COMPENG 4DK4 LAB2

Richard Qiu – 400318681 – Group 1

Experiment

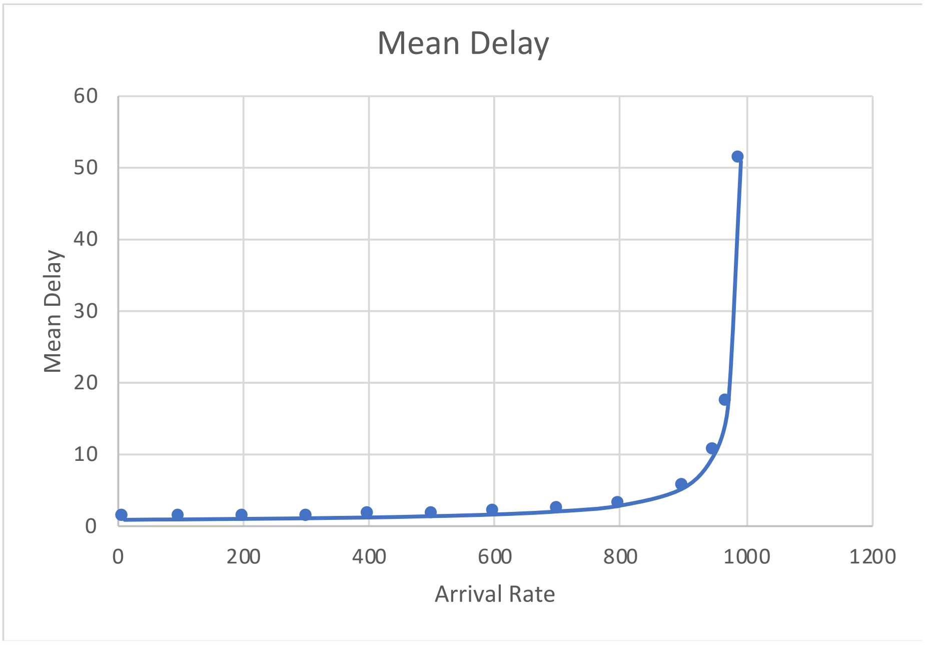
2.

The Packet length is 1000 bits, and the output link operator has a bit rate of 1 Mbps, to make sure there are no packet losses during the transmission, we need to meet the below requirement.

0 < PACKET_ARRIVAL_RATE * PACKET_LENGTH < LINK_BIT_RATE

By simulating each arrival rate with 10 different random seeds and find the average mean delay. The result can be obtained below.

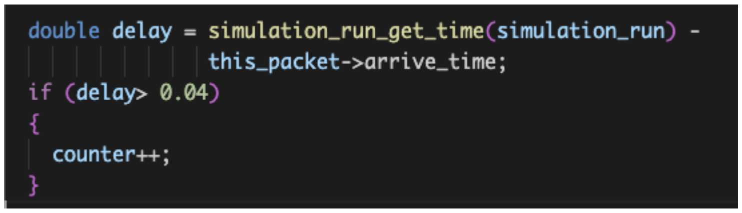
Arrival Rate	Mean Delay
10	1.01
100	1.06
200	1.13
300	1.21
400	1.33
500	1.5
600	1.75
700	2.17
800	3
900	5.5
950	10.51
970	17.2
990	51.11



As we can see from the simulation graph, the mean delay curve starts from the y-axis intercept at 1.00 when x=0 and approaches the infinity when the arrival rate approaches the x-asymptote 1000.

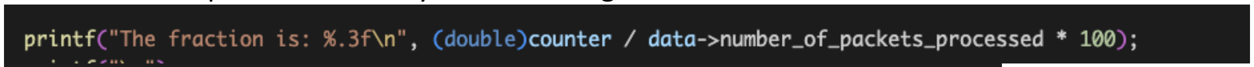
3.

By adding a counter to count how many packets have the delay exceeds 40msec while manually adjust the packet arrival rate and simulation with 10 different random seeds, I find out to meet the 2% packet delay requirement, when the arrival rate is at 133 packets/second, the average 40msec delay fraction is 1.969% which meets the question requirement.

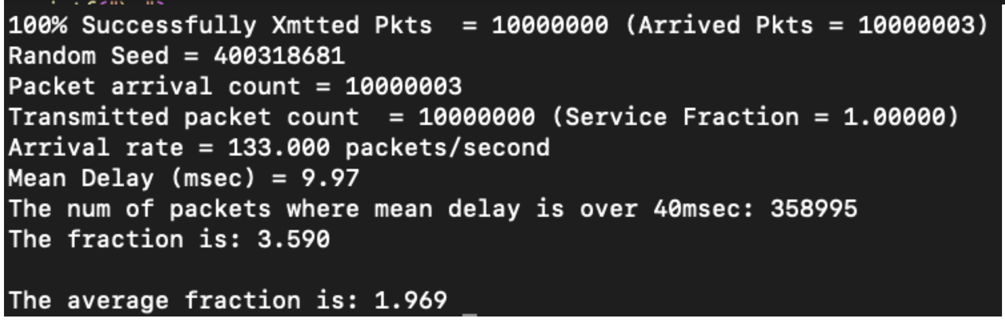


```
double delay = simulation_run_get_time(simulation_run) -
                this_packet->arrive_time;
if (delay> 0.04)
{
  counter++;
}
```

And we can simple find the delay fraction using the below code.



```
printf("The fraction is: %.3f\n", (double)counter / data->number_of_packets_processed * 100);
```

