University of Waterloo

Faculty of Engineering

Department of Electrical and Computer Engineering

ECE 358 Project 2: CSMA/CD Performance Evaluation

Prepared by

Alex Chun Kei Chiu

ackchiu

20615465

Zi Chao Liang

zcliang

20617479

Table of Contents

[1 1-Persistent CSMA/CD Protocol 3](#_Toc3142930)

[1.1 Efficiency 3](#_Toc3142931)

[1.2 Throughput 4](#_Toc3142932)

[2 Non-persistent CSMA/CD Protocol 5](#_Toc3142933)

[2.1 Efficiency 5](#_Toc3142934)

[2.2 Throughput 6](#_Toc3142935)

[3 Source Code and Design Decisions 6](#_Toc3142936)

[Source Code: Lab2.py 10](#_Toc3142937)

[Source Code: PersistentCSMASimulator.py 10](#_Toc3142938)

[Source Code: NonpersistentCSMASimulator.py 12](#_Toc3142939)

[Source Code: Node.py 14](#_Toc3142940)

[Source Code: Packet.py 16](#_Toc3142941)

# 1 1-Persistent CSMA/CD Protocol

The 1-persistent CSMA/CD protocol was simulated with our code for N (number of packets): 20, 40, 60, 80, 100 and A (average packet rate/s) of 7, 10, and 20. With these values, we attained the graphs and data below with regard to efficiency and throughput.

## Efficiency

Figure 1: Efficiency of 1-Persistent CSMA/CD Protocol

The efficiency of this protocol is a decreasing linear relationship for lower values of A and increasing N (as seen with A = 7). This is expected as when you add more nodes that are trying to transmit, there are bound to be more collision as more nodes are trying to transmit at the same time. If there are more collisions, then the efficiency of the protocol decreases as the equation for efficiency is the following:

However, as you increase the value of A and keep the range of N constant, the relationship becomes more of an inversely proportional curve (as seen with A = 10 and A = 20). This is due to the fact that when the average packets per second sent for each node increases, the nodes are all trying to send more and more packets so the nodes will have higher collision counters and will be in exponential backoff for longer and longer times. This makes it such that almost all the nodes have a different exponential backoff timer and so the rate at which collisions increase will decrease. This in turn will make the rate at which efficiency drops will slow down as you increase the number of nodes.

## Throughput

Figure 2: Throughput of 1-Persistent CSMA/CD Protocol

The throughput of the protocol has the relationship above. The equation for throughput is

For each value of average rate of packets per second (A), the throughput increases at a logarithmic relationship plateauing at around 0.9 Mbps rate as the number of nodes (N) increase. The difference between the different values of A is that the rate at which the throughput increases as you increase the number of nodes increases as the value of A is increased. This results in larger A values plateauing at fewer nodes than lower values of A. This is expected as although the efficiency of the three experimented values vary with relationships described in the previous section, by increasing the average rate at which each node sends out packets, the total amount of packets sent out as the number of nodes increase is much higher so the throughput will also increase a lot faster.

# 2 Non-persistent CSMA/CD Protocol

The Nonpersistent CSMA/CD protocol was simulated with our code for N (number of packets): 20, 40, 60, 80, 100 and A (average packet rate/s) of 7, 10, and 20. With these values, we attained the graphs and data below with regard to efficiency and throughput. This protocol is identical to the 1-persistent CSMA/CD protocol except that when a node senses that the medium is busy, it waits an exponential backoff before sensing the medium again.

## 2.1 Efficiency

Figure 2: Efficiency of Nonpersistent CSMA/CD Protocol

The efficiency of this protocol is very high nearing essentially 100% efficiency for all of A = 7, 10, 20. This is expected as before, when the medium is busy, nodes that are trying to send a packet will constantly poll the medium to be free and when the medium is free, they will all try to send a packet which will result in a collision. From adding an exponential backoff after nodes detect the medium as busy, the nodes will sleep for a random amount of time and will therefore not all collide with each other after the medium is free again. There is a slight difference in the efficiencies between the three rates A = 7, 10, 20 because as the rate of packets being queued increases, there will be slightly more collisions as there are more nodes trying to be sent.

