ECE 358: Computer Networks

Student Name: Qian, Yan Liang (David)

Date of submission: September 15, 2018

Submitted By: David Qian

Student ID: 20568467

Waterloo Email: ylqian@edu.uwaterloo.ca

Marks Received:

Marked by:

Contents

1 Question 1

To generate an exponential random variable from the uniform distribution, all we have to do is:

1. Generate a random number in [0, 1] from an uniform distribution. This can be done by calling the `rand()` function built into C++.
2. Return $X = \mathrm{F}^{-1}(U)$, where F^{-1} is the inverse of the exponential distribution function. This is derived below.

$$
\begin{aligned}
y &= 1 - e^{-\lambda x} \\
x &= 1 - e^{-\lambda y} \\
1 - x &= e^{-\lambda y} \\
F^{-1} &= -\frac{\ln(1-x)}{\lambda}
\end{aligned}
$$

The implementation for the method that generates exponentially distributed random numbers `genExpRand(double lambda)` can be found in `utils.cc`.

The implementation for the C++ code that generates 1000 numbers from an exponential distribution with $\lambda = 75$ is in `main.cc` and is named `validExpGen()` and is run as a sanity check before the simulation. From our experiment, the mean is 0.0129 and variance is 0.000166. The theoretical values for the mean and variance are $\mu = \frac{1}{\lambda} = \frac{1}{75} \approx 0.013$ and

$\sigma = \frac{1}{\lambda^2} = \frac{1}{5625} \approx 0.00017$. Our experimental values are within 1% of the expected values. Thus, we verify that our generator is valid.

2 Question 2

I decided to build the M/M/1/K queue first and let the M/M/1 queue be a special case of it where K = ∞. In my implementation, I decoupled the *simulation* from the *queue* itself.

In `simulation.cc`, I wrote a method `simulateMM1Queue(double alpha, doubled L, double lambda, double C, double T)`, where these variables follow the definition stated in the lab outline. This method generates the observer event times and packet arrival times via the Poission distribution method `genPoissionDistr(double lambda, double T)`, which can be found in `utils.cc`. This method repeatedly calls our exponential random number generator to generate intervals between events.

The packet sizes were generated using the exponential random number generator. To get the packet departure time for packet p, simply calculate $\frac{L_p}{C} +$ `max`$(lastPacketDepartureTime, packetArrivalTime)$. We combine these events into a vector of `Events` and then sort them by their event time. We then iterate through the events and feed them to the Network queue. At the end, we inspect the counters for the network queue and calculate our performance metrics.

In `queue.cc`, we initialize our counters and expose a method `handle(Event e)` which consumes an event and adjusts the state of counters to reflect the event. Specifically, we have counters for `numIdle` to reflect the number of observations in which the queue was idle, `numObservations` for the total number of observations, `numArrival` to count the number of packets that arrived at the queue, `numDepartures` for the number of packets that departed the queue (sent off by the server), and an accumulator `sumSampledQueueSize` which accumulates the size of the queue every time we observe it. Depending on the type of the event (specified by an enum), we do several different things:

- OBSERVER: increment `numObservations`, increment `numIdle` if the queue is idle (`currentSize = numArrivals - numDepartures`), and add the current size of the queue to a sum (`sumSampledQueueSize`) to calculate the average size later on.
- ARRIVAL: increment `numArrivals`.
- DEPARTURE: increment `numDepartures`.

You may have noticed that the `handle` method also generates an event. This is for the M/M/1/K queue case, where dynamic events are generated on the fly.

In particular, departure events may be generated on packet arrival. We will discuss this later on.

