# Synthetic Random Packet Traffic Generator and Simulations of 802.11 DCF MAC and RTS/CTS

Shane Bennett, *FSU* Grady Denton, *FSU*

## 1. Packet Traffic Generator

We created a python "generator" script to generate simulated random packet traffic. This generator program accepts 6 parameters and 1 optional flag.

```
num_node: Number of Nodes – default of 2

avg_pkt_size: Packet Size – default of 100

offered_load: Offered Load – a number in the
range [0.01, 10.0] – default of 1

num_pkts_per_node: Number of Packets Each
Node Will Send – default of 1

seed: Seed Value for Random Number Generator
– defaults to timestamp

output: Output Filename – defaults "traffic"

-e: exponential distribution, default is
uniform distribution
```

The main function of our generator script is a nested for-loop. For each node in the system, num_pkts_per_node packet entries are generated. These packet entries will consist of five parts: a packet ID, a source node ID, a destination node ID, a packet size value, and the time the packet is sent.

The unique packet ID is calculated using the following formula:

```
packet id = node_id + packet# * num_nodes
```

For example, in the case of a 10 node system, the first 3 packets that node_id=7 sends will be packet_id=7, 17,27.

The source node ID is simply the ID of the node that is sending the packet. If the system contains 10 nodes this value will be in the integer range [0, 9].

The destination node ID is selected randomly from the total node IDs. In a 10 node system this value will be in the integer range [0,9]. If the destination node ID generate happens to be the source node ID, we simply choose another random value.

The avg_pkt_size user input is used to generate packets of varying size. The packet sizes generated follow a uniform distribution by default. So, if the user uses the default avg_pkt_size of 100, the packets sizes will be selected as a random integer in the range [1, 2*100], or, more generally, [1, 2*avg_pkt_size]. Adding the -e flag will cause the generator.py program to generate packet sizes according to an exponential curve.

The time value is calculated using the following process. We first use the packet size, number of nodes, and offered load parameters to calculate an inter-packet time gap for each node. This equation is used:

```
Gap = ((pkt_size * num_node / offered_load) -
pkt_size)
```

Each node will send their first packet at a random time in the range of [0, 2*gap]. Next, they will wait the packet transmission time (which is calculated by avg_pkt_size / dataRate) before selecting a new random time in the range of [0, 2*gap] to send their next packet. We declared a function which selected when the next packet should send based on when the last packet was sent:

```
Def next_pkt_time(last_pkt_time)
   random_gap = randint(0, 2*gap)
   Return last_pkt_time
   + transmission_time
   + random_gap
```

As each packet is generated it is stored in a vector in the format: packet_id, source_node_id, destination_node_id, pkt_size, time. When all num_node*num_pkts_per_node packets have been generated, we sort this vector by the time entry. Once sorted, the total number of packets (num_node*num_pkts_per_node) is printed to the first line of the designated output file. The sorted vector is then printed to the output file.

## 2. 802.11 DCF MAC

Our implementation of DCF MAC contains two parts, a file parsing section and a larger main loop. The file parsing section simply reads in a traffic file line by line. Each of these lines represents one packet, and each packet is pushed unto a queue based on the src_id indicated in the packet. For example, if there are 10 nodes then 10 queues will be created. Packets from

src_node=1 will be pushed into the queue associated with node 1.

Each node is associated with a node state structure. The node state structure contains a node state value which is a integer equal to 0, 1, 2, 3, 4, or 5 if the node is currently waiting for a packet from the application, waiting for DIFS, waiting for backoff slots, waiting for transmission time, waiting for ACK, or waiting for the medium to become free to continue with DIFS, respectively.

```
Node State:
   0 -- waiting for packet from application
   1 -- waiting for DIFS
   2 -- waiting for slots
   3 -- waiting for packet transmission
   4 -- waiting for packet ACK
   5 -- interrupted, waiting for free medium
```