## 2.2 Throughput

Figure 4: Throughput of Nonpersistent CSMA/CD Protocol

The throughput of this protocol is relatively linear because the efficiency is approximately the same so when the number of nodes increases at a linear rate, then the throughput of the nodes will also increase at a linear rate. The throughput for higher A values will have a higher throughput because an increase in the average amount of packets sent will directly increase throughput as the efficiency of the protocol is approximately 100%.

# 3 Source Code and Design Decisions

A class was written to generate exponential random variables whenever the class instance calls a genValue method. In this method, a uniform random variable is generated and uses it along with an inputted lambda value to generate a single exponential random via the inverse method. Theoretically, exponential distributions should have an expected value of and a variance of . This is the same exponential random variable generator that was used in Lab 1. This class was used to generate the arrival times in the queue of each created node. The lambda used for this distribution is the value of “A” or the average packet arrival rate since the average of a Poisson distribution is

The Packet class was used again in for cleanliness of code and ease of understanding the flow of the simulation.

We created a new class called Node that would handle basic functions by the node and commonly used functions associated with it. The *genPacketArrivalEvents* function will populate the queue for the node with arrival times based on a Poisson distribution where the value of lambda (average for the distribution) is the value of “A”. This function is called upon the object’s creation so that each Node will always be populated with a list of packets. There are also other class functions like *waitExponentialBackoff* and *waitExponentialBackoffMediumSensing* that add an exponential backoff to the arrival times of packets and *bufferPackets* that serve to shift the arrival times of all packets within a range of timestamps to a certain value. These functions make designing the simulator easier as we can just call the appropriate class functions when necessary. Lastly, there are helper functions that just serve to simply the code like *removeFirstPacket*, *getFirstPacketTimestamp*, *genExponentialBackoffTime* where it serves to just add a level of transparency to the code we write in the main simulator.

Our two main classes are PersistentCSMASimulator and NonpersistentCSMASimulator that solves questions 1 and 2 respectively. Each of the classes have variables *transmittedPackets* and *succcessfullyTransmittedPackets* in order to properly evaluate the performance of the protocol.

PersistentCSMASimulator has three main stages – createNodes, processPackets, and printResults. The createNodes stage will create the appropriate amount of nodes for the simulation and store it in a local variable.

