University of Waterloo

ECE 358 Lab2 Report

Prepared by:

Group 27

Victor Yan
vlyan
20612514

Lun Jing
l7jing
20558988

Table of Contents

Summary *4*

Design *5*

Nodes and Packets **5**

Start Transmission **6**

Other Nodes Activities During Transmission **6**

Busy Detect 7

Collision Detect 7

Reschedule the Transmit Time 8

Generating Next Packet **8**

Simulation Results and Analysis *10*

1-persistent **10**

Non-persistent **12**

Stability Check **15**

Summary

In this lab, we are using python to design a CSMD/CD simulator to test the performance of a LAN with several end nodes connected with one signal bus. Through the simulation, we will see how the performance, the transmission efficiency and network throughput, will be affected by the number of nodes along the bus and the rate of generating the packets on each nodes.

Before we running the simulation, here are some assumptions we have:

- According to the lab manual, we assumed that all nodes are distributed along the bus with same distance away from each other. Moreover, all nodes will have same rate on generating the packets and all packets will have same length. Finally, all nodes will have same transmission speed.
- For non-persistent, when a node see the bus is busy more than 10 times, we either cap the iteration limit or drop the packets. We decided to keep the packet and cap the iteration limit.

Design

In this section, we are going to briefly describe how we design the simulator, with the code snippets.

Nodes and Packets

In this lab, we created a Packet class to represent each packet generated by node. Figure 1 shows the attributes of a Packet object and the constructor function.

```
# Packet class
# Class encapsulating all data regarding an individual packet, including
# the node it belongs to. All "node" data is contained within the head
# packets instead (e.g. collision/channel busy counters).
You, a few seconds ago | 1 author (You)
class Packet:
    t_trans_delay = T_trans # transmission delay

    def __init__(self, node, t_arrival):
        self.t_arrival = t_arrival      # the time when the packet is generated on node
        self.node = node                # the index of node where this packet generated
        self.t_trans = t_arrival        # the actual transmission start time
        self.c_collision = 0            # the number of collisions the packet has expereienced when it is being transmitted
        self.t_remove = 0               # the time when the packet has been removed from the node, either been transmitted or droped
        self.c_channel_busy = 0         # the counter to record how many times the packet found the bus is busy when it attempts to tranmit
```

Figure 1 Packet class

Figure 2a and 2b below shows how we initialize the ndoes. For the nodes, instead of generating a queue for each node and generating all the packets at once before the simulation for each node, to save memory, we decided to use a queue with size N to store the first packets for each node only, named “node_head”. Since this is a queue structure, the following packets will not affect the transmission for the current packet.

```
# N number of nodes, all heads begin empty
node_head = [None,] * N
end_time = 0

for i in range(N):
    node_head[i] = Packet(i, generate_random(A))
```

Figure 2a initialize nodes and first packets of each node

```
# Generate value from a random distribution
# Generate a random value from the Poisson distribution given parameter
# lambda (typically the arrival rate of the simulation).
# @param lambda_para - float:   Parameter for the random distribution.
# @return float:                Random value following the distribution.
def generate_random(lambda_para):
    return - (1 / lambda_para) * math.log(1 - random.uniform(0, 1))
```

Figure 2b generate value from a random distribution

Start Transmission

After we initialize the packets and nodes, we start transmission. As lab manual mentioned, we are going to find the earliest arrival packet and start transmission. Figure 3 shows how we start transmission and how we reset the counters and flag for the collision detect.

```
while(not allNodesEmpty(node_head)):
    # Find the earliest node_head
    trans_node, trans_start_at_src = findNextPacket(node_head)
    # Transmit the targeted head packet
    trans_packet = node_head[trans_node]
    trans_end_at_src = trans_start_at_src + trans_packet.t_trans_delay
    c_tx_attempts+=1
    node_head[trans_node].c_channel_busy = 0

    # Flag to check if collision occur
    f_collision = False
    collision_nodes = []
    t_collision_detected = -1
```

Figure 3 initialize counters and flag for each new transmission

Other Nodes Activities During Transmission

When one packet is transmitting, the other nodes will either find the bus busy or have a collision.

