Lab 2: CSMA/CD Performance Evaluation

Manthan Shah, 20658832
Heather D'Souza, 20671903

Faculty of Engineering
Department of Electrical and Computer Engineering

November 10, 2020
Course Instructor: Dr. A. Wasef

1 Persistent CSMA/CD protocol simulation

A persistent CSMA/CD simulator was developed, to analyze the efficiency and throughput metrics of a network with a varying number of nodes and packet arrival rates. The design of the simulator has two components; a method called populate_nodes which generates arrival events for each node, and another method called persistant_csma_cd which is responsible for the transmission of nodes, and handling of any collisions. There are also two data structures, Packet and Node, to simplify the design and operation of the simulator.

1.1 Simulator data structures

The Packet data structure defines each individual packet that would be transmitted by a node, and it has two fields; arrival_time and collisions. The arrival_time field is used to keep track of the timestamp of the packet relative to the node and simulator. The collisions field is used to keep track of how many times a particular packet has been involved in a collision on the bus. The bus_busy_counter is the collision counter used for medium sensing.

```
class Packet:
    def __init__(self, arrival_time, collisions,
bus_busy_counter):
        self.arrival_time = arrival_time
        self.collisions = collisions
        self.bus_busy_counter = bus_busy_counter
```

The Node data structure models a real-world node that would be attached to a bus on a computer network. It has one field, packets, which is a queue of Packet objects (described above). This field models all the packets that must be transmitted by the node to some other node.

```
class Node:
   def __init__(self, packets):
       self.packets = packets
```

1.2 Arrival event generation

The populate_nodes method is responsible for creating all the nodes, and also populating them with a queue of packets to be transmitted. It takes three parameters; N, the number of nodes on the network, A, the arrival rate for packets at each node, and T_sim, the maximum simulation time.

In the method, we first create an empty list representing all the nodes. Then, we repeat the process of creating and populating a node, for the number of desired nodes (N) - this is

represented by the for loop. Inside the loop, we keep track of a curr_time variable, which is with respect to the maximum simulator time (T_sim), and a double-ended queue of packets, called packets. We keep creating timestamps using the random variable generation method following a Poisson distribution from lab 1 until T_sim, and add these to the packets queue. At the end, we create a node with these packets, and append it to the overall node list, nodes. As mentioned, this process is repeated N times, where N is the number of nodes.

```
# Creates nodes and populates them with a queue of packets, based
on inputs.
# N: number of nodes.
# A: arrival rate of packets at each node.
# T_sim: desired simulation time.
# The output is a list of nodes.
def populate_nodes(N, A, T_sim):
   nodes = []

   # Create packet arrival times using Poisson distribution with
packet arrival
   # rate defined by A. We create a deque, and as long as each
arrival time is valid we
   # add to the deque. We then create a node with those packets at
the end and repeat this
   # process for all nodes, defined by N.
   for i in range(N):
       curr_time = 0
       packets = collections.deque()

       while curr_time < T_sim:
           curr_time += generate_random_variable(A)
           if curr_time < T_sim:
               packets.append(Packet(curr_time, 0))

       nodes.append(Node(packets))

   return nodes
```

1.3 Simulation method design

The persistent_csma_cd method is responsible for the actual network simulation, and computation of efficiency and throughput. The method takes seven different parameters. N represents the total number of nodes on the network. A represents the arrival rate of packets at each of these nodes on the network. T_sim is the total simulation_time for the simulator. D is the physical distance between each node on the bus. S represents the propagation speed of the medium. L is the size of each packet that would be emitted by each node. Finally, R is the transmission rate over the link. Let us now walk through the code inside this method.

We first begin by creating a list of nodes that have a queue of packets each, by calling the populate_nodes method. Then we set up state tracking variables such as success_tx to track successful transmissions, total_tx for total transmissions, curr_time to represent the simulation time, and min_queue_idx to keep track of the index of the node which is transmitting at a particular instance.

