ECE358: Computer Networks

Lab Report 2

A Report Prepared For:
The University of Waterloo

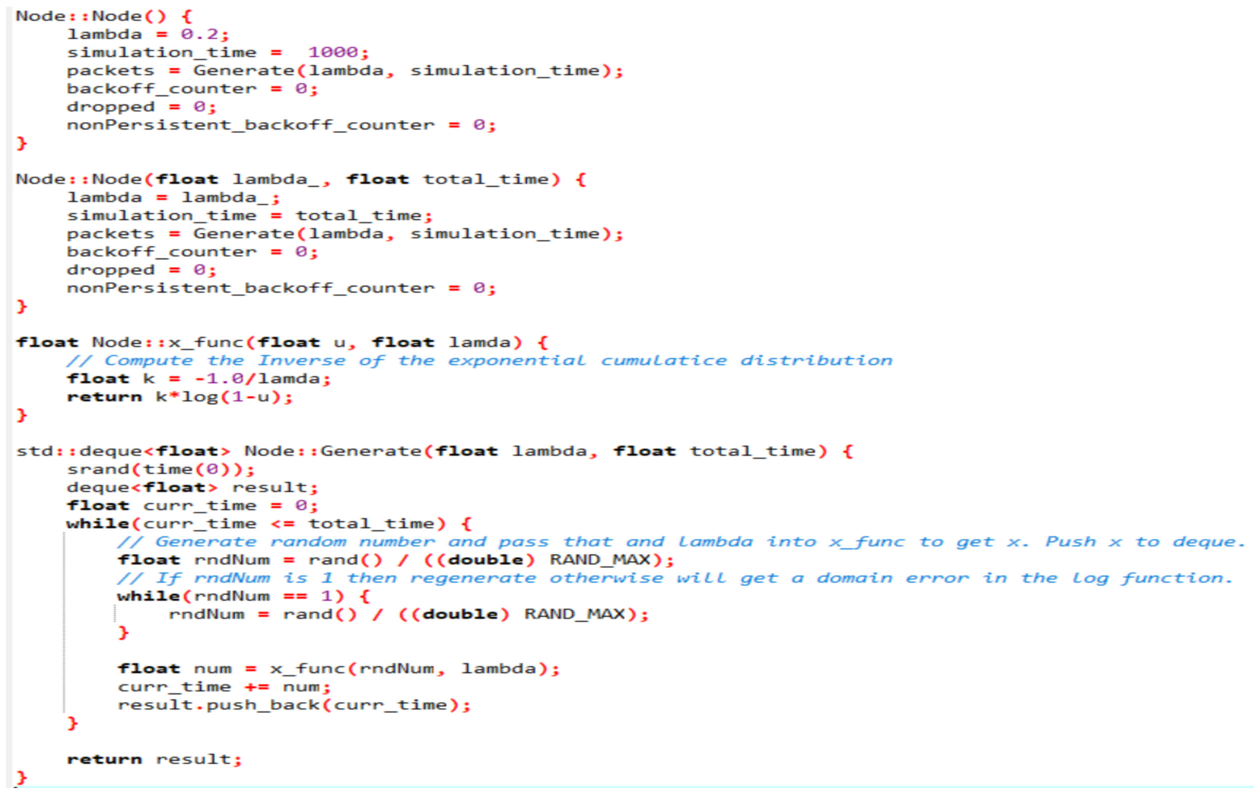
Prepared By:
Group 68
Youssef Roman
Alistair Fink

November 6, 2019

1.0 Simulator Design

1.0 Node Data Structure

The learning objective of this lab is to design and implement a discrete event simulator to evaluate the performance of local area networks (LAN) using CSMA/CD protocol. The main program“main.cpp” utilizes the Node class which can be found in figure 1 below. An instance of a node takes the average number of packets generated ‘lambda’ and the simulation time and uses the Generate function which follows the same process as lab 1.

