ECE361 Lab 4
Medium Access Control (MAC)

Due Mar. 24, 2020 @ 11:59 PM

1 Overview

One of the fundamental design challenges in networking is the issue of sharing access to a single communication medium. This challenge is apparent in wireless networks, but also exists in wired networks, as you will see in the second half of this lab. Medium Access Control (MAC) protocols enable multiple stations to share access to a single medium; and the choice of protocol (and its various parameters) directly affects the efficiency of the channel utilization.

In this lab you will simulate two alternative Medium Access Control (MAC) protocols and explore their properties. In the first part, you will be given a working simulator for the ALOHA MAC protocol ("Pure ALOHA"), which you must modify to implement Slotted ALOHA. The goal is to simply see the positive effect on channel utilization when stations are synchronized by a central clock, versus a completely unsynchronized and distributed solution. In the second part of the lab, you will implement the core logic of a Carrier Sense Multiple Access with Collision Detection (CSMA/CD) simulator. The goal in this part is to explore and observe how various network characteristics can influence the ability to accurately detect collisions.

2 Lab Initialization

This lab includes starter code that uses a custom-built Python 3 library called `ece361`. A Python virtual environment will be created, that will include the library, where you will do all the work in this lab. We have provided an initialization script for you that will setup this virtual environment, along with the starter code, and the custom library.

It's highly recommended to update your VM before starting each lab. Open a terminal (in the VM), and type `ece361-update`

To run the initialization script for lab 4, open a terminal and type `ece361-lab-init 4`. You should see something similar to Listing 1.

```
ubuntu@ece361:~$ ece361-lab-init 4
Finding available UG EECG host...
Warning: Permanently added 'ug251.eecg.utoronto.ca,128.100.13.251' (ECDSA) to the list of known
    hosts.
Creating working directory for lab 4 at /home/ubuntu/lab4
Creating Python3 virtual environment...
Installing libraries...

...

Done. Now run source /home/ubuntu/lab4/sourceMe to activate the virtual environment.
```

Listing 1: Initializing lab 4.

The initialization script creates a working directory for this lab, located at `~/lab4`. To activate the virtual environment, run the source command as shown in Listing 1. Once the virtual environment is activated, you should see a `(.venv)` appear at the beginning of your prompt.

You will need to activate the virtual environment **in each terminal** that runs code belonging to this lab.

The `ece361` library used in this lab is different than the ones from the previous three labs. Your code from the previous labs will not work in this new virtual environment.

In your lab 4 working directory, you should have the following files:

- **`slotted_aloha.py`**
  - Simulator for Slotted ALOHA. By default, it implements ALOHA (i.e. non-slotted). You will have to modify it to implement Slotted ALOHA.
- **`csma_cd.py`**
  - Simulator for CSMA/CD, separated from the core simulation logic.
- **`simulation_logic.py`**
  - Starter code for the CSMA/CD simulator's logic. You will have to complete this file.

If you discover a bug in any of the files provided, please report it on Piazza.

3 Part 1: Slotted ALOHA

In this part, you will take a fully working ALOHA simulator (given to you) and modify it to become Slotted ALOHA. In Slotted ALOHA, stations are synchronized to transmit frames only at the start of a slot interval, and thus, it is simply a discretized version of ALOHA. The provided simulator takes a single command-line parameter, `G`, and runs the simulation based on that parameter. Run the simulator by simply calling it with a parameter for `G`, as shown in Listing 2. Each run of the simulator should last roughly a minute.

```
(.venv) ubuntu@361:~/lab4$ python3 slotted_aloha.py
USAGE: python3 slotted_aloha.py <total arrival rate G>
(.venv) ubuntu@361:~/lab4$
(.venv) ubuntu@361:~/lab4$ python3 slotted_aloha.py 0.5
2020-03-08 22:28:41,829 slotted_aloha INFO: Frame size = 704 bits
2020-03-08 22:28:41,829 slotted_aloha INFO: Tx Rate = 96000 bps
2020-03-08 22:28:41,830 slotted_aloha INFO: Serializaiton delay = 0.007333333333333333 seconds
2020-03-08 22:28:41,830 slotted_aloha INFO: Propagation delay = 0 seconds
2020-03-08 22:28:41,830 slotted_aloha INFO: Global arrival rate = 68.18181818181819 frames / second
2020-03-08 22:28:41,830 slotted_aloha INFO: Number of stations = 3
```

```
2020-03-08 22:28:41,830 slotted_aloha INFO: Per-station arrival rate = 22.72727272727273 frames /
    second
2020-03-08 22:28:41,831 slotted_aloha INFO: ====================

...
```

