Synthetic Random Packet Traffic Generator and Simulations of ALOHA, Slotted ALOHA and CSMA 1-persistent protocols

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.

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

pkt_size: Packet Size – default of 1

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"
```

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 packet size value is assumed to be the same for every node, and is provided by the input argument pkt_size.

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 equivalent to pkt_size at 1Gbps) 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. ALOHA

We created a python "Aloha" program to simulate the Aloha MAC protocol. For analysis purposes the program keeps a counter of every packet that is successfully transmitted. To determine whether a packet transmits without collision, we declare a queue which acts as the physical medium. If this queue contains at least one packet then the medium is said to be busy. Packets are read sequentially from the traffic file produced by the "generator" program.

When a packet is read from the traffic file, its time is compared with all packets in the queue and any queued

packets that should be finished sending are popped from the queue. For any finished packets we print an appropriate "failed" or "successfully transmitted" message to the output file in the format:

```
current_time  packet_id  source_id
destination_id  pkt_size
time_pkt_started_sending  "finish sending:
failed"/"finish sending: successfully
transmitted"
```

Next, if the medium is still busy (if there are still packets in the queue), then the current packet is flagged as a collision prior to being pushed to the queue. If there is a single packet in the queue prior to pushing the new collided packet, that first packet must also be flagged as a collision. As we push the new packet to the queue we print an appropriate "packet sending" or "packet sending: collision" message to the output file in the format:

```
current_time  packet_id  source_id
destination_id  pkt_size
time_pkt_started_sending  "start
sending"/"start sending: collision"
```

Once the end of the traffic file is reached, we output "finish sending" messages for any remaining packets in the queue.

3. Slotted ALOHA

We slightly modified the python "Aloha" program described previously to simulate the slotted aloha MAC protocol.

When a packet is read from the traffic file that packet's time entry is overwritten with the time value of the next time slot. The following equation is used to round up to the next time slot:

```
Slotted_time = time + (transmission_time -
(time % transmission_time)
```

This is the only change made to the "Aloha" program.

4. CSMA

To simulate CSMA we added an additional queue for packets that have detected a busy medium and are waiting for it to become idle. As with the aloha and slotted aloha implementations, we process each packet from the traffic input file sequentially.

When the a new current packet is read from the file, we first perform a check to see if the packets in the sending queue should have finished sending before the current packet will attempt to send. If they should have finished sending, they are flushed from the sending queue and "finished sending" statements are printed to the output file for each flushed packet.

Next, we check to see if the sending queue is empty and if there are packets in the waiting queue. If both of these checks return true, then we move all waiting queue packets over to the sending queue and print "start sending" messages for them. If more than one packet has been moved from the the waiting queue to the sending queue, then we flag all as having collided with each other.

