ECE358: Computer Networks

Winter 2014

Project 1: Queue Simulation

Date of submission: Feb 3, 2014

Submitted by:

Student ID: (daskinne) 20373565

Davd Skinner

[daskinne@uwaterloo.ca](mailto:daskinne@uwaterloo.ca)

Marks received:

Marked by:

Table of Contents

Question1 2

Question 2 3

Question 2 Code Listing 4

Question 3 10

Question 4 11

Question 5 11

Question 6 12

# Question1

What is the mean and variance of the 1000 random variables you generated?

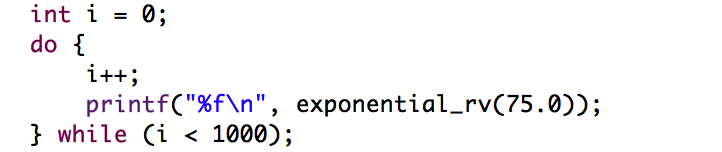
F(x) = exp(-1\*lambda \* x)  
x = -ln(F(x))/lambda

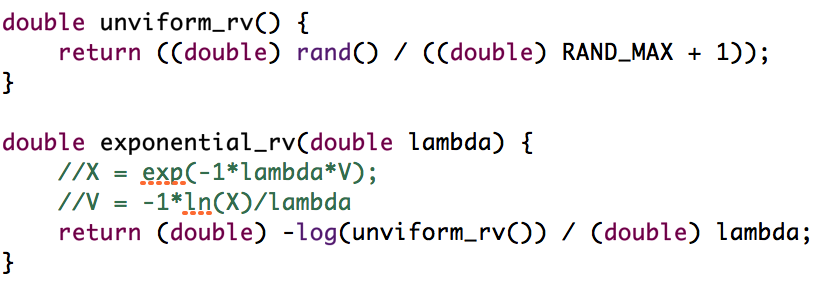
Where F(x) == U(0,1)

(calculated) => (observed)

Mean: 1/lambda = 0.133- => 0.0128652

Variance: 1/lambda^2 = 0.000177- => 0.00017793





# Question 2

To create the simulator I defined a class Simulator which encapsulates the event scheduler (ES) and all of the functional requirements. Each stage of the simulation is performed by a self describing function which accepts the relevant parameters. The re-used parameters: link rate, queue size, duration (T), and average packet size (L) are defined in the constructor to avoid misuse. The event scheduler itself is defined as a list of simEvent structs. These structs are assigned a type (EventType) corresponding to arrivals, departures and observations. The statistic calculations are stored within the main simulator class and are public so as to be accessible when the simulation is complete. As can be seen in the main, the simulator usage is as follows.

Sim::Simulator\* sim = **new** Sim::Simulator(C, queue\_size, T, L);

//Generate observation events, poisson parameter alpha ( if less than T)

sim->generate\_observations(alpha);

//Packet arrival times (parameter lambda)

sim->generate\_arrivals(lambda);

sim->order\_events();

//Find departure times

sim->calculate\_departures();

//packet arrivals so far, packets departures so far

sim->order\_events();

sim->observe\_events();

The challenges I faced with implementing the scheduler were as follows:

* Brushing up on c++, as it has been a long time since I have had to use my pointer knowledge for good
* Efficiently sorting the ES only when necessary
* Cleaning up memory between simulation runs

The overall logic followed naturally from the description in the lab manual, however the definition of the random variable generator could use some improvement. Thankfully Albert’s description in the lab presentation was sufficient to arrive at the result below. I would appreciate if this section of the lab manual was improved, as well as the difference between the Poisson and exponential R.V. generators, I think my understanding would have been greater had this been more explicit vs. relying on the internet for clarification.

## Question 2 Code Listing

**namespace** Sim {

**enum** EventType {

*ARRIVAL*, *DEPARTURE*, *OBSERVATION*

};

**struct** simEvent {

EventType type;

**int** id;

**double** packetLength;

**double** time;

**bool** dropped;

} SimEvent;

**bool** **compare\_times**(simEvent& first, simEvent& second) {

**return** first.time < second.time;

}

;

**class** Simulator {

**int** Na; //Arrivals

**int** Nd; //Departures

**int** No; //Observations

**int** linkRate;

**int** queueSize;

**int** durationT;

**int** avgPacketSize;

list<simEvent> ES;