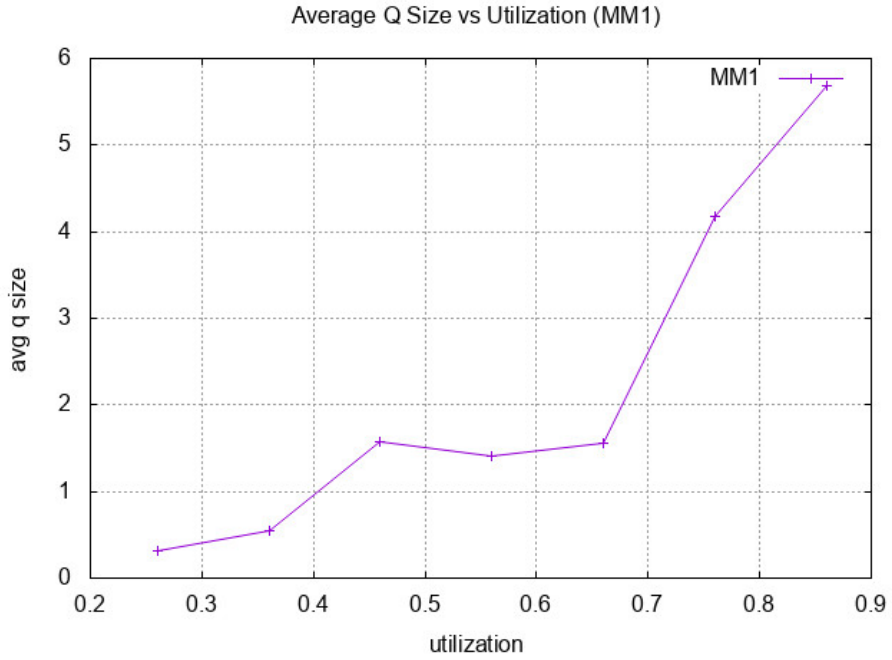
The core of the implementation is made available in the Appendix. For implementation of lower level functions, such as the random number generator, please refer to the source code submitted with the lab.

3 Question 3

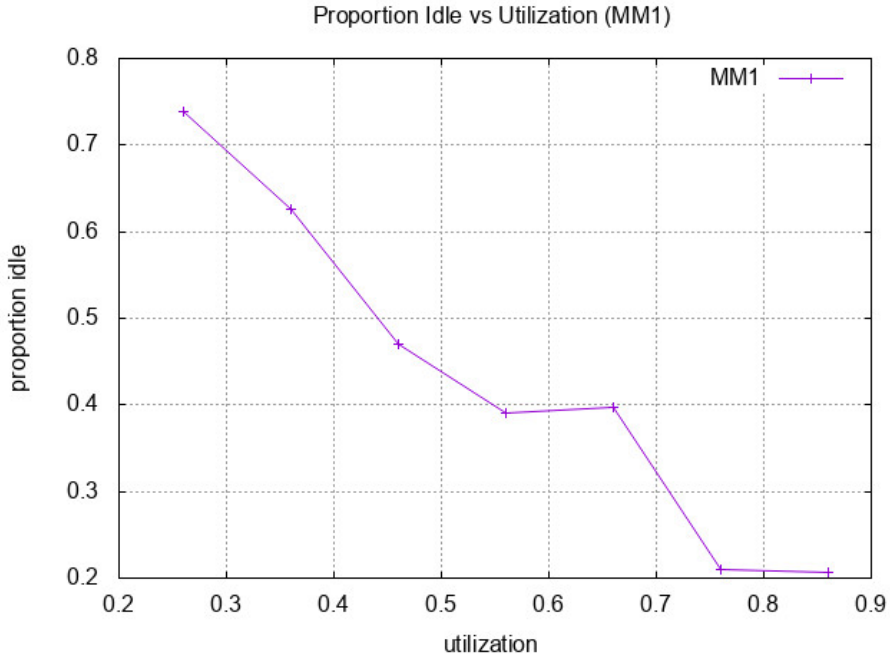
Setting $L = 12000$, $C = 10^6$, $T = 10$, $\alpha = 100$, we must determine λ as a function of ρ, where $0.25 < \rho < 0.95$. This is an easy re-arrangement of $\lambda = \frac{\rho C}{L}$. We then create a for loop from the lower and upper bounds of ρ and determine the necessary parameters for the simulation. We record the results below.