Next, we enter a while loop, which is where the main persistent, CSMA/CD simulation takes place. We begin by determining which node has a packet at the front of its packet queue with the earliest arrival time. This is done by looping over all nodes and peeking at the arrival time of the packet at the front of each node, and keeping track of the node (and its associated index) with the earliest packet time. After that, we check for an exit condition of curr_time equal to the maximum system float. This is because, if all nodes had empty queues, there would be no node with anything to transmit, and thus we must break out of the simulation.

Now, we set up a variable to keep track of any collisions between the transmitting node (node with min_queue_index) and any other node, called collision_detected. The process begins by scanning all nodes (except the transmitting one) for a packet that could collide with the transmitting node packet - this is done via the for loop. Inside the for loop, we check for a collision by checking if the current node has a packet arrival time before the first bit of the transmitting node's packet arrives at the current node. If so, then we update the total_tx, collision_detected variables, and the current node's colliding packet's collision field. After, we check if that particular current node packet has experienced more than 10 collisions, in which case we drop that packet, and update the front arrival time of the node. Otherwise, we must reschedule the colliding packet of the current node. We do this by computing an exponential backoff:

$$T_{backoff} = K\{0, 2^m - 1\} \cdot \frac{512}{R}$$

We apply this backoff to the arrival time of the colliding packet of the current node. Thus, this reschedules the packet to prevent a future collision immediately.

Now, there are two cases which we must handle. The first is if the transmission of the packet by the transmitting node was unsuccessful (i.e. collisions with packets in any other nodes). We determine this by checking against the collision_detected variable, which would have been updated when a collision was detected between transmitting and some other node. If there is a collision, we must either drop the transmitting packet, or reschedule it based on the backoff, similar to what was done to the packets of the other colliding nodes. If the transmitting node has already experienced more than 10 collisions while trying to transmit this packet, we drop it, and update the arrival time of the next-in-line packet. We update the arrival time such that it is the recently dropped packet's arrival time or new packet's arrival time - whichever one is larger. This is done so that when there are collisions, we don't have to query through all preceding packets and update times, we do this computation at the point

of packet removal. If there are less than 10 collisions on the transmitting node, we compute an exponential backoff, and apply it to the transmitting packet's arrival times to reschedule it for a later simulation time.

The second case is that there were no collisions between the transmitting node's packet and any other nodes. In this case, we first update the success_tx, and total_tx variable, and then remove the transmitted packet from the transmitting node's packet queue, and update the arrival time for the next-in-line packet. Now, we must do an additional check on all nodes, to determine if there was any front packet in any nodes packet queue that was scheduled to be transmitted while the bus was busy. This involves looping through all the nodes and determining if any nodes packet was to be sent between the arrival of the first and last bits of the transmitting packet. If there was such a packet in such a node, then we reschedule that packet's arrival time to be after the last bit of the transmitting node's packet has passed that node.