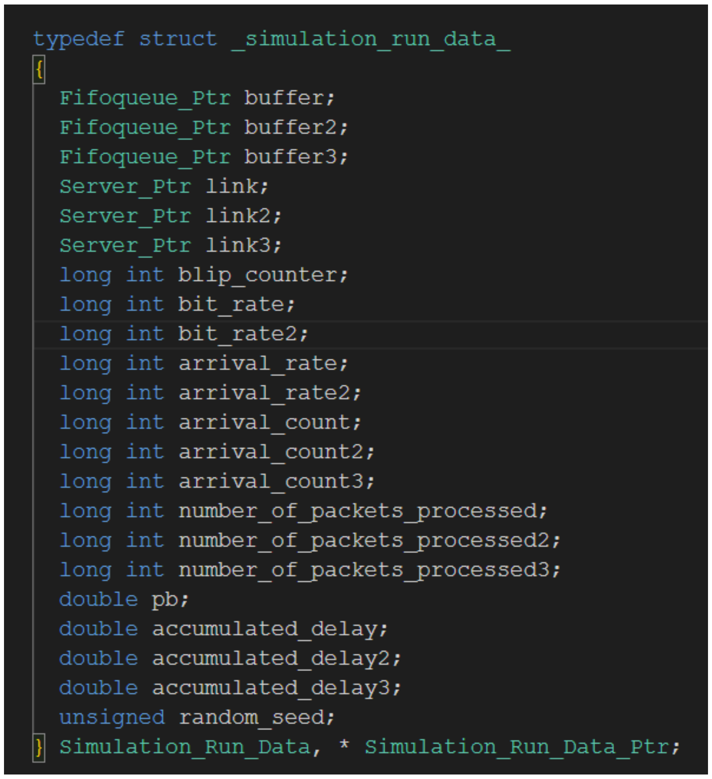


```
100% Successfully Xmtted Pkts  = 10000000 (Arrived Pkts = 10000003)
Random Seed = 400318681
Packet arrival count = 10000003
Transmitted packet count  = 10000000 (Service Fraction = 1.00000)
Arrival rate = 133.000 packets/second
Mean Delay (msec) = 9.97
The num of packets where mean delay is over 40msec: 358995
The fraction is: 3.590

The average fraction is: 1.969 _
```

The random seed list is {333333, 444444, 555555, 666666, 777777, 888888, 1000000, 2000000, 3000000, 400318681};

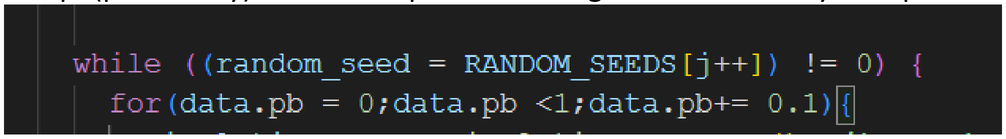
4.

To perform a three packet switches, I have added two buffers and two links and other necessary members in the data struct as shown below.



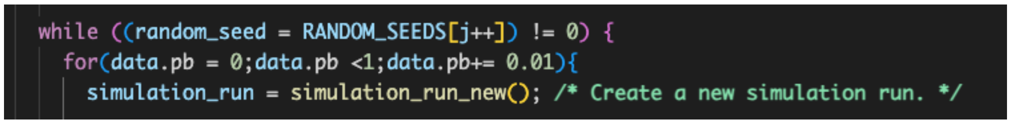
```
typedef struct _simulation_run_data_
{
  Fifoqueue_Ptr buffer;
  Fifoqueue_Ptr buffer2;
  Fifoqueue_Ptr buffer3;
  Server_Ptr link;
  Server_Ptr link2;
  Server_Ptr link3;
  long int blip_counter;
  long int bit_rate;
  long int bit_rate2;
  long int arrival_rate;
  long int arrival_rate2;
  long int arrival_count;
  long int arrival_count2;
  long int arrival_count3;
  long int number_of_packets_processed;
  long int number_of_packets_processed2;
  long int number_of_packets_processed3;
  double pb;
  double accumulated_delay;
  double accumulated_delay2;
  double accumulated_delay3;
  unsigned random_seed;
} Simulation_Run_Data, * Simulation_Run_Data_Ptr;
```

The pb(probability) will be independent change to model this system performance.



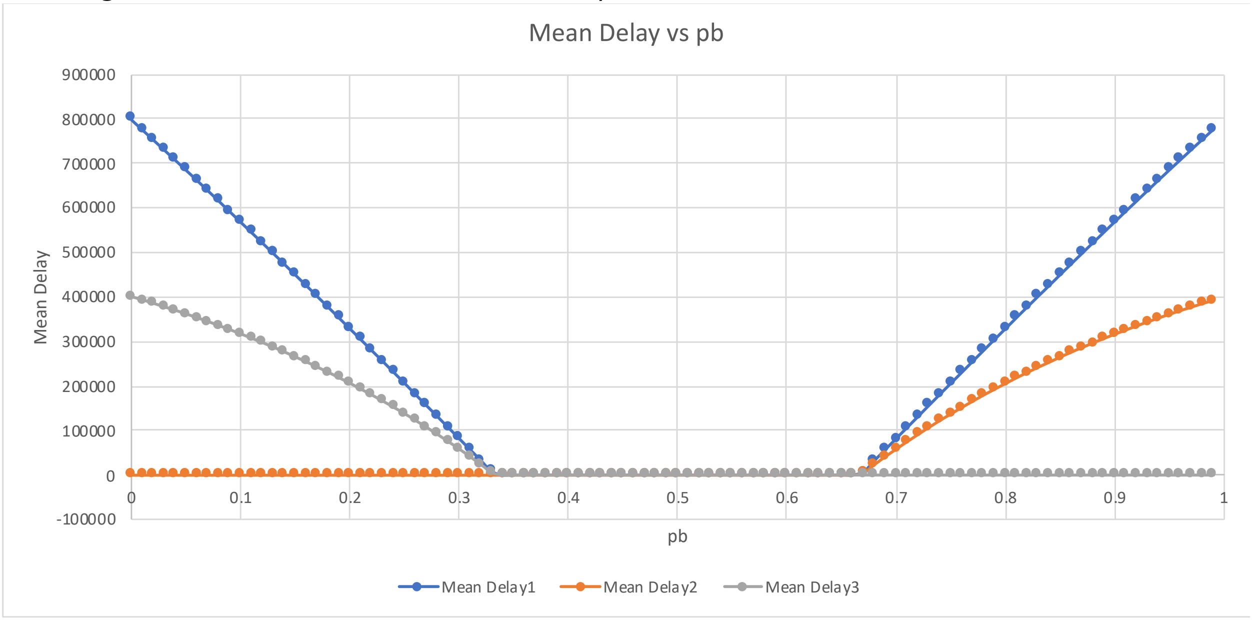
```
while ((random_seed = RANDOM_SEEDS[j++]) != 0) {
  for(data.pb = 0;data.pb <1;data.pb+= 0.1){
```

Firstly, I performed 100 simulations runs where increment the pb by 1 to see how this model performed. Since the data density is too big, only one random seed(my student number) is used in this case.