ρ	E[N]	P_{IDLE}
0.26	0.314	0.740
0.36	0.540	0.626
0.46	1.57	0.470
0.56	1.41	0.390
0.66	1.56	0.397
0.76	4.18	0.209
0.86	5.69	0.207

1. E[N]: To calculate this value, every time the queue processes an `OBSERVER` event, we add the `currentSize` = `numArrivals` − `numDepartures` to an accumulator `sumSampledQueueSize`. At the end of the simulation, we find that $E[N] = \frac{\texttt{sumSampledQueueSize}}{\texttt{numObservations}}$. The results are given below.



2. P_{IDLE}: To calculate this value, every time the queue processes an `OBSERVER` event, we increment our idle counter like so `numIdle` $+=$ `(currentSize) == 0)`. Then at the end of the simulation $P_{IDLE} = \frac{\texttt{numIdle}}{\texttt{numObservations}}$.



4 Question 4

For $\rho = 1.2$, we observe that $E[n] \approx 92$, which is a sharp increase from the values above. At the same time, $P_{IDLE} \approx 0.004$, which is close to the previous values.

The reason for this change is because the network queue is receiving data at a higher rate than it can output data (throughput). This leads to an increase in average *queuing delay*. As (average) queue utilization increases past 1, we see that more data arrives ($L\lambda$) then the server can output (C), on average. This means that there is an increased likelihood that an incoming packet will be waiting in the queue. This increase in average queuing delay is reflected in the fact that the average queue size is now sharply greater than before. Note that the proportion idle of the queue did not change significantly as for $\rho \approx 1$, the queue is already being heavily utilized and so the proportion of time it is idle is already pretty low. Increasing ρ any further will not change this metric by too much.

5 Question 5

As mentioned previously, M/M/1 is just a special case of M/M/1/K. Thus, our implementation strategy for the queue did not change, although now we execute

a different portion of the queue code.

In `simulation.cc` we implement the method `simulateMM1KQueue(double alpha, double L, double lambda, double C, double T, int K)`. These parameter are synonymous in meaning to those defined in the lab outline. Note that we define a new parameter K, which now indicates the max size of the queue. Like previously, we use `genPoissionDistr(double lambda, int T)` to generate the observer event times and packet arrival times, however the difference this time is that we no longer geneerate the departure times and packet sizes.

We fuse observer and arrival events into a single event queue using a 2 pointer approach. Then, we create a new queue called `dynamicEvents` which is responsible for storing events generated on the fly (i.e. departure events). We use another queue for departure events instead of inserting into one event queue because we would have to binary search for the proper index, which is computationally more expensive than having two queues. Every time we consume an event, all we have to do is compare the front of the two queues `events` and `dyanmicEvents` to choose the earliest one, and all we have to do to insert a new event is to push it back to the end of the `dynamicEvents` queue because packets arriving later will depart later due to the FIFO property. We then feed events from these two queues into the network queue and computer our performance metrics at the end.

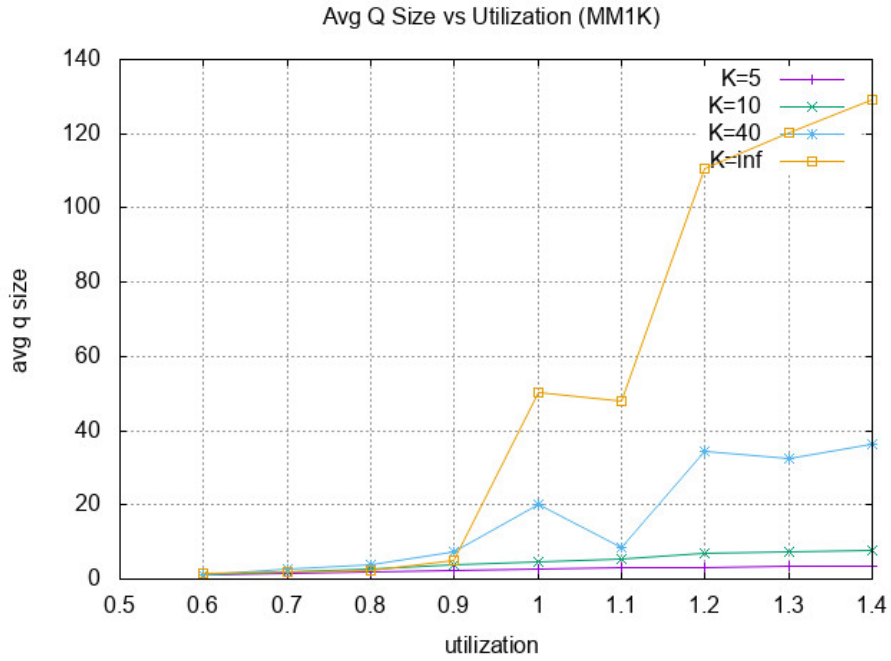
There is no new code in `queue.cc`. However, for the M/M/1/K we do use two additional counters - `numDropped` to record the number of packets dropped and `numPacketsGenerated` to keep track of the total number of packets generated. We set in the queue constructor the `isGenDepartureEvents` flag to allow it to generate departure events, pass `linkCapacity` so that we can compute service time on the fly, and input the `maxSize` of the queue. On `ARRIVAL` events, we now check if the queue is at max capacity and drop the packet and increment the dropped counter if it is. We also generate a new departure event for packets that aren't dropped via the same method as previously (using the last departure time and the service time). This departure event is added to the `dynamicEvents` queue and automatically consumed by the network queue later on.

