




CS 213/293 Assignment Discrete Event Simulation Design Document

Mridul Ravi Jain
110040083



Have you submitted a working version of the simulator?

- Yes-Part 1 only



Source Code Files Information

| File Name | Brief Description |
|-------------|--|
| Events.h | Declares Event class and its methods |
| Events.cpp | Implements Event class methods |
| Request.h | Declares a Request class and its methods |
| Request.cpp | Implements Request class methods |



Source Code Files Information

File Name

Brief Description

QueueingSystem.h

Declares QueueingSystem class and its methods

QueueingSystem.cpp

Implements Queueing System class methods

main.cpp

Main file to start the simulation

Server.h

Declares Server class and implements its methods



Source Code Files Information

| File Name | Brief Description |
|----------------|---|
| Simulation.h | Declares Simulation class and its methods |
| Simulation.cpp | Implements Simulation class methods |
| declarations.h | File to hold random declarations |



Class Design

| Class Name | Brief Description |
|------------|--|
| Request | Describes a request in queueing system. Defined by an ID, its arrival time and its service time. |
| Event | Describes a simulation event. Defined by an ID, type, time and server number allotted-in case of departures. |



Class Design

| Class Name | Brief Description |
|------------|---|
| Server | Stores information about the server (ID, state-idle/busy, pointer to request in service) |
| Simulation | Consists of an event list, a simulation clock, a queueing system that is being simulated and some counters and metrics. Also, has different event handlers. |



Class Design

| Class Name | Brief Description |
|----------------|--|
| QueueingSystem | Captures all aspects of the queueing system being simulated. Defined by a buffer for queueing requests, a list of servers and the performance metrics associated with the system. |



Data Structures Used

| Purpose for which data structure is used | Data Structure Used | Whether Own Implementation or STL |
|--|---------------------|-----------------------------------|
| Future event list | Priority queue | STL |
| Buffer(waiting requests only) | Queue | STL |
| Servers list | List | STL |



Data Structures Used

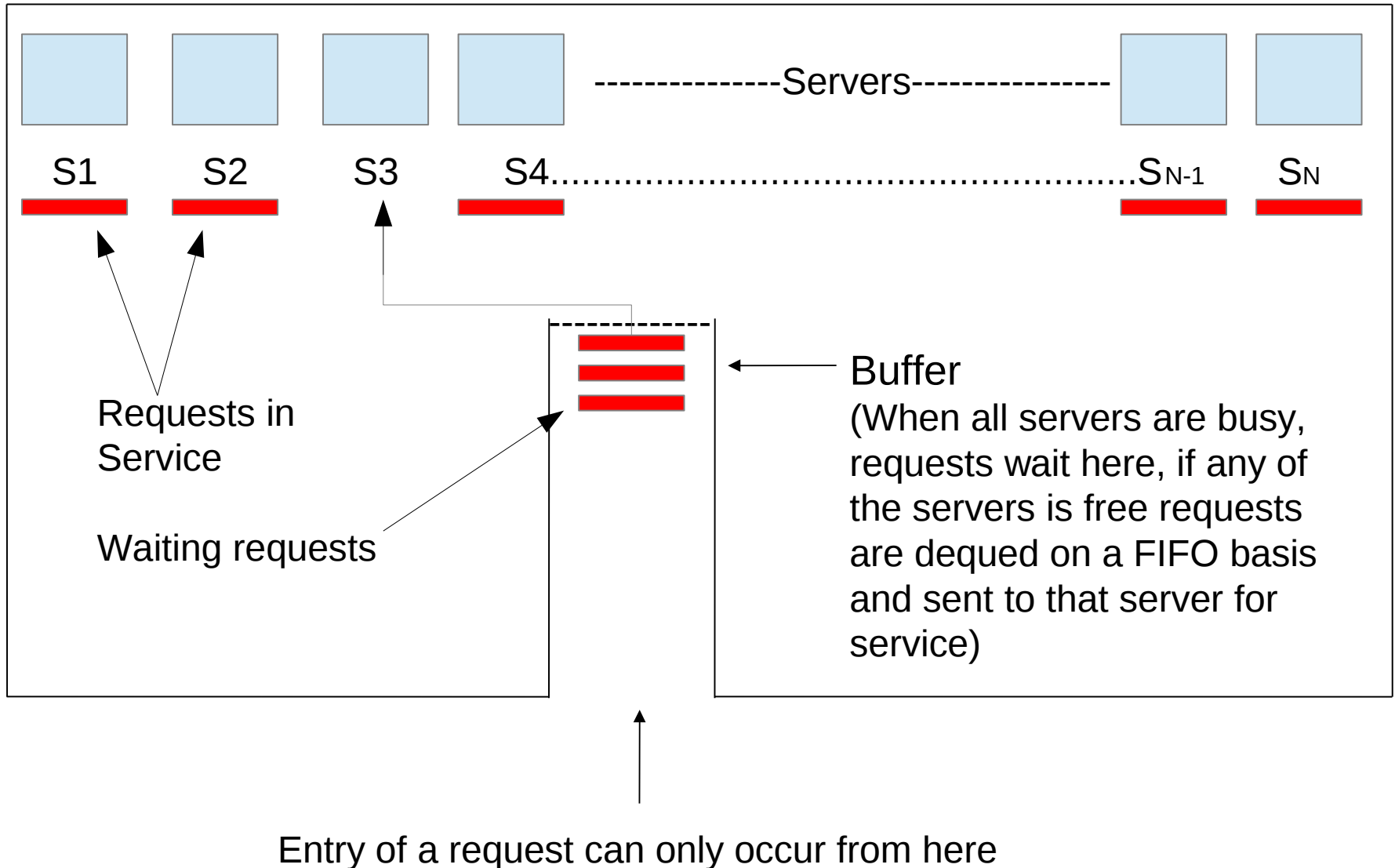
| Purpose for which data structure is used | Data Structure Used | Whether Own Implementation or STL |
|--|---------------------|-----------------------------------|
| Storing Waiting times | Vector | STL |
| Mapping Arrival Time with Service Time | Map | STL |