Busy Detect

Figure 4a shows the logic on detecting bus busy. Once a packet and wait until the transmission finished (1-persistent).

```
# Bus busy or node idle through entire transmission
if node_head[i].t_trans > trans_start_at_src + getPropagationDelay(trans_node, i):

    # Bus detected to be busy
    if node_head[i].t_trans < trans_end_at_src + getPropagationDelay(trans_node, i):
        if persistent_simulation:
            # Greedy; set start of transmission time immediately to when the current transmission seems to end
            node_head[i].t_trans = trans_end_at_src + getPropagationDelay(trans_node, i)
```

Figure 4a bus busy detection

Or, the packet wait for some random period (non-persistent). Figure 4b is the logic for handling busy detection on non-persistent situation. In this case, sometimes the length of the wait time is not long enough, which means the node may find bus busy twice in a single transmission. So we keep generating until the busy time is long enough.

```
if not persistent_simulation:
    # It is possible that despite adding some backoff/waiting time, the channel is still detected to be busy
    # due to the same transmitting node. In this case, loop until the wait time is sufficiently large.
    while node_head[i] is not None and node_head[i].t_trans < trans_end_at_src + getPropagationDelay(trans_node, i):

        # This implementation caps the counter limit at 10 and continuously waits to transmit the same packet
        if node_head[i].c_channel_busy < 10:
            node_head[i].c_channel_busy += 1
        node_head[i].t_trans += calcExpBackoff(node_head[i].c_channel_busy)
```

Figure 4b reschedule when detect busy for non-persistent

Collision Detect

Figure 5a shows the logic on detect the collision situations. We also recorded the earliest time when the collision happened in the collision, since there may be multiple nodes have collisions. We also have a temperate array to store the indexes of the nodes that are involved in the collision.

```
# Collision
else:
    f_collision = True
    collision_nodes.append(i)
    c_tx_attempts+=1
    # Node i has sensed the channel to be idle and so began transmission before colliding into the currently-transmitting node
    # The channel busy counter is reset since the channel was sensed as idle
    node_head[i].c_channel_busy = 0
    # This determines when the currently-transmitting node first detects a collision - through the earliest time
    if t_collision_detected == -1 or t_collision_detected > (node_head[i].t_trans + getPropagationDelay(i, trans_node)):
        t_collision_detected = node_head[i].t_trans + getPropagationDelay(i, trans_node)
```

Figure 5a collision detection

Reschedule the Transmit Time

After the collision, the packets those have collision times not greater than 10 will be reschedule. Figure 5b shows how we reschedule the packets.

```
# If a collision has occurred
if f_collision == True:
    collision_nodes.append(trans_node)
    for i in collision_nodes:
    # update the wait time
        node_head[i].c_collision += 1
        if node_head[i].c_collision <= 10:
            # Assuming all collision detections are relative to collision detected by
            # transmitting node, + propagation delay from transmitting node to colliding nodes
            node_head[i].t_trans = t_collision_detected + getPropagationDelay(i, trans_node) + calcExpBackoff(node_head[i].c_collision)

        else:
            # Drop packet, move next packet to node head
            node_head[i] = getNextPacket(node_head, i, t_collision_detected + getPropagationDelay(i, trans_node))
```

Figure 5b reschedule packet transmite time when collision occurs

Generating Next Packet

In our design, we basically generate packet on demand. Since the arrival time of the packet is independent with the transmission time, we only generate the packet after one node pops its first packet (either drop or transmite it successfully). Figure 6 is how we generate the packets and update its transmission started time based on the previous dropped/transmitted time of the previous packet.

```python
# Get the next packet in the queue
# Get the next packet to put into a node's head of the queue. This packet
# is generated on the fly and the "initial" arrival time is based on the
# previous packet's "initial" arrival time. This should act the same as if
# the nodes were all generated beforehand, and should only serve the
# purpose of saving memory.
# @param node_head - list[Packet]:  List of the nodes' packet queue heads.
# @param i - int:                   Index of the node to get a new packet for.
# @param previous_pkt_done - float: Current time of when the previous packet
#                                   was "done" (successfully transmitted or
#                                   dropped) and when the new packet is to
#                                   be the new head of the node's queue.
# @return Packet:                   New packet for the head of the node's queue.
def getNextPacket(node_head, i, previous_pkt_done):
    # Generate the next packet that arriving the node
    previous_pkt = node_head[i]
    next_pkt_arrival = previous_pkt.t_arrival + generate_random(A)
    next_pkt = Packet(i, next_pkt_arrival)

    # Check T exceed
    if next_pkt_arrival > T:
        return None
    else:
        # Check the arrival time with the transmit finish time
        if next_pkt_arrival < previous_pkt_done:
            next_pkt.t_trans = previous_pkt_done

    return next_pkt
```

Figure 6 generate next packet for the node pops one packet

Simulation Results and Analysis

1-persistent

1. Figures 7 and 8 below show the graphs for the efficiency and throughput of the persistent CSMA/CD simulation as a function of the number of nodes N respectively.