6 Question 6

Setting $L = 12000$, $C = 10^6$, $T = 10$, $\alpha = 100$, we must determine λ as a function of ρ using the same method previously.

1. E[N]: To calculate this value, every time the queue processes an `OBSERVER` event, we add the `currentSize` = `numArrivals` − `numDepartures` to an accumulator. At the end of the simulation, we find that $E[N] = \frac{\texttt{sumSampledQueueSize}}{\texttt{numObservations}}$. The results are given below.

ρ	K = 1	K = 10	K = 40	K = inf
0.6	1.13078	1.11565	1.19658	1.44925
0.7	1.47799	1.93219	2.87003	1.77495
0.8	2.03437	2.77208	3.94422	2.29563
0.9	2.4027	3.86413	7.33369	4.93624
1	2.5391	4.67211	19.9806	50.2159
1.1	2.99901	5.55578	8.31594	48.0913
1.2	3.05712	6.94859	34.399	110.481
1.3	3.31509	7.50448	32.6788	120.194
1.4	3.359	7.55765	36.1759	129.229

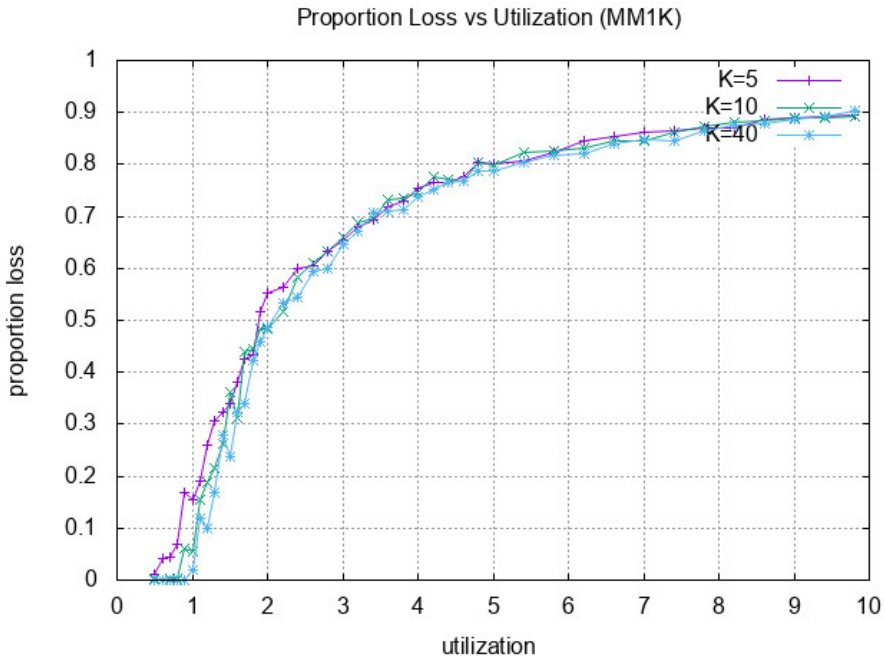


For the cases where K is bounded, we see that as queue utilization surpasses 1.0, the average queue size approaches K, that is the maximum queue size. This makes sense, since for utilization greater than 1, we see that on average, the queue receives more data it can handle and the average queue size increases to match, until it is capped at its maximum K. Newly arriving packets are dropped/lost when the queue is at its maximum size. However, for $K = \infty$, we can we see that the average queue size continue to grow and does not converge to a maximum. For the infinite case, the queuing delay still increases as its throughput is the same as the other cases - however, no packets are dropped and so the queue continues

to grow longer as it's size is not bounded. Comparing it with the other cases, we can roughly see the proportion of dropped packets by seeing how much larger the average queue size of the infinite case is.