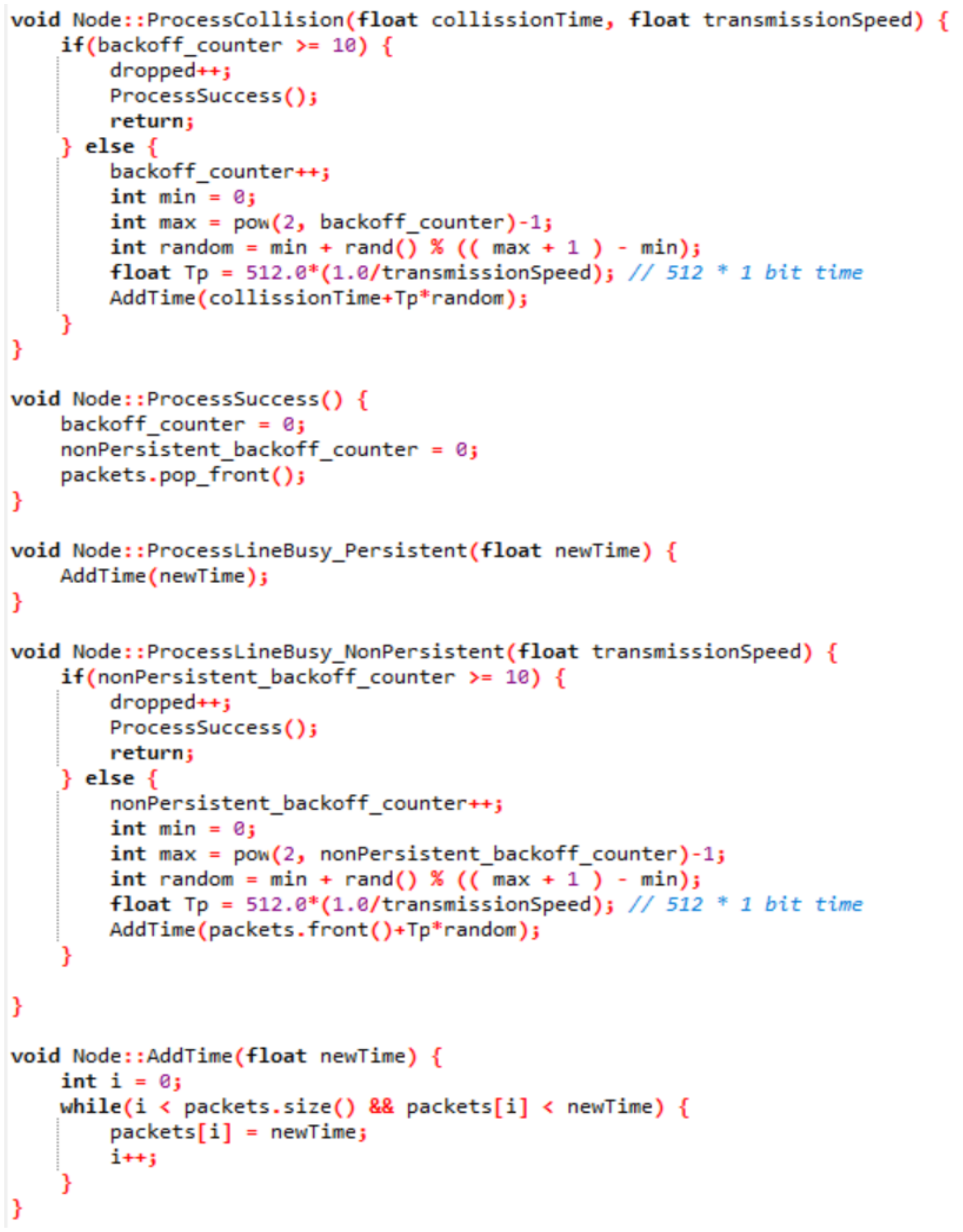


```
Node::Node() {
    lambda = 0.2;
    simulation_time =  1000;
    packets = Generate(lambda, simulation_time);
    backoff_counter = 0;
    dropped = 0;
    nonPersistent_backoff_counter = 0;
}

Node::Node(float lambda_, float total_time) {
    lambda = lambda_;
    simulation_time = total_time;
    packets = Generate(lambda, simulation_time);
    backoff_counter = 0;
    dropped = 0;
    nonPersistent_backoff_counter = 0;
}

float Node::x_func(float u, float lamda) {
    // Compute the Inverse of the exponential cumulatice distribution
    float k = -1.0/lamda;
    return k*log(1-u);
}

std::deque<float> Node::Generate(float lambda, float total_time) {
    srand(time(0));
    deque<float> result;
    float curr_time = 0;
    while(curr_time <= total_time) {
        // Generate random number and pass that and lambda into x_func to get x. Push x to deque.
        float rndNum = rand() / ((double) RAND_MAX);
        // If rndNum is 1 then regenerate otherwise will get a domain error in the log function.
        while(rndNum == 1) {
            rndNum = rand() / ((double) RAND_MAX);
        }

        float num = x_func(rndNum, lambda);
        curr_time += num;
        result.push_back(curr_time);
    }

    return result;
}
```

Figure 1: Node Constructor and Random Generator

Figure 2 below contains the functions used when nodes are either successful in transmitting its packet or when collision occurs. The “ProcessCollision” function checks the condition of the counter limit and drops the packet when the limit is reached to 10 trials. If it does not reach the limit, the backoff counter is incremented and the “AddTime” function is used to add a random time multiplied by 512 bit times which uses the transmission speed that is in the argument. The “ProcessSuccess function on the other hand, resets the back off counters for persistent and non persistent and removes the packet indicating that it has transmitted. The

“ProcessLineBusy_Persistent” function is used in persistent mode and it uses the “AddTime” function which adds the time needed for the node to check back the medium again. As for the non persistent mode, the “ProcessLineBusy_NonPersistent” function is used and checks if the backoff counter reached its limit of 10 trials. If the limit is reached, the packet is dropped. Otherwise, the back Off counter is incremented and the “AddTime” function is called to add the calculated time in terms of a random value time multiplied by 512 bit times.



```
void Node::ProcessCollision(float collissionTime, float transmissionSpeed) {
    if(backoff_counter >= 10) {
        dropped++;
        ProcessSuccess();
        return;
    } else {
        backoff_counter++;
        int min = 0;
        int max = pow(2, backoff_counter)-1;
        int random = min + rand() % (( max + 1 ) - min);
        float Tp = 512.0*(1.0/transmissionSpeed); // 512 * 1 bit time
        AddTime(collissionTime+Tp*random);
    }
}

void Node::ProcessSuccess() {
    backoff_counter = 0;
    nonPersistent_backoff_counter = 0;
    packets.pop_front();
}

void Node::ProcessLineBusy_Persistent(float newTime) {
    AddTime(newTime);
}

void Node::ProcessLineBusy_NonPersistent(float transmissionSpeed) {
    if(nonPersistent_backoff_counter >= 10) {
        dropped++;
        ProcessSuccess();
        return;
    } else {
        nonPersistent_backoff_counter++;
        int min = 0;
        int max = pow(2, nonPersistent_backoff_counter)-1;
        int random = min + rand() % (( max + 1 ) - min);
        float Tp = 512.0*(1.0/transmissionSpeed); // 512 * 1 bit time
        AddTime(packets.front()+Tp*random);
    }

}

void Node::AddTime(float newTime) {
    int i = 0;
    while(i < packets.size() && packets[i] < newTime) {
        packets[i] = newTime;
        i++;
    }
}
```