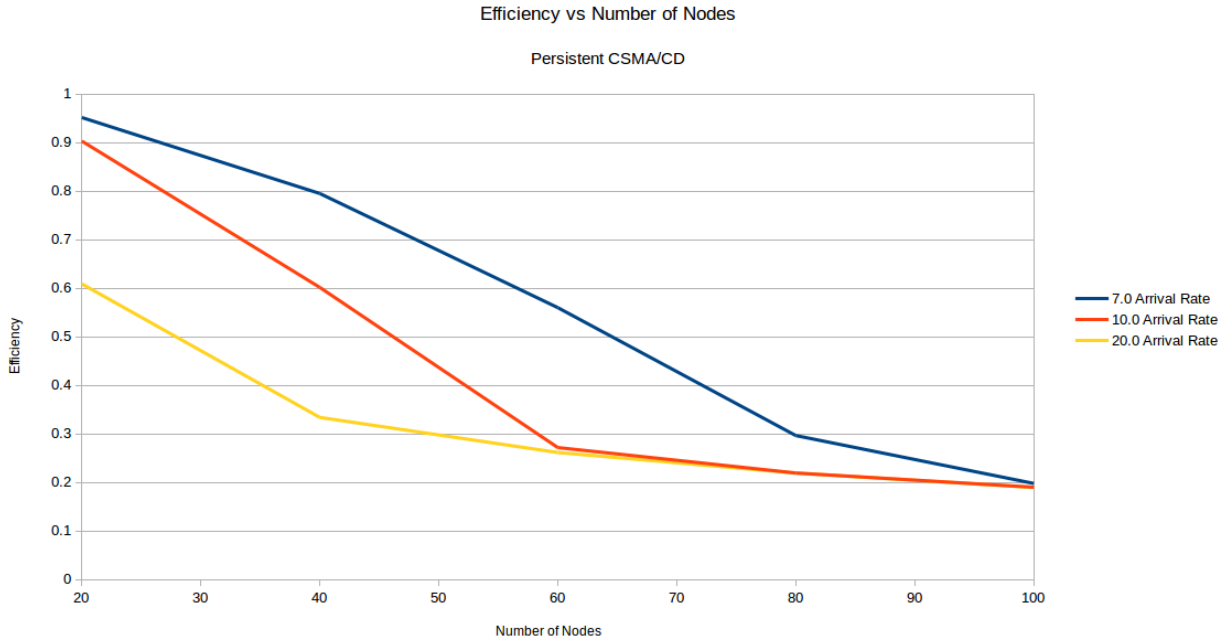

Figure 7 efficiency vs number of nodes for 1-persistent, T = 1000

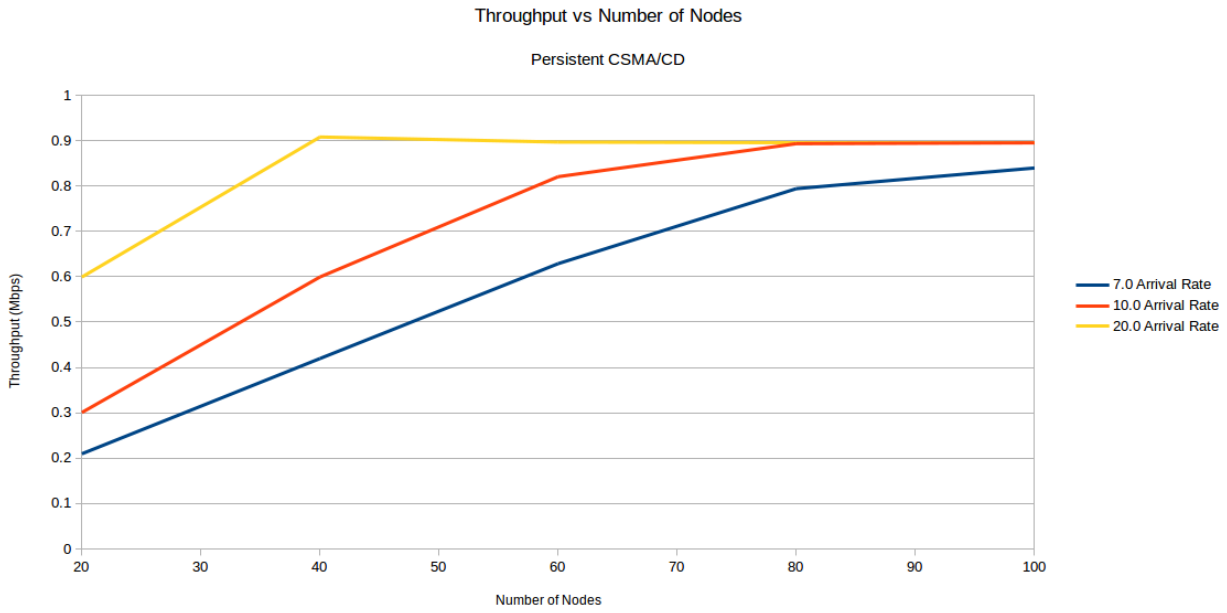

Figure 8 throughput vs number of nodes for 1-persistent, T = 1000

It can be seen clearly from Figure 1 that the efficiency of the persistent CSMA/CD protocol decreases as the number of nodes in the network increase, and that a faster arrival rate also causes the efficiency to decrease. This means that as N or A increase, the number of collisions increase.

It can also be seen that despite the arrival rate increasing to 10 or 20, all three lines approach a similar efficiency with a higher enough number of nodes of somewhere between 15-20 %. This seems to suggest that there is a limit for how low the efficiency can reach. It is possible that this hypothetical limit is reached when the arrival rate is high enough, the nodes are always backed up with packets in their respective queues to send, and thus will always be colliding at a similar rate as it is the highest rate possible, which inversely causes efficiency to be the lowest possible.

Figure 2 shows an increase in throughput as the number of users increase and/or as the arrival rate increases as well. There appears to be a rather steady increase in throughput as the number of nodes increase, saturating at approximately 900 Kbps, or 90 % of the theoretical limit (i.e. the rate of transmission, 1 Mbps). It's understandable that increasing the number of users or rate of arrival for packets would increase throughput as adding both adds a higher chance for a packet to arrive at any point in time during the simulation. This makes it more likely for the channel to be used by any node to transmit said packets. Before reaching the limit, the channel remains idle for certain period of time during the simulation due to no packets having arrived for the nodes to need to transmit.

The limit of 90 % may arise from a similar cause as the one described in the explanation for the efficiency graph in Figure 1. By reaching the limit of having packets constantly queued up at each node to transmit as soon as possible, perhaps the throughput cannot be increased any further as there is a similar pattern of collisions that occur each time at the end of transmission for some packets.

Non-persistent

2. Figures 9 and 10 display the graphs for efficiency and throughput of the non-persistent CSMA/CD simulation as a function of the number of nodes N respectively.