Listing 2: Running the Slotted ALOHA simulator (default behaviour is ALOHA)

The result of the simulation will be appended to a file called `out.txt` (one will be created if it doesn't already exist). It will be a simple comma-delimited entry showing `G` and `S`, respectively. If you run the simulation multiple times with varying values of `G`, you can populate the output file with a series of data points, as seen in Listing 3.

```
0.03125,0.029624704097476643
0.0625,0.057893199350203525
0.125,0.10240504689659247
0.25,0.15737031469013515
0.5,0.19284254160579692
1.0,0.1651676868095355
2.0,0.04724612507409784
```

Listing 3: Sample data from `out.txt` for ALOHA

Try the unmodified simulator now, by running it with the `G` values shown in Listing 3 (i.e. the value before the comma). See if you can produce a similar set of numbers with a similar pattern.

You can use this data to then graph the `G` vs `S` curves you have seen from lecture (e.g. see Fig. 1). There are other parameters within the simulation code as well, but for the purposes of the lab, we have hard-coded them for you.

Do NOT modify the hard-coded parameters, it may affect the simulator's ability to produce the correct results. If you wish, you may alter the `RUN_TIME` parameter. A longer running time would produce more stable results.

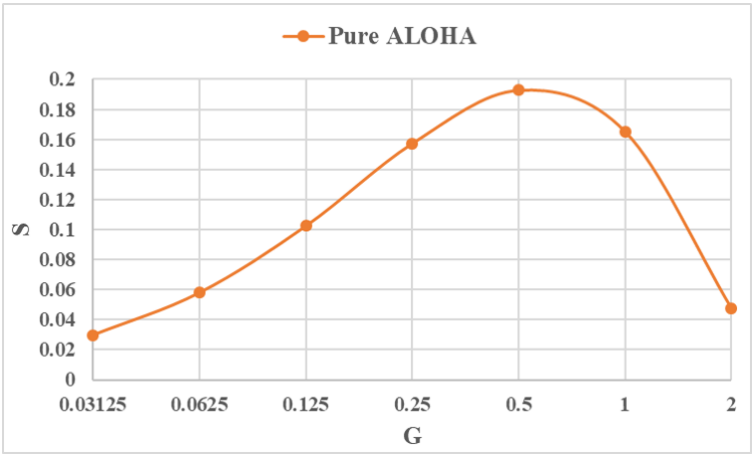

Figure 1: Sample graph of ALOHA based on simulator's output.

Your task is to discretize the simulator by syncing the frame transmissions to the start of each interval. The provided code has already calculated the slot interval, stored in a variable called `SLOT_INTERVAL`.

Focus on the `Station` function, which implements the logic of the stations. Looking at the code, see if you can figure out how the stations decide when to transmit their next frame. That is all you need to modify.

After you have made your modifications, generate a new `G` vs `S` curve and see if it matches the theoretical improvement of Slotted ALOHA over ALOHA (see Fig. 2).

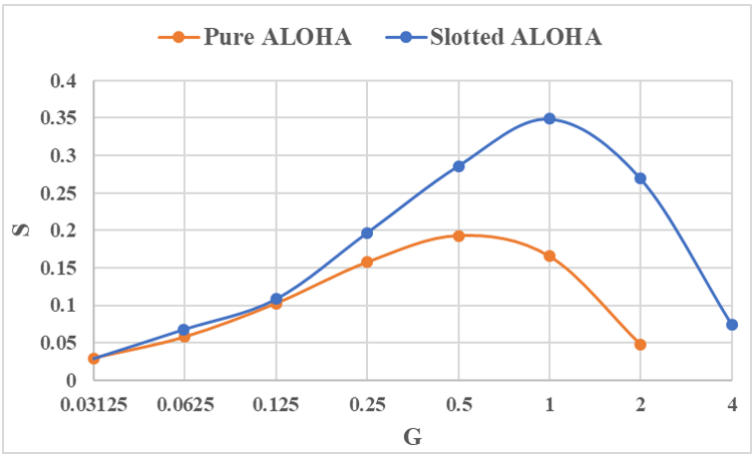

Figure 2: Sample graph of Pure ALOHA vs Slotted ALOHA `G` vs `S` curves

4 Part 2: Carrier Sense Multiple Access with Collision Detection (CSMA/CD)

Carrier Sense Multiple Access with Collision Detection (CSMA/CD) is a protocol for medium access in which stations "sense" the medium and wait until it is quiet before transmitting a frame (the "CS" part). While a station is transmitting, it continues sensing the medium to detect for transmissions from other stations, which would indicate a collision (the "CD" part). When a collision is detected, the station immediately stops, and reschedules the frame after some time.

