1 Written questions

1.1 Different router architectures (10 points)

Consider a router with $N$ input ports and $N$ output ports. Answer the following questions.

1. How many enqueues and dequeues does the memory of a shared-memory router need to support on every tick? (2 points)

   N enqueues and N dequeues

2. How many enqueues and dequeues does each memory of an output-queued router need to support on every tick? How many such memories are required? (3 points)

   N enqueues and 1 dequeue. N such memories.

3. How many enqueues and dequeues does each memory of an input-queued router need to support on every tick? How many such memories are required? (3 points)

   1 enqueue and 1 dequeue. N such memories.

4. What is the advantage of a shared-memory router relative to an input/output-queued router? What is the disadvantage? (2 points)

   Advantage : Shared memory router can dynamically allocate memory between different output ports depending on traffic.
   Disadvantage : As the size of N increases it is hard to achieve N enqueues and N dequeues per tick for the shared-memory.

1.2 The ALOHA protocol (10 points)

Assume there are $N$ users who cannot hear each other communicating with a centralized coordinator in the ALOHA protocol.

1. Derive the optimal value of the transmission probability for each user in the ALOHA protocol. (5 points)
1.3 Differences between ALOHA, WiFi, and Ethernet (20 points)

1. Explain the difference in operating conditions between ALOHA and WiFi. (5 points)

ALOHA operated over a large distance where one user can’t detect the transmission of another user. WiFi operated over small distance where one user can detect transmission of other user.

2. How does this manifest itself as differences in the design of the ALOHA and WiFi MAC protocols? (5 points)

ALOHA tries to avoid collisions by random access, whereas WiFi avoid collision by not transmitting when the medium is busy (i.e., carrier sense).

3. Explain the difference between the physical layer of WiFi and Ethernet. (5 points)

WiFi can’t send and receive transmissions simultaneously (due to interference of EM waves), whereas Ethernet can perform simultaneous transmission and reception.

4. How does this manifest itself as differences in the design of the WiFi and Ethernet MAC protocols? (5 points)

As WiFi can’t receive detect a collision while transmitting, it tries to avoid collisions which led to CSMA/CA protocol. Ethernet can detect a collision during transmission which led to CSMA/CD protocol.

In class notes. Equating the derivative of

\[ N.p.(1 - p)^{N-1} \]  

to zero as follows:

\[ N((1 - p)^{N-1} - p(N - 1)(1 - p)^{N-2}) = 0 \]  \hspace{1cm} (2)

\[ (1 - p) - p(N - 1) = 0 \]  \hspace{1cm} (3)

\[ 1 - p - Np + p = 0 \]  \hspace{1cm} (4)

\[ p = \frac{1}{N} \]  \hspace{1cm} (5)

2. What is the overall utilization of the shared medium when each user uses the optimal transmission probability? Show your work. (5 points)

The following equation

\[ (1 - \frac{1}{N})^{N-1} \]  \hspace{1cm} (6)

tends to \( \frac{1}{e} \) (or 37%) as \( N \) tends to infinity.
1.4 Other questions on MAC protocols (30 points)

1. What is carrier sense? (5 points)

It is a process of listening to shared medium before transmission to avoid collisions. This process allows the sender to determine whether someone else is currently transmitting on the medium or not.

2. What is one possible implementation of the carrier sense capability? (5 points)

By checking the signal intensity of EM waves in case of WiFi or checking the voltage in case of Ethernet bus. If the signal intensity or voltage is much higher than a threshold, it is a strong indicator that some other user is currently transmitting on the medium.

3. Give an example of the hidden terminal problem. (5 points)

Two laptops A and B connected to a WiFi access point C. A and B are within the range of C but A and B can’t detect each other.

4. Explain how WiFi solves the hidden terminal problem. (5 points)

It is solved by RTS/CTS signals from hosts. Hosts A and B both send RTS signal to access point C. C will give access to either A or B and broadcasts CTS signal. As the signal is broadcasted, the host which was not given permission to transmit will wait the amount of time set in CTS before performing carrier sense again.

5. Explain the TDMA and FDMA MAC protocols. (5 points)

TDMA: Giving different time slots to hosts so they can transmit in a shared medium.
FDMA: Giving different frequency range slots to hosts so that they can transmit in a shared medium.

6. Where would you use TDMA/FDMA-style MAC protocols? (2 points)

Where the medium is shared between multiple hosts and a centralized coordinator is available to perform the assignment of time slots or frequency ranges to different hosts.

7. How are they different from the MAC protocols used in WiFi, ALOHA, or Ethernet? (3 points)

There is no need to detect collisions in TDMA/FDMA-style MAC protocols because the centralized coordinators arranges for hosts to transmit at different times/frequencies to avoid collisions in the first place.

