Bisakh Mondal
Roll: 001810501079 (A3)
BCSE-III
Computer Networks
Assignment-III (CSMA)

INDEX

Problem: Implement 1-persistent, non-persistent and p-persistent CSMA techniques.

In this assignment, you have to implement 1-persistent, non-persistent and p-persistent CSMA techniques. Measure the performance parameters like throughput (i.e., average amount of data bits successfully transmitted per unit time) and forwarding delay (i.e., average end-to-end delay, including the queuing delay and the transmission delay) experienced by the CSMA frames (IEEE 802.3). Plot the comparison graphs for throughput and forwarding delay by varying p. State your observations on the impact of performance of different CSMA techniques.

Deadline: 16 December 2020
Submission: 16 December 2020

Design:

The purpose of the program is to simulate the end to end communication between sender and receiver using different CSMA techniques.

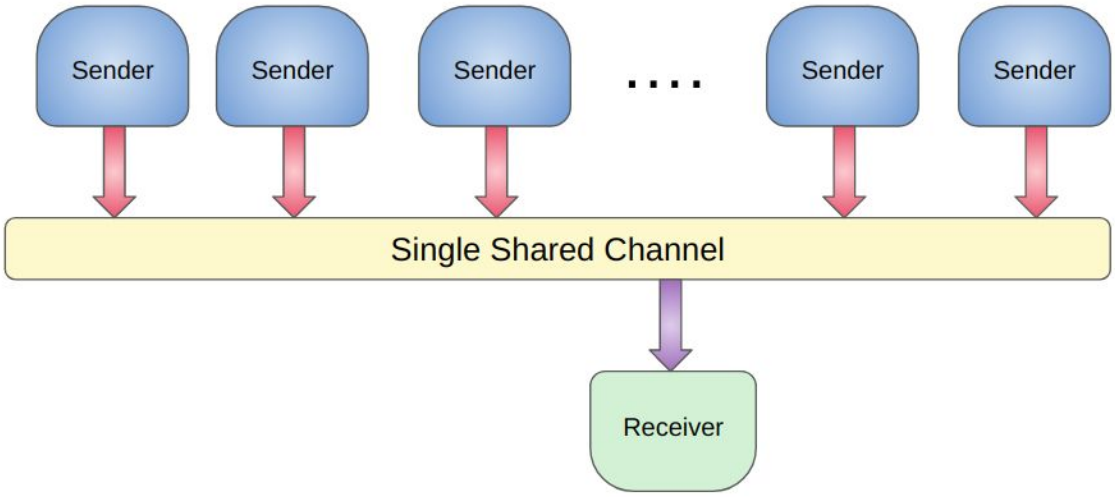
CSMA or Carrier Sense Multiple Access is a MAC (Medium access Control) method used in earlier ethernet or LAN technology. The idea behind such implementation comes into place when there is more than one number of nodes(sender and receivers) connected in a **single shared link/channel** and dependent upon it solely for data transmission - a suitable example is the nodes in **Bus Topology**. CSMA itself suggests carrier-sensing before initiating transmission on the shared channel to defer collisions among data of different sender nodes.

So in my implementation I have considered the **NUM_NODES** number of sender nodes and **single** receiver nodes and a channel connecting them. The purpose is to show the performance of different persistent schemes, so keeping a static receiver in a domain of multiple dynamic senders serves the purpose while making the design slightest simpler. There is three main **package** in the whole design technique:

- sender

- receiver
- channel
- Utils (Supportive package)

One thing need to be mentioned here, as CSMA/CD is **not dependent on acknowledge system,** in my design I am creating two central goroutines which are responsible for delivering the **status of the medium** (i.e. IDLE or BUSY), and **collision status** (if a collision occurred during transmission) to the nodes connected to the channel when they start sensing.



Generic Overview

THE COLLISION STATUS AUTHORITY

```
package utils


const (
      COLLIDED = 0
      NCOLLIDED = 1
)

var addcountC = make(chan int) // set current collision status
var counterC = make(chan int) //  get current collsion track

func SetNotCollided() { addcountC <- NCOLLIDED }
func SetCollided() { addcountC <- COLLIDED }
```

```
func CollisionStatus() int { return <-counterC }


func TellerCollision() {
	var curStatus int = NCOLLIDED // Status of transmission
	for {
		select {
		case curStatus = <-addcountC:
		case counterC <- curStatus:
		}
	}
}
```

THE MEDIUM STATUS AUTHORITY

```
package utils

const (
	BUSY = 1
	IDLE = 0
)
var addcount = make(chan int) // set current medium status
var counter = make(chan int) //  get current medium status

func SetMediumBusy() { addcount <- BUSY }
func SetMediumIdle() { addcount <- IDLE }

func SenseMedium() int { return <-counter }


func TellerMedium() {
	var curStatus int = IDLE // Status of medium is confined to only Teller
goroutine
	for {
		select {
		case curStatus = <-addcount:
		case counter <- curStatus:
		}
	}
```

```
}
```

Both `TellerCollision() & TellerMedium()` are concurrent threads providing the status regarding medium to the **concurrent** sender nodes in a **thread-safe** manner without the overhead of **incorporating mutex.**

Frame Format

Here for better generalization and uniformity, **IEEE 802.3** frame format has been used throughout the communication. The fields are
Preamble + SFD + DEST_MAC + SRC_MAC + Node_Info(custom addition) + DATA + CRC. 7 + 1 + 6 + 6 + 2 +46 + 4 = 72 Bytes
Each data packet is encoded with **CRC32** for detecting transmission error.