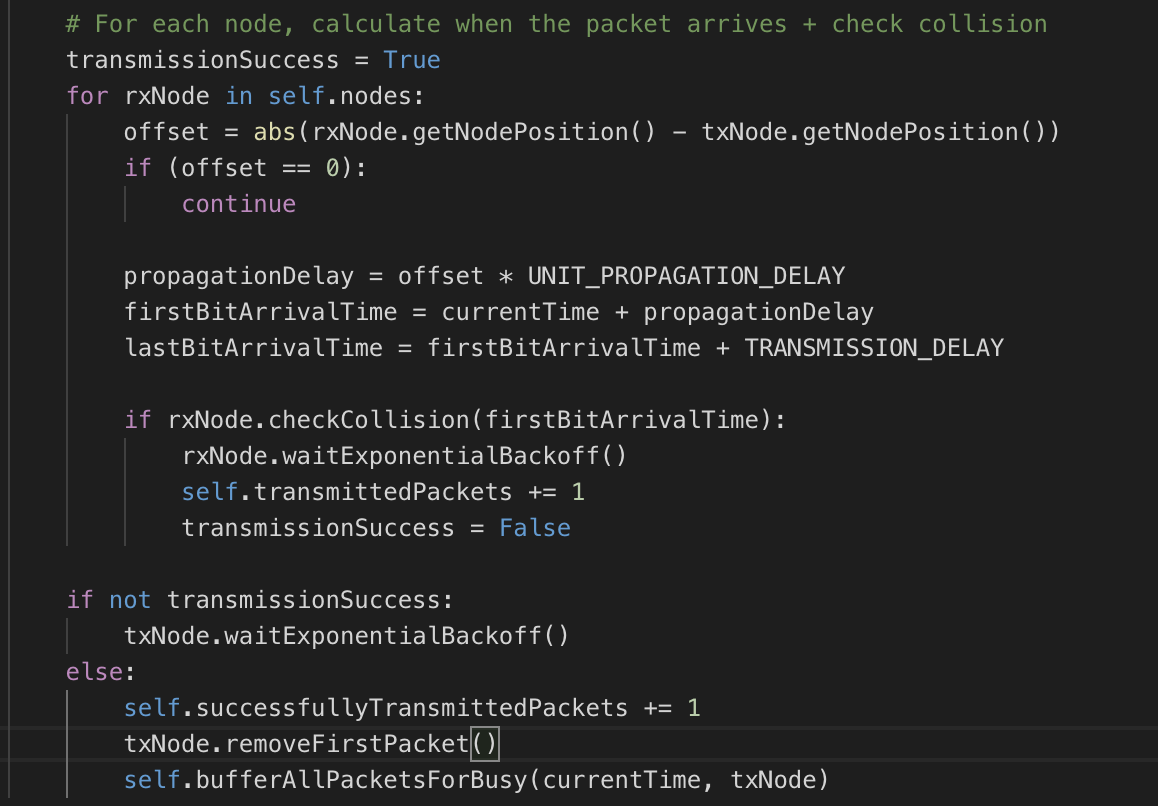


Figure 5: Main logic within each packet transmission

The processPackets stage starts by taking the smallest arrival time within all the nodes and seeing if it collides with any other packets from any other nodes (the *transmittedPackets* counter is incremented regardless is there is a collision). If there is a collision, then both the transmitter and receiver nodes will have exponential backoff and the *transmittedPackets* variable will be incremented for each transmission along with the collision counter incrementing in each node. If a collision does not happen, then the *successfullyTransmittedPackets* counter will be incremented and the rest of the nodes will be checked to see if they had any packets that had arrival times between the timestamps of the first and last bits to arrive at each node. If a node has packets waiting to be sent between these timestamps, then their arrival timestamps would be updated to be the end of the transmission. Lastly, we print the values of the efficiency and throughput of the protocol.

The NonpersistentCSMASimulator is very similar to the PersistentCSMASimulator except that when a node senses that the medium is busy, it doesn’t just try to sense again immediately but instead waits for an exponential backoff time period before trying to sense again. This is done through implementing a new function called *waitExponentialBackoffMediumSensing* function

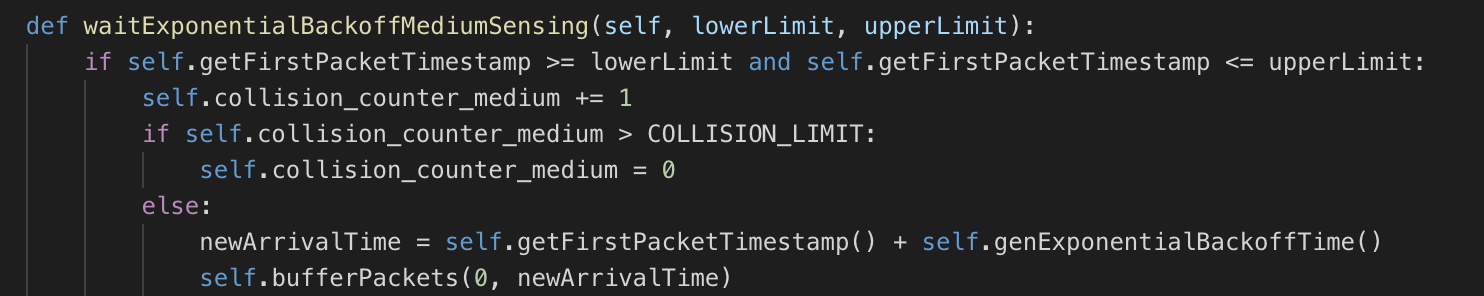


Figure 6: Function to Wait Exponential Backoff for Medium Sensing

that will get called instead of the *bufferPackets* function at the end the *bufferAllPacketsForBusy* function which tells all other nodes that the medium is currently busy.

Through these classes, we were able to attain the results from the previous section. The lab2.py script allows easy execution of the classes described above and its use guide is in the README.md file included.

