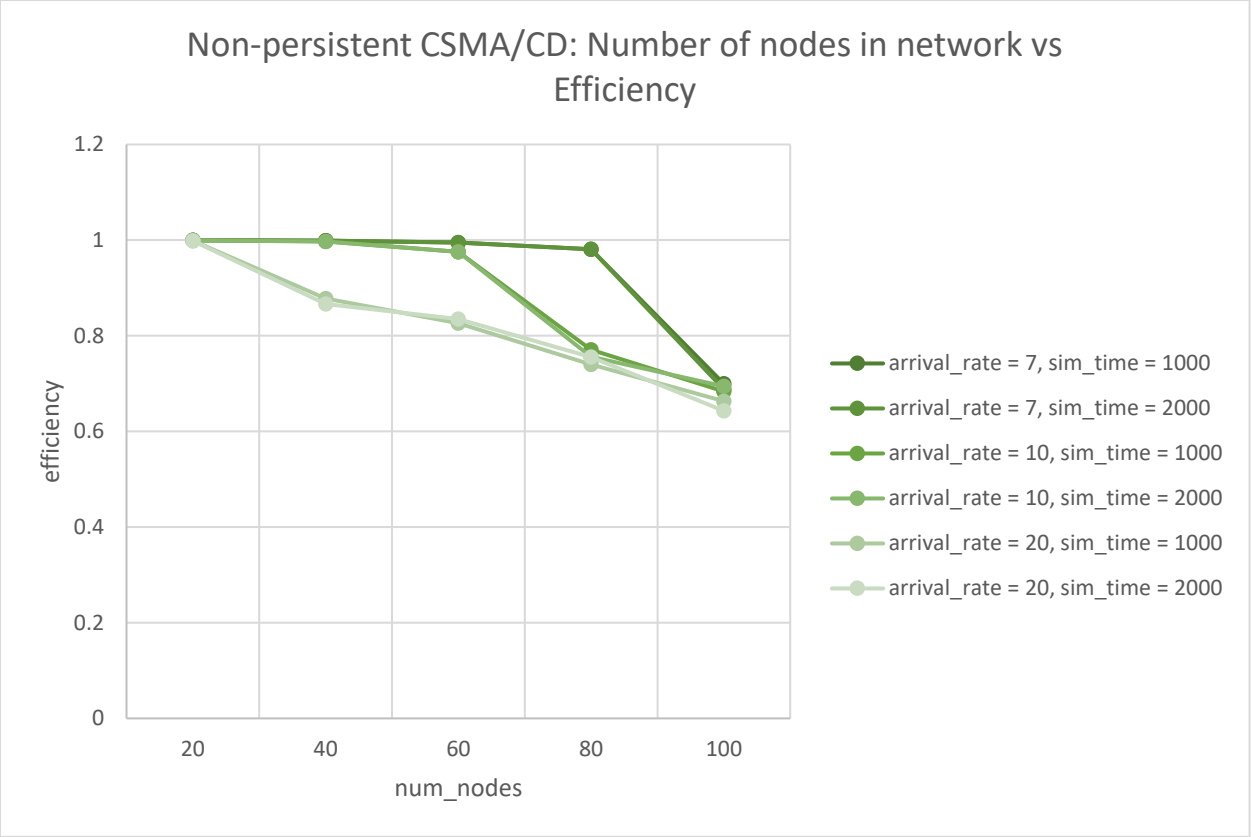

Figure 3 Graph showing relationship between number of nodes in network and efficiency computed by the simulation (for different arrival rates and total simulation times)

Firstly, in Figure 3 we see that several of the line plots are overlapping each other, namely the line plots associated with the same arrival rates. This shows that the efficiency of the simulated network doesn't vary with how long the simulation lasts. We can conclude from this that the system is stable.

We see a general negative trend for each of the line plots, meaning efficiency decreases with increasing number of nodes in the network. Also, we see that a higher arrival rate yields lower efficiency in the network. These results make sense intuitively, more nodes → higher network traffic intensity → higher queue utilization. We know that $\lambda = \frac{\rho L}{C}$ (from Lab 1), where the packet arrival rate λ is proportional to the queue utilization factor ρ and inversely proportional to the data transmission rate C.

Another way to think about it is that the efficiency is the ratio between successfully transmitted packets to the total number of transmitted packets. When there are more nodes (or larger arrival rate, which means more packets in the queue of each node), that means there is a higher probability for collision. Efficiency is negatively impacted with a higher number of collisions happening since that means extra time must be taken to re-transmit packets. So, it's reasonable to conclude that more nodes/packets in a network lead to lower efficiency.