Figure 2: Node Processing

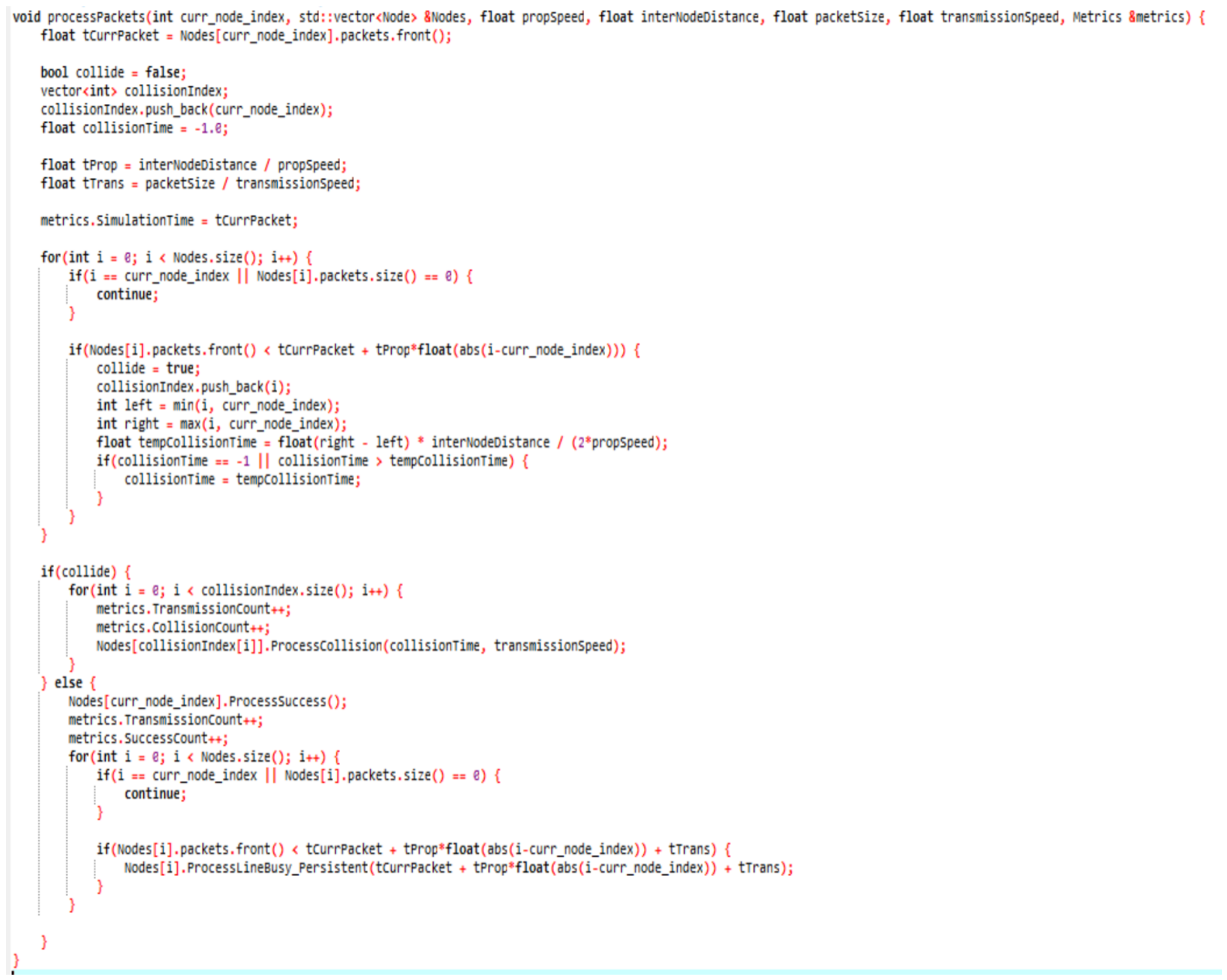
1.2 Process Packets in Persistent Mode

The “ProcessPackets” is the main function used in “Presistent.cpp” seen in figure 3 below. This function takes the current node index that needs to be processed and the vector that contains a list of the nodes. For computation and analysis, the function also takes the inter node distance, propagation speed, transmission speed, and an instance of a metrics class to update the counters. At first, the current packet of the node is saved into “tCurrPacket” float variable and boolean flag “collide” is set to false. An integer vector called “colissionIndex” is created and the current node that is transmitting add its node index into the collision index vector.