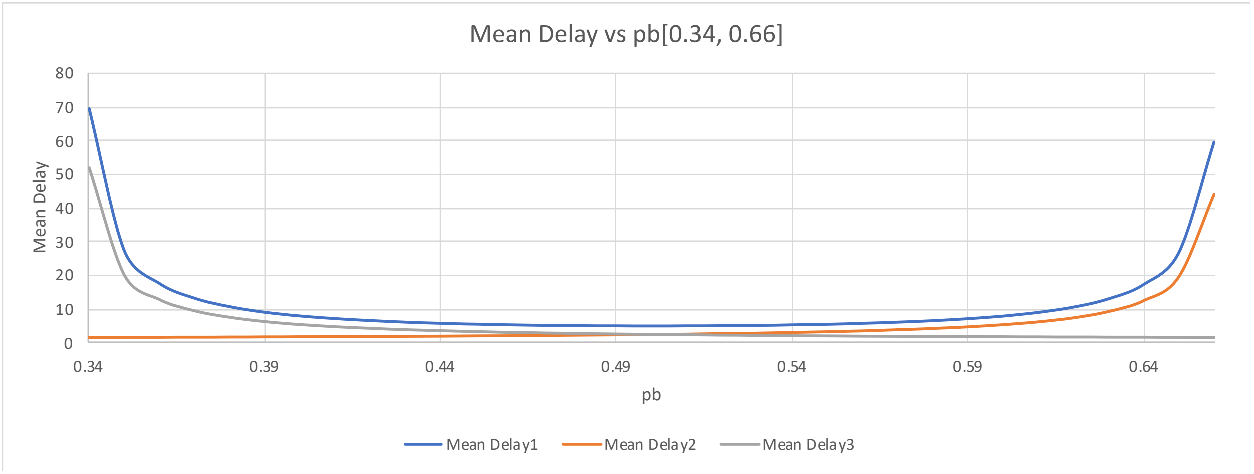


```
while ((random_seed = RANDOM_SEEDS[j++]) != 0) {
  for(data.pb = 0;data.pb <1;data.pb+= 0.01){
   simulation_run = simulation_run_new(); /* Create a new simulation run. */
```

After it generates the data, we can obtain the plot below.

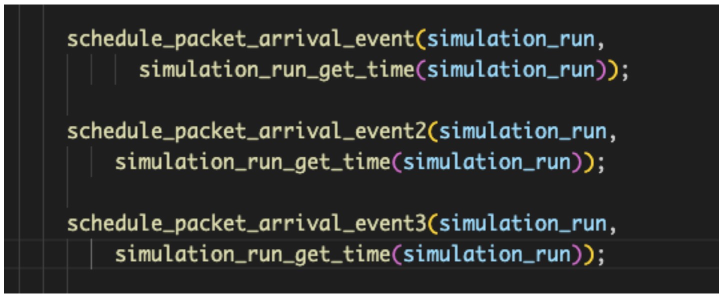


To explore further, we can zoom in the plot within the range [0.34, 0.66] of its x values since the further value will increase significantly of the mean delay as we obtained from the data.



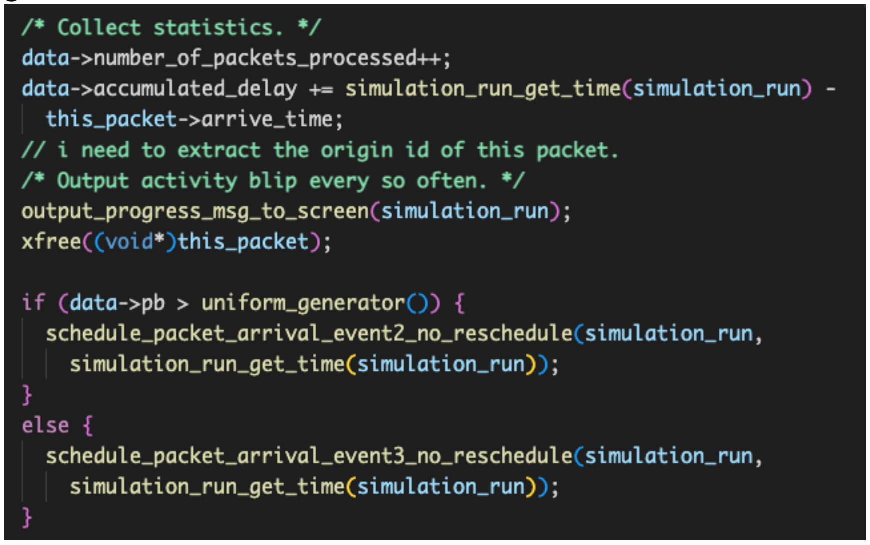
From this plot and data results, we can see the mean delay at switch 1 is always larger than the mean delay at switch 2 and 3 because the data packets will experience two delays from the switch 1 and either switch 2 or 3. And when the pb is equal to 0.5, the three switches will all experience the lowest delay. That is because now the packets from switch 1 can be equally distribute to the switch 2 or switch 3 so it will use the maximum resources to speed the transmission process.

What I have done with the code is first added three schedule events, the event2 and event3 will be just a copy of the schedule events but with different arrival rates. The arrival rate of each event is defined in the Data struct and initialized at the first.