2. P_{LOSS}: To calculate this value, we simply look at $P_{LOSS} = \frac{\texttt{numDropped}}{\texttt{numGenerate}}$. This is proportion of the packets that were dropped (i.e.lost) out of all the packets that were generated. We see that as utilization surpasses 1.0 for all the queues, the proportion lost starts to increase pretty fast. For small queue sizes, the proportion loss grows slightly slower than the larger queue sizes but not by much likely because the maximum queue sizes aren't too different.

ρ	K = 1	K = 10	K = 40
0.5	0.011655	0.00218341	0
0.6	0.0427184	0	0
0.7	0.0429752	0.00369686	0
0.8	0.0679612	0.00158983	0
0.9	0.16775	0.0597015	0
1	0.155131	0.0552291	0.0206422
1.1	0.190639	0.155153	0.118077
1.2	0.260654	0.189216	0.0990991
1.3	0.305685	0.215631	0.168819
1.4	0.3223	0.263296	0.28021
1.5	0.340278	0.362319	0.237131
1.6	0.381095	0.313386	0.321857
1.7	0.426056	0.439499	0.340333
1.8	0.434929	0.444966	0.422431
1.9	0.515319	0.483496	0.459339
2	0.551457	0.483949	0.486293
2.2	0.564525	0.515442	0.531789
2.4	0.599027	0.583754	0.545216
2.6	0.60613	0.609317	0.593304
2.8	0.631834	0.632035	0.598371
3	0.653815	0.659839	0.647781
3.2	0.679395	0.687339	0.670509
3.4	0.694084	0.695092	0.705862
3.6	0.718057	0.732596	0.710169
3.8	0.729051	0.735108	0.714017
4	0.753094	0.7463	0.737444
4.2	0.764156	0.776921	0.751569
4.4	0.765463	0.770772	0.765139
4.6	0.775715	0.768233	0.76697
4.8	0.805083	0.804795	0.787398
5	0.802193	0.799184	0.788475
5.4	0.806048	0.823413	0.805051
5.8	0.823492	0.826238	0.818182
6.2	0.846125	0.830518	0.820089
6.6	0.853726	0.844981	0.839211
7	0.860944	0.846434	0.849332
7.4	0.863806	0.861075	0.846154
7.8	0.870373	0.872278	0.863419
8.2	0.869099	0.881761	0.874689
8.6	0.887692	0.884476	0.877634
9	0.888336	0.889403	0.887613
9.4	0.893008	0.890247	0.891016
9.8	0.894852	0.892804	0.902492


Proportion Loss vs Utilization (MM1K)
proportion loss
utilization
K=5
K=10
K=40
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
1
2
3
4
5
6
7
8
9
10