A for loop is used to iterate through all the nodes in the vector and compare the current node that is transmitting with other nodes. The “tnodePacket” is a float variable that saves the time of the node packet that is to be compared to. The distance, propagation time and transmission time are being evaluated between the current node that is being compared to the original node that is transmitting. Then they are saved into float variables, “distance”, ”tProp” and “tTransmission”. Then an if condition is added to check if the compared node has a time value less than the node that is transmitting which follows the order, (Nodes[i].packets.front() < tCurrPacket + tProp*float(abs(i-curr_node_index))). If the condition is satisfied, a collision occurs and the collision flag is set to true. The current node updates its index into the collision vector. Then a left integer variable is used to save the index of the minimum index value between the current compared node and the node that is transmitting.The right integer variable is used to save the maximum index value between the transmitting and compared node. The left and right variables are used to evaluate temporary collision time “tempCollisionTime” as (float(right - left) * interNodeDistance / (2*propSpeed)). Then the collision time variable is updated to the calculated “tempCollisionTime” variable as long as (collisionTime>tempCollisionTime).

After Iterating through all the nodes in the list and updating the counters, the collide flag is checked and a for loop is used to iterate through the number of collided nodes. The collision collision and transmission counters are incremented end each collided node calls the “ProcessCollide” function passing in the collision time and transmission speed as arguments.

When no collision occurs, the transmitted node calls the “ProcessSuccess” function and both transmission and success counters are incremented. Then a for loop is used to iterate through all the compared nodes and the condition (Nodes[i].packets.front() < tCurrPacket + tProp*float(abs(i-curr_node_index)) + tTrans) is used for the nodes that are sensing the medium when its busy. Therefore, these nodes call the “ProcessLineBusy_Persistent” function passing in the new time to be added until it can check the medium again.



```
void processPackets(int curr_node_index, std::vector<Node> &Nodes, float propSpeed, float interNodeDistance, float packetSize, float transmissionSpeed, Metrics &metrics) {
    float tCurrPacket = Nodes[curr_node_index].packets.front();

    bool collide = false;
    vector<int> collisionIndex;
    collisionIndex.push_back(curr_node_index);
    float collisionTime = -1.0;

    float tProp = interNodeDistance / propSpeed;
    float tTrans = packetSize / transmissionSpeed;

    metrics.SimulationTime = tCurrPacket;

    for(int i = 0; i < Nodes.size(); i++) {
        if(i == curr_node_index || Nodes[i].packets.size() == 0) {
            continue;
        }

        if(Nodes[i].packets.front() < tCurrPacket + tProp*float(abs(i-curr_node_index))) {
            collide = true;
            collisionIndex.push_back(i);
            int left = min(i, curr_node_index);
            int right = max(i, curr_node_index);
            float tempCollisionTime = float(right - left) * interNodeDistance / (2*propSpeed);
            if(collisionTime == -1 || collisionTime > tempCollisionTime) {
                collisionTime = tempCollisionTime;
            }
        }
    }

    if(collide) {
        for(int i = 0; i < collisionIndex.size(); i++) {
            metrics.TransmissionCount++;
            metrics.CollisionCount++;
            Nodes[collisionIndex[i]].ProcessCollision(collisionTime, transmissionSpeed);
        }
    } else {
        Nodes[curr_node_index].ProcessSuccess();
        metrics.TransmissionCount++;
        metrics.SuccessCount++;
        for(int i = 0; i < Nodes.size(); i++) {
            if(i == curr_node_index || Nodes[i].packets.size() == 0) {
                continue;
            }

            if(Nodes[i].packets.front() < tCurrPacket + tProp*float(abs(i-curr_node_index)) + tTrans) {
                Nodes[i].ProcessLineBusy_Persistent(tCurrPacket + tProp*float(abs(i-curr_node_index)) + tTrans);
            }
        }

    }
}
```

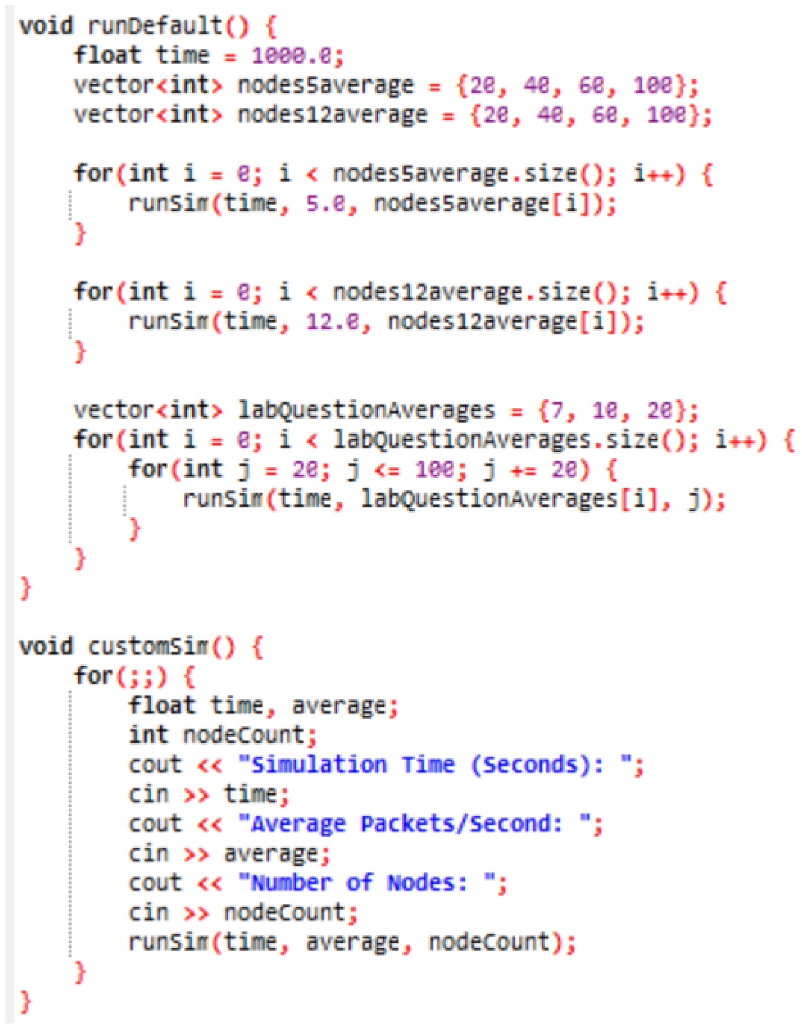
Figure 3: Process Packets Function Persistent Mode