How are you representing multiple servers (or multiple requests in service) ?

- To extend single server to multiple server, I have created a list of servers.
- Now when we read the request, we push into buffer. It does not mean that we are making it to wait, but rather our buffer is the only path through which a request can enter into the system.
- Since all servers are “idle” initially, first request goes first server for service.

Queueing System






On a departure event how do you know which request to remove from the queueing system ?

On a departure event, only one of the following two cases can happen:

- Either there is no request waiting in the buffer.
In this case we set the server state (where departure occurred) to “idle” and (pointer to)request in service is set to NULL



On a departure event how do you know which request to remove from the queueing system ?

- Or there is a request waiting in the buffer.

In this case, we dequeue the buffer on a FIFO basis. And set the request dequeued is set as the request in service at the server (where departure occurred). So the server state remains “busy”.

At this moment we also schedule the departure of this request, at time = Simulation clock time + service time, and push this event into event list.



Any change in overall queueing system representation ?

- I created a server class.
- Now, since we are doing multiple server, there cannot be a single variable to indicate the busy server state for all servers.
- So now we have a list of servers, and each server will have an ID, a state determining variable and a pointer to request in service.
- Initially, all server states are set to “idle” and pointer to request in service = NULL.



Any change in the Simulation of the queueing system – i.e. event class?

- Yes. Now in event class we have another data member to denote the “server ID” where that event occurs.
- In the simulation, an arrival denotes arrival into the system and NOT an arrival at a server for service. So when scheduling an arrival event, server ID is given as 0(zero). There is actually no server with ID as 0. Server IDs starts from 1.
- So, 0 is just to avoid conflicting with server IDs while scanning through the server list.



Any change in the Simulation of the queueing system – e.g. event class?

- However, a departure always occurs at a server, and therefore while scheduling a departure event, server ID is given as the ID of server where the request is being processed (whose departure we are scheduling).
- So, whenever departure occurs we always know at which server departure is occurring.



How do you handle the arrival of a request?

- We first create a request object from the event information
- Then we check if buffer is not full.
- If buffer is full, request is dropped and no change in system state occurs.
- If not buffer is not full, push the request into buffer (again note that buffer is our only way of entry).
- Now we scan through the server list to look for an idle server.



How do you handle the arrival of a request?

- If idle server found, dequeue from the buffer and put it into service at this idle server.
- Else, no dequeuing is done and there is no change in system state except the queuing of a request .