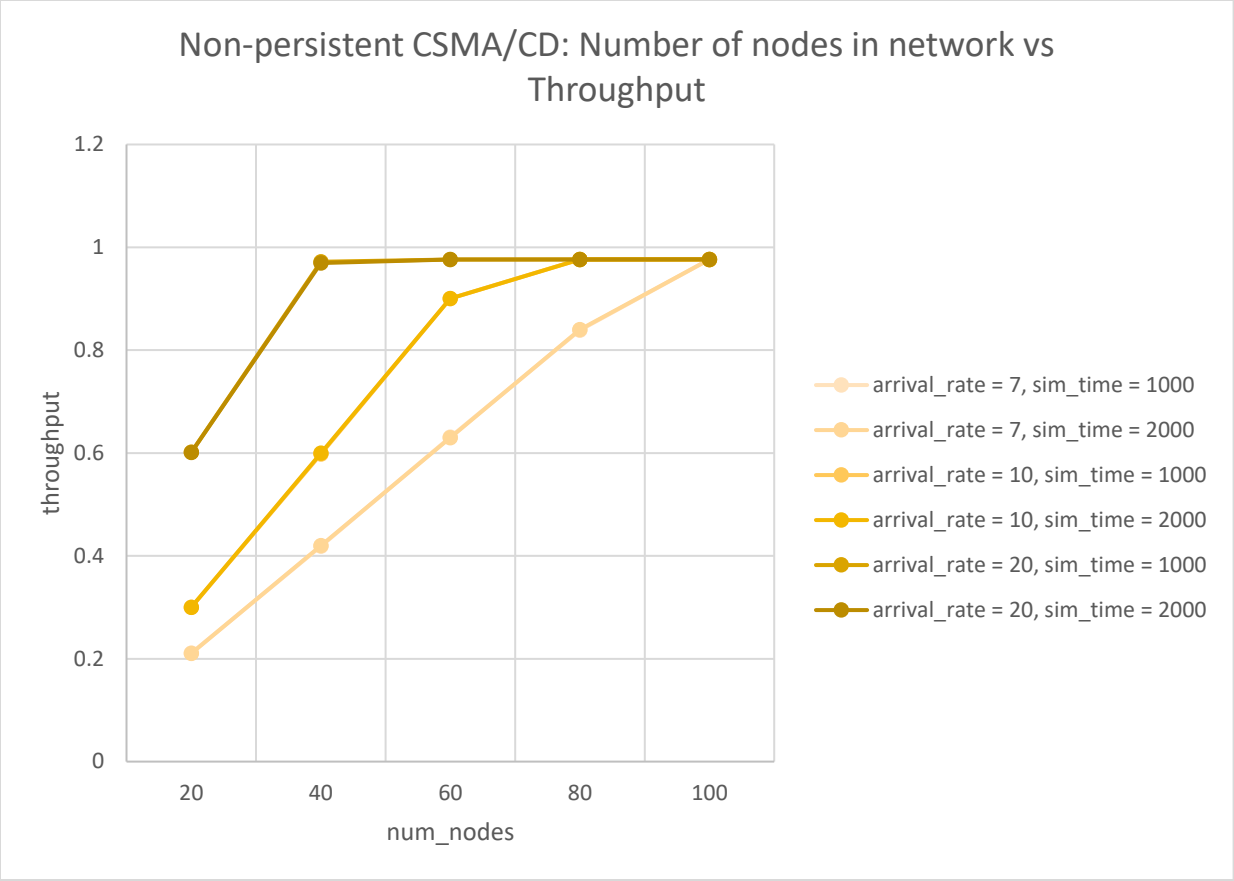

Figure 4 Graph showing relationship between number of nodes in network and throughput computed by the simulation (for different arrival rates and total simulation times)

Firstly, in Figure 4 we see that several of the line plots are overlapping each other, namely the line plots associated with the same arrival rates. This shows that the throughput of the simulated network doesn't vary with how long the simulation lasts. We can conclude from this that the system is stable.

We see a general positive trend for each of the line plots, up till a threshold of about 97-100% at which point they plateau. We can also see that higher arrival rates achieve larger throughput. These results intuitively make sense, higher arrival rate means more packets are being transmitted which means throughput generally increases.

1-persistent vs. non-persistent

First, some immediate observations.

- Efficiency in the non-persistent simulation is overall much better than in 1-persistent. Non-persistent maintains efficiency very well as the number of nodes scale, whereas 1-persistent suffers heavy efficiency loss with increase in number of nodes
- Throughput in 1-persistent and in non-persistent are overall similar. 1-persistent has marginal advantage when the number of nodes is small but non-persistent plateaus at a higher maximum throughput

These conclusions make sense when considering how the two modes differ. It is reasonable that non-persistent would have a large efficiency advantage because nodes in the network aren't "busy-waiting" on the bus to be free. When packets across different nodes are scheduled to transmit at the end of the busy time for the bus, it is essentially generating more collisions and postponing when they happen. Using an exponential back-off scheme of rescheduling packet transmissions does a much better job at staggering the transmission times. Probabilistically speaking, this staggering creates fewer collisions in the future which in turn means a larger efficiency advantage.

The results for throughput also make sense. Maximum throughput in the non-persistent case is higher than the maximum throughput in the 1-persistent case for the same reason non-persistent has better efficiency than 1-persistent. The reduced number of collisions in the non-persistent mode increases the overall number of successfully transmitted packets, which is proportional to throughput.