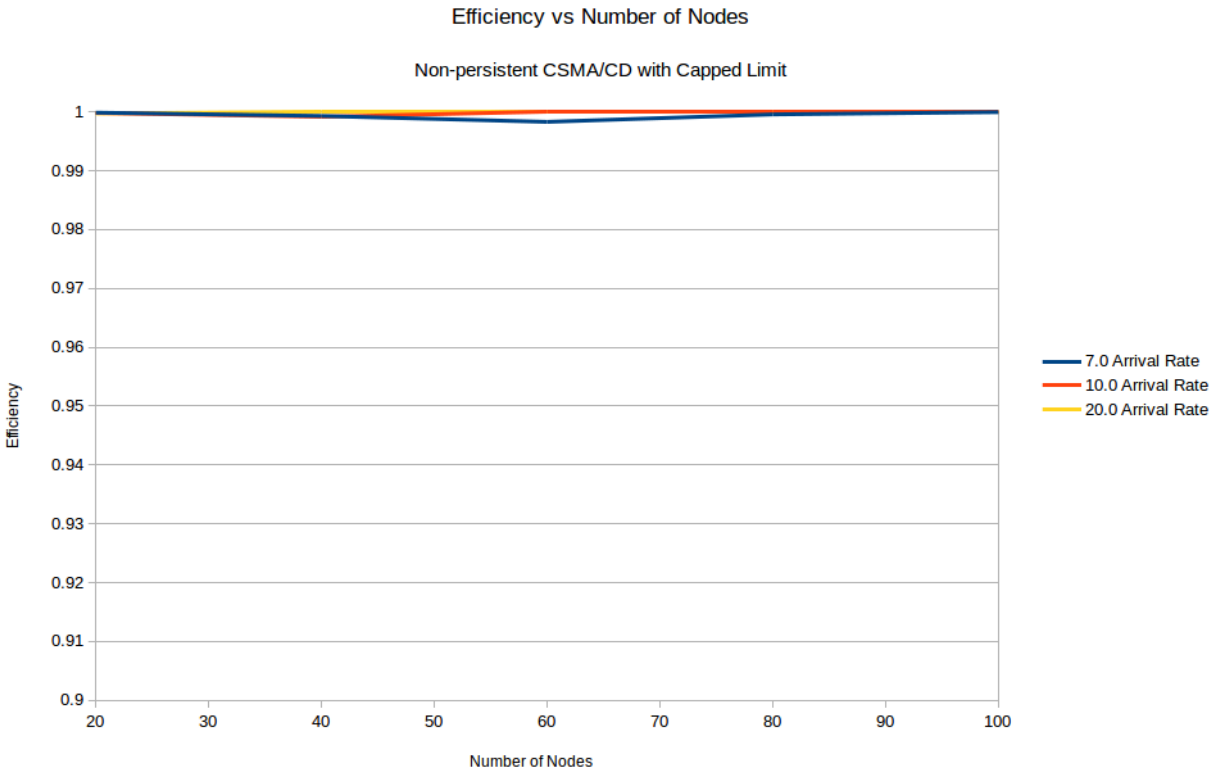

Figure 9 efficiency vs number of nodes for non-persistent, T = 1000

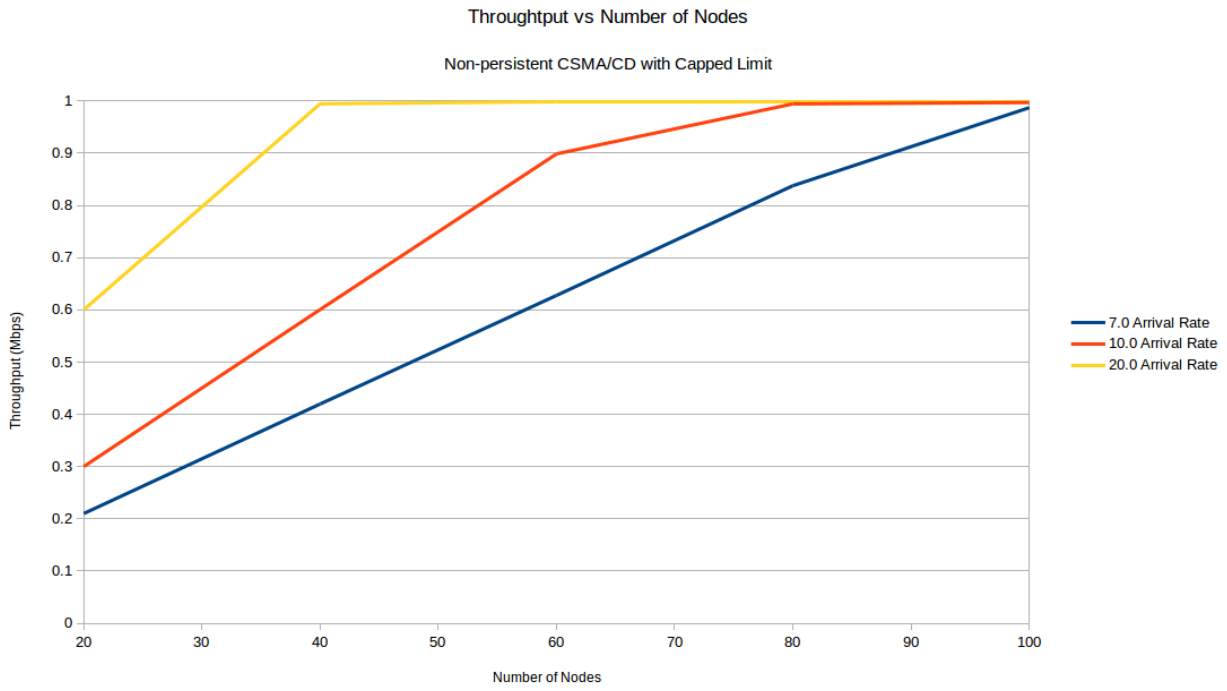

Figure 10 throughput vs number of nodes for non-persistent, T = 1000

Figure 3 indicates that the efficiency for the non-persistent CSMA/CD protocol is always high (nearly 100 %) and is unaffected by both the number of users as well as the arrival rate of packets. This is most likely due to nodes using this protocol being less greedy, and tend to not all sense the medium to be idle near-simultaneously (i.e. as soon as the currently transmitting node is done, plus the appropriate propagation delay). In the case of non-persistent CSMA/CD, the random backoff time after sensing the medium to be busy prevents the chance of multiple nodes immediately sensing the channel to be idle. Thus, while packets may still wait to be transmitted, the nodes are less likely to collide with other nodes' transmissions and more likely to sense the channel to be busy more accurately, resulting in the efficiency being high.

This kind of behaviour is independent of the number of nodes and the arrival rate since an increase in packets to be sent or number of nodes wishing to transmit will not cause more collisions as all nodes will still be more likely to accurately sense the channel to be busy rather than colliding.

Clearly, the efficiency for the non-persistent CSMA/CD protocol is far higher than the efficiency for the persistent CSMA/CD protocol, and more importantly, unlike the persistent protocol, the non-persistent protocol is unaffected by the increase in the number of nodes or the arrival rate of packets at each node.