# Source Code: Lab2.py

|  |
| --- |
| import numpy as np  import time  from PersistentCSMASimulator import PersistentCSMASimulator  from NonpersistentCSMASimulator import NonpersistentCSMASimulator  from ExponentialRandomVariableGenerator import ExponentialRandomVariableGenerator  def question\_1():  for A in [7, 10, 20]:  for N in [20, 40, 60, 80, 100]:  simulator = PersistentCSMASimulator(N, A).run()  def question\_2():  for A in [7, 10, 20]:  for N in [20, 40, 60, 80, 100]:  simulator = NonpersistentCSMASimulator(N, A).run()  # main  question\_number = raw\_input("Enter Question Number [1, 2] ")  question\_number = int(question\_number)  start\_time = time.time()  if question\_number == 1:  question\_1()  elif question\_number == 2:  question\_2() |

# Source Code: PersistentCSMASimulator.py

|  |
| --- |
| from \_\_future\_\_ import division  from Node import Node  SIMULATION\_TIME = 1000 # 1000s  TRANSMISSION\_RATE = 1000000 # 1 Mbps  PACKET\_LENGTH = 1500 # assume all packets are the same length  TRANSMISSION\_DELAY = PACKET\_LENGTH / TRANSMISSION\_RATE  DISTANCE\_BETWEEN\_NODES = 10  PROPAGATION\_SPEED = (2/3) \* 300000000  UNIT\_PROPAGATION\_DELAY = DISTANCE\_BETWEEN\_NODES / PROPAGATION\_SPEED  class PersistentCSMASimulator:  def \_\_init\_\_(self, numNodes, avgPacketArrivalRate):  self.nodes = []  self.numNodes = numNodes  self.avgPacketArrivalRate = avgPacketArrivalRate  # metrics  self.transmittedPackets = 0  self.successfullyTransmittedPackets = 0  def run(self):  self.createNodes()  self.processPackets()  self.printResults()  def createNodes(self):  for i in range(self.numNodes):  self.nodes.append(Node(i, self.avgPacketArrivalRate, SIMULATION\_TIME))  def bufferAllPacketsForBusy(self, currentTime, txNode):  maxOffset = abs((self.numNodes - 1) - txNode.getNodePosition())  maxPropagationDelay = maxOffset \* UNIT\_PROPAGATION\_DELAY  maxFirstBitArrivalTime = currentTime + maxPropagationDelay  maxLastBitArrivalTime = maxFirstBitArrivalTime + TRANSMISSION\_DELAY  for node in self.nodes:  offset = abs(node.getNodePosition() - txNode.getNodePosition())  propagationDelay = offset \* UNIT\_PROPAGATION\_DELAY  firstBitArrivalTime = currentTime + propagationDelay  node.bufferPackets(firstBitArrivalTime, maxLastBitArrivalTime)  def processPackets(self):  while True:  # get the sender node which has the smallest packet arrival time  txNode = min(self.nodes, key=lambda node: node.getFirstPacketTimestamp())  # update the currentTime  currentTime = txNode.getFirstPacketTimestamp()  if currentTime > SIMULATION\_TIME:  break  # A packet is trying to be sent  self.transmittedPackets += 1  # For each node, calculate when the packet arrives + check collision  transmissionSuccess = True  for rxNode in self.nodes:  offset = abs(rxNode.getNodePosition() - txNode.getNodePosition())  if (offset == 0):  continue    propagationDelay = offset \* UNIT\_PROPAGATION\_DELAY  firstBitArrivalTime = currentTime + propagationDelay  lastBitArrivalTime = firstBitArrivalTime + TRANSMISSION\_DELAY  if rxNode.checkCollision(firstBitArrivalTime):  rxNode.waitExponentialBackoff()  self.transmittedPackets += 1  transmissionSuccess = False  if not transmissionSuccess:  txNode.waitExponentialBackoff()  else:  self.successfullyTransmittedPackets += 1  self.bufferAllPacketsForBusy(currentTime, txNode)  txNode.removeFirstPacket()  def printResults(self):  print("================ RESULTS ================")  print("Arrival Rate: {}, NumNodes: {}".format(self.avgPacketArrivalRate, self.numNodes))  print("SuccessFully Transmitted Packets: {}".format(self.successfullyTransmittedPackets))  print("Total Transmitted Packets: {}".format(self.transmittedPackets))  print("Efficiency of CSMA/CD: {}".format((self.successfullyTransmittedPackets / self.transmittedPackets)))  print("Throughput of CSMA/CD: {} Mbps".format(((self.successfullyTransmittedPackets \* PACKET\_LENGTH / 1000000) / SIMULATION\_TIME))) |