1.3 Process Packets in Non Persistent Mode

For the non Persistent mode, the same logic for processing packets is followed. The only difference is that the nodes that are checking the medium when it is busy call the “ProcessLineBusy_NonPresistent” function. Since this function accounts for a limit counter which drops the packet if it reaches a limit of 10.

1.4 Simulator Run Through

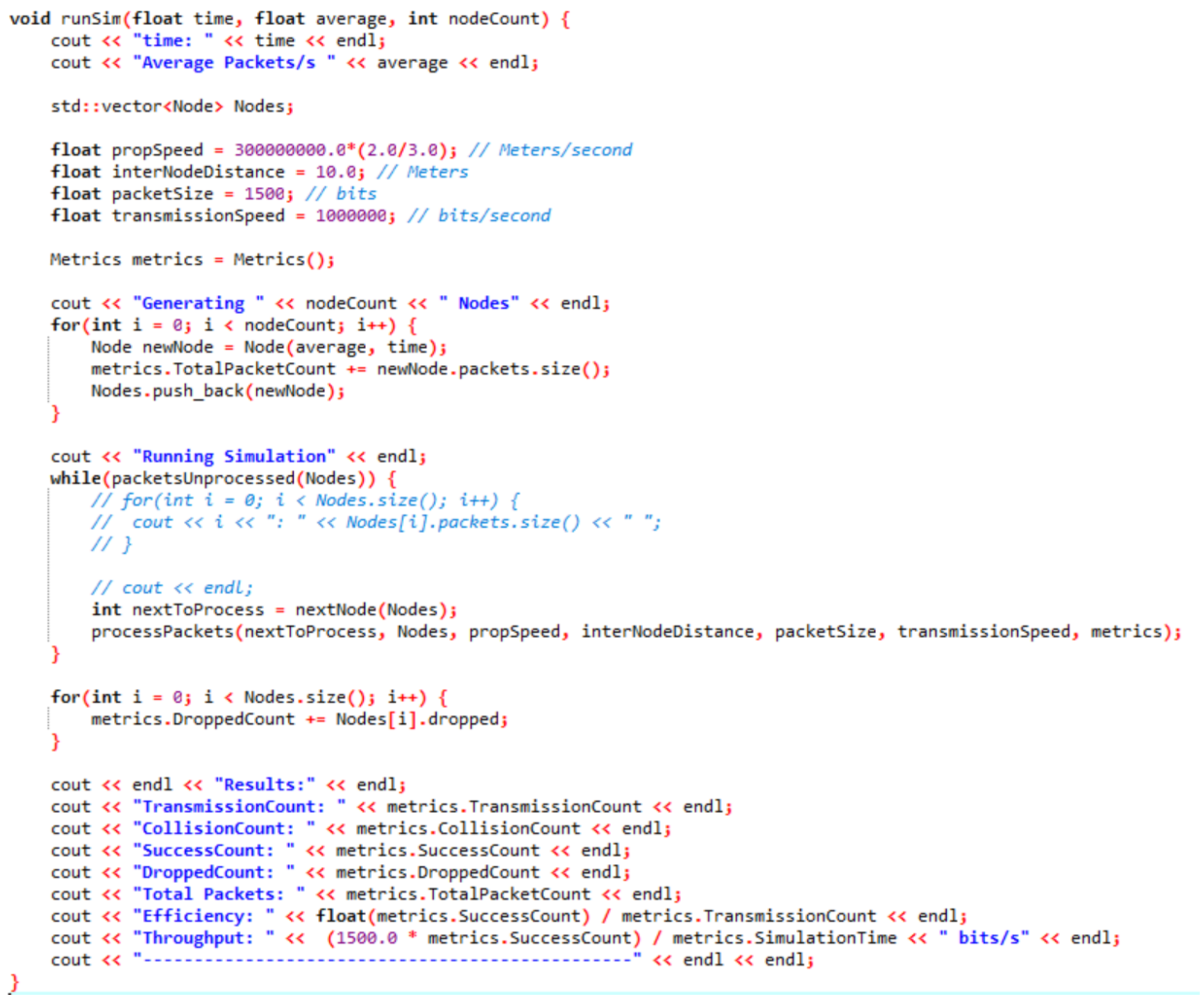
There are 2 modes for running the simulator for both persistent and non persistent modes which are shown in figure 4 below. The “runDefault” function runs the default combinations of average packets per second and node counts as outlined in the question section of the lab manual.The “customSim” function is another mode which runs a simulation according to the user’s input for number of nodes, average packet rate and simulation time.



```
void runDefault() {
    float time = 1000.0;
    vector<int> nodes5average = {20, 40, 60, 100};
    vector<int> nodes12average = {20, 40, 60, 100};

    for(int i = 0; i < nodes5average.size(); i++) {
        runSim(time, 5.0, nodes5average[i]);
    }

    for(int i = 0; i < nodes12average.size(); i++) {
        runSim(time, 12.0, nodes12average[i]);
    }

    vector<int> labQuestionAverages = {7, 10, 20};
    for(int i = 0; i < labQuestionAverages.size(); i++) {
        for(int j = 20; j <= 100; j += 20) {
            runSim(time, labQuestionAverages[i], j);
        }
    }
}

void customSim() {
    for(;;) {
        float time, average;
        int nodeCount;
        cout << "Simulation Time (Seconds): ";
        cin >> time;
        cout << "Average Packets/Second: ";
        cin >> average;
        cout << "Number of Nodes: ";
        cin >> nodeCount;
        runSim(time, average, nodeCount);
    }
}
```