Augmented IEEE 802.3 frame format

Sender Node

The `node.go` file in the **sender package** contains a generic structure of a single sender node, which comprises of

- `buffer [] Bytestream` -> A temporary **buffer where frames are coming at real time.**
- `S2C chan <- Bytestream` -> A buffered pipe between sender and channel
- `collisions int` -> the collision count of current frame that is being sent
- `NodeNo int` -> The information about the node
- `srcMac int` -> Node's Mac Address
- `destMac int` -> Destination Mac address

Each node has a **PACKET_ARRIVAL_RATE,** i.e number of frames coming to the sender per second. The constant signifies that the making a frame from raw data incurs some **frame creation time.** So the function `GenerateFrames` adds frame to the buffer in real time. The three files `onePersistent.go` `nonPersistent.go` `pPersistent.go` contains the logic of three different persistent schemes. The member function of the sender node `Init()` sends packets to the channel depending upon the desired scheme.

Channel

It plays a crucial role in frame transmission between sender and receiver while also detecting collision during transmission. It is responsible for discarding frames if collision is detected while also transmitting a collision signal to the **Collision authority** or in terms the sender node to take proper action depending upon the scheme. If no collision is detected the passes the frame to the receiver node. The channel structure

- `C2S chan string` -> Buffered pipe connecting multiple sender to the channel
- `C2R chan string` -> Unbuffered pipe connecting channel and single receiver.
- `Buffer * SyncBuffer` -> A thread synchronised buffer to temporarily hold data for sending to receiver.
- `monitor * StatsMonitor` -> A Stats monitor utility which is responsible for calculation of throughput and efficiency.

The `channel.go` in the channel package contains the logic behind the channel operation.

It receives frames from sender

```
func (c* Channel)receivefromSender(){
      for {
            //make the collision flag unset
            SetNotCollided()

            if SenseMedium() == BUSY{
                  //if more than one frames in the pipe then collision

                  if len(c.C2S) > 1 {
                        //collision
                        SetCollided()
```

```
                        //log into monitor for throughput and efficiency
                        c.monitor.Transmitted(len(c.C2S))

                        log.Println("Collision")
                        //free the pipe, data is unusable
                        for len(c.C2S) > 0 {
                              <-c.C2S
                        }

                  }else{
                        //there is a single frame inside the channel
                        //putting it into buffer
                        bitstream := <-c.C2S
                        c.Buffer.Push(bitstream)

                        //log into monitor for throughput and efficiency
                        c.monitor.Success()
                  }

                  //the data needs to be captured from medium which will
incur some time penalty
                  time.Sleep(800* time.Microsecond)

                  //making the medium idle again
                  SetMediumIdle()
            }

      }
}
```

A goroutine/thread responsible for delivering frames stored buffer to receiver.

```
func (c *Channel)send2receiver(wg  *sync.WaitGroup){
      defer wg.Done()
      for {

            if c.Buffer.Len()==0{
                  time.Sleep(500*time.Microsecond)
```

```
                continue
            }

            bytestream := c.Buffer.Pop()
            c.C2R <- bytestream
        }
        log.Println("exited from s2r")
```

The StatsMonitor Utility

During Channel operation it is responsible for keeping track of the efficiency and throughput.

```
type StatsMonitor struct {
    TotalTransmittedPacket, SuccessfulTransmittedPacket int

}

func NewStatusMonitor() *StatsMonitor{
    return &StatsMonitor{0,0}
}

func (s * StatsMonitor)Success(){
    s.SuccessfulTransmittedPacket++
    s.TotalTransmittedPacket++
}
func (s *StatsMonitor)Transmitted(cnt int){
    s.TotalTransmittedPacket+=cnt
}

func (s *StatsMonitor)Stats(timetaken float64){
    //timetaken in microseconds
    log.Println("Efficiency: ", float64(s.SuccessfulTransmittedPacket)
```

```
/ float64(s.TotalTransmittedPacket))
      //bits transferred per second
      throughput := float64(s.SuccessfulTransmittedPacket * FRAMELENGTH)
/ timetaken //in bits per microsecond
      throughput = throughput * math.Pow(10, -6) * math.Pow(10, 6)
      //print(float64(s.SuccessfulTransmittedPacket)/
      //
float64(s.TotalTransmittedPacket), ",",throughput)
      log.Println("Throughput: ", throughput," Mbps")
}
```

Receiver Node

Here the role of the receiver is very simple: receive the frame that has arrived without any collision. The `receiver.go` in the **receiver package** contains the necessary logic. The structure of receiver node is,

`C2R chan Bytestream` -> The pipe through which frames are coming.

It is getting initialized by `Init()` function which continuously receives the incoming frame.

Different Access Methods

One Persistent CSMA

1-persistent CSMA is an aggressive transmission algorithm. When the transmitting node is ready to transmit, it senses the transmission medium for idle or busy. If idle, then it transmits immediately. If busy, then it senses the transmission medium continuously until it becomes idle, then transmits the frame unconditionally (i.e. with probability=1). In case of a collision, the sender waits for a random period of time (**backoff algorithm)** and attempts the same procedure again

The **algorithm**: (in `onePersistent.go` inside the **sender package**)

```
func (s * Sender)OnePersistent(){
	curFrame := s.buffer[0]

	for len(s.buffer)!=0 {

		if SenseMedium() ==IDLE {
			//marking the medium busy to gain access
			SetMediumBusy()
			//sending the frame
			s.S2C <- curFrame
			//wait for if collision signal during propagation
			time.Sleep(500*time.Microsecond)

			if CollisionStatus() == COLLIDED{
				//if collided
				s.collisions++

				if s.collisions >= MAX_COLLISIONS {
					//drop the frame
					s.buffer = s.buffer[1:]
					//reset collision count
					s.collisions=0
					if len(s.buffer) !=0{
						curFrame = s.buffer[0]
					}
					continue //no need to wait further start afresh
				}

				//wait for timeslots depending the back-off algorithm
				k := rand.Intn(int(math.Pow(2,
float64(s.collisions))))

				time.Sleep(time.Duration(k*TIMESLOTS)*
time.Microsecond)

			}else{
				//transmission successful
				s.buffer = s.buffer[1:]
				//reset collision count
				s.collisions=0
				if len(s.buffer) !=0{
					curFrame = s.buffer[0]
				}
```

```
                }

            } else{
                //do nothing Aggressive algorithm
            }
      }
}
```

Non Persistent CSMA

Non persistent CSMA is a non aggressive transmission algorithm. When the transmitting node is ready to transmit data, it senses the transmission medium for idle or busy. If idle, then it transmits immediately. If busy, then it waits for a random period of time (during which it does not sense the transmission medium) before repeating the whole logic cycle (which started with sensing the transmission medium for idle or busy) again. This approach reduces collision.