The node state structure additionally contains a timeUntilNextEvent value that indicates how many microseconds must pass before the next event will happen on that node. The meaning of this value depends on the node state value. For example, if node_state==**4** and timeUntilNextEvent==100, for node 2, then in 100 microseconds node 2 will have completed waiting for an ACK. However if node_state==**3** and timeUntilNextEvent==100, for node 2, then in 100 microseconds node 2 will have fully transmitted a packet. The meanings of timeUntilNextEvent based on node_state value are as follows:

```
node_state: timeUntilNextEvent meaning
         0: time until a packet arrives
         1: time until DIFS is completed
         2: time until slots reach 0
         3: time until transmission completion
         4: time until ACK fully received
```

Just prior to the main loop, we initialize the timeUntilNextEvent for each node to be the time that the first packet is received from an application on that node. Once this value is initialized for each node we may enter the Main Loop.

The Main Loop of our implementation has four parts each cycle: first, the next node that will trigger an event is determined by comparing the timeUntilNextEvent of each node and picking the node with the lowest value; second, the node that has been selected must have completed a node_state, so appropriate print statements are output; third, we update the timeUntilNextEvent of every node other than the node selected to reflect the appropriate passage of time; last, we update the node_state and timeUntilNextEvent values of the node that was selected.

A more in depth explanation of the four parts is as follows:

Part 1: if the medium is not currently busy then a node of any node_state value may be selected if it has the lowest timeUntilNextEvent. If the medium is currently busy then we do not allow nodes that are waiting for DIFS to begin counting down, we also do not allow nodes that are waiting for a nonbusy medium to reach their events. So, the lowest timeUntilNext is selected from the nodes with node_state==0, 3, or 4 is selected, if the medium is busy.

Part 2: if the selected node's node_state==0 then it must have received a packet from the application. We print nothing in this situation.

If the selected node's node_state==1 then it must have completed waiting for DIFS, we print that that node has finished waiting for DIFS..

If the selected node's node_state==2 then it must have completed waiting for its chosen slottime, we print that that node has finished waiting and is ready to send a packet.

If the selected node's node_state==3 then it must have completed transmitting its packet, if a collision has been detected at this time, then we print that the node has recognized the collision. If there was not a collision, then we print nothing.

If the selected node's node_state==4 then it must have fully received an ACK, we print that the node has sent X number of bits, where X is the size of the packet that was sent.

Part 3: we update the node state struct of all nodes other than the node that was selected, based on whether the medium is busy or not.

If the medium is not busy, we update the timeUntilNextEvent value of each node other than the selected node. We do the update by subtracting the selected node's timeUntilNextEvent value from all other nodes' timeUntilNextEvent values. For example, if two nodes were waiting for DIFS and one had timeUntilNextEvent==10, and the second had timeUntilNextEvent==20, then the first node would be selected to have completed waiting for DIFS first and the second nodes value will be updated as timeUntilNextEvent==20-10=10.

In the special case that the selected node's node_state==2, then the selected node must have finished counting down its slots and will take control of the medium. This will interrupt any nodes waiting for DIFS and any node still waiting for slots. Interrupted

nodes will be set to node_state==5 and have their timeUntilNextEvent value set to 0.

If the medium is busy in part three of the Main Loop, then only those nodes in node_state==0 or node_state==3 should have their timeUntilNextEvent updated. This is because packets will continue to arrive from applications even while the medium is busy, and because nodes with node_state==3 must be the other nodes that have collided with the current node and are using the medium at the same time.

Part 4: we must update the state of the selected node.

```
If node_state==0:
  node_state=1
  timeUntilNext=DIFStime
If node_state==1:
  node_state=2
  timeUntilNext=slotTime
If node_state==2:
  node_state=3
  timeUntilNext=transmission_time
If node_state==3:
  node_state=4
  timeUntilNext=SIFStime + ACKtime
If node_state==4:
  node_state=0
  timeUntilNext=timeNextPacketWillArrive
```

The above pseudocode is the skeleton of the last part of the Main Loop code, additional logic must be included to detect and handle collisions.