Finally, we compute the efficiency and throughput values. The efficiency can be calculated by dividing the successful transmissions by the total number of transmissions. The throughput can be found by multiplying the successful transmissions by the size of each packet, and then dividing that by the total simulation time; this gives us the rate of transfer of bits/sec from sender to receiver. We divide by 10^6 to convert megabits per second for convenience.

```
# Simulates the persistent CSMA/CD network scenario.
# N: number of the nodes on the network.
# A: arrival rate of packets at each node.
# T_sim: desired simulation time.
# D: distance between each node.
# S: propagation speed of the medium.
# L: packet size.
# R: transmission rate over the link.
# The method outputs the efficiency and throughput of the network
based on the inputs.
def persistent_csma_cd(N, A, T_sim, D, S, L, R):
   # Generate arrival packets at each node up to the simulation
time.
   nodes = populate_nodes(N, A, T_sim)

   # Variables to keep track of successful and overall number of
transmissions.
   success_tx, total_tx = 0, 0

   # Variables to keep track of simulation time and transmitting
node index.
   curr_time, min_queue_idx = 0, 0

   while curr_time < T_sim:
```

```
        # Compute the node (and associated node index) that has
smallest packet arrival time, by checking
        # if a node's latest packet arrival time is smaller than
previous, smallest packet arrival time.
        curr_time = float('inf')
        for i in range(N):
            if not len(nodes[i].packets) > 0:
                continue
            if nodes[i].packets[0].arrival_time < curr_time and
nodes[i].packets[0].arrival_time < T_sim:
                min_queue_idx = i
                curr_time = nodes[i].packets[0].arrival_time

        # This indicates that all nodes are empty - exit condition
from simulation.
        if curr_time == float('inf'):
            break

        print(curr_time)

        # To keep track of any collisions between transmitting node
and all other nodes.
        collision_detected = False

        # Scan all nodes (except transmitting node) to determine if
there are any collisions.
        for idx, node in enumerate(nodes):
            if idx == min_queue_idx:
                continue
            if not len(node.packets) > 0:
                continue

            packet = node.packets[0]
            T_prop = (D / S) * abs(idx - min_queue_idx) #
Propagation delay based on distance between transmitting and
current node.

            # Determine if node's latest packet time is to be
transmitted before first bit of transmitting
            # node is received - this indicates a collision between
the transmitting and current loop node.
            if packet.arrival_time <=
nodes[min_queue_idx].packets[0].arrival_time + T_prop:
                total_tx += 1
                collision_detected = True

                packet.collisions += 1
```

```
                # If the packet has been involved in more than 10
collisions, drop it and update the arrival time of the next packet
in the node
                # if there is one. We only change the packet's
arrival time if it is less than the dropped packet's arrival time.
                if packet.collisions > 10:
                    last_packet = node.packets.popleft()
                    if len(node.packets) > 0:
                        node.packets[0].arrival_time =
max(node.packets[0].arrival_time, last_packet.arrival_time)
                else:
                    # Apply an exponential backoff to the node's
packet time based on number of collisions - time node must wait
before it can
                    # retransmit this packet.
                    T_backoff = random.randint(0,
2**packet.collisions - 1) * (512 / R)
                    packet.arrival_time += T_backoff

        # Helper variables to access transmitter node and packet.
        transmitter_node = nodes[min_queue_idx]
        transmitter_node_packet = transmitter_node.packets[0]

        # If the transmitting node collided with any other nodes,
it's packet arrival time must be updated (or dropped). Otherwise,
the packet
        # must be removed from the transmitting node's packet
queue.
        if collision_detected:
            total_tx += 1

            transmitter_node_packet.collisions += 1
            if transmitter_node_packet.collisions > 10:
                last_packet = transmitter_node.packets.popleft()
                if len(transmitter_node.packets) > 0:
                    transmitter_node.packets[0].arrival_time =
max(transmitter_node.packets[0].arrival_time,
last_packet.arrival_time)
            else:
                T_backoff = random.randint(0,
2**transmitter_node_packet.collisions - 1) * (512 / R)
                transmitter_node_packet.arrival_time += T_backoff
        else:
            success_tx += 1
            total_tx += 1

            last_packet = transmitter_node.packets.popleft()
```

```
            if len(transmitter_node.packets) > 0:
                transmitter_node.packets[0].arrival_time =
max(transmitter_node.packets[0].arrival_time,
last_packet.arrival_time)

            # Scan every node on the bus and update the latest
packet arrivals in the case that there were packets that were to
be transmitted
            # during a busy bus (while the transmitting node was
transmitting).
            for i in range(len(nodes)):
                if not len(nodes[i].packets) > 0:
                    continue

                packet = nodes[i].packets[0]
                T_prop = abs(i - min_queue_idx) * (D / S)

                # If this node's packet was to be transmitted after
the first bit, but before the last bit of the current transmitting
node's packet,
                # we must re-schedule the packet to after the last
bit of the current transmitting node's packet passes this node  on
the bus.
                if transmitter_node_packet.arrival_time + T_prop <=
packet.arrival_time < transmitter_node_packet.arrival_time +
T_prop + L / R:
                    packet.arrival_time =
transmitter_node_packet.arrival_time + T_prop + L / R

    print("Done simulation!")
    efficiency = success_tx / total_tx
    throughput = ((success_tx * L) / T_sim) / (10**6)
    print(efficiency, throughput)
    return (efficiency, throughput)
```