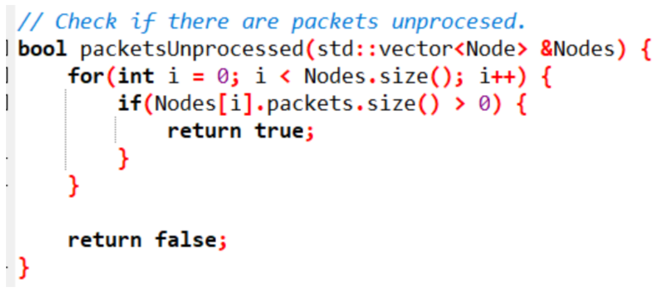
Figure 4: Custom and Default Simulation Mode

Both of the modes explained above run the simulation using the “runSim” function shown in figure 5 below which take in an average rate of packet/second and the node count. The function starts of by outputting to the console the simulation time and average packet rate. Then it creates a vector of Nodes to be used for metrics calculations. Then the parameter variables that follow the lab manual are created to store the propagation speed, transmission speed, the inter node distance and the packet size. An instance of metrics is created to store the counters. A for loop is applied to iterate through the number of nodes and adds the node to the vector. Then the simulation is run by calling the “packetUnprocessed” function shown in figure 6 below. This function takes in the list of nodes and returns false if the packets in all nodes are empty which is the exit condition for the simulation. As long as there are still packets that need to be transmitted by the nodes in the list, each node in the list calls the “ProcessPackets” function as explained above. After every packet is processed in the nodes list, the dropped packets counter is being added through a for loop which iterates through all nodes. Finally, the metrics parameters and counters are being output to the console.



```cpp
void runSim(float time, float average, int nodeCount) {
    cout << "time: " << time << endl;
    cout << "Average Packets/s " << average << endl;

    std::vector<Node> Nodes;

    float propSpeed = 300000000.0*(2.0/3.0); // Meters/second
    float interNodeDistance = 10.0; // Meters
    float packetSize = 1500; // bits
    float transmissionSpeed = 1000000; // bits/second

    Metrics metrics = Metrics();

    cout << "Generating " << nodeCount << " Nodes" << endl;
    for(int i = 0; i < nodeCount; i++) {
        Node newNode = Node(average, time);
        metrics.TotalPacketCount += newNode.packets.size();
        Nodes.push_back(newNode);
    }

    cout << "Running Simulation" << endl;
    while(packetsUnprocessed(Nodes)) {
        // for(int i = 0; i < Nodes.size(); i++) {
        //  cout << i << ": " << Nodes[i].packets.size() << " ";
        // }

        // cout << endl;
        int nextToProcess = nextNode(Nodes);
        processPackets(nextToProcess, Nodes, propSpeed, interNodeDistance, packetSize, transmissionSpeed, metrics);
    }

    for(int i = 0; i < Nodes.size(); i++) {
        metrics.DroppedCount += Nodes[i].dropped;
    }

    cout << endl << "Results:" << endl;
    cout << "TransmissionCount: " << metrics.TransmissionCount << endl;
    cout << "CollisionCount: " << metrics.CollisionCount << endl;
    cout << "SuccessCount: " << metrics.SuccessCount << endl;
    cout << "DroppedCount: " << metrics.DroppedCount << endl;
    cout << "Total Packets: " << metrics.TotalPacketCount << endl;
    cout << "Efficiency: " << float(metrics.SuccessCount) / metrics.TransmissionCount << endl;
    cout << "Throughput: " <<  (1500.0 * metrics.SuccessCount) / metrics.SimulationTime << " bits/s" << endl;
    cout << "-------------------------------------------------" << endl << endl;
}
```