## 3. 802.11 RTS/CTS MAC

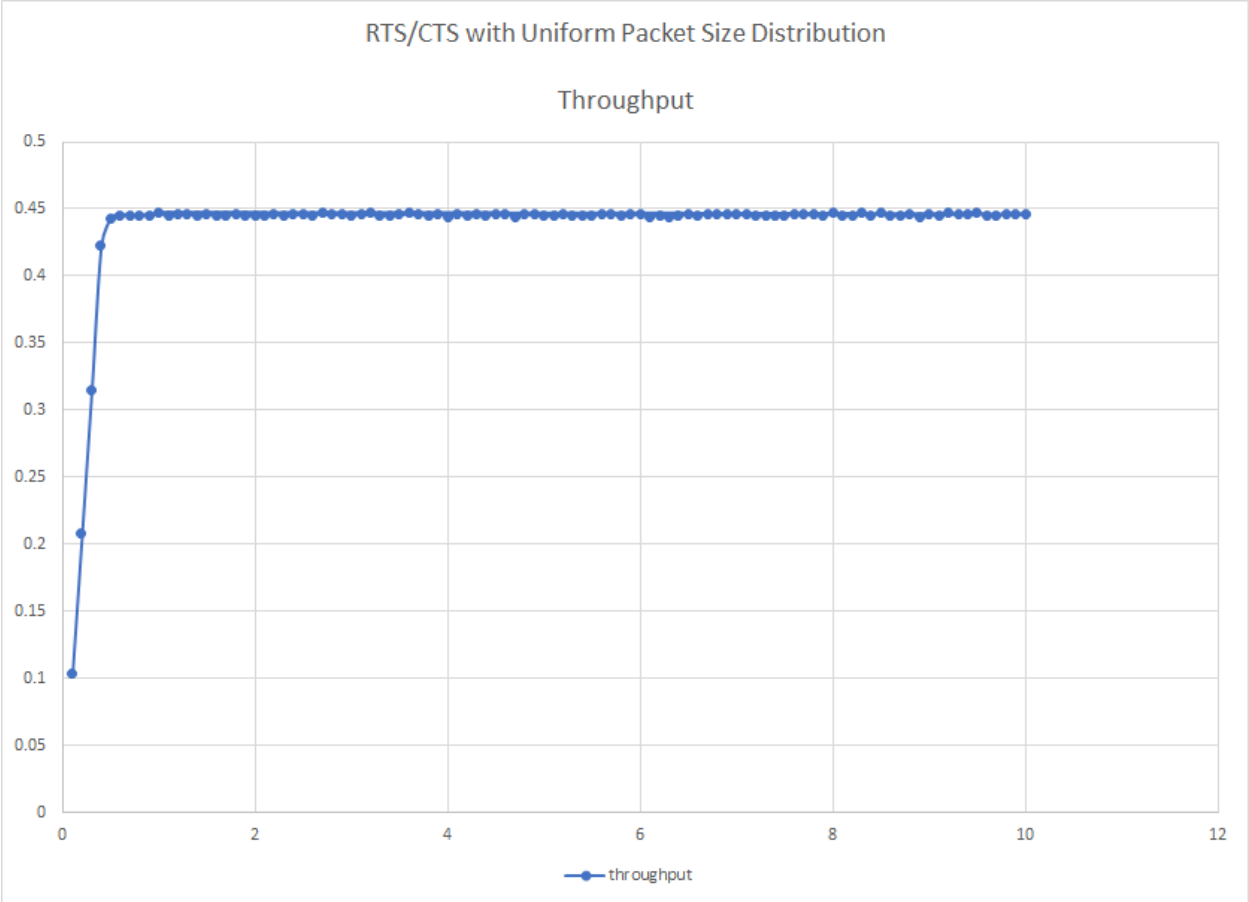
The above skeleton pseudocode is slightly altered to handle RTS/CTS MAC protocol:

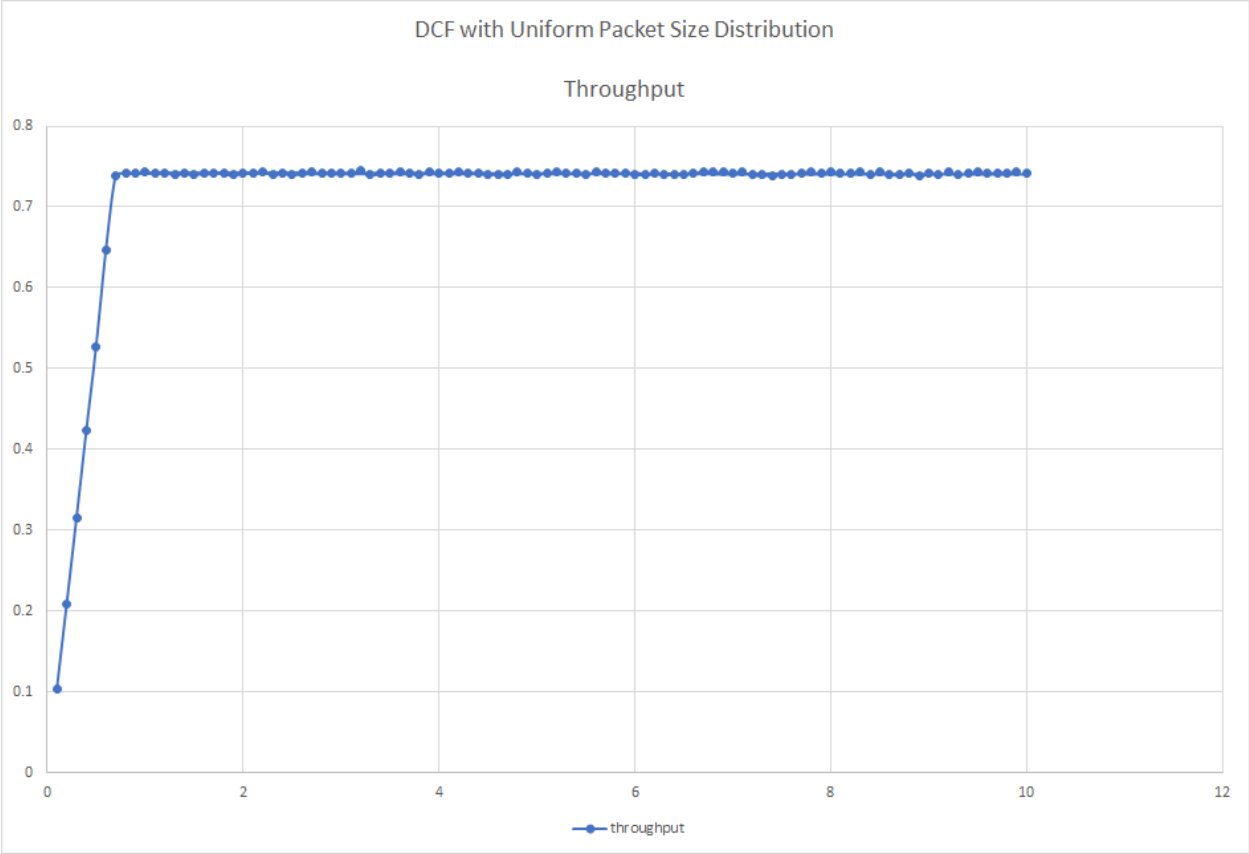
```
If node_state==2:
  node_state=3
  timeUntilNext= RTStime + SIFStime
                 + CTStime
If node_state==3:
  node_state=4
  timeUntilNext=SIFStime
                + transmission_time
                + SIFStime + ACKtime
```