Hence, a station would only be aware of a collision if it receives a transmission from another station while it is sending. If a station finishes a transmission before the first bit of another station's frame arrives, then it will be unaware of the collision and falsely believe it successfully transmitted its prior frame. For CSMA/CD to properly function, stations must be accurately aware of all collisions so that they can reschedule collided frames.

In this part of the lab, you will complete a CSMA/CD simulator that simply counts collisions: actual collisions versus station-aware collisions. If the number of actual collisions does not match the number of station-aware collisions, then the stations will inadvertently lose data. As you'll see, the ability of a station to detect a collision varies depending on different network characteristics and parameters.

4.1 Simulated Station Setup

This simulation models the setup of the stations in the following ways:

- Assume the stations form a star topology, so all stations are equidistant. Thus, propagation delay is constant.
- Assume the stations are inter-connected by a hub (not a switch), and hence forms a single broadcast medium.
- The stations will pre-generate a sequence of arrival times for N number of frames (N is a configurable parameter). If a frame cannot be sent at a given time (i.e. due to the medium being busy), that frame is rescheduled to the end of the arrival sequence.
- All frames during the simulation are constant length (a configurable parameter).

4.2 Simplified CSMA/CD Protocol

The actual protocol you will complete is a simplified version of CSMA/CD. The key modifications, which will make the simulation easier, are as follows:

- When two frames collide, the frame that began transmission earlier in time "loses" (i.e. considered as 1 collision), while the frame that began transmission later in time "wins" (i.e. it can potentially be counted as a success, if and only if no subsequent frame collides with it).
- Upon collision detection, transmission does not stop (i.e. the stations involved will finish transmitting their frames). The source station should not attempt to re-transmit this frame.
- Upon collision, no jamming or collision enforcement is done by the stations.
- No inter-frame spacing (i.e. once the medium is silent, stations with a frame can send).
- No backoff mechanism. If a station has a frame to transmit and the medium is busy, it reschedules the frame to the end of its arrival sequence.

4.3 Logic of Simulator

For each of the stations, the simulator will pre-generate the arrival times of frames to the station. Another way to think about these arrival times is that they are the frame's transmission start times (assuming a completely free medium). Using this information, the simulator can find the earliest frame arrival time, and compare it with the second-earliest arrival time, to see if a collision will occur. It then "shifts forward", comparing the second-earliest with the third-earliest. Then the third-earliest is compared against the fourth-earliest. The simulator will repeat this pattern for each pair. You can visualize this as a set of N frame arrivals per station, like that shown in Fig. 3.

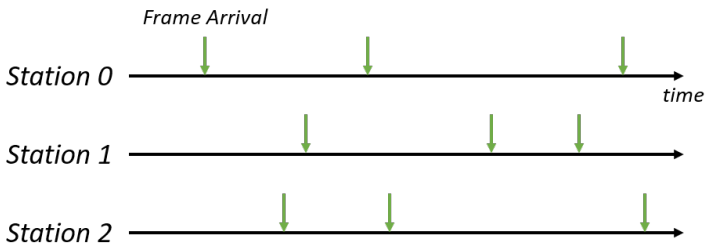

Figure 3: Visualization of each station's frame arrival times.

When analyzing the arrival times of two subsequent frames (e.g. the left-most two arrows in Fig. 3, from station 0 and station 2), one can determine if a collision will occur based on: 1) their arrival times; 2) the inter-station propagation delay (which is constant in this lab due to the assumed topology); and 3) the frame's serialization delay (which is constant for a given simulation run). To determine if two subsequent frames will collide, there are three general cases to consider.

Case 1: Given two frame arrival times from two different stations, the latter frame arrives to find the medium free, and starts transmitting before the first bit of the former frame is able to reach the second station. This can be visualized as seen in Fig. 4.

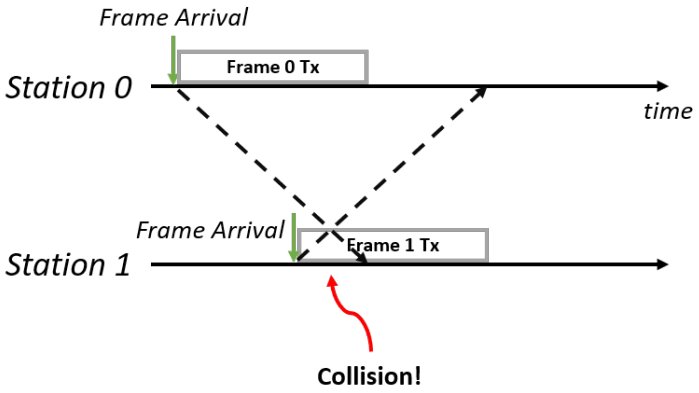

Figure 4: Example of when a frame collides, and the source station is unaware of the collision.