The **algorithm**: (in `nonPersistent.go` inside the **sender package**)

```
func (s * Sender)NonPersistent(){
      curFrame := s.buffer[0]

      for len(s.buffer)!=0 {

            if SenseMedium() ==IDLE {
                  //marking the medium busy to gain access
                  SetMediumBusy()
                  //sending the frame
                  s.S2C <- curFrame
                  //wait for collision signal during propagation
                  time.Sleep(500*time.Microsecond)

                  if CollisionStatus() == COLLIDED{
```

```
                        //if collided
                        s.collisions++

                        if s.collisions >= MAX_COLLISIONS {
                              //drop the frame
                              s.buffer = s.buffer[1:]
                              //reset collision count
                              s.collisions=0
                              if len(s.buffer) !=0{
                                    curFrame = s.buffer[0]
                              }
                              continue //no need to wait further start
afresh
                        }

                        //wait for timeslots depending the back-off
algorithm
                        k := rand.Intn(int(math.Pow(2,
float64(s.collisions))))

                        time.Sleep(time.Duration(k*TIMESLOTS)*
time.Microsecond)

                  }else{
                        //transmission successful
                        s.buffer = s.buffer[1:]
                        //reset collision count
                        s.collisions=0
                        if len(s.buffer) !=0{
                              curFrame = s.buffer[0]
                        }
                  }

            } else{
                  //Wait Non Aggressive algorithm wait for random amount
of time
                  randn := rand.Intn(5) //within 2.5 second
                  time.Sleep(time.Duration(randn* TIMESLOTS)*
time.Microsecond)
```

```
            }
      }
}
```

P Persistent CSMA

This is an approach between 1-persistent and non-persistent CSMA access modes. When the transmitting node is ready to transmit data, it senses the transmission medium for idle or busy. If idle, then it transmits immediately. If busy, then it senses the transmission medium continuously until it becomes idle, then transmits with probability p. If the node does not transmit (the probability of this event is 1-p), it waits until the next available time slot. If the transmission medium is not busy, it transmits again with the same probability p. This probabilistic hold-off repeats until the frame is finally transmitted or when the medium is found to become busy again.

The **algorithm**: (in `pPersistent.go` inside the **sender package**)

```
const Probability = 0.5

func (s * Sender)PPersistent(){
      curFrame := s.buffer[0]

      for len(s.buffer)!=0 {

            if SenseMedium() ==IDLE {

                  //send with probability "Probability"
                  if rand.Float64() <= Probability {

                        //marking the medium busy to gain access
                        SetMediumBusy()
```

```
                        //sending the frame
                        s.S2C <- curFrame
                        //wait for collision signal during propagation
                        time.Sleep(500 * time.Microsecond)

                        if CollisionStatus() == COLLIDED {
                              //if collided
                              s.collisions++

                              if s.collisions >= MAX_COLLISIONS {
                                    //drop the frame
                                    s.buffer = s.buffer[1:]
                                    //reset collision count
                                    s.collisions = 0
                                    if len(s.buffer) != 0 {
                                          curFrame = s.buffer[0]
                                    }
                                    continue //no need to wait further
start afresh
                              }

                              //wait for timeslots depending the back-off
algorithm
                              k := rand.Intn(int(math.Pow(2,
float64(s.collisions))))

                              time.Sleep(time.Duration(k*TIMESLOTS) *
time.Microsecond)

                        } else {
                              //transmission successful
                              s.buffer = s.buffer[1:]
                              //reset collision count
                              s.collisions = 0
                              if len(s.buffer) != 0 {
                                    curFrame = s.buffer[0]
                              }
                        }
                  }else{
```

```
                    //wait for next time slot with probability (1-p)
                    time.Sleep(time.Duration(TIMESLOTS)*
time.Microsecond)
                }

            } else{
                //sense continuously if busy
            }
        }
}
```

Test Cases & Outputs:

As mentioned earlier the sender nodes generates frames containing random data in real time and appends it in its own buffer. The **PACKET_ARRIVAL_RATE** decides the number of packets arriving per second to the particular node until the total time is less than the **MAX_SIMULATION_TIME.**

Keeping the following constants

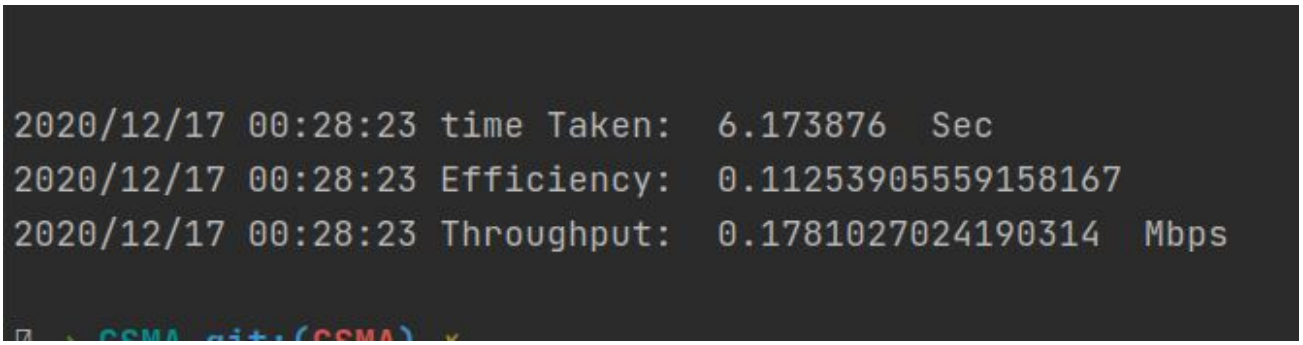
```
MAXSIMULATETIME= 10 //simulate for this number of second Seconds

NUM_NODES = 5 //Number of Sender Node

PACKET_ARRIVAL_RATE=7//Number of packets arriving per Second
```