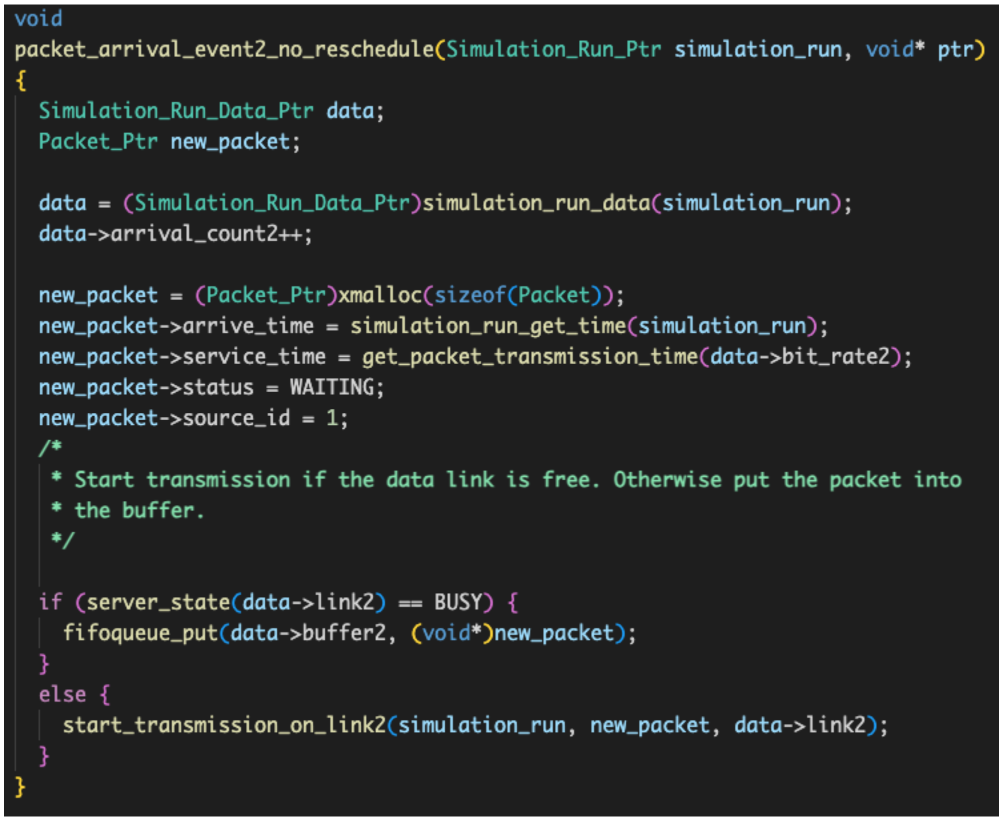
```
    schedule_packet_arrival_event(simulation_run,
          simulation_run_get_time(simulation_run));

    schedule_packet_arrival_event2(simulation_run,
        simulation_run_get_time(simulation_run));

    schedule_packet_arrival_event3(simulation_run,
        simulation_run_get_time(simulation_run));
```

And for every packet coming off the switch1, we also need to add the transmission to either the switch 2 or 3, the following code will compare the predefined pb with the random number generator and choose which switch should arrive.



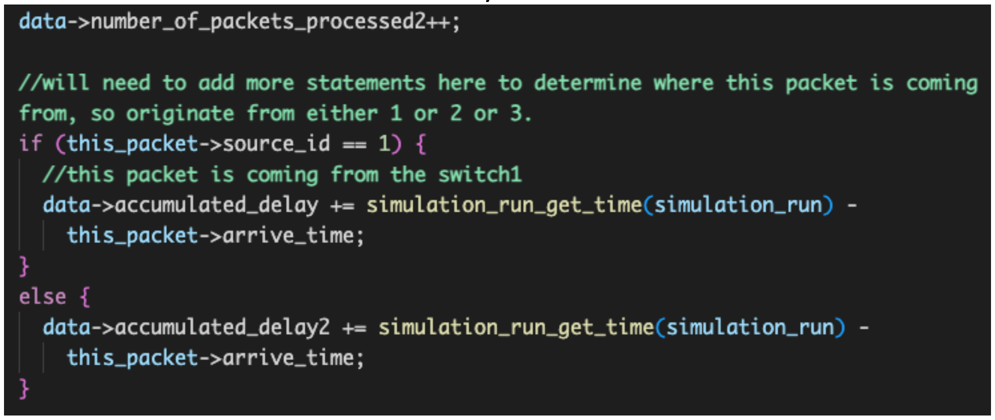
```
/* Collect statistics. */
data->number_of_packets_processed++;
data->accumulated_delay += simulation_run_get_time(simulation_run) -
  this_packet->arrive_time;
// i need to extract the origin id of this packet.
/* Output activity blip every so often. */
output_progress_msg_to_screen(simulation_run);
xfree((void*)this_packet);

if (data->pb > uniform_generator()) {
  schedule_packet_arrival_event2_no_reschedule(simulation_run,
    simulation_run_get_time(simulation_run));
}
else {
  schedule_packet_arrival_event3_no_reschedule(simulation_run,
    simulation_run_get_time(simulation_run));
}
```

I define the next packet transmission below as a function as shown above. What it does is to trigger the no reschedule packet arrival event which it won’t schedule the next packet in that function because the packet is coming from the switch 1 and it will be scheduled in the switch1 packet arrival event.



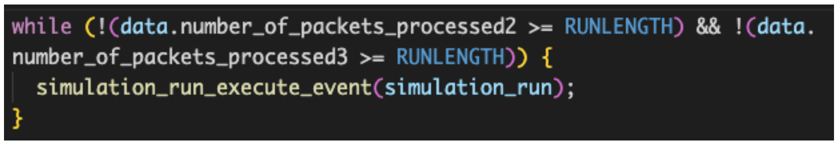
```c
void
packet_arrival_event2_no_reschedule(Simulation_Run_Ptr simulation_run, void* ptr)
{
  Simulation_Run_Data_Ptr data;
  Packet_Ptr new_packet;

  data = (Simulation_Run_Data_Ptr)simulation_run_data(simulation_run);
  data->arrival_count2++;

  new_packet = (Packet_Ptr)xmalloc(sizeof(Packet));
  new_packet->arrive_time = simulation_run_get_time(simulation_run);
  new_packet->service_time = get_packet_transmission_time(data->bit_rate2);
  new_packet->status = WAITING;
  new_packet->source_id = 1;
  /*
   * Start transmission if the data link is free. Otherwise put the packet into
   * the buffer.
   */

  if (server_state(data->link2) == BUSY) {
    fifoqueue_put(data->buffer2, (void*)new_packet);
  }
  else {
    start_transmission_on_link2(simulation_run, new_packet, data->link2);
  }
}
```

It will also assign the source id equal to 1 so at the end it will know where this packer is origin from and added the accumulate delay to the switch 1 as we seen below.



```c
data->number_of_packets_processed2++;

//will need to add more statements here to determine where this packet is coming
from, so originate from either 1 or 2 or 3.
if (this_packet->source_id == 1) {
  //this packet is coming from the switch1
  data->accumulated_delay += simulation_run_get_time(simulation_run) -
    this_packet->arrive_time;
}
else {
  data->accumulated_delay2 += simulation_run_get_time(simulation_run) -
    this_packet->arrive_time;
}
```

And we also limit the Run length equal to 10e6, as either of the packet finished first, it will stop the simulation.