**public**:

/////////STATS/////

**int** num\_observations;

**int** num\_packets;

**double** num\_packets\_in\_buffer;

**double** sojourn\_time;

**double** pIdle;

**double** pLoss;

//////////////////

**Simulator**(**int** linkRate, **int** queueSize, **int** durationT, **int** L) {

**this**->linkRate = linkRate;

**this**->queueSize = queueSize;

**this**->durationT = durationT;

**this**->avgPacketSize = L;

**this**->num\_packets\_in\_buffer = 0;

**this**->sojourn\_time = 0;

**this**->pIdle = 0;

**this**->pLoss = 0;

}

**~Simulator**() {

ES.clear();

}

**void** **generate\_observations**(**double** alpha) {

**double** time = 0;

**int** i = 1;

**while** (**true**) {

simEvent e;

e.type = *OBSERVATION*;

e.id = i;

e.dropped = **false**;

time += exponential\_rv(alpha);

e.time = time;

**if** (e.time > durationT) {

**break**;

}

i++;

ES.push\_back(e);

}

num\_observations = i;

}

**void** **generate\_arrivals**(**double** lambda) {

**double** time = 0;

**int** i = 1;

**while** (**true**) {

simEvent e;

e.type = *ARRIVAL*;

e.id = i;

e.dropped = **false**;

e.packetLength = exponential\_rv(1.0 / (**double**) avgPacketSize);

time += exponential\_rv(lambda);

e.time = time;

**if** (e.time > durationT) {

**break**;

}

i++;

ES.push\_back(e);

}

num\_packets = i;

}

**void** **calculate\_departures**() {

**double** current\_time = 0;

**int** packet\_count = 0;

**for** (list<simEvent>::iterator it = ES.begin(); it != ES.end(); ++it) {

**if** (it->type != *ARRIVAL* || it->dropped) {

**continue**;

}

//How long has this packet been waiting?

//printf("Packet at time %f (delta)%f: id %d size: %f \n", it->time, (current\_time - it->time), it->id, it->packetLength);

**if** (it->time > current\_time) {

current\_time = it->time;

}

//add departure event at

**double** departureTime = (**double**) current\_time

+ (it->packetLength / (**double**) linkRate);

**if** (queueSize > 0) {

//if finite queue, mark dropped packets by considering the number left in the queue while servicing

//count # events between current time and departureTime, if > queueSize, drop packet

**int** numQueued = 0;

**for** (list<simEvent>::iterator dq = it; dq != ES.end(); ++dq) {

**if** (dq->type != *ARRIVAL* || dq->dropped) {

//only check arrivals and packets that have not yet been dropped

**continue**;

}

**if** (dq->time > departureTime) {

**break**;

//We have moved beyond the affected interval

}

**if** (numQueued == queueSize) {

//Start dropping packets

//printf("Packet id %d dropped (time:%f)\n", dq->id, dq->time);

dq->dropped = **true**;

pLoss++;

} **else** {

numQueued++;

}

}

}

//add departure event

simEvent e;

e.id = it->id;

e.type = *DEPARTURE*;

e.time = departureTime;

//printf("Packet departs at time %f id %d \n", e.time, e.id);

ES.push\_back(e);

sojourn\_time += it->time;

packet\_count++;

//time advances to take new packet

current\_time = departureTime;

}

//average sojourn time

sojourn\_time /= (**double**) packet\_count;

pLoss /= (**double**) num\_packets;

}

**void** **order\_events**() {

ES.sort(compare\_times);

}

**void** **observe\_event**(simEvent\* se) {

**switch** (se->type) {

**case** *ARRIVAL*:

**if** (!se->dropped) {

Na++;

}

**break**;

**case** *DEPARTURE*:

Nd++;

**break**;

**case** *OBSERVATION*:

No++;

num\_packets\_in\_buffer += Na - Nd;

**if** (Na == Nd) {

pIdle++;

}

//printf("Time: %f No %d Na %d Nd %d\n", se->time, No, Na, Nd);

**break**;

}

}

**void** **observe\_events**() {

Na = 0;

Nd = 0;

No = 0;

**for** (list<simEvent>::iterator it = ES.begin(); it != ES.end(); ++it) {

observe\_event(&\*it);

}

pIdle /= num\_observations;

num\_packets\_in\_buffer /= (**double**) num\_observations;

}

};

}

**int** **main**(**void**) {

Sim::Simulator\* sim = **new** Sim::Simulator(C, queue\_size, T, L);

//Generate observation events, poisson parameter alpha ( if less than T)

sim->generate\_observations(alpha);