One Persistent:

Nodes: 50



Nodes: 5

```
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  17822
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  21299
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  19941
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  32388
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  4841
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  445
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  20506
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  22930
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  704
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  4372
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  13176
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  16185
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  21048
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  25309
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  20966
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  771
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  14342
2020/12/17 00:26:45 Frame Received: Node 4  | Data:  10285


2020/12/17 00:26:45 time Taken:  0.740154  Sec
2020/12/17 00:26:45 Efficiency:  0.2913752913752914
2020/12/17 00:26:45 Throughput:  0.29183115946140936  Mbps

 → CSMA git:(CSMA) ✗
```

Non Persistent:

Nodes: 50

```
2020/12/17 00:29:46 time Taken:  8.753883  Sec
2020/12/17 00:29:46 Efficiency:  0.5742983751846381
2020/12/17 00:29:46 Throughput:  0.38374193486479086  Mbps
```

Nodes: 5

```
2020/12/17 00:25:24 Frame Received: Node 4  | Data:  4499
2020/12/17 00:25:24 Frame Received: Node 1  | Data:  6104
2020/12/17 00:25:24 Frame Received: Node 3  | Data:  18458
2020/12/17 00:25:24 Collision
2020/12/17 00:25:24 Frame Received: Node 1  | Data:  8723
2020/12/17 00:25:24 Frame Received: Node 3  | Data:  8846
2020/12/17 00:25:24 Frame Received: Node 1  | Data:  13643
2020/12/17 00:25:24 Frame Received: Node 2  | Data:  5207
2020/12/17 00:25:24 Frame Received: Node 1  | Data:  15879
2020/12/17 00:25:24 Frame Received: Node 2  | Data:  11040
2020/12/17 00:25:24 Frame Received: Node 3  | Data:  5607
2020/12/17 00:25:24 Frame Received: Node 1  | Data:  27813
2020/12/17 00:25:24 Frame Received: Node 1  | Data:  11163
2020/12/17 00:25:24 Frame Received: Node 1  | Data:  4556
2020/12/17 00:25:24 Frame Received: Node 3  | Data:  31454
2020/12/17 00:25:24 Frame Received: Node 1  | Data:  7317
2020/12/17 00:25:24 Frame Received: Node 1  | Data:  11601
2020/12/17 00:25:24 Frame Received: Node 3  | Data:  22008
2020/12/17 00:25:24 Frame Received: Node 1  | Data:  30921
2020/12/17 00:25:24 Frame Received: Node 3  | Data:  32289
2020/12/17 00:25:24 Frame Received: Node 3  | Data:  7776


2020/12/17 00:25:24 Frame Received: Node 3  | Data:  24230
2020/12/17 00:25:24 time Taken:  0.992243  Sec
2020/12/17 00:25:24 Efficiency:  0.9131614654002713
2020/12/17 00:25:24 Throughput:  0.3906784930707498  Mbps

→ CSMA git:(CSMA) ✗
```

P persistent (p=0.5)

Nodes: 50

```
2020/12/17 00:32:44 time Taken:  7.470269999999999  Sec
2020/12/17 00:32:44 Efficiency:  0.136325852036575 24
2020/12/17 00:32:44 Throughput:  0.1896798910882739  Mbps

 → CSMA git:(CSMA) ✗
```

Nodes: 5

```
2020/12/17 00:33:31 Frame Received: Node 5  | Data:  23716
2020/12/17 00:33:31 Frame Received: Node 3  | Data:  28879
2020/12/17 00:33:31 Collision
2020/12/17 00:33:31 Frame Received: Node 1  | Data:  658
2020/12/17 00:33:31 Frame Received: Node 1  | Data:  7555
2020/12/17 00:33:31 Collision
2020/12/17 00:33:31 Frame Received: Node 5  | Data:  21293
2020/12/17 00:33:31 Frame Received: Node 5  | Data:  16634
2020/12/17 00:33:31 Collision
2020/12/17 00:33:31 Frame Received: Node 5  | Data:  93
2020/12/17 00:33:31 Collision
2020/12/17 00:33:31 Frame Received: Node 1  | Data:  27988
2020/12/17 00:33:31 Frame Received: Node 3  | Data:  13546
2020/12/17 00:33:31 Frame Received: Node 3  | Data:  24602
2020/12/17 00:33:31 Collision
2020/12/17 00:33:31 Frame Received: Node 1  | Data:  19266
2020/12/17 00:33:31 Frame Received: Node 5  | Data:  4494
```

```
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  17559
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  14266
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  14803
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  25464
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  14210
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  18166
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  15980
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  3051
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  14242
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  12412
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  4364
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  13831
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  4899
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  2309
2020/12/17 00:33:32 Frame Received: Node 2  | Data:  2803
2020/12/17 00:33:32 time Taken:  1.1525159999999999  Sec
2020/12/17 00:33:32 Efficiency:  0.6392276422764228
2020/12/17 00:33:32 Throughput:  0.31435919327801O9  Mbps
```

ANALYSIS

To have a better idea how the algorithms are performed several tests have been performed **varying the number of nodes from 1 to 1000** (extremely crowded situation) and different metrics have been calculated, for eg.