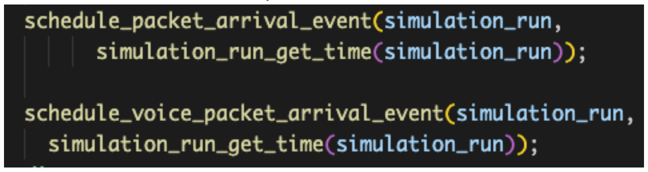


```c
while (!(data.number_of_packets_processed2 >= RUNLENGTH) && !(data.
number_of_packets_processed3 >= RUNLENGTH)) {
  simulation_run_execute_event(simulation_run);
}
```

The full implementation can be found in the attached zip file.

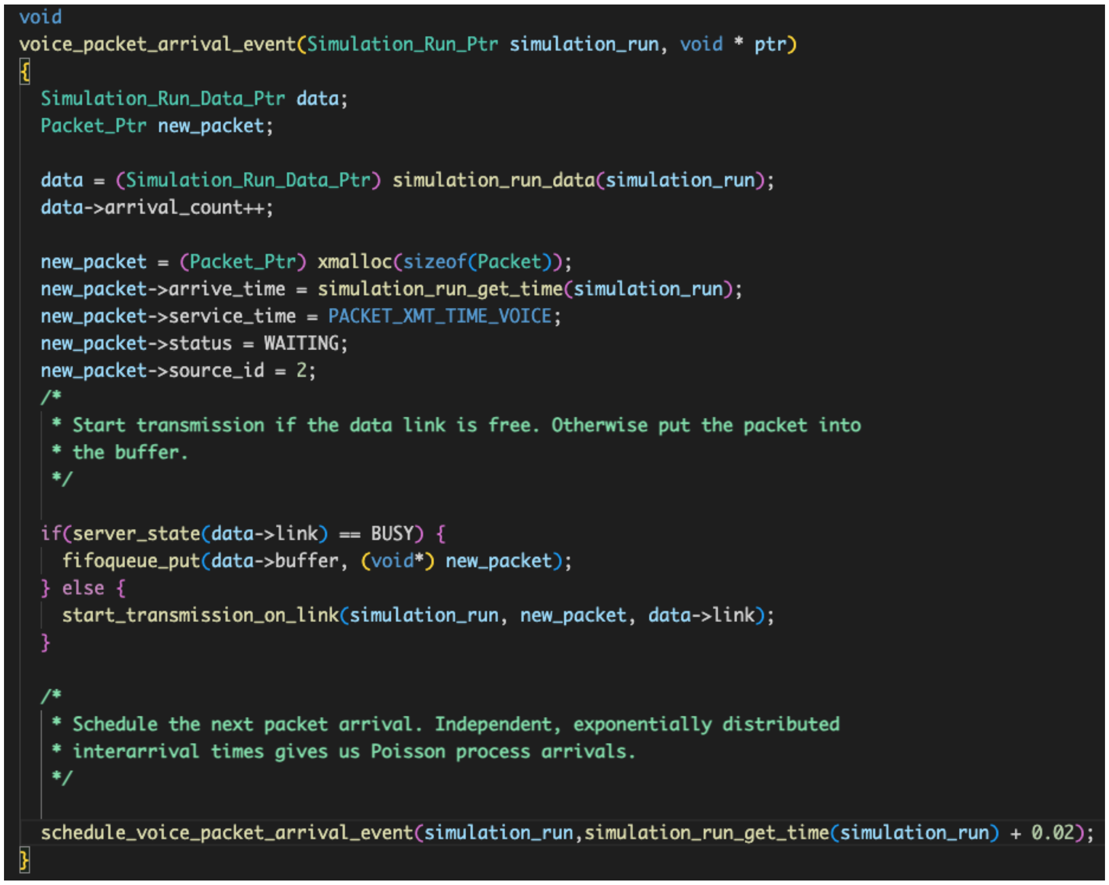
5.

To allow the switching link to serve both data and voice stream packets in the same buffer and served in FCFS order, we can add another voice arrival event on behalf of the part 2.



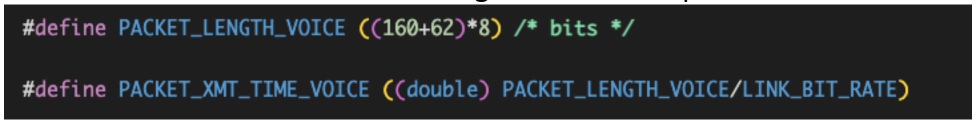
```c
schedule_packet_arrival_event(simulation_run,
      simulation_run_get_time(simulation_run));

schedule_voice_packet_arrival_event(simulation_run,
  simulation_run_get_time(simulation_run));
```