1.4 Efficiency and throughput of the LAN

The efficiency and throughput were plotted as functions of the number of nodes, N for a range of arrival event rates, A. N was spanned from N = 20 to N = 100, and A was set to 7, 10, and 20 for each iteration of sweeping N through its range.

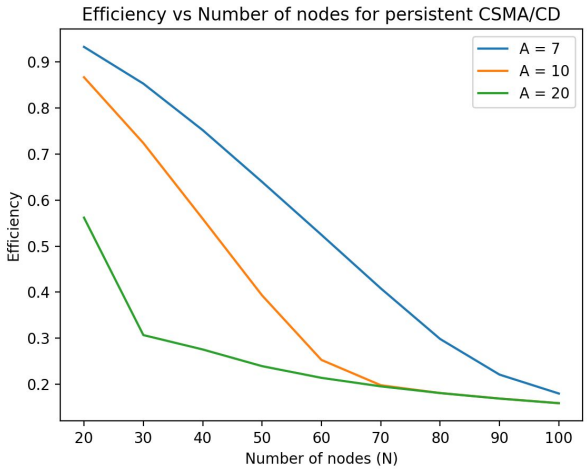

Figure 1: plot of efficiency vs number of nodes for persistent CSMA/CD simulation

The efficiency is shown in Figure 1. It can be observed that the persistent CSMA/CD protocol is comparatively efficient at lower values of N and A. It is most efficient when N = 20 and A = 7 (90%+ efficiency), and the LAN is most efficient for all cases of A at N = 20. It can be observed that as the number of nodes increases, at all arrival rates, the efficiency tends to decrease drastically. This makes sense as more nodes on the bus would mean higher likelihood of collisions on each packet transmission. Also, as the arrival rate is increased, the efficiency drops altogether compared to lower arrival rates, but the decreasing efficiency trend as N increases can still be observed. Conceptually, this makes sense as increased frequency of packet arrivals at nodes means less time in between each packet between all nodes on the bus, thus greatly increasing the likelihood of overall collisions on the bus

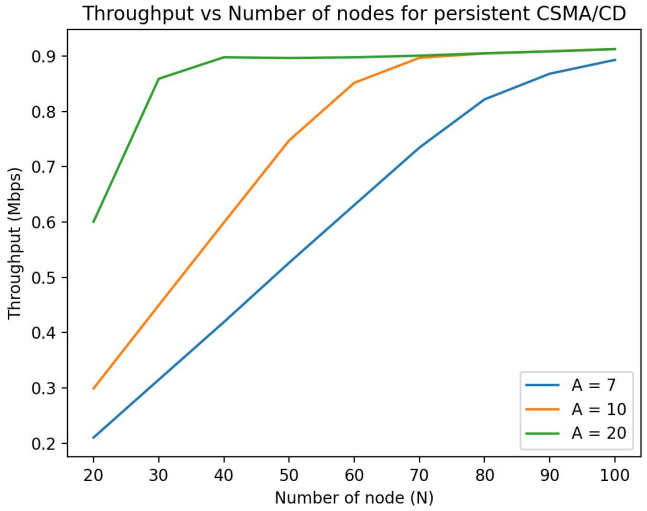

Figure 2: plot of throughput vs number of nodes for persistent CSMA/CD simulation

The throughput is shown in figure 2. I can be observed that the throughput tends to increase as N increases. This is because as N increases, the total number of transmissions will also increase since the bus is a lot more crowded with nodes. Since the proportion of successful transmissions will also increase, although at a much lower efficiency, the overall throughput increases; throughput is just the total number of bits transmitted per second, so more successful transmissions means more bits transferred. This is also the reason why as the arrival rate of packets at each node increases, the throughput increases. The increase in volume of packets means more overall transmissions, thus increased proportional successful transmissions, thus increased amounts of bits transferred.