- **Efficiency :** successfully transmitted packets **/** total transmitted packets
- **Throughput:** amount of successful bits transferred per second though the medium
- **TTS**: Total transmission time of different algorithms
- **Forwarding Delay**: average delay per successfully transmitted packets during transmission.

Collected Data

One Persistent

Nodes	TTS (seconds) -1P	Efficiency -1P	Throughput (Mbps) -1P
1	0.21	1.00	0.41
5	1.08	0.44	0.29
10	2.41	0.35	0.24
20	4.75	0.20	0.18
30	6.97	0.19	0.18
50	10.95	0.14	0.17
75	15.08	0.11	0.16
100	20.44	0.09	0.15
250	44.51	0.04	0.10
500	81.27	0.02	0.07
1000	152.71	0.01	0.03

Non Persistent

Nodes	TTS (seconds) -NP	Efficiency -NP	Throughput (Mbps) -NP
1	0.33	1.00	0.23
5	1.16	0.88	0.34
10	2.29	0.77	0.32
20	4.42	0.69	0.32
30	6.35	0.64	0.33
50	10.36	0.49	0.30
75	15.11	0.40	0.29
100	19.63	0.35	0.27
250	46.30	0.17	0.20
500	79.14	0.08	0.14
1000	142.09	0.03	0.08

0.1 Persistent

Nodes	TTS (seconds) -0.1P	Efficiency -0.1P	Throughput (Mbps) -0.1P
1	0.91	1.00	0.09
5	2.96	0.88	0.13
10	3.13	0.59	0.23
20	6.59	0.38	0.18
30	9.54	0.29	0.17
50	17.15	0.20	0.14
75	21.91	0.16	0.15
100	30.49	0.13	0.13
250	73.22	0.05	0.07
500	112.75	0.02	0.04
1000	251.32	0.01	0.03

0.5 Persistent

Nodes	TTS (seconds) -0.5P	Efficiency -0.5P	Throughput (Mbps) -0.5P
1	0.26	1.00	0.31
5	1.13	0.48	0.29
10	2.75	0.42	0.22
20	5.70	0.25	0.18
30	7.23	0.20	0.19
50	18.54	0.15	0.10
75	16.56	0.11	0.16
100	20.59	0.09	0.14
250	53.98	0.04	0.09
500	90.22	0.02	0.06
1000	142.88	0.01	0.04

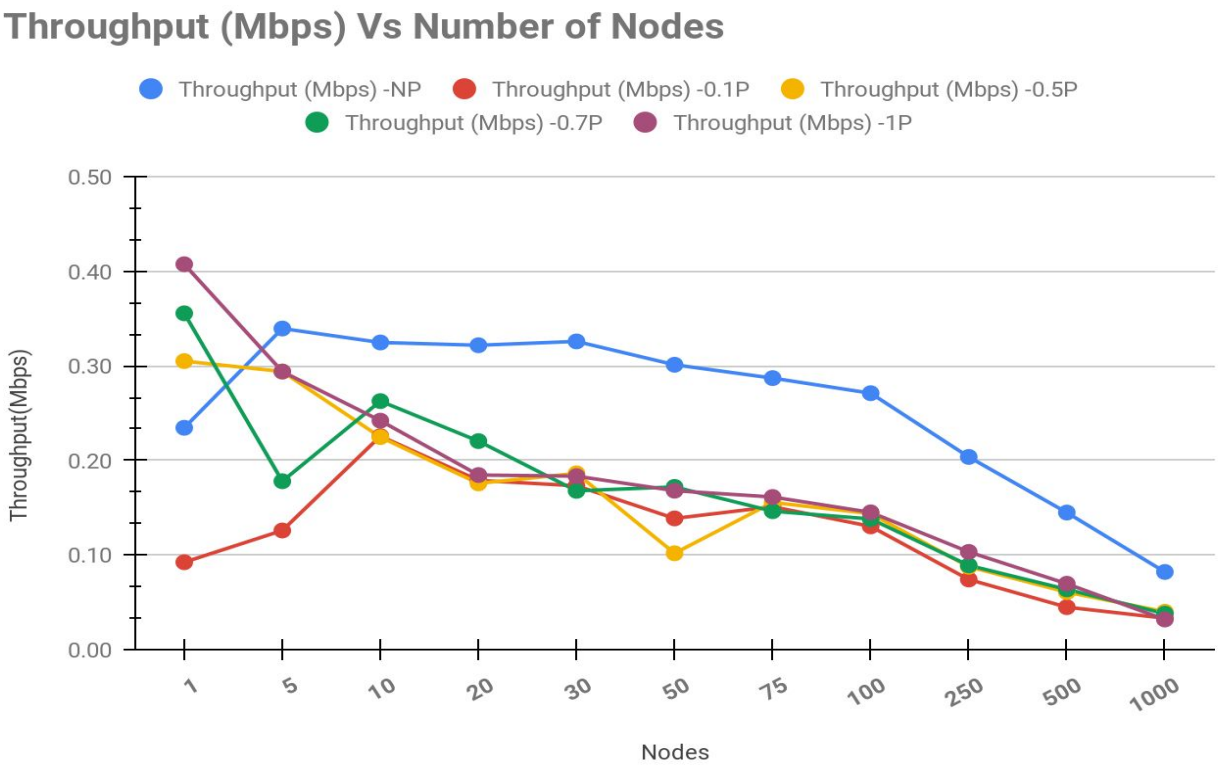
0.7 Persistent

Nodes	TTS (seconds) -0.7P	Efficiency -0.7P	Throughput (Mbps) -0.7P
1	0.22	1.00	0.36
5	1.90	0.49	0.18
10	2.13	0.33	0.26
20	4.53	0.25	0.22
30	7.43	0.18	0.17
50	11.04	0.15	0.17
75	16.32	0.11	0.15
100	21.50	0.09	0.14
250	49.37	0.04	0.09
500	85.05	0.02	0.06
1000	138.26	0.01	0.04

Charts:

- The efficiency is a fraction between 0 and 1.
- The throughput is measured in Mbps i.e Megabits per second.
- The forwarding delay is calculated in Milliseconds.
- The Total transmission time i.e TTS has been calculated in Seconds.