```
if not sending_queue and waiting_queue:
  sending_queue = deepcopy(waiting_queue)
  waiting_queue.clear()
  if len(sending_queue) > 1:
    flag all as collided
    print sending messages for all
  elif len(packet_queue) == 1:
    print sending message
```

Now we check to see if the new sending queue contains packets that should have finished sending before the current packet will attempt to send. If it does, then we flush the sending queue and print appropriate "finished sending" statements for each packet that was flushed.

Finally, we decide where to push the current packet. If the sending queue contains any packets, then we push the current packet to the waiting queue and set its time-to-send value equal to the time the packets in the sending queue will complete their transmission. Else, if the sending queue is empty, we simply push the current packet into the sending queue as is:

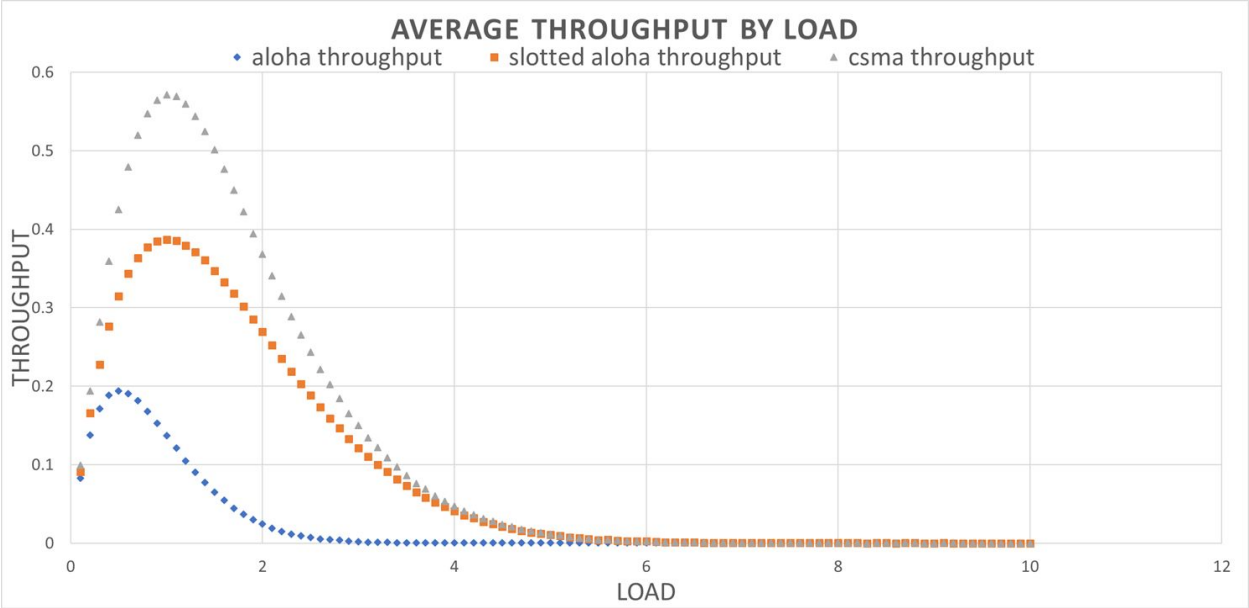
```
if sending_queue:
   current_pkt.time = sending_queue[0].time +
   transmission_time
   waiting_queue.append(current_pkt)
else:
   sending_queue.append(current_pkt)
```

When the end of the traffic file has been reached, we

flush both the sending and waiting queues and print appropriate messages for all flushed packets.

5. Throughput Analysis

Our analysis of our implementations of ALOHA, Slotted ALOHA and CSMA 1-persistent closely match those reported in figure 4-4 of ``Computer Networks,'' by Andrew S. Tanenbaum. Similar to the throughput analysis reported in the text, our implementations of ALOHA, Slotted ALOHA and CSMA 1-persistent reach a peak throughput of about 20%, 40%, and 60%, respectively. Below is a comparison between the throughput graph generated by our implementation compared to the graph from figure 4-4.



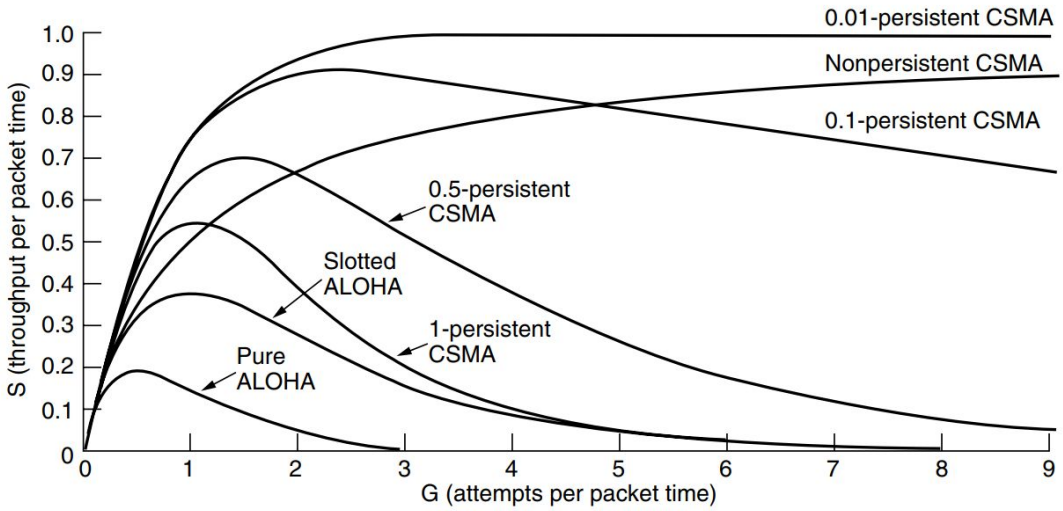


Figure 4-4. Comparison of the channel utilization versus load for various random access protocols.