Patrick Lyons

CS 1699 – Wireless Networks

Professor Xerandy

CSMA 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
* Various .txt files
  + Includes 1PacketRun1.txt, 1PacketRun2.txt, 5PacketRun1.txt, and 5PacketRun2.txt. These files simply include the output from 2 separate runs of the csma executable in which the 8 devices send 1 or 5 packets each, respectively. These files were used to plot the data, and contain the data used in my graphs in the Analysis section below.
* CSMA results.xlsx
  + Contains tables of data from the .txt files, and corresponding graphs of the values of total time. The second page of the file shows the graphs, but with the device numbers in the order in which they finish transmitting the data.

**How To Run**

Execute the following commands in a working unix terminal, ensuring that g++ is installed, and can reference the c++ 11 standard library, along with the -pthread argument.

1. chmod +x run.sh
2. ./run.sh
3. [You will be prompted by csma.cpp to enter a number between 1 and 6. Do so and hit enter]
4. [You will be prompted by csma.cpp to enter a number between .3 and .6. Do so and hit enter]
5. [You will be prompted 8 times to enter values between 1 and 100. Do so and hit enter for each]
6. The program will now execute with your given standards

**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. The run.sh file compiled csma.cpp using the following command.

g++ -std=c++11 -pthread csma.cpp -o csma

**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. In main, we prompt the user for the number of packets that each device will send, the multiplier for the packet transmission time (number between .3 and .6 to be multiplied by td), and 8 prompts to get the activity rate of each device (between 1 and 100).
* **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 the current 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 Boolean 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 to the user through the terminal.

**CSMA Cycle**

****

**Analysis and Overview of Data**

The graphs in the excel file, and shown below highlights some crucial, yet expected results of the data. In our first run, the randomly generated probability of having data ready was only 9%. As a result, it took a little longer to run than the other runs did. At first glance, it would appear that there was an error, since all mediums took 825 ms to send their packet, from the time they realized they had data ready to send, until it was sent and acknowledged. The graph is not incorrect. This consistency is as a result of no two devices having data ready to send at any given time… Since it took some time for a device to finally have data ready (a random number generated between 1 and 9 out of 100), a device could complete the transfer process without fighting for access to the medium. As a result, we see here that the minimum time to send 1 packet of data through our network, with a medium that is ‘always available’ and a time ts equal to 5 ms (made small to complete the analysis quicker), is 825ms.

In concurrent runs, we see a general upward trend in data as concurrent devices begin to have data ready to send. There seems to be a nearly constant slope in each graph, with some discrepancy. Looking at the output of the runs (in the respective .txt files) we see that these occur when two devices have multiple packets to send, and fight for control of the medium… one begins to send data, but before it can send all five packets, another device gains control of the medium, and may finish sending data before finally freeing the medium for the first device to finish its send cycle.

The data shown correctly mimics a Carrier Sense Multiple Access (CSMA) network, where each device with data to send will senses the traffic on the medium, waits a given amount of time, check the medium again, and finally begin its transmission phase. Concurrent devices tend to have higher transmission times, as the medium is blocked by devices who had data ready to be sent earlier. Activity rate is not necessarily something that can be changed as easily in real life practice as in this simulation, but as we see, the transfer of data from start to finish happens much quicker when the Probability to Send is higher. For the data below:

* number of packets was set to a constant of either 1 or 5,
* tp was set to a constant .5 \* td
* P was set to a constant, randomly generated number for each run

This was done for easier calculations and for references to be made easier as well.

\*\*\*\*Please note that I hardcoded in values for numPackets, tp, and P for my testing, but changed it to accept user input for these values at the TAs request\*\*\*\*

**Data**