Comparing figure 1 and 2, it can be observed that increasing throughput by increasing the number of nodes on the bus, is at the expense of efficiency. Increasing nodes gives higher throughput, but the efficiency is much lower. Similarly, increasing the arrival rate yields higher overall throughput at low and high numbers of nodes on the bus. This is also at the expense of efficiency, as it drastically decreases when the arrival rate is increased.

1.5 Simulator Verification

To ensure that the simulator performance is stable, the lower and upper thresholds of N (number of nodes) were tested at simulation times, T = 1000 s and T = 2000 s for A = 7 and A = 20, respectively. Table 1 describes the results that were obtained as a result.

Table 1: changes in efficiency and throughput at thresholds of N and A

Number of nodes (N)	Arrival packets rate (A)	Simulation time (s)	Efficiency (%)	Throughput (Mbps)
20	7	1000	93.45	0.2106
20	7	2000	93.43	0.2097
100	7	1000	18.05	0.8919
100	7	2000	17.92	0.8932
20	20	1000	56.22	0.5980
20	20	2000	56.59	0.5984
100	20	1000	15.88	0.9124
100	20	2000	15.87	0.9123

It can be observed from table 1, that when the simulation time was doubled, the results (efficiency and throughput) were consistently within 5% of each other (between the simulation times). The values of N = 20 and N = 100 were chosen as testing targets of N, and

A = 7 and A = 20 for A, because they represent the lower and upper bounds of the number of nodes (N) and arrival packets rate (A) presented in the lab manual. It allows us to approximate and extrapolate that if the performance is stable at the boundaries, it will likely be stable for values of N and A in between. This comes with a tradeoff of not testing every single possible value of N and A, which in a real-world situation would be a very expensive process and unfeasible. Thus, only the extremities of N and A were used for simulation stability testing. This sufficiently demonstrates the simulator is stable.

2 Non-persistent CSMA/CD protocol simulation

2.1 Simulation method design

The non-persistent variation of the CSMA/CD protocol differs from the persistent design in that an additional exponential backoff may be used to delay the transmission of a packet. In this variation, the node senses the medium when it has a packet to send. If the medium is idle, the packet is transmitted. If it is not idle, the node must wait for a random amount of time before it can sense the medium again.