In this case, a collision has clearly occurred. However, whether or not the source station for the first frame becomes aware of the collision depends on whether the first bit of the second frame is able to propagate back to the first station before the first station finishes sending its frame. In the case of Fig. 4, the answer is no. Thus, the first station will falsely believe it successfully transmitted and delivered the frame.

On the other hand, if the first bit of the second frame propagates back to the first station while it is still transmitting, then the first station becomes aware of the collision and properly detects it. This scenario is illustrated in Fig. 5.

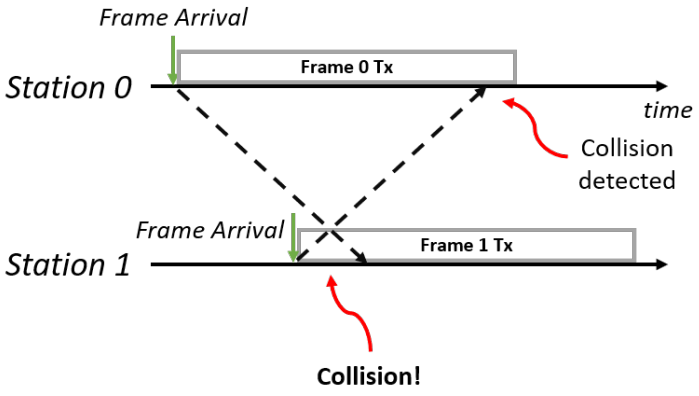

Figure 5: Example of when a frame collides, and the source station is aware of the collision.

Case 2: Given two frame arrival times from two different stations, the latter frame arrives to find the medium busy, and reschedules the frame (i.e. puts it to the end of its arrival sequence). This is illustrated in Fig. 6, where the frame arriving at Station 1 will have to be rescheduled.

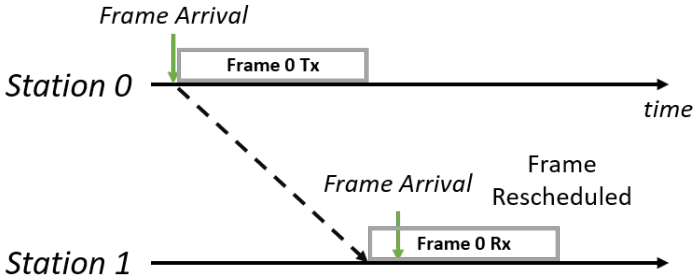

Figure 6: Example of when a frame arrives while the medium is busy.

Case 3: Given two frame arrival times from two different stations, the latter frame arrives to find the medium free, and starts transmitting after the final bit of the former frame reaches the second station. This is the best case scenario and means there is no collision or rescheduling. This case is illustrated in Fig. 7.

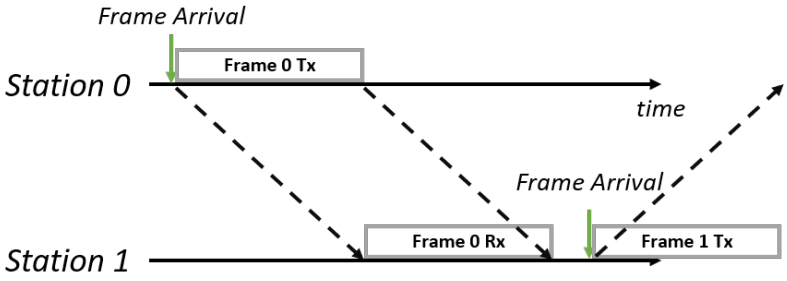

Figure 7: Example of when a frame arrives with no conflict with the past frame.

Given the three scenarios illustrated, consider how the network diameter (i.e. the station-to-station distance), the frame size, and channel bandwidth can all affect the stations' ability to detect collisions.

4.4 Completing the Simulator

We have provided you with `csma_cd.py` and `simulation_logic.py`.

In the `simulation_logic.py` file, complete the `RunSimulation()` function to implement the simulation logic as described in section 4.3.

Read the comments in the file carefully. In particular, the differences between the `numActualCollisions`, `numStnObsvCollisions`, and `numStnObsvSuccess` variables, which the `RunSimulation()` function must return.

4.4.1 The `FrameMetadata` Class

To help you with your task, a class called `FrameMetadata` has been provided for you to work with. It is a simple object to hold metadata about a single frame, with two member variables:

- `FrameMetadata.stnID`: Denotes the source station this frame is from; and
- `FrameMetadata.arrTime`: Denotes the arrival time of the frame to the station (i.e. its transmission start time, if the medium is free).