# Source Code: NonpersistentCSMASimulator.py

|  |
| --- |
| from \_\_future\_\_ import division  from Node import Node  SIMULATION\_TIME = 1000 # 1000s  TRANSMISSION\_RATE = 1000000 # 1 Mbps  PACKET\_LENGTH = 1500 # assume all packets are the same length  TRANSMISSION\_DELAY = PACKET\_LENGTH / TRANSMISSION\_RATE  DISTANCE\_BETWEEN\_NODES = 10  PROPAGATION\_SPEED = (2/3) \* 300000000  UNIT\_PROPAGATION\_DELAY = DISTANCE\_BETWEEN\_NODES / PROPAGATION\_SPEED  class NonpersistentCSMASimulator:  def \_\_init\_\_(self, numNodes, avgPacketArrivalRate):  self.nodes = []  self.numNodes = numNodes  self.avgPacketArrivalRate = avgPacketArrivalRate  # metrics  self.transmittedPackets = 0  self.successfullyTransmittedPackets = 0  def run(self):  self.createNodes()  self.processPackets()  self.printResults()  def createNodes(self):  for i in range(self.numNodes):  self.nodes.append(Node(i, self.avgPacketArrivalRate, SIMULATION\_TIME))  def createNodes(self):  for i in range(self.numNodes):  self.nodes.append(Node(i, self.avgPacketArrivalRate, SIMULATION\_TIME))  def bufferAllPacketsForBusy(self, currentTime, txNode):  maxOffset = abs((self.numNodes - 1) - txNode.getNodePosition())  maxPropagationDelay = maxOffset \* UNIT\_PROPAGATION\_DELAY  maxFirstBitArrivalTime = currentTime + maxPropagationDelay  maxLastBitArrivalTime = maxFirstBitArrivalTime + TRANSMISSION\_DELAY  for node in self.nodes:  offset = abs(node.getNodePosition() - txNode.getNodePosition())  propagationDelay = offset \* UNIT\_PROPAGATION\_DELAY  firstBitArrivalTime = currentTime + propagationDelay  node.waitExponentialBackoffMediumSensing(firstBitArrivalTime, maxLastBitArrivalTime)  def processPackets(self):  while True:  # get the sender node which has the smallest packet arrival time  txNode = min(self.nodes, key=lambda node: node.getFirstPacketTimestamp())  if not txNode.queue:  break  # update the currentTime  currentTime = txNode.getFirstPacketTimestamp()  # A packet is trying to be sent  self.transmittedPackets += 1  # For each node, calculate when the packet arrives + check collision  transmissionSuccess = True  for rxNode in self.nodes:  offset = abs(rxNode.getNodePosition() - txNode.getNodePosition())  if (offset == 0):  continue    propagationDelay = offset \* UNIT\_PROPAGATION\_DELAY  firstBitArrivalTime = currentTime + propagationDelay  lastBitArrivalTime = firstBitArrivalTime + TRANSMISSION\_DELAY  if rxNode.checkCollision(firstBitArrivalTime):  rxNode.waitExponentialBackoff()  self.transmittedPackets += 1  transmissionSuccess = False  if not transmissionSuccess:  txNode.waitExponentialBackoff()  else:  self.successfullyTransmittedPackets += 1  txNode.removeFirstPacket()  self.bufferAllPacketsForBusy(currentTime, txNode)  def printResults(self):  print("================ RESULTS ================")  print("Arrival Rate: {}, NumNodes: {}".format(self.avgPacketArrivalRate, self.numNodes))  print("SuccessFully Transmitted Packets: {}".format(self.successfullyTransmittedPackets))  print("Total Transmitted Packets: {}".format(self.transmittedPackets))  print("Efficiency of CSMA/CD: {}".format((self.successfullyTransmittedPackets / self.transmittedPackets)))  print("Throughput of CSMA/CD: {} Mbps".format(((self.successfullyTransmittedPackets \* PACKET\_LENGTH / 1000000) / SIMULATION\_TIME))) |