The implemented design builds off of the persistent CSMA/CD design. It takes the same arguments and uses the same methods for arrival packet generation and collision detection. In this design, when a node successfully transmits a packet, the bus_busy_counter of the packet is reset to zero. This happens because if a node was able to transmit, it checked the bus and found that it was idle. Similar to the persistent design, the first packet of each node, excluding the transmitting node, is scanned to determine if another node wants to transmit a packet at the period of time this node senses the bus as busy. However, instead of updating the arrival times of the first packet at each node to occur after the time at which the last bit of the currently transmitting packet has passed, a random wait time is applied. The formula for the random wait time is the same as the one described in section 1.3. A node finds the bus busy when the arrival time of its packet is between the first bit of the currently transmitting packet (transmitter_node_packet.arrival_time + T_prop) and the last bit of the currently transmitting packet (transmitter_node_packet.arrival_time + T_prop + L / R). If this is true for the first packet in the node, its bus_busy_counter is incremented. If the busy_busy_counter is less than 10, its arrival time is rescheduled to its current arrival time plus the random wait time. The bus_busy_counter is incremented and the new arrival time is calculated as many times as needed until the arrival time of the packet occurs after the transmitting packet has passed the node. This explains the while loop in which packet.arrival_time is compared to transmitter_node_packet.arrival_time + T_prop and transmitter_node_packet.arrival_time + T_prop + L / R. If the busy_busy_counter is greater than 10, the packet is dropped.

```
# Simulates the persistent CSMA/CD network scenario.
# N: number of the nodes on the network.
# A: arrival rate of packets at each node.
# T_sim: desired simulation time.
# D: distance between each node.
```

```
# S: propagation speed of the medium.
# L: packet size.
# R: transmission rate over the link.
# The method outputs the efficiency and throughput of the network
based on the inputs.
def non_persistent_csma_cd(N, A, T_sim, D, S, L, R):
    # Generate arrival packets at each node up to the simulation
time.
    nodes = populate_nodes(N, A, T_sim)

    # Variables to keep track of successful and overall number of
transmissions.
    success_tx, total_tx = 0, 0

    # Variables to keep track of simulation time and transmitting
node index.
    curr_time, min_queue_idx = 0, 0

    while curr_time < T_sim:

        # Compute the node (and associated node index) that has
smallest packet arrival time, by checking
        # if a node's latest packet arrival time is smaller than
previous, smallest packet arrival time.
        curr_time = float('inf')
        for i in range(N):
            if not len(nodes[i].packets) > 0:
                continue
            if nodes[i].packets[0].arrival_time < curr_time and
nodes[i].packets[0].arrival_time < T_sim:
                min_queue_idx = i
                curr_time = nodes[i].packets[0].arrival_time

        # This indicates that all nodes are empty - exit condition
from simulation.
        if curr_time == float('inf'):
            break

        # To keep track of any collisions between transmitting
node and all other nodes.
        collision_detected = False

        # Scan all nodes (except transmitting node) to determine
if there are any collisions.
        for idx, node in enumerate(nodes):
            if idx == min_queue_idx:
                continue
            if not len(node.packets) > 0:
```

```
                continue

            packet = node.packets[0]
            T_prop = (D / S) * abs(idx - min_queue_idx) #
Propagation delay based on distance between transmitting and
current node.

            # Determine if node's latest packet time is to be
transmitted before first bit of transmitting
            # node is received - this indicates a collision
between the transmitting and current loop node.
            if packet.arrival_time <=