7 Appendix

simulation.cc

```
PerformanceMetrics simulateMM1Queue(double alpha, double L, double lambda,
                                    double C, double T) {
    // determine observer times
    auto observerEventTimes = genPoissionDistr(alpha, T);

    // determine packet arrival time/sizes
    auto packetArrivialTimes = genPoissionDistr(lambda, T);
    auto packetSizes = genExpDistr(1.0/L, packetArrivialTimes->size());

    // snap sizes to ints
    for (int i = 0; i < packetSizes->size(); i++) {
        packetSizes->at(i) = round(packetSizes->at(i));
    }


    // determine departure times
    shared_ptr<vector<double>> packetDepartureTimes = make_shared<vector<double>>();
    double lastDepartureTime = 0;

    for (int i = 0; i < packetArrivialTimes->size(); i++) {
        double arrivalTime = packetArrivialTimes->at(i);
        double packetSize = packetSizes->at(i);

        double serviceTime = packetSize/C;

        if (arrivalTime >= lastDepartureTime) {
            lastDepartureTime = arrivalTime + serviceTime;
        } else {
            lastDepartureTime += serviceTime;
        }

        packetDepartureTimes->push_back(lastDepartureTime);
    }

    // combine into an event queue for easier simulation
    vector<Event> events;

    for (int i = 0; i < observerEventTimes->size(); i++) {
        Event event = Event(EventType::OBSERVER, observerEventTimes->at(i));
        events.push_back(event);
    }

    for (int i = 0; i < packetArrivialTimes->size(); i++) {
        Event event = Event(EventType::ARRIVAL, packetArrivialTimes->at(i));
        event._packetSize = packetSizes->at(i);
        events.push_back(event);
    }

    for (int i = 0; i < packetDepartureTimes->size(); i++) {
        Event event = Event(EventType::DEPARTURE, packetDepartureTimes->at(i));
        events.push_back(event);
    }

    sort(events.begin(), events.end(),
```

```
        [](Event a, Event b) {return a._eventTime < b._eventTime;}
      );

    // simulate
    NetworkQueue q;
    for (int i = 0; i < events.size(); i += 1) {
        q.handle(events.at(i));
    }

    double averageNumberOfPackets = 1.0 * q._sumSampledQueueSize / q._numObservations;
    double proportionIdle = 1.0 * q._numIdle / q._numObservations;

    /* cout << "Num Arrivals: " << q._numArrivals << endl; */
    /* cout << "Num Departures: " << q._numDepartures << endl; */
    /* cout << "Num Observations: " << q._numObservations << endl; */


    /* cout << "Average Number of Packets: " << averageNumberOfPackets << endl; */
    /* cout << "Proprtion Idle: " << proportionIdle << endl << endl; */

    PerformanceMetrics metrics;

    metrics.pIdle = proportionIdle;
    metrics.pLoss = 0.0;
    metrics.averageQSize = averageNumberOfPackets;

    return metrics;
}

PerformanceMetrics simulateMM1KQueue(double alpha, double L, double lambda,
                                     double C, double T, int K) {
    // determine observer times
    auto observerEventTimes = genPoissionDistr(alpha, T);

    // determine packet arrival time/sizes
    auto packetArrivialTimes = genPoissionDistr(lambda, T);

    // combine into an event queue for easier simulation
    queue<Event> events;

    int p1 = 0, p2 = 0;
    while (p1 < observerEventTimes->size() || p2 < packetArrivialTimes->size()) {
        if (p1 < observerEventTimes->size() && p2 < packetArrivialTimes->size()) {
            if (observerEventTimes->at(p1) < packetArrivialTimes->at(p2)) {
                events.push(Event(EventType::OBSERVER, observerEventTimes->at(p1++)));
            } else {
                events.push(Event(EventType::ARRIVAL, packetArrivialTimes->at(p2++)));
                events.back()._averagePacketLength = L;
            }
            continue;
        }

        if (p1 < observerEventTimes->size()) {
            events.push(Event(EventType::OBSERVER, observerEventTimes->at(p1++)));
        } else {
            events.push(Event(EventType::ARRIVAL, packetArrivialTimes->at(p2++)));
        }
```

```
    }

    queue<Event> dynamicEvents;  // dynamic events generated on the fly

    // simulate
    NetworkQueue q(true, K, C);

    while (!events.empty() || !dynamicEvents.empty()) {
        if (!events.empty() && !dynamicEvents.empty()) {
            if (events.front()._eventTime < dynamicEvents.front()._eventTime) {
                Event newEvent = q.handle(events.front());
                events.pop();

                if (newEvent._type != EventType::NONE) dynamicEvents.push(newEvent);
            } else {
                Event newEvent = q.handle(dynamicEvents.front());
                dynamicEvents.pop();

                if (newEvent._type != EventType::NONE) dynamicEvents.push(newEvent);
            }
            continue;
        }

        if (!events.empty()) {
            Event newEvent = q.handle(events.front());
            events.pop();
            if (newEvent._type != EventType::NONE) dynamicEvents.push(newEvent);
        } else {
            Event newEvent = q.handle(dynamicEvents.front());
            dynamicEvents.pop();
            if (newEvent._type != EventType::NONE) dynamicEvents.push(newEvent);
        }
    }

    double averageNumberOfPackets = 1.0 * q._sumSampledQueueSize / q._numObservations;
    double proportionIdle = 1.0 * q._numIdle / q._numObservations;
    double proportionLost = 1.0 * q._numDropped / q._numPacketsGenerated;

    /* cout << "Num Arrivals: " << q._numArrivals << endl; */
    /* cout << "Num Departures: " << q._numDepartures << endl; */
    /* cout << "Num Observations: " << q._numObservations << endl; */
    /* cout << "Num Dropped: " << q._numDropped << endl; */
    /* cout << "Num Generated: " << q._numPacketsGenerated << endl; */


    /* cout << "Average Number of Packets: " << averageNumberOfPackets << endl; */
    /* cout << "Proprtion Idle: " << proportionIdle << endl; */
    /* cout << "Proprtion Lost: " << proportionLost << endl << endl; */

    PerformanceMetrics metrics;

    metrics.pIdle = proportionIdle;
    metrics.pLoss = proportionLost;
    metrics.averageQSize = averageNumberOfPackets;

    return metrics;
}
```