# Source Code: Node.py

|  |
| --- |
| from \_\_future\_\_ import division  from ExponentialRandomVariableGenerator import ExponentialRandomVariableGenerator  from Packet import Packet  from collections import deque  import random  COLLISION\_LIMIT = 10  TRANSMISSION\_RATE = 1000000 # 1 Mbps  class Node:  def \_\_init\_\_(self, position, arrivalTimeLambda, simulationTime):  self.queue = deque()  self.position = position  self.arrivalTimeLambda = arrivalTimeLambda  self.simulationTime = simulationTime  self.collision\_counter = 0  self.collision\_counter\_medium = 0  self.genPacketArrivalEvents()    def genPacketArrivalEvents(self):  # create Arrival Time generator  arrivalTimeGenerator = ExponentialRandomVariableGenerator(lmbda=self.arrivalTimeLambda)  # create arrival events for the simulation  currentTime = 0  while currentTime < self.simulationTime:  # add inter-arrival time to arrive at current timestamp  interArrivalTime = arrivalTimeGenerator.genValue()  currentTime += interArrivalTime  # add packet to queue  self.queue.append(Packet(currentTime))  # Checks if next packet is during a transmission. If next packet  # Arrives before the sender's first bit arrives, bus appears to be idle  def checkIfBusy(self, firstBitArrivalTime, lastBitArrivaltime):  return firstBitArrivalTime < self.getFirstPacketTimestamp() and self.getFirstPacketTimestamp() < lastBitArrivaltime  # If packet arrival < arrival of transmitted first bit, bus appears to be idle  def checkCollision(self, firstBitArrivalTime):  return self.getFirstPacketTimestamp() <= firstBitArrivalTime  def waitExponentialBackoff(self):  self.collision\_counter += 1  if self.collision\_counter > COLLISION\_LIMIT:  self.removeFirstPacket()  else:  # Each node waits backoff time. Means we start waiting from our first packet time  newArrivalTime = self.getFirstPacketTimestamp() + self.genExponentialBackoffTime()  self.bufferPackets(0, newArrivalTime)  def waitExponentialBackoffMediumSensing(self, lowerLimit, upperLimit):  if self.getFirstPacketTimestamp >= lowerLimit and self.getFirstPacketTimestamp <= upperLimit:  self.collision\_counter\_medium += 1  if self.collision\_counter\_medium > COLLISION\_LIMIT:  self.collision\_counter\_medium = 0  else:  newArrivalTime = self.getFirstPacketTimestamp() + self.genExponentialBackoffTime()  self.bufferPackets(0, newArrivalTime)  # Pushes packet timestamps to an upper limit given a range  def bufferPackets(self, lowerLimit, upperLimit):  for packet in self.queue:  if packet.timestamp >= lowerLimit and packet.timestamp <= upperLimit:  packet.timestamp = upperLimit  elif packet.timestamp > upperLimit:  break  def genExponentialBackoffTime(self):  # generate a random number between 0 and 2^i-1  R = random.randint(0, (2\*\*self.collision\_counter) - 1)  # random number \* 512 bit-time  backoff = R \* 512 \* (1.0 / TRANSMISSION\_RATE)  return backoff  def removeFirstPacket(self):  self.queue.popleft()  self.collision\_counter = 0  def getFirstPacketTimestamp(self):  if self.queue:  return self.queue[0].timestamp  else:  return float('inf')  def getNodePosition(self):  return self.position |

# Source Code: Packet.py

|  |
| --- |
| TRANSMISSION\_RATE = 1000000 # 1 Mbps  class Packet:  def \_\_init\_\_(self, length):  self.length = length  def getTransmissionTime(self):  return self.length / TRANSMISSION\_RATE |