Complexity analysis for multiple server extension

Number servers: s , Wait buffer size: w , Size of event list: N

Number of requests serviced : n

- Running time analysis of main Simulation loop i.e.

run(QueueingSystem q) :

- while(eventlist not empty)- $O(N)$
- Pop next event- $O(\text{constant})$
- Sort waiting times : $O(n \log n)$
- Overall running time : **$O(N+n \log n)$**

Complexity analysis for multiple server extension

Number servers: s , Wait buffer size: w , Size of event list: N

- Running time analysis of Arrival Event Handling:
 - Creating a request object from event information- $O(\log N)$
 - Checking if buffer is not full- $O(\text{constant})$
 - Enqueue request into the buffer- $O(\text{constant})$
 - Scan through server list- $O(s)$
 - Update metrics- $O(\text{constant})$
 - Schedule departure of this request(if idle server found)- $O(\text{constant})$
 - Schedule next arrival- $O(\text{constant})$
 - Therefore, overall running time : **$O(s + \log N)$**

Complexity analysis for multiple server extension

Number servers: s , Wait buffer size: w , Size of event list: N

■ Running time analysis of Departure Event Handling:


- If buffer is not empty-
 - Dequeue from the buffer(if not empty)- $O(\text{constant})$
 - Scan through the server list to find where departure occurred- $O(s)$
 - Updating system metrics and server metrics- $O(\text{constant})$
 - Scheduling next departure- $O(\text{constant})$
- If buffer is empty-
 - Scan through the server list to find where departure occurred- $O(s)$
 - Update server metrics- $O(\text{constant})$

Overall running time of departure event handling : **$O(s)$**



Extendability analysis of your implementation

- What changes are required to extend to heterogeneous servers?
 - Till now, what I have done is taken the processing size of an instruction and since we were dealing with constant speed of a server (in Part 1), I divided size by speed to get the service time when reading input from stdin. And the map was arrival time->service time.

- 
- But in case of heterogeneous servers, we can not construct such a map.
 - So change the map to `<arrival time, processing_size>` map in request class.
 - Server class will have an additional component as speed.
 - No change in departure event handler, QueueingSystem, Event class etc.
 - Only change required is scheduling of departure event which will now be scheduled at $\text{time} = \text{simTime} + (\text{processing_size} / \text{speed})$, where speed is of the server where departure occurs.



Extendability analysis of your implementation

- **Server metrics:** My code is extend-able in terms of calculating the server metrics such as cumulative server busy time, average utilization of a server, number of requests it serviced, etc. due to having a separate server class.
- **Timeouts:** code can be extended to implement timeouts by creating another event handler for timeouts and using an $O(w)$ extra space to update the buffer
- Also, we can extend the code to find out the average job queue length.



Summarized tradeoffs

- My code is not very efficient in terms of space & time complexity, however it is extendable and readable (easy to understand).
- My code is an extendable version of the single server simulation code provided to us. I could not find out any other way to implement the discrete event simulation more elegantly.
- It can be used to implement timeouts, heterogeneous servers, and calculate many other metrics, though it will require more memory usage.

How are you representing the “probe”?

- Probes are being dealt as a third kind of event. Therefore a third event handler has been made.
- The first probe is scheduled at time P .
- Depending on queue length server speed may change.
- In `probe_event_handler`, we also schedule the next probe at $\text{time} = \text{simTime} + P$.
- Probe events are pushed into the event list only till the time when no arrivals are left and no departures are left in the event list.

How are you maintaining speeds for the servers?

- Map is now $\langle \text{arrivalTime}, \text{processing_size} \rangle$.
- Arrival event handler remains the same and departure of a request is scheduled at time $\text{simTime} + (\text{processing_size} / \text{speed})$.
- At a probe, whenever job queue length is seen above threshold J_{High} , speed is increased by S (upto max). Whenever job queue length is seen below threshold J_{low} , speed is decreased by S (upto min).
- After every probe if server speeds change, all departures are rescheduled.

Recalculation of departure time

- New departure time of a request =

simTime

+

$(\text{processing_size} - (\text{prev_server_speed} * (\text{simTime} - \text{processing_start_time})))$

new_server_speed

where, simTime is the time when probe occurs

This will be done for all requests whose service is currently going on.