Figure 5: Run Simulation Function

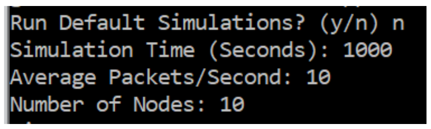


```cpp
// Check if there are packets unprocesed.
bool packetsUnprocessed(std::vector<Node> &Nodes) {
    for(int i = 0; i < Nodes.size(); i++) {
        if(Nodes[i].packets.size() > 0) {
            return true;
        }
    }

    return false;
}
```

Figure 6: Unprocessed Packet Boolean Function

1.5 Simulator Usage

The persistent simulator can be run by running the command “make -f Persistent.mk”. This command will both compile and run the simulator. Upon execution of this command the user will be prompted with whether or not they’d like to run the default simulation. Selecting ‘yes’ will run the default combinations of average packets per second and node counts as outlined in the question section of the lab manual. Selecting ‘no’ will allow the user to input custom simulation time, average packets per seconds, and number of nodes in order to run a custom simulation as shown in Figure 7.



```
Run Default Simulations? (y/n) n
Simulation Time (Seconds): 1000
Average Packets/Second: 10
Number of Nodes: 10
```

Figure 7: Persistent Simulator Custom Simulation

Similarly, the non-persistent simulator can be run by running the command “make -f NonPersistent.mk” which will compile and then run the simulator. Upon execution the non-persistent simulator will also prompt the user to run the default simulation. In both cases the non-persistent simulator will run in the same way as previously described for the persistent simulator.

Note, when running the default simulation for the persistent simulator some scenarios may take hours to run.

2.0 Persistent Simulator Results

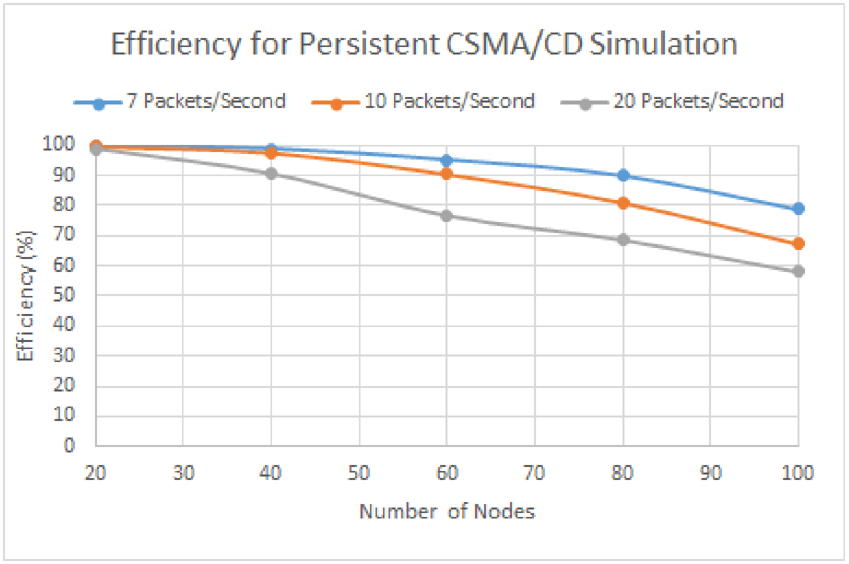


Figure 8: Persistent Simulation Efficiency Results

Running the persistent simulation for average packets per second of 7, 10, and 20 with node counts of 20, 40, 60, 80, and 100 nodes each yields the efficiencies as shown in Figure 8. As can be seen while each set of simulations starts at or close to 100% efficiency it steadily declines as more nodes are added with the rate at which they decline increasing with the amount of average packets per second. This is most likely due to the shorter average time period between packets when the amount of average packets per second is increased. With more average packets per seconds the packet timings become much closer and as such have much higher potential for collisions. As the number of nodes is increased this increases the potential for collisions and as such the efficiency drops as both of these factors are increased.