We can make a copy of the first data schedule event for the voice packet events as shown below.



```c
void
voice_packet_arrival_event(Simulation_Run_Ptr simulation_run, void * ptr)
{
  Simulation_Run_Data_Ptr data;
  Packet_Ptr new_packet;

  data = (Simulation_Run_Data_Ptr) simulation_run_data(simulation_run);
  data->arrival_count++;

  new_packet = (Packet_Ptr) xmalloc(sizeof(Packet));
  new_packet->arrive_time = simulation_run_get_time(simulation_run);
  new_packet->service_time = PACKET_XMT_TIME_VOICE;
  new_packet->status = WAITING;
  new_packet->source_id = 2;
  /*
   * Start transmission if the data link is free. Otherwise put the packet into
   * the buffer.
   */

  if(server_state(data->link) == BUSY) {
    fifoqueue_put(data->buffer, (void*) new_packet);
  } else {
    start_transmission_on_link(simulation_run, new_packet, data->link);
  }

  /*
   * Schedule the next packet arrival. Independent, exponentially distributed
   * interarrival times gives us Poisson process arrivals.
   */

  schedule_voice_packet_arrival_event(simulation_run,simulation_run_get_time(simulation_run) + 0.02);
}
```

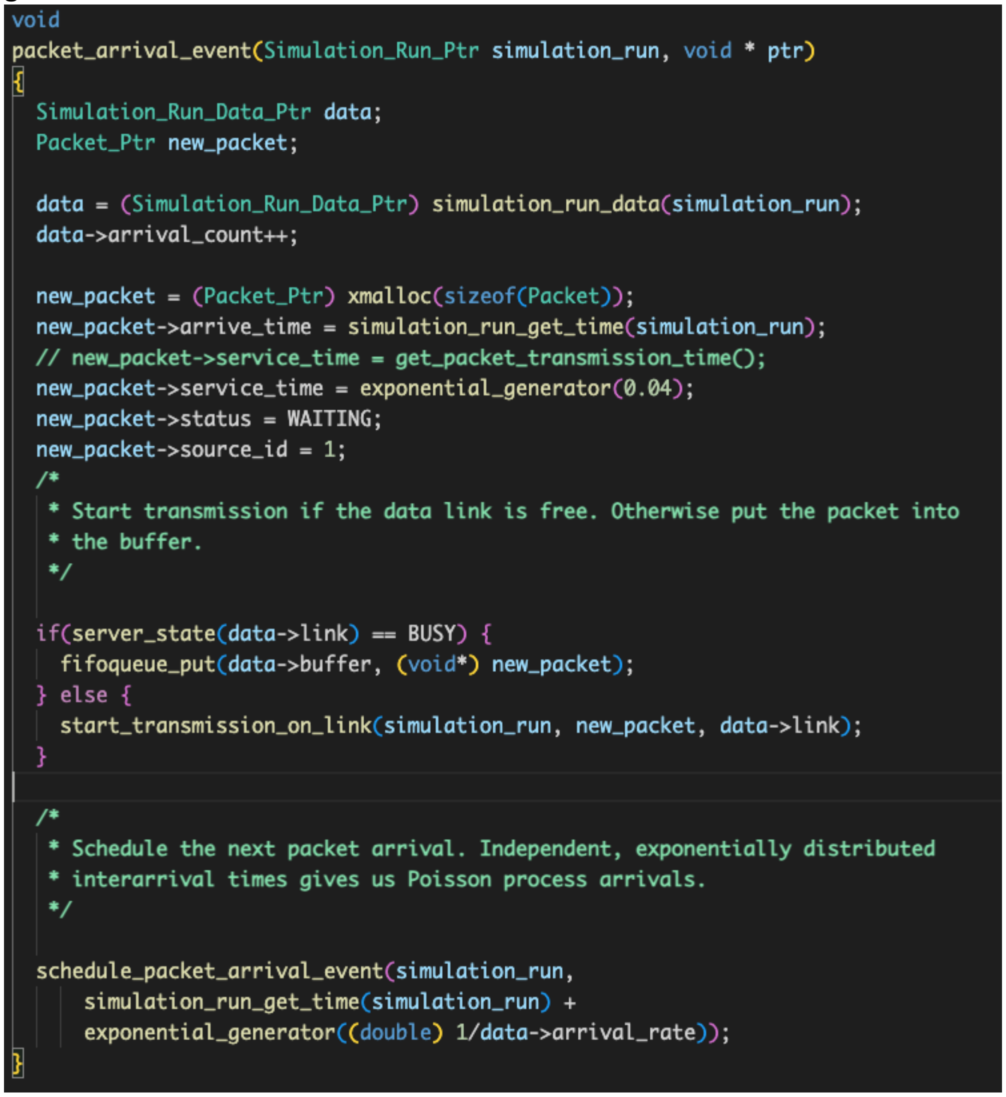
And in order to calculate the mean delay, we need to know where the packet is origin from at the end of the transmission, so we assign a source id of 2 here to know it is coming from the voice and the source id of 1 is coming from the data packets.



```c
#define PACKET_LENGTH_VOICE ((160+62)*8) /* bits */

#define PACKET_XMT_TIME_VOICE ((double) PACKET_LENGTH_VOICE/LINK_BIT_RATE)
```

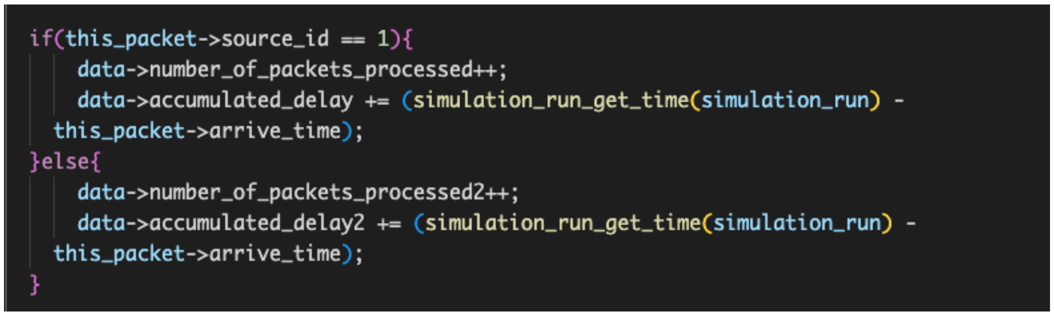
And we also found the G.711 (64 Kbps) voice codec has a packet size of 160bytes raw data and in this case we also need to add 62bytes of header here and we can calculate the service time by divided by the link bit rate which is 1Mbps here. And the packets arrived with fixed inter-packet arrival times equal to tv =20ms, so we just need schedule the next packet arrival event by adding 0.02 plus the current time in the code.