queue.cc

```
enum class EventType { NONE, OBSERVER, ARRIVAL, DEPARTURE };

struct Event {
    double _eventTime;
    EventType _type;

    // potential metadata
    int _packetSize = -1;
    double _averagePacketLength = 0.0;

    Event(EventType type, double eventTime = 0);
};

struct NetworkQueue {
    // counters
    int _numArrivals = 0, _numDepartures = 0;
    int _numObservations = 0, _numIdle = 0;

    int _sumSampledQueueSize = 0;

    // M/M/1/N QUEUE IMPL START
    bool _isGenDepartureEvents;       // let the queue generate its own departure events
    double _lastDepartureTime = 0;    // the expected departure time of the last packet in q
    double _linkCapacity;
    int _maxSize;                     // default infinite size
    int _numPacketsGenerated = 0, _numDropped = 0;
    // M/M/1/N QUEUE IMPL END


    NetworkQueue(
        bool isGenDepartureEvents = false,
        int maxSize = -1,
        double linkCapacity = 1000000
    );

    Event handle(Event e);
};


Event NetworkQueue::handle(Event e) {
    int currentSize = _numArrivals - _numDepartures;

    if (e._type == EventType::OBSERVER) {
        _numObservations += 1;
        _numIdle += (currentSize == 0);

        _sumSampledQueueSize += currentSize;
    } else if (e._type == EventType::ARRIVAL) {
        _numPacketsGenerated += 1;

        if (currentSize >= _maxSize && _maxSize != -1) {
            // packets dropped
            _numDropped += 1;
        } else {
            // packet not dropped
```

```
            _numArrivals += 1;
            currentSize += 1;

            // M/M/1/N QUEUE IMPL START
            if (e._packetSize == -1) {
                // packet size undetermined = randomly generate
                e._packetSize = genExpRand(1.0/e._averagePacketLength);
            }

            if (_isGenDepartureEvents) {
                double serviceTime = e._packetSize / _linkCapacity;

                if (currentSize == 1) {
                    // this packet just arrived
                    _lastDepartureTime = e._eventTime + serviceTime;
                } else {
                    _lastDepartureTime = _lastDepartureTime + serviceTime;
                }

                return Event(EventType::DEPARTURE, _lastDepartureTime);
            }
            // M/M/1/N QUEUE IMPL END
        }

    } else if (e._type == EventType::DEPARTURE) {
        _numDepartures += 1;
    }

    return Event(EventType::NONE);  // event type is none branches here
}
```