nodes[min_queue_idx].packets[0].arrival_time + T_prop:
                total_tx += 1
                collision_detected = True

                packet.collisions += 1

                # If the packet has been involved in more than 10
collisions, drop it and update the arrival time of the next packet
in the node
                # if there is one. We only change the packet's
arrival time if it is less than the dropped packet's arrival time.
                if packet.collisions > 10:
                    last_packet = node.packets.popleft()
                    if len(node.packets) > 0:
                        node.packets[0].arrival_time =
max(node.packets[0].arrival_time, last_packet.arrival_time)
                else:
                    # Apply an exponential backoff to the node's
packet time based on number of collisions - time node must wait
before it can
                    # retransmit this packet.
                    T_backoff = random.randint(0,
2**packet.collisions - 1) * (512 / R)
                    packet.arrival_time += T_backoff

        # Helper variables to access transmitter node and packet.
        transmitter_node = nodes[min_queue_idx]
        transmitter_node_packet = transmitter_node.packets[0]

        # If the transmitting node collided with any other nodes,
it's packet arrival time must be updated (or dropped). Otherwise,
the packet
        # must be removed from the transmitting node's packet
queue.
        if collision_detected:
            total_tx += 1
```

```
            transmitter_node_packet.collisions += 1
            if transmitter_node_packet.collisions > 10:
                last_packet = transmitter_node.packets.popleft()
                if len(transmitter_node.packets) > 0:
                    transmitter_node.packets[0].arrival_time =
max(transmitter_node.packets[0].arrival_time,
last_packet.arrival_time)
            else:
                T_backoff = random.randint(0,
2**transmitter_node_packet.collisions - 1) * (512 / R)
                transmitter_node_packet.arrival_time += T_backoff
        else:
            success_tx += 1
            total_tx += 1

            # Node was able to successfully transmit the packet,
so we reset the busy counter on the packet.
            transmitter_node_packet.bus_busy_counter = 0

            last_packet = transmitter_node.packets.popleft()
            if len(transmitter_node.packets) > 0:
                transmitter_node.packets[0].arrival_time =
max(transmitter_node.packets[0].arrival_time,
last_packet.arrival_time)

            # Scan every node on the bus and update the latest
packet arrivals in the case that there were packets that were to
be transmitted
            # during a busy bus (while the transmitting node was
transmitting).
            for i in range(len(nodes)):
                if not len(nodes[i].packets) > 0:
                    continue

                packet = nodes[i].packets[0]
                T_prop = abs(i - min_queue_idx) * (D / S)

                # If this node's packet was to be transmitted
after the first bit, but before the last bit of the current
transmitting node's packet,
                # we must re-schedule the packet to be its current
time plus an exponential backoff
                while transmitter_node_packet.arrival_time +
T_prop <= packet.arrival_time <
transmitter_node_packet.arrival_time + T_prop + L / R:
                    if packet.bus_busy_counter < 10:
                        packet.bus_busy_counter += 1
```

```
                        T_random_wait = random.randint(0,
2**packet.bus_busy_counter - 1) * (512 / R)
                        packet.arrival_time += T_random_wait
                    else:
                        last_packet = nodes[i].packets.popleft()
                        if len(nodes[i].packets) > 0:
                            nodes[i].packets[0].arrival_time =
max(nodes[i].packets[0].arrival_time, last_packet.arrival_time)
                        break

    print("Done simulation!")
    efficiency = success_tx / total_tx
    throughput = ((success_tx * L) / T_sim) / (10**6)
    print(efficiency, throughput)
    return (efficiency, throughput)
```