4.4.2 The `StationArrivals` Class

Another class you will use is the `StationArrivals` class. The first parameter passed into the `RunSimulation()` function is an instance of this class. In particular, there are two class methods you need to be aware of:

- `StationArrivals.getNextArrival()`: Finds the next earliest frame arrival among all stations, and returns an instance of `FrameMetadata`. If every station has transmitted all their frames, then it returns `None`.
- `StationArrivals.rescheduleFrame(frameMeta)`: Given an instance of `FrameMetadata`, reschedules the frame to the end of the station's arrival sequence. Does not return anything.

4.5 Running the Simulator

To run the simulator, simply execute `csma_cd.py`, as shown in Listing 4.

```
1 (.venv) ubuntu@361:~/lab4$ python3 csma_cd.py
2
3 ====================
4 SIMULATION PARAMETERS:
```

```
  - NUM_STATIONS = 3
  - NUM_FRAMES = 1000000 (per station)
  - ARR_RATE = 10000.0 (frames / sec)
  - FRAME_SIZE = 512 (bits)
  - MEDIUM_BW = 10000000 (bits / sec)
  - MEDIUM_DIAMETER = 2500 (metres)
  - MEDIUM_VELOCITY = 97656250 (metres / sec)
  - SERIALIZATION_DELAY = 5.12e-05 (sec)
  - PROP_DELAY = 2.56e-05 (sec)
  - NORM_DELAY_BW = 0.5

====================
Generating arrival times...
Generated 3000000 arrivals in 0.8805019855499268 seconds

====================
Starting CSMA-CD simulation...

...
```

Listing 4: Running the CSMA/CD simulator

The default configuration simulates a 10 Mbps channel with a network diameter of 2500 metres, and a frame size of 512 bits. If you completed the simulator correctly, the results at the end of the simulation will show that the number of collisions (perceived by the stations) should match the number of actual collisions that occured. Similarly, the number of successful transmissions (perceived by the stations) should match the number of actual successful transmissions. This is seen in the sample output shown in Listing 5.

If you want a deterministic output, you can seed the random generation of frame arrivals by running the simulator with the `--seed` flag. e.g. `python3 csma_cd.py --seed 12345`

```
...

====================
Stations' statistics (stations' point-of-view):
  - Number of successful transmissions: 2072048 (69.068 %)
  - Number of frame collisions: 927952
  - Efficiency of channel utilization: 41.694 %

Actual statistics:
  - Number of successful transmissions: 2072048 (69.068 %)
  - Number of frame collisions: 927952
  - Efficiency of channel utilization: 41.694 %
```

Listing 5: Sample output of CSMA/CD simulator, where the stations accurately counted the number of collisions.

Once you have achieved this with the default parameters, then you can begin altering the default configurations by changing various aspects of the simulation.

If you simply run `python3 csma_cd.py -h`, you can see the list of parameters you can alter by using command-line flags. These include: the number of stations, the number of frames per station, the frame arrival rate per station, the frame size, the channel bandwidth, the network diameter, a seed value for deterministic arrival patterns, and a flag to quiet the output.

As you slowly increase the frame size, while leaving every other parameter as default, you'll see the efficiency of the channel utilization rise. Why?

If you increase the network diameter, while leaving every other parameter as default, the stations start to believe they had more successful frame transmissions than actually occurred. Why? Similarly, try just increasing the bandwidth, or just decreasing the frame size. What key parameter from class did you learn about that relates to all this?

If you look on Wikipedia's page about the Ethernet frame, you may have noticed that it lacks a key header that most other protocols have: length. Its minimum frame size must be exactly 64 Bytes. Can you understand why?

5 Exerciser

You can use the exerciser to help test the correctness of your implementation. The exerciser will run a set of public test cases against your code. However, note that the tests in the exerciser is not complete and it is your responsibility to test your code to make sure it conforms to the requirement of the lab. Ensure you have the following files all within the same directory:

- **slotted_aloha.py**: Your completed Slotted ALOHA simulator.
- **simulation_logic.py**: Your completed simulation logic for the CSMA/CD simulator.

In terminal, browse to the directory containing the files and type `ece361-exercise 4`.

6 Submission

Once you are confident of your implementation, you can run the submission process (which will invoke the exerciser before submitting). Only one person in the group needs to submit.

From the same directory as where you ran the exerciser, type `ece361-submit submit 4`.

You can then verify the submission by typing `ece361-submit list 4`.

At some point after the lab's due date, private test cases will be run against your submission to calculate your final mark.