Figure 4 shows the throughput of the non-persistent CSMA/CD protocol to follow a similar trend to the persistent CSMA/CD protocol simulation results, increasing as N and A increase, but now saturating at about 1 Mbps (or 100 % of the transmission rate) instead of 90 %. It's understandable that the same trends occur as the reasoning for the increase in throughput is the same for both protocols. As for the different levels at which the protocols saturate in throughput, that may be due to the fact that there are far less collisions in the non-persistent CSMA/CD protocol (as seen from the near-100 % efficiency in Figure 3), and thus there is less time spent

transmitting colliding packets and having all nodes perform the backoff. In the case of non-persistent CSMA/CD, it is more likely for nodes to properly sense the channel to be busy and thus not collide with the currently transmitting node, allowing the throughput to be in constant use with little to no collisions slowing down the process.

Very similar trends are seen in the throughput graphs in Figure 2 and 4, including even the beginning throughput at N = 20 for the different arrival rates. The biggest difference lies in the throughput for the non-persistent protocol saturating at 1 Mbps, where as it saturates at 900 Kbps for the persistent protocol. This key difference is, as explained above, most likely due to the high efficiency of the non-persistent protocol allowing the channel to be used more efficiently without wasting bandwidth on colliding transmissions and backoffs.

Stability Check

Figure 11 to figure 14 are for stability checks, with T = 2000 for both persistent and non-persistent.