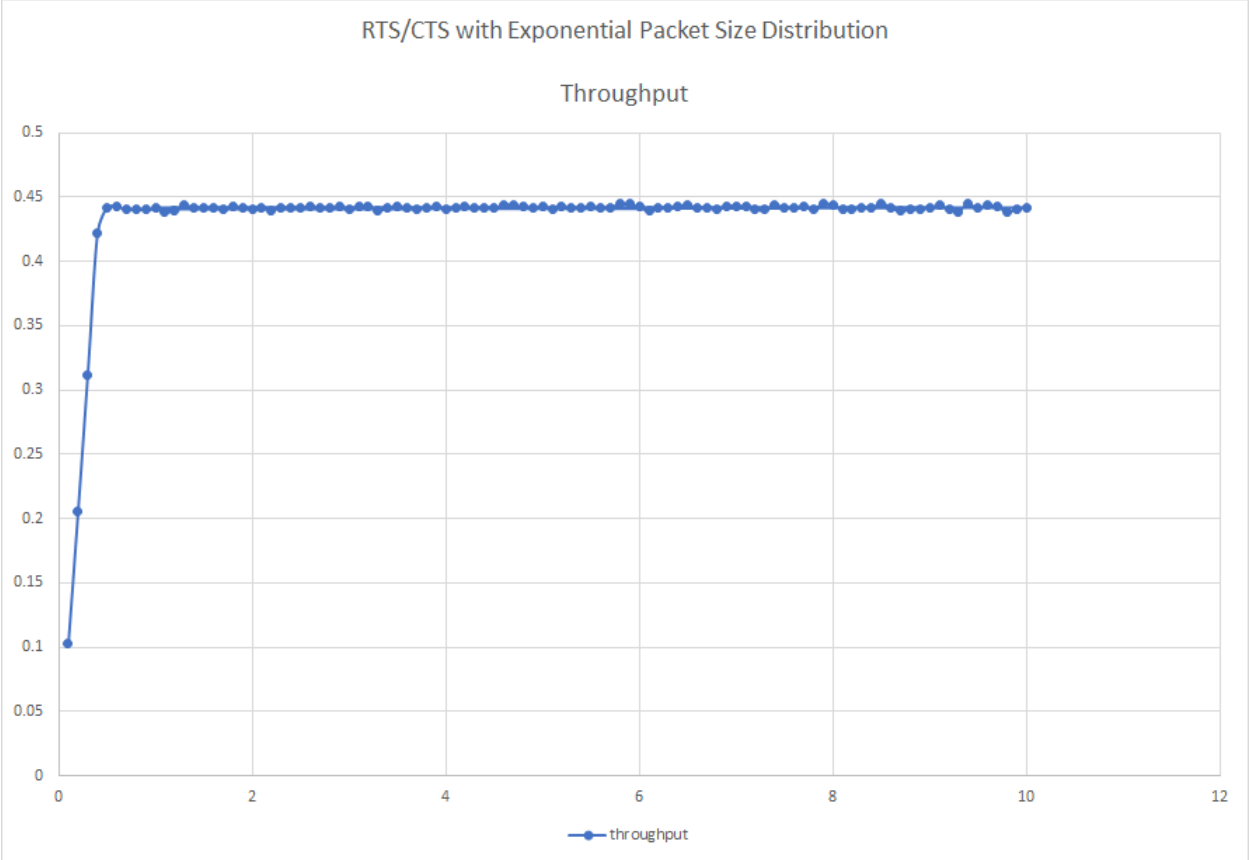
## 4. Throughput Analysis

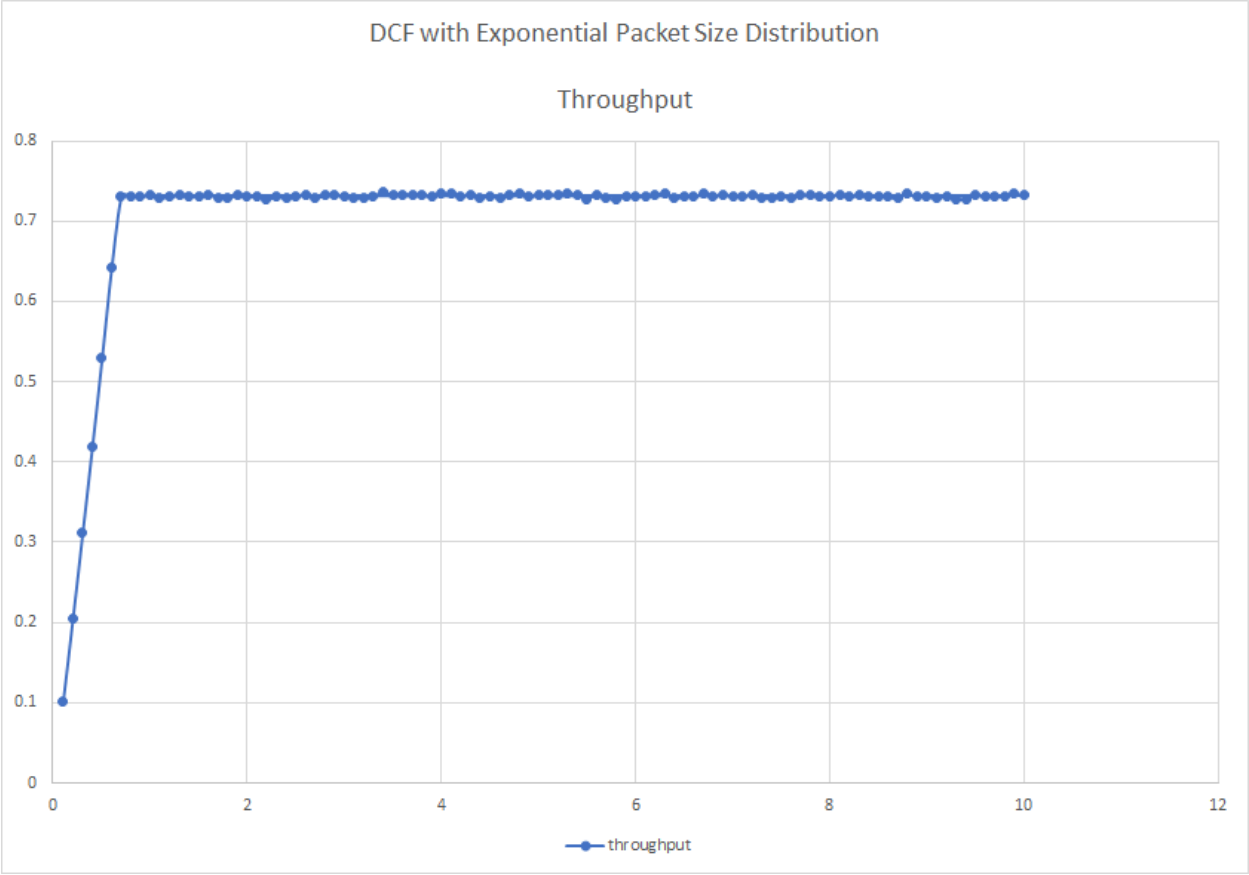
To analyze our implementations of DCF and RTS/CTS we first created many traffic files. We created traffic files in which each node sent 10,000 packets and there are 10 total nodes. We generated traffic files for offered_load values of 0.1, 0.2, 0.3, …, 9.8, 9.9, 10.0. We generated these traffic files once using uniformly distributed pkt_sizes and once using exponentially distributed pkt_sizes. Next, we tested DCF on the two sets of traffic files. Finally, we tested RTS/CTS on the two sets of traffic files.

Our analysis of our implementations of DCF and of RTS/CTS produced the following graphs. As can be seen, DCF reaches a much higher throughput than RTS/CTS when tested with both uniformly and exponentially distributed packet sizes.


RTS/CTS with Uniform Packet Size Distribution
Throughput
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
2
4
6
8
10
12
throughput


DCF with Uniform Packet Size Distribution
Throughput
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8
10
12
throughput


RTS/CTS with Exponential Packet Size Distribution
Throughput
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
2
4
6
8
10
12
throughput


DCF with Exponential Packet Size Distribution
Throughput
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8
10
12
throughput