Patrick Lyons

CSMA – CA Simple Simulator Project

Final Project Analysis

**Files**

* csma.cpp
  + This was my source file for compiling and running my project. All logic and output comes solely from this file
* run.sh
  + A simple bash script created to compile and run the csma.cpp file. It compiles using g++, specifically using the c++ 11 standard library, and also compiling with the -pthread argument. We create the csma executable and run it with this single file. Please note that if the run.sh file does not run, you may need to make it executable using the following command in a unix system:
    - chmod +x run.sh

**System**

The code was developed, compiled, and run using g++ on a 2017 Macbook Pro running MacOS 10.13 High Sierra. The built in terminal was used for compilation.

**Functions Used**

The following functions were used in the csma.cpp file:

* main(int argc, char \*argv[])
  + Main function of the program creates 8 threads to represent the mobile devices. It also seeds the random number generator with the current unix time to allow for actual random numbers. When mobile devices are created, we call sleep\_and\_detect(deviceID, probability), and wait for them all to finish their cycle before joining the threads together and exiting.
* **random\_number\_generator(int n)**
  + This function takes in an integer n, and generates a random number between 1 and n inclusive. Returns this number.
* sleep\_and\_detect(int deviceID, int probabilityToSend)
  + The main function in the program, which cycles through the csma protocol. We give it the ID of the calling thread (deviceID) and the probabilityToSend variable, which simply holds the given probability that a device has data to send.
  + We start by deciding if our devices have data to send by generating a random number between 1 and 100. If the number generated is less than or equal to our probabilityToSend variable, we set the readyToSend variable to True, indicating that we are ready to proceed in the process. If the probability comparison does not indicate that we have data to send, the calling thread should sleep for time td before checking again.
  + When we are ready to send, we use a series of do-while loops, compounded with if statements to determine where in the process we are, based on the chart that we were given below.
  + Being that the logic is in the chart below, I will forgo explaining every loop and check.
  + Essentially, what we need to know is that while devices still have packets to send, we will constantly run through the process of checking the medium, and calling the send\_data() function, all the while incrementing our total amount of time taken up.
* **send\_data(int deviceID, int& totalTime)**
  + Passed in the deviceID of the calling thread, and a reference to the totalTime variable so that we can continue to accurately keep track of it. The first thing a device will do in here is call a mutex lock on the function to keep all other devices from being able to change anything while the current thread is functioning.
  + Once this is done, the device will call the set\_status function and set the medium status to false, indicating that It is busy. Any other devices that call check\_status will see the status and will continue to sleep and check it periodically until It is free.
  + Here, the device will imitate the packet send by sleeping for time tp, and simulate receiving an ACK by sleeping for time tifs
  + Finally, before finishing, set the medium status to true, and unlock the mutex to allow other threads to begin their sending procedure.
* **check\_status()**
  + Check the status of the medium. False means it’s unavailable, True means it’s available.
* **set\_status()**
  + Set the status of the medium using the same rules listed above
* **print\_results(**int deviceID, int totalTime**)**
  + Print out the final results of each device, simply printing the ID of the device, and the total time taken to send a packet