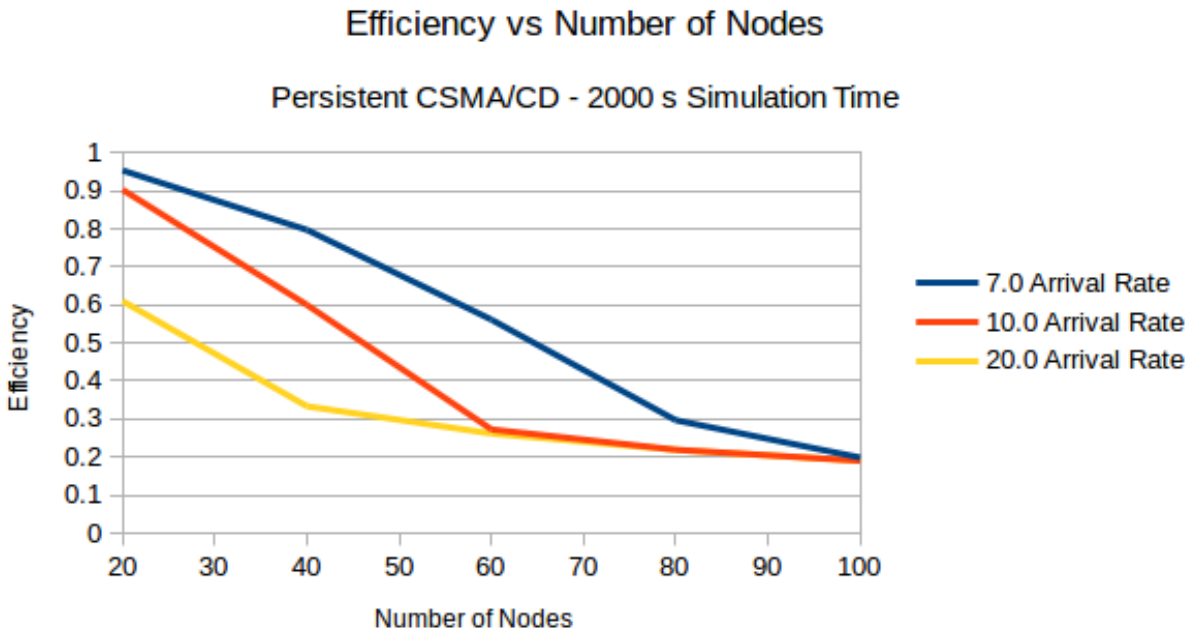

Figure 11 efficiency vs number of nodes for 1-persistent, T = 2000

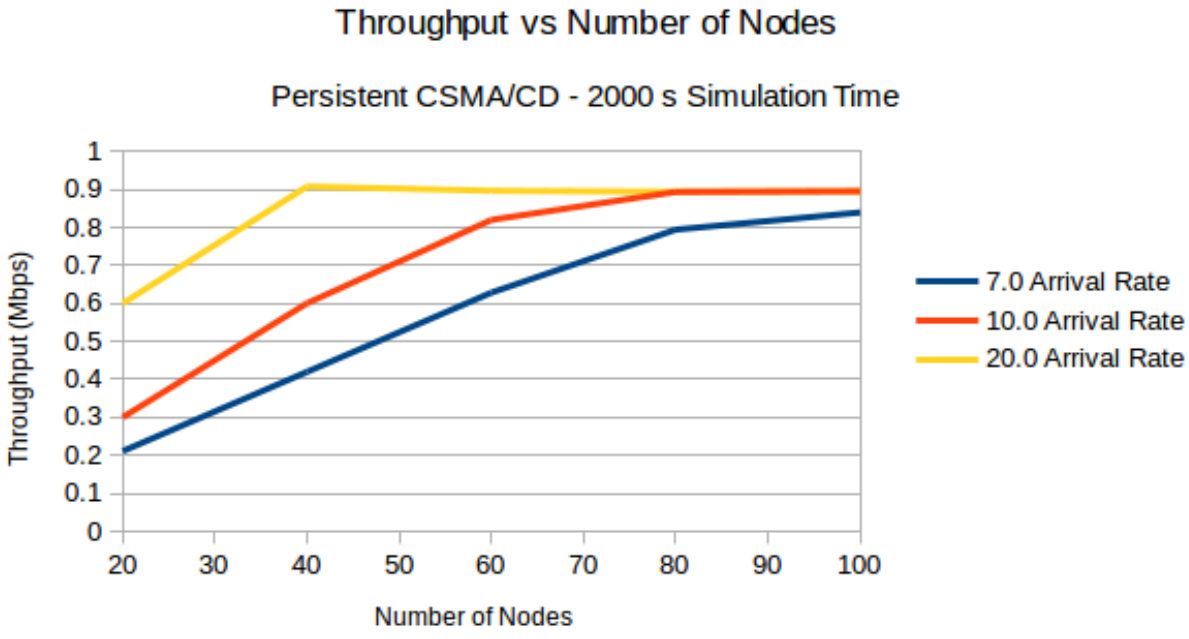

Figure 12 throughput vs number of nodes for 1-persistent, T = 2000