Throughput Vs Total Number of nodes

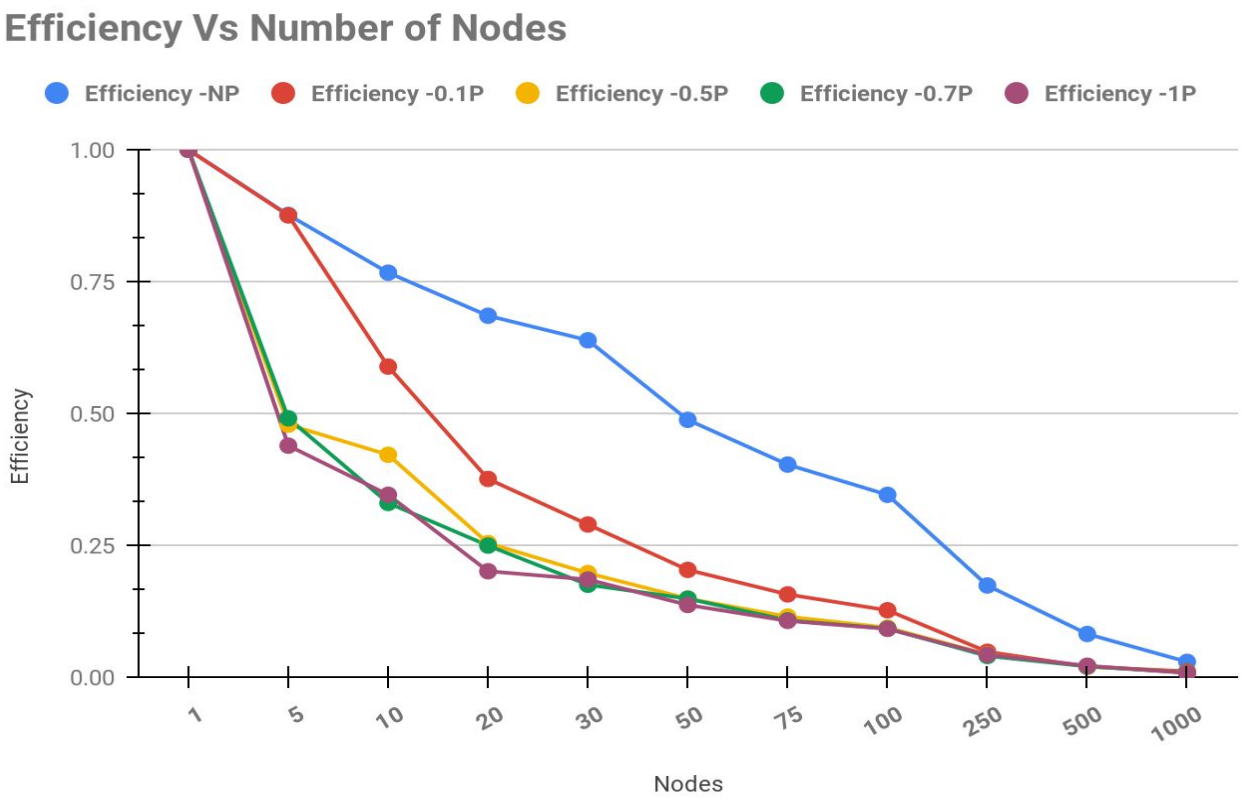


Observations:

- The throughput is much higher in 1-Persistent scheme when the number of concurrent senders is very low due to its high aggressive nature.

- But the non-persistent method overtooks all the algorithms in terms of throughput in the channel but it takes some good amount of time while doing so.

Efficiency Vs Total Number of node



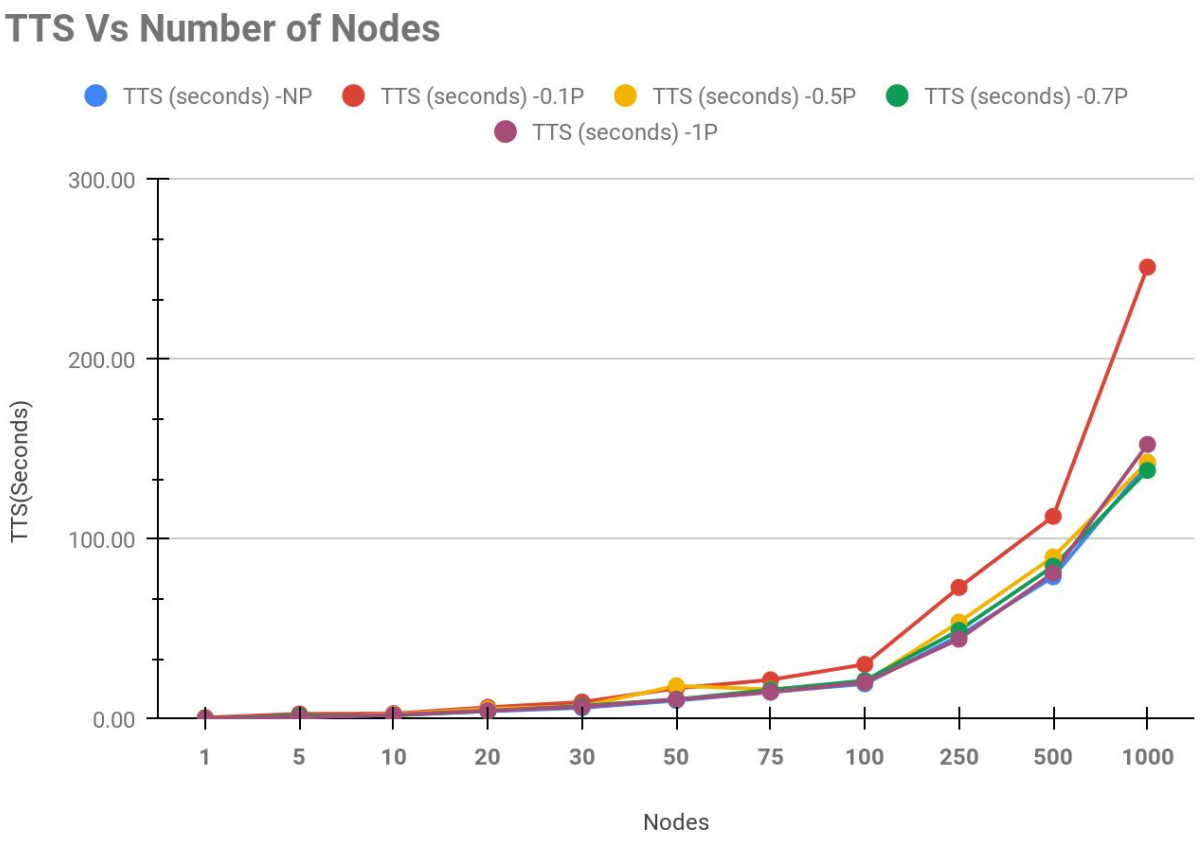
Observations:

- All have same efficiency in single node i.e 100%

- Again Non persistent maintains higher efficiency even in the extremely high collision domain.

- 1-persistent performs very poorly if the number of nodes gets gradually increased.

TTS Vs Total Number of nodes



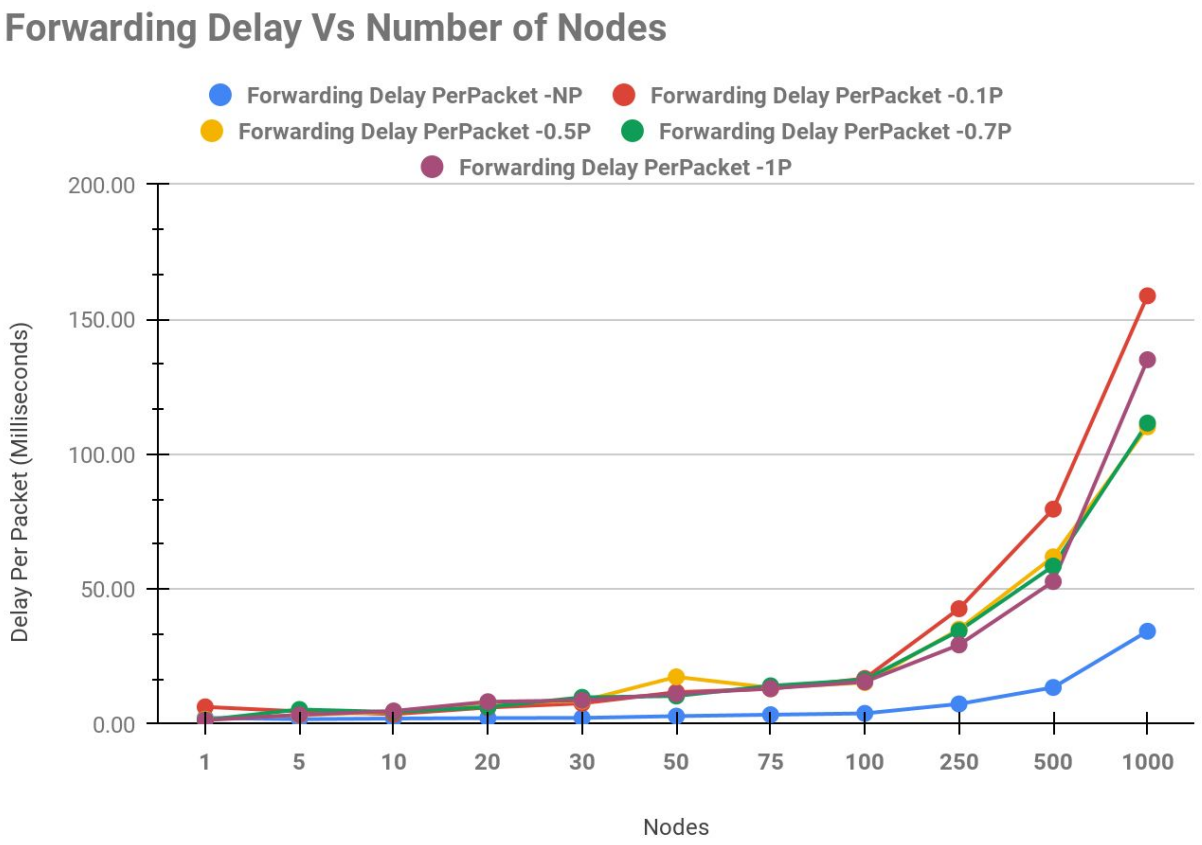
Observations:

- It's obvious that total transmission time will be increased w.r.t the number of nodes as the channel has to transfer more amount of data.

- More collisions make the graph steeper.

- **One important observation**: For total nodes greater than 50, the total transmission time of 0.1- persistent is greater than the 1-Persistent method.