Also for the data packet arrival event, we need to set the service time to an exponential generate time of 40ms as shown below.



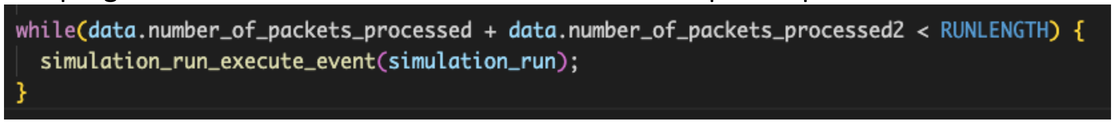
```c
void
packet_arrival_event(Simulation_Run_Ptr simulation_run, void * ptr)
{
  Simulation_Run_Data_Ptr data;
  Packet_Ptr new_packet;

  data = (Simulation_Run_Data_Ptr) simulation_run_data(simulation_run);
  data->arrival_count++;

  new_packet = (Packet_Ptr) xmalloc(sizeof(Packet));
  new_packet->arrive_time = simulation_run_get_time(simulation_run);
  // new_packet->service_time = get_packet_transmission_time();
  new_packet->service_time = exponential_generator(0.04);
  new_packet->status = WAITING;
  new_packet->source_id = 1;
  /*
   * Start transmission if the data link is free. Otherwise put the packet into
   * the buffer.
   */

  if(server_state(data->link) == BUSY) {
    fifoqueue_put(data->buffer, (void*) new_packet);
  } else {
    start_transmission_on_link(simulation_run, new_packet, data->link);
  }

  /*
   * Schedule the next packet arrival. Independent, exponentially distributed
   * interarrival times gives us Poisson process arrivals.
   */

  schedule_packet_arrival_event(simulation_run,
      simulation_run_get_time(simulation_run) +
      exponential_generator((double) 1/data->arrival_rate));
}
```

Therefore, in the packet_transmission.c file, for the end packet transmission event, we can collect all the data.



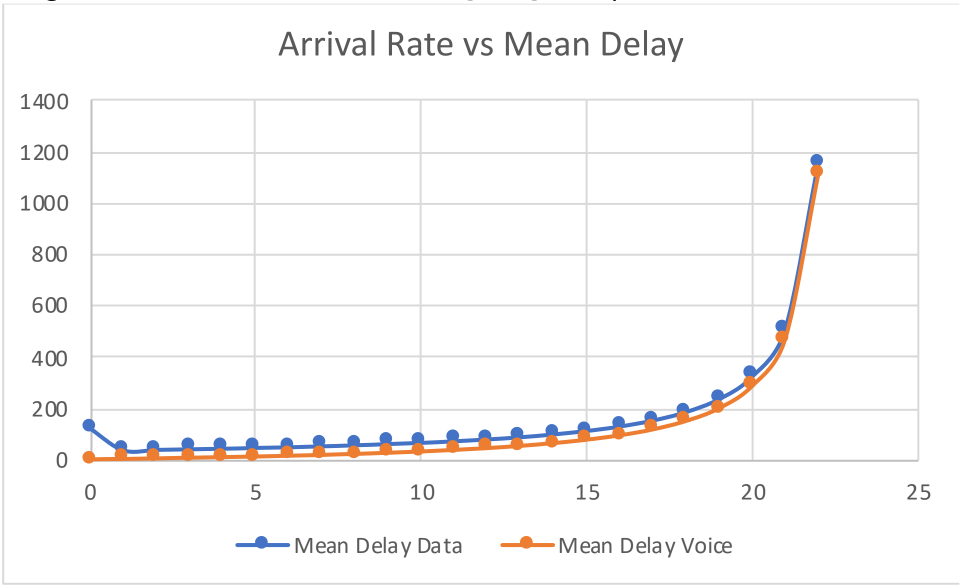
```c
  if(this_packet->source_id == 1){
      data->number_of_packets_processed++;
      data->accumulated_delay += (simulation_run_get_time(simulation_run) -
    this_packet->arrive_time);
  }else{
      data->number_of_packets_processed2++;
      data->accumulated_delay2 += (simulation_run_get_time(simulation_run) -
    this_packet->arrive_time);
  }
```

The program will end when the sum of voice and data packet processed reach the run length.



```c
while(data.number_of_packets_processed + data.number_of_packets_processed2 < RUNLENGTH) {
  simulation_run_execute_event(simulation_run);
}
```

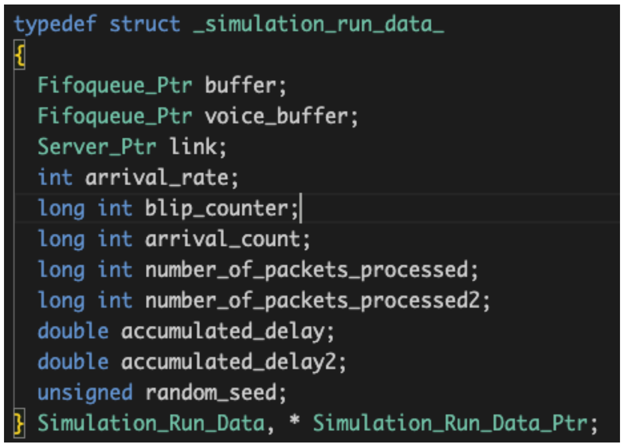
By playing around with the arrival rate, I found when it is below 30 it will generate a low mean delay and I increment the arrival rate by 1 for 0 to 30 to get the results and found an acceptable range of the arrival rate is between [1,22]. The plot is shown below.



As we can see from the plot, the mean delay data is sharing the same trendline as the mean delay voice and is slight larger when it is below 20 because the packet arrival rate will be lower than the voice packet arrival since it is a Poisson process based on the current arrival rate.

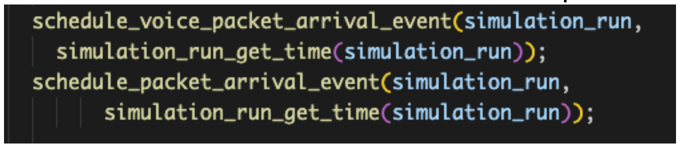
6.