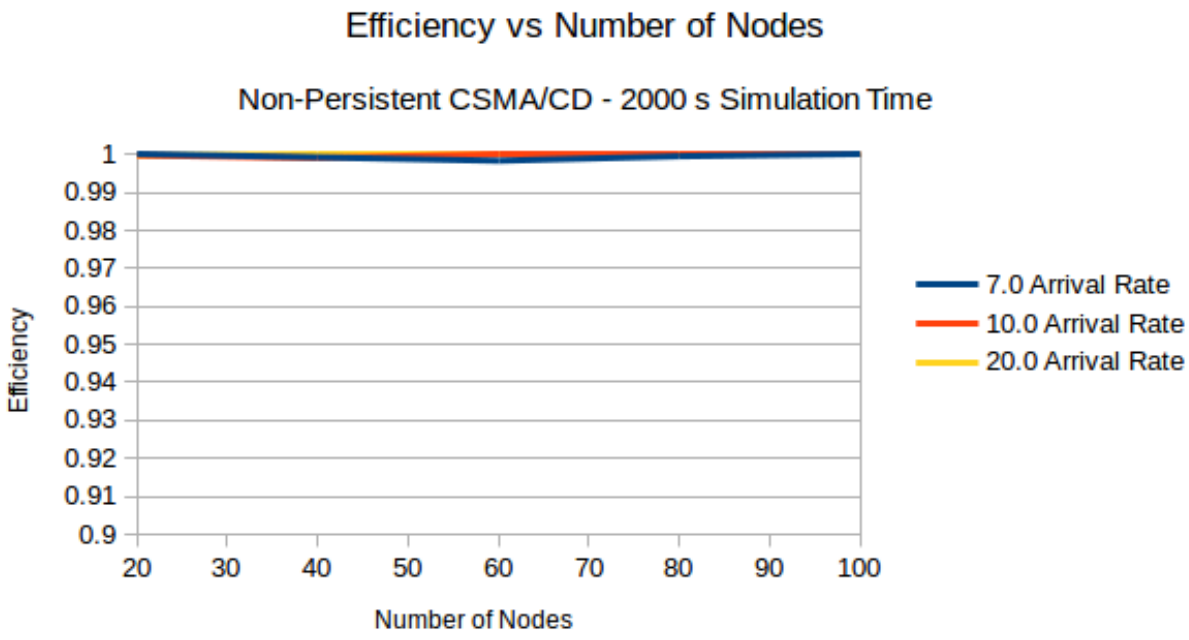

Figure 13 efficiency vs number of nodes for non-persistent, T = 2000

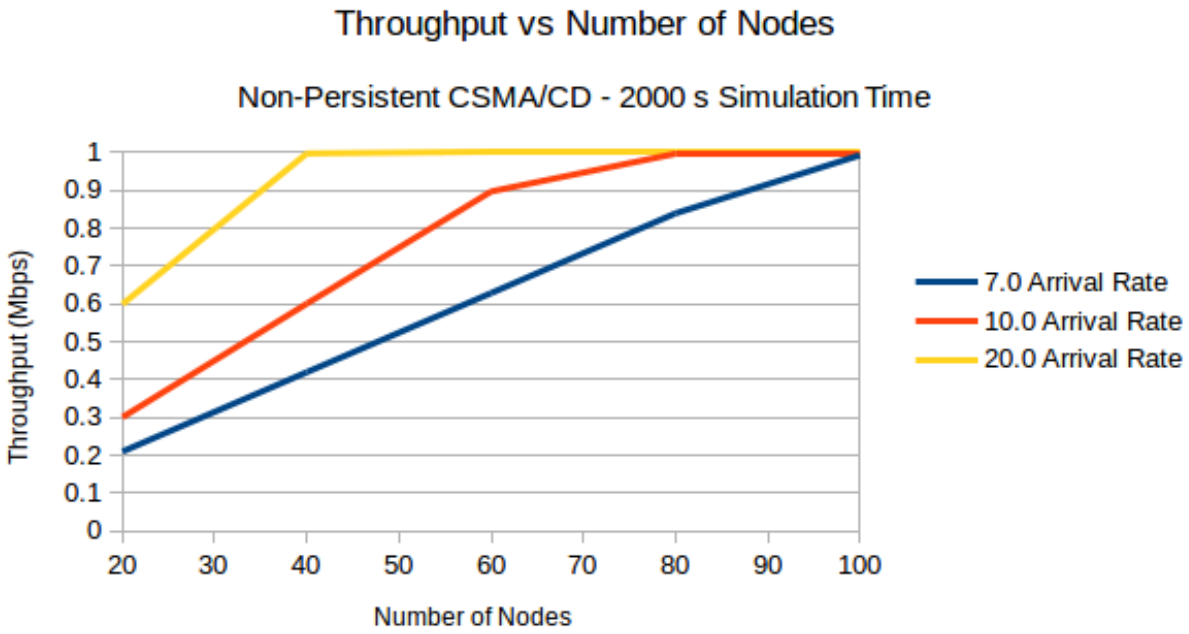

Figure 14 throughput vs number of nodes for non-persistent, T = 2000