2 Coding questions (30 points)

Use the starter code (hash_tables.py) to implement the three different schemes for exact matching we discussed in class: standard hashing (5 points), 2-choice hashing (15 points), and 2-left hashing (15 points).
2.1 Code walkthrough

hash_tables.py is the only Python file you’ll need to work with for this assignment. It takes two arguments. The first one is the occupancy of the hash table (i.e., what fraction of the hash table is populated with entries). The second one is the name of the hashing scheme (“2choice”, “2left”, or “standard”). For instance, to run hash_tables.py with the standard hashing scheme with an occupancy of 50%, you would run the following command:

Anirudhs-Air:sims anirudh$ python3 hash_tables.py 0.3 standard

Fraction of trials in which slot size did not exceed capacity 0.671

The output of running this command tells you how often the hash table did not overflow. To understand how this is calculated, let’s walk through hash_tables.py.

The constants at the top of hash_tables.py are:

1. The occupancy of the hash table as a fraction between 0 and 1. This is fed in by the user.
2. The number of slots in the hash table. For all three hashing schemes, we structure the hash table as an array with a certain number of slots, where each slot can hold up to a certain number of elements.
3. The capacity of each slot. This reflects hardware realities of how many elements can be held in each slot. We have set this to 5 because it is in the same ballpark as the number of elements in each slot for real hardware implementations.
4. The number of trials. For each trial, we check if the hash table overflowed: did the hash table run out of space because a particular slot that an element hashed into was already full? Note that we say that the hash table overflowed if even a single slot’s capacity has been exceeded—even though the hash table might still have space in other slots. This is why the fraction of trials in which the hash table did not overflow reaches 0 long before the occupancy of the hash table reaches 1.
5. The number of elements in the hash table. This is computed by multiplying the occupancy fraction with the product of the number of slots and the capacity of each slot (i.e., the maximum occupancy of the entire hash table).

After the constants have been initialized (either to default or user-supplied values), the main for loop in hash_tables.py runs (for seed in range(TOTAL_TRIALS)). This for loop runs a number of independent trials, each initialized with a different random seed to ensure the randomness in each trial is independent of other trials.

In each trial, we run through the total number of elements in the hash table in another for loop (for i in range(NUM_ELEMENTS)). For each element, we compute the slot it hashes into depending on the hashing scheme, which is another user-supplied input. You need to implement three hashing schemes.

1. standard: We compute a single slot number for each new element regardless of the current occupancy of each slot.
2. 2choice: We compute 2 slot numbers using two independent hash functions, and then pick the slot number that has lower occupancy, breaking ties randomly.
3. 2left: We conceptually divide the hash table into two subtables with equal number of slots in each subtable. We compute a slot number in each subtable using two independent hash functions. Of these 2 slots in 2 subtables, we pick the slot that has lower occupancy, but always break ties in favor of one of the two subtables.

To compute a random slot number, you do not need to use an actual hash function for this assignment. Instead, you can use the Python function call (random.randint(U, L)) to generate a random slot number between U and L (both U and L included). You can test the occupancy at a particular slot number using the occupancy array in hash_tables.py. For each algorithm, make sure that the slot number you pick is written into the variable (slot_number) in hash_tables.py because this is the index variable we use to update the occupancy array.
2.2 Testing your code

The easiest way to test your code is to remember that for a given occupancy, the fraction of trials in which the hash table does not overflow is going to be highest for 2left, followed by 2choice, followed by standard. For instance, when I run my solutions for all three algorithms with an occupancy of 0.6, here’s what I see.

```
Anirudhs-Air:sims anirudh$ python3 hash_tables.py 0.6 2left
Fraction of trials in which slot size did not exceed capacity 0.999
Anirudhs-Air:sims anirudh$ python3 hash_tables.py 0.6 2choice
Fraction of trials in which slot size did not exceed capacity 0.994
Anirudhs-Air:sims anirudh$ python3 hash_tables.py 0.6 standard
Fraction of trials in which slot size did not exceed capacity 0.0
```

Also, when the occupancy of the hash table goes up, the performance of all three schemes must degrade: the fraction of trials in which the hash table does not overflow should go down. When I run the three algorithms with an occupancy of 0.7, here’s what I see.

```
Anirudhs-Air:sims anirudh$ python3 hash_tables.py 0.7 2left
Fraction of trials in which slot size did not exceed capacity 0.95
Anirudhs-Air:sims anirudh$ python3 hash_tables.py 0.7 2choice
Fraction of trials in which slot size did not exceed capacity 0.844
Anirudhs-Air:sims anirudh$ python3 hash_tables.py 0.7 standard
Fraction of trials in which slot size did not exceed capacity 0.0
```

3 Submitting the assignment

Go to https://classroom.github.com/a/xsdkXV3N to create a private github repo to submit the assignment.