Updating the departure time –data structure

- Now, since we have to update the events in a priority_queue, we can't use STL priority_queue.
- Thus we will make a new kind of data structure where we can reschedule the events.
- This will be a priority queue with locator capability.
- That is, there must be a way to directly access a specific item and change its priority value (timestamp, in our case).
- So we create a class indexedHeap.

class indexedHeap

- `idType insert(objectT o)`: insert into indexedHeap
- `DeleteMin`: find out item with lowest priority
- `changeKey(idType id, newKey)`: this will change the key value of the object whose ID in the heap was “id” to newKey.
- `increaseKey(position p, delta)`: increase key of object at index p in the heap vector by delta.
- `decreaseKey(position p, delta)`: decrease k of object at index p in the heap vector by delta.

class indexedHeap

- PercolateUp:
 - PercolateDown:
 - void makeEmpty():
 - bool isEmpty():
- Private data members:
- vector<Comparable> array : basic heap vector
 - int position[Nmax] : This keeps the position of the object whose ID is “i” at the index i

Updating the departure event-algorithm

- We calculate the new departure times according to previous formula.
- For any event that is potentially “reschedulable” we keep the ID that now the indexedHeap will return, with the concerned entity.
- Now, in the event list we know all the departure events, each event will have an ID.
- At a probe we call chageKey on the ID of departure events at each server with new departure times.
- So our priority queue gets updated.



Appendix: Class Details

P.S. : Ignore the comments in Part-1 submission

Server class

- Describes a generic server of a queueing system.
- int ID: each server has an ID(starts from 1)
- bool busy : denotes state of a server,
 - 0 if idle
 - 1 if busy
- Request* reqInService : pointer to request in service.
- bool isBusy() : returns state of a server

Request class

- int ID : ID to uniquely identify the request
- double arrivalTime : Time of arrival of request
- double serviceTime : Time required to process the request
- Various accessor functions
 - double getArrivalTime()
 - double getServiceTime()
 - int getID()

Event class

- Event class describes a simulation "event".
- int ID : Event is identified by an ID
- eventType type : type of event (arrival/departure)
- double time : time of occurrence of an event
- int sn : server where event occurs
 - 0 for Arrival Event
 - serverID for Departure Event
- Accessor methods :
 - double getTime()
 - eventType getType()
 - int getServerNo() ;

QueueingSystem class

- QueueingSystem is the class that captures all aspects of the queueing system being simulated. It is defined by:
- queue<Request> buffer: a buffer for queueing requests
- the performance metrics associated with the system
 - double avgResponseTime : average time from joining the queue till finishing service
 - double avgWaitingTime : average time from joining the queue till processing start time
 - double avgServiceTime : average time from processing start time till processing end time

QueueingSystem class

- Operations are to enqueue, dequeue, setting the metrics, and calculating buffer size:
 - void enqueue(Request r) : Enqueue request r in buffer
 - Request dequeue() :
 - Request nextReq() : Returns request at the front of the buffer does not remove the request , this is not the request in service but first waiting request
 - int Buffersize() : returns no. of waiting requests in the buffer.
 - double setAvgResponseTime(double Ravg) : setting the Average Response Time
 - double setAvgWaitingTime(double Wavg) : setting the Average Waiting Time
 - double setAvgServiceTime(double Savg) : setting the Average Service Time

Simulation class

- Main Simulation class. Comprises of an event list, a current simulation clock, a queueing system that is being simulated, server list and some counters and metrics.
 - **priority_queue<Event> eventList** : Eventlist implemented using STL priority queue. This will allows us to get the "most imminent event" in $O(1)$ time.
 - **double simTime** : Simulation clock.
 - **QueueingSystem sys** : The system being simulated.
 - **list<Server> ServerList** : servers list
- Performance metircs
- **int customersArrived** : How many customers have arrived for service
 - **int customersDeparted** : How many customers have finished service
 - **int customersDropped** : How many customers gone without being served because of full buffer

Simulation class

- **int Sn** : no of servers
- **int Buffer_Size** : maximum permissible buffer length
- **double S** : step size by which server speed is changed at a probe
- **double P** : probe interval

Min speed, Max speed that the server can operate at (in MIPS)

- **double min_speed, max_speed;**
- **int Jlow, Jhigh** : Job Queue size thresholds
- **int reqID** : request no.

for output to server.log and request.log

- **ofstream ReqOut**
- **ofstream SerOut**
- **Event Handlers:**
 - **void arrival_event_handler(Event e);**
 - **void departure_event_handler(Event e);**