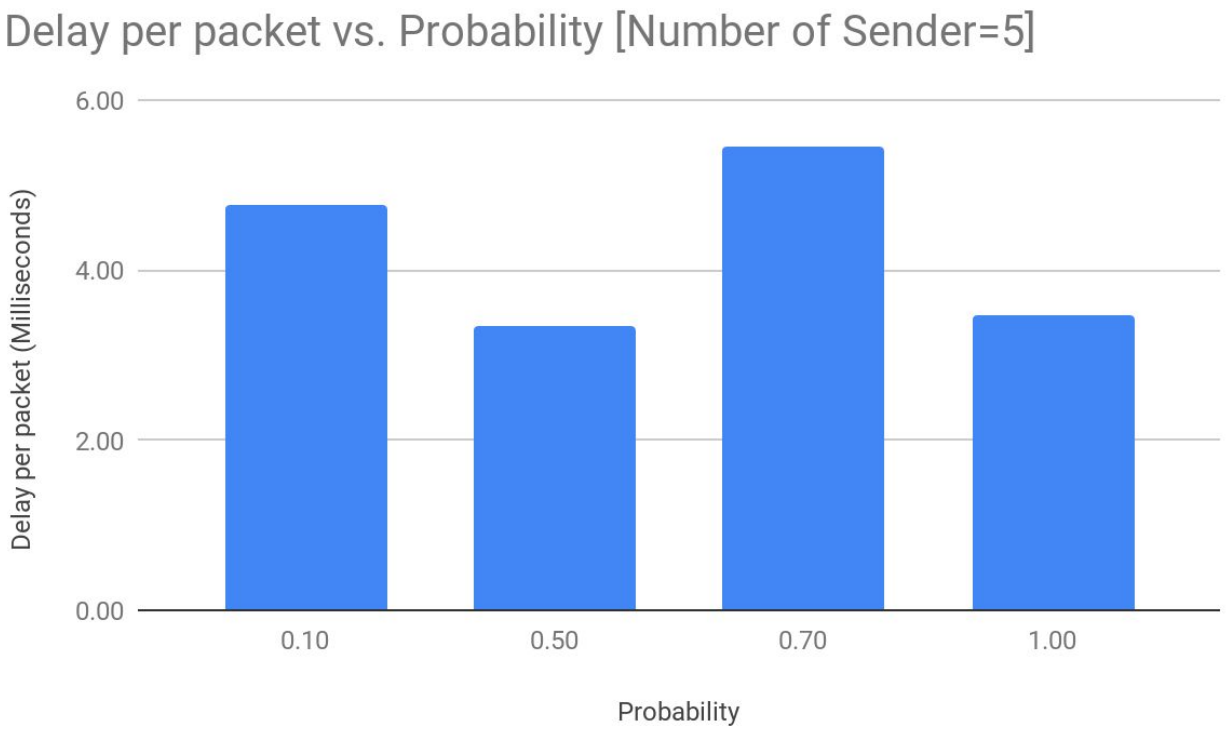
Forwarding delay Vs Total Number of nodes



Data Gathered:

Nodes	Forwarding Delay PerPacket -1P	Forwarding Delay PerPacket -NP	Forwarding Delay PerPacket -0.1P	Forwarding Delay PerPacket -0.5P	Forwarding Delay PerPacket -0.7P
1	1.51	2.33	6.41	1.83	1.57
5	3.46	1.86	4.76	3.33	5.46
10	4.90	2.10	3.74	4.59	4.53
20	8.32	2.27	6.18	7.89	6.39
30	8.84	2.33	7.74	8.60	9.95
50	11.27	2.99	11.87	17.53	10.45

75	13.26	3.52	13.11	13.54	14.22
100	15.71	4.00	16.93	15.44	16.44
250	29.39	7.50	42.79	35.21	34.61
500	52.86	13.62	79.69	62.02	58.61
1000	135.06	34.44	158.74	110.21	111.58



Another Key Observation

Capture effect due to Back- off algorithm

As upon receiving a collision signal if the sender node detects that it's transmission is a part of the collision it should wait for **a particular time slot** and retransmit the frame. But this method has **probability of collision is 1 again**. So the back-off algorithm says to wait a factor of a random in between

$[0, 2^{\text{number of collisions}} - 1]$ * the time slot time.

Lets have an example suppose S1 has collision count 1, and S2 has collision count 2, so S1 can send after a timeslot of [0,1] and S2 can do so after [0,1,2,3] but out of which 2 cases can cause collision when both chooses 0 or 1 which results in collision probability

= 2 / (2 * 4)

= 1 / 4 --which is a success compared to 1 in earlier case.

But inherently due to the algorithm, a node with less collision count captures the channel for longer time, which results in **starvation among other nodes.**

```
2020/12/15 19:54:20 Frame Received: Node 0  | Data:  14245
2020/12/15 19:54:21 Collision
2020/12/15 19:54:22 Frame Received: Node 0  | Data:  17149
2020/12/15 19:54:23 Frame Received: Node 0  | Data:  5773
2020/12/15 19:54:23 Collision
2020/12/15 19:54:25 Frame Received: Node 0  | Data:  23025
2020/12/15 19:54:25 Frame Received: Node 1  | Data:  30909
2020/12/15 19:54:26 Frame Received: Node 1  | Data:  8681
2020/12/15 19:54:27 Frame Received: Node 1  | Data:  3659
2020/12/15 19:54:27 Exiting sender 1
2020/12/15 19:54:28 Frame Received: Node 0  | Data:  8981
2020/12/15 19:54:29 Frame Received: Node 0  | Data:  7336
2020/12/15 19:54:29 Frame Received: Node 0  | Data:  31432
2020/12/15 19:54:30 Frame Received: Node 0  | Data:  771
2020/12/15 19:54:31 Frame Received: Node 0  | Data:  5482
2020/12/15 19:54:32 Frame Received: Node 0  | Data:  3743
2020/12/15 19:54:33 Frame Received: Node 0  | Data:  234
2020/12/15 19:54:33 Exiting sender 0
```

COMMENTS:

It was indeed a nice assignment to practically implement different CSMA techniques. The results are really good which gives the vibes of the real world scenario. Though it's not a much used technique nowadays, understanding the different persistent algorithms and also code them was really a lot of fun.

Thank you.