2.2 Efficiency and throughput of the LAN

The calculations for efficiency and throughput are the same as the ones in the persistent design. It is clear from the graphs that efficiency is much higher in the non-persistent method, compared to the persistent design (Figure 3). In the non-persistent design, efficiencies are greater than 0.98 for the combinations of number of nodes and arrival rate tested. This occurs because the fraction of successful transmissions increases. This is a result of the exponential backoff added to the packet arrival time when a node senses the bus as busy. This results in fewer collisions and therefore a higher percentage of successful packets. It is observed that for each packet arrival rate tested, the efficiency decreases as the number of nodes increases. There is crossover between the lines representing the various arrival rates. This is due to the fact that the efficiency values are very close. It should be noted that on occasion, the efficiencies observed for a particular arrival rate might go up for a simulation with N nodes and go back down for a simulation with more nodes. However, this is also a result of the efficiency values being very close to each other. In general, the trend that efficiency decreases as the number of nodes increases is preserved. This trend make sense as an increase in nodes would result in more packets and more collisions.

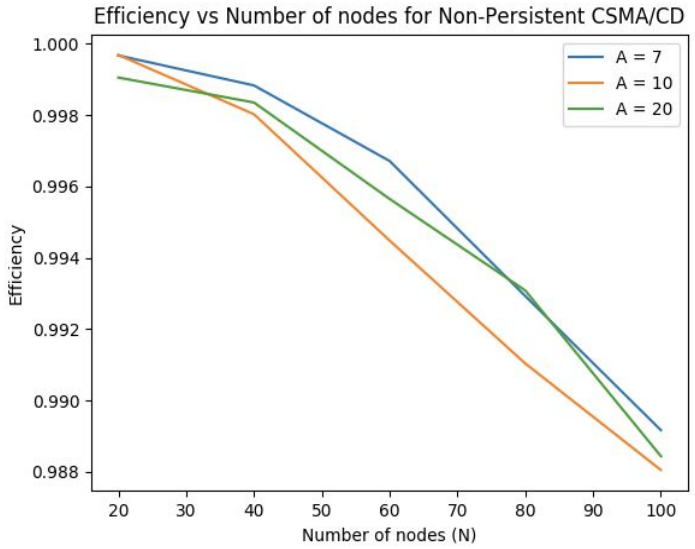

Figure 3: plot of efficiency vs number of nodes for non-persistent CSMA/CD simulation

The graph for the throughput (Figure 4) resembles the throughput graph for the persistent case (Figure 2). The throughput observed when the simulation is run with 20 nodes and either A = 7, A = 10, or A = 20, matches the throughput in the persistent case. However, the throughput in the persistent case tapers off faster than the throughput in the non-persistent case. In the persistent case, the throughput levels off at around 1 Mbps at 40 nodes, 70 nodes, and 100 nodes for A = 20, A = 10, and A = 7, respectively. In the non-persistent case, the throughput does not level off. Instead, the maximum throughput observed for A = 10 and A = 20 was reached at 100 nodes at a rate just under 1 Mbps. For A = 7, a throughput just less than 0.9 Mbps is reached at 100 nodes. It is clear that although the non-persistent design has better efficiency than the persistent design, it performs worse in regards to throughput.

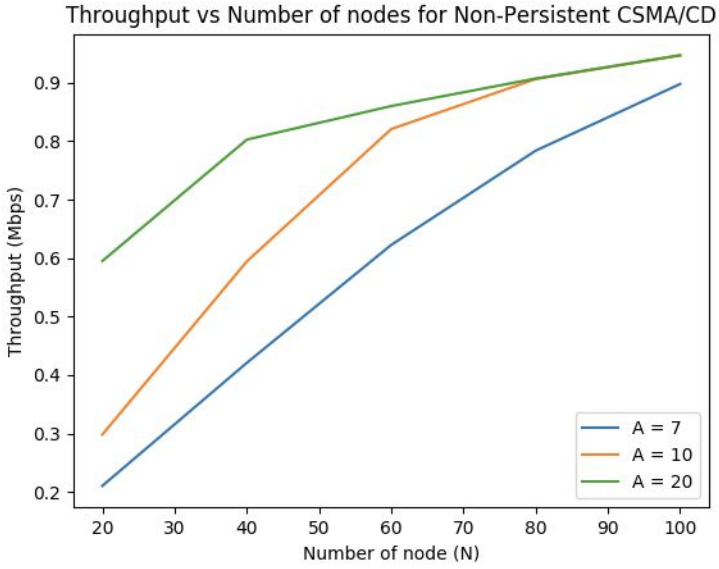

Figure 4: plot of throughput vs number of nodes for non-persistent CSMA/CD simulation

2.3 Simulator Verification

Various combinations of number of nodes and packet arrival rate were tested to determine the appropriate duration of the simulation (Table 2). The non-persistent simulation was run for 1000 and 2000 seconds for each combination and efficiency and throughput values were calculated.

Table 2: changes in efficiency and throughput at thresholds of N and A

Number of nodes (N)	Arrival packets rate (A)	Simulation time (s)	Efficiency (%)	Throughput (Mbps)
20	7	1000	99.983	0.21010
20	7	2000	99.979	0.20958
100	7	1000	98.905	0.89845
100	7	2000	98.918	0.89843
20	20	1000	99.914	0.59280
20	20	2000	99.883	0.59606
100	20	1000	99.734	0.94827
100	20	2000	98.607	0.94721

The efficiency and throughput values were within 5% of each other between the 1000-second and 2000-second tests. Therefore, a simulation time of 1000 seconds was used when running the simulations for non-persistent CSMA/CD.