//Packet arrival times (parameter lambda)

sim->generate\_arrivals(lambda);

sim->order\_events();

//Find departure times

sim->calculate\_departures();

//packet arrivals so far, packets departures so far

sim->order\_events();

sim->observe\_events();

**printf**("%f\t%d\t%f\t%f\t%f\t%f\t%f\t%f\t\n", p, T, lambda, alpha,

sim->num\_packets\_in\_buffer, sim->sojourn\_time, sim->pIdle,

sim->pLoss);

//number of observations

**delete** sim;

}

}

**return** EXIT\_SUCCESS;

}

;

# Question 3

The simulator was run with lambda=alpha calculated as follows.

Lambda=alpha= p\*L/C

The cases were run for T=11000, and 2T=22000. The %difference between the results was nominally < 2.5%. This was well within the defined tolerance of 5%, as such future simulations were run with a T of 11000.

E[N] = Number Arrivals – Number Departures, recorded on each observation event, and the sum divided by the number of observation events.

pIdle = #(Arrivals-Departures == 0), recorded on each observation event, and divided by the number of observation events.

The figures corresponded with my intuition. As the interarrival times were exponentially distributed you would expect the exponential utilization. The idle time relates linearly to the utilization, and you would expect a utilization of 100% to correspond to 0% idle. The realized value for p = 0.95 was ~1% and the series was highly linear.

# Question 4

When the utilization (p) is >1, we would describe the system as oversubscribed. One would expect the E[N] to ‘explode’ as the system is backed up with more packets than it can accommodate. In practice, the value of E[N] ‘exploded’ as expected with a value of 90k. pIdle should also tend towards 0 as the system is constantly busy servicing packets. This was also observed as the value recorded was 0% idle.

# Question 5

To introduce the bounded queue, I had to add a condition to look ahead and mark packets as dropped when calculating the departure times. This required adding another parameter to the simulator class for queue size, and a loop to seek through the ES to count off the number of allowed packets and mark all packets beyond this threshold and with arrival times less than the calculated departure of the current packet as dropped. These packets were then skipped when reached by the departure time calculation method, and ignored when observing the final queue.

The relevant changes are included in the listing below. These are provided in context in the previous code listing to avoid duplication.

**if** (queueSize > 0) {

//if finite queue, mark dropped packets by considering the number left in the queue while servicing

//count # events between current time and departureTime, if > queueSize, drop packet

**int** numQueued = 0;

**for** (list<simEvent>::iterator dq = it; dq != ES.end(); ++dq) {

**if** (dq->type != *ARRIVAL* || dq->dropped) {

//only check arrivals and packets that have not yet been dropped

**continue**;

}

**if** (dq->time > departureTime) {

**break**;

//We have moved beyond the affected interval

}

**if** (numQueued == queueSize) {

//Start dropping packets

//printf("Packet id %d dropped (time:%f)\n", dq->id, dq->time);

dq->dropped = **true**;

pLoss++;

} **else** {

numQueued++;

}

}

}

# Question 6

1. We can see with this plot that as p increases past 100% utilization, the number of packets in the queue tends towards the size of the queue (k). What is interesting is that the queue does not 100% saturate at p=1.4. I hypothesize that this may be because there were not enough observer events, though it may be that the service time of the packets is quick enough to clear the queue at this level. We can see that at p=1, k=40 we achieve a E[N] approximately equal to that of the unbounded queue. This would indicate that the service time of the packets is following the same distribution as the unbounded queue. We know this to be true as the packet size (L) is equal for both the bounded and unbounded cases. Furthermore, it is very unlikely to have a statistically constant 100% full queue, as the packets follow a Poisson process and packets arriving very close together are likely to result in dropped packets creating a gap in arrivals.

pLoss = #droppedPackets / #Packets, calculated by counting the number of dropped packets when calculating the departure times and dividing by the total number of packets generated over the entire simulation interval.

This was the most interesting of all of the experiments. We see that regardless of queue size the pLoss is highly correlated. While the relative percentage loss is different for p < 1 (the smaller our buffer the larger the % lost as the queue size is having a larger impact on the loss than the oversubscription of events); it is nearly 100% correlated for p>1. The three series have the exact same asymptotic performance as degradation of the system is dominated by the oversubscription of events for utilization levels >100%. We see that for all of these series pLoss => 0.9 as packets will continue to be serviced despite the high drop rate. It is expected that at p => infinity this would continue to approach 100% packet loss.