In part 6, we want to give priority over data packets. We can achieve that by creating the separate voice and data packet queues and also the separate arrival and departure events. As usual, we can define a data buffer and voice buffer in our data struct and initialize them in the main.c file.



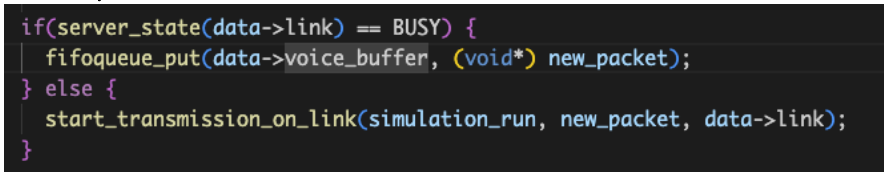
```
typedef struct _simulation_run_data_
{
  Fifoqueue_Ptr buffer;
  Fifoqueue_Ptr voice_buffer;
  Server_Ptr link;
  int arrival_rate;
  long int blip_counter;
  long int arrival_count;
  long int number_of_packets_processed;
  long int number_of_packets_processed2;
  double accumulated_delay;
  double accumulated_delay2;
  unsigned random_seed;
} Simulation_Run_Data, * Simulation_Run_Data_Ptr;
```

And we need to first schedule these two separate events for the clock time when it is zero.



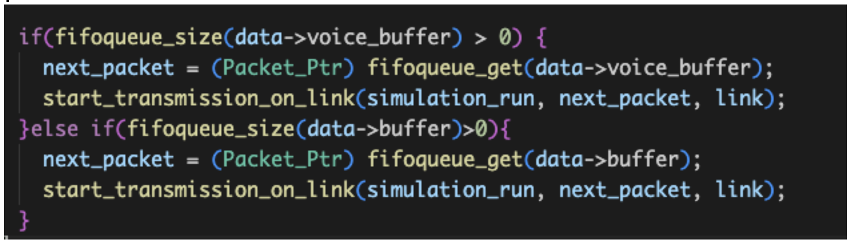
```
  schedule_voice_packet_arrival_event(simulation_run,
    simulation_run_get_time(simulation_run));
  schedule_packet_arrival_event(simulation_run,
        simulation_run_get_time(simulation_run));
```

The packet arrival event logic should be similar as part6 except this time we will put the packet on a separate buffer as shown below.



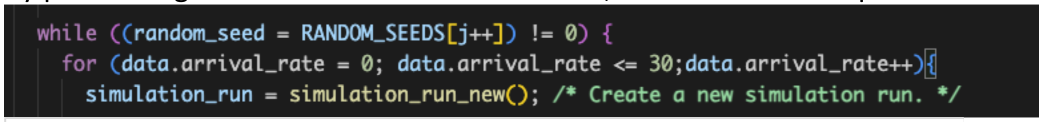
```
if(server_state(data->link) == BUSY) {
  fifoqueue_put(data->voice_buffer, (void*) new_packet);
} else {
  start_transmission_on_link(simulation_run, new_packet, data->link);
}
```

And also at the end of each transmission, we will add a logic check to see if the voice buffer size is zero, if it is not, it will start the transmission of the voice packet first than transmit the data packet when it is not zero.

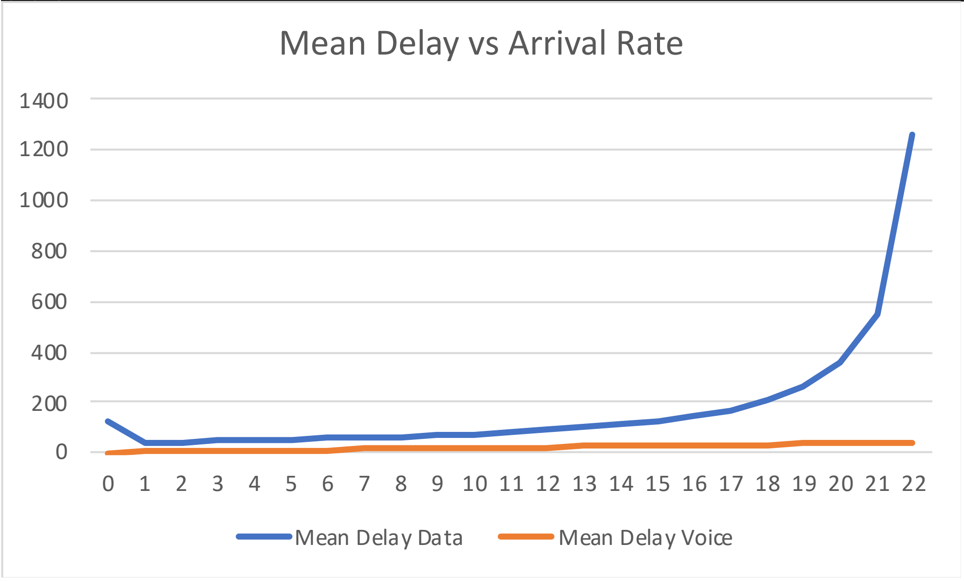


```
if(fifoqueue_size(data->voice_buffer) > 0) {
  next_packet = (Packet_Ptr) fifoqueue_get(data->voice_buffer);
  start_transmission_on_link(simulation_run, next_packet, link);
}else if(fifoqueue_size(data->buffer)>0){
  next_packet = (Packet_Ptr) fifoqueue_get(data->buffer);
  start_transmission_on_link(simulation_run, next_packet, link);
}
```

By performing the same arrival rate increment, we can obtain the plot below.



```
 while ((random_seed = RANDOM_SEEDS[j++]) != 0) {
   for (data.arrival_rate = 0; data.arrival_rate <= 30;data.arrival_rate++){
     simulation_run = simulation_run_new(); /* Create a new simulation run. */
```



As we can see from the plot, the mean delay of voice is showing almost a constant linear trendline while the data packets mean delay will increase when the arrival rate is going to increase which it satisfies the requirement to give the voice packet the priority to transmit.