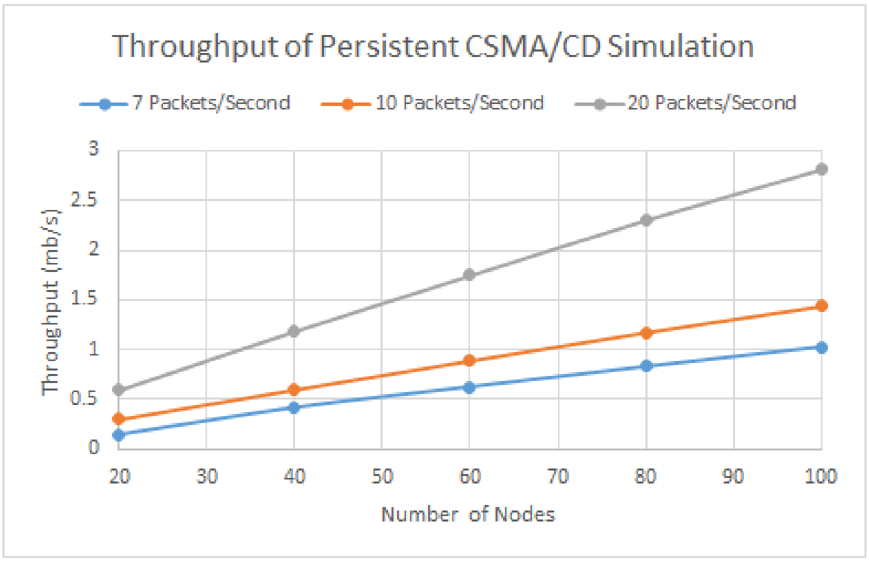


Figure 9: Persistent Simulation Throughput Results

Running the simulation also produces the results for throughput as shown in Figure 9. As can be seen the throughput seems to linearly increase in each simulation set and steadily rises with increasing nodes. This can most likely be attributed to the raw increase in the number of packets within the system in each scenario. The average total number of packets within a simulation can be calculated by taking the simulation time and multiplying it by the total average number of packets per second which can be calculated by multiplying the number of nodes by the average number of packets per second. This means that by holding the number of nodes constant for each scenario and then increasing the number of packets per second the amount of data being processed in the same time frame will also increase. As a result the throughput will increase.

3.0 Non-Persistent Simulator Results

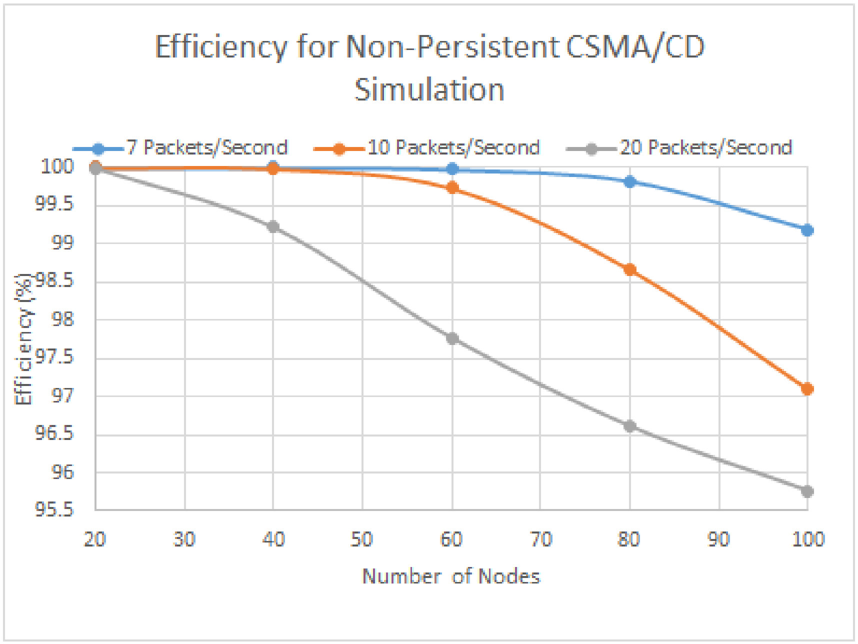


Figure 10: Non-Persistent Simulation Efficiency Results

Running the simulation for the non-persistent simulation yields the efficiency results as shown in Figure 10. As can be seen while each set of simulations gradually lowers in efficiency the actual efficiency only drops to about 95% at the lowest in all the scenarios. This is in contrast to the persistent model which has efficiencies that drastically decrease to about 58% at the lowest. This is most likely due to the fact that when the line is sensed as busy a node will on average wait longer than in the persistent simulation. As a result there is a much lower chance for constant collisions as the packets will naturally space themselves farther apart. This is in contrast to the persistent model which will have nodes try to broadcast as soon as the line is free which will introduce more chances for collisions.

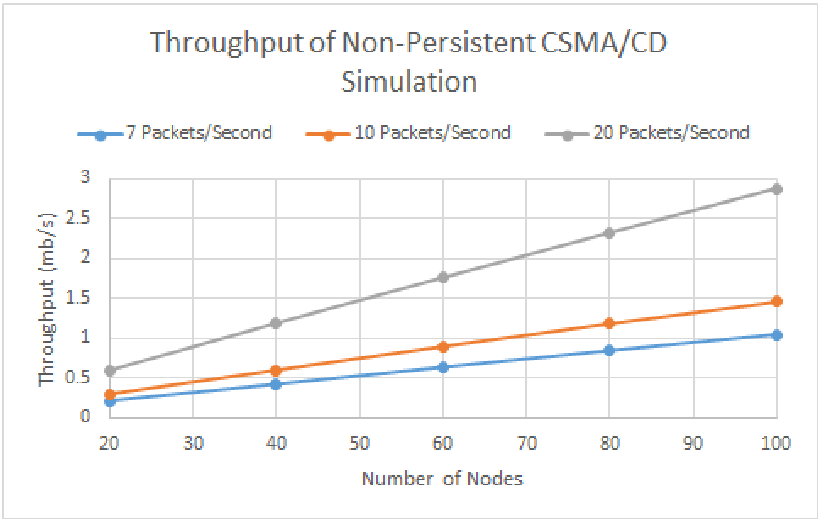


Figure 11: Non-Persistent Simulation Throughput Results

As can be seen in Figure 11 the throughput of each simulation set is linearly increasing and is very similar to the persistent model. This is most likely due to the fact that the average total number of packets in each scenario is the exact same as that of the persistent scenarios. By looking at the end of the graph for the 20 packets per second scenario with 100 nodes we see that the difference is about 0.07 mbps higher than that of the persistent model. This increase makes sense since the efficiency of this scenario is higher for the non-persistent simulation than that of the persistent simulation.