

# EP2200 Course Project 2016 - Simulating a Spotify Server

Submitted by: Parth Singh (ITA) (930724-8535)

Course Coordinator: Viktoria Fodor

For simulation, a pseudo code is developed for the given scenario. Below is the basic pseudo code explaining the flow of requests in back-end server of Spotify. Furthermore, the detailed pseudo code explaining each step (each logic) in detail is given in Appendix.

## Pseudo Code

**Step1:** Create Poisson arrivals list till maximum time (System will run for a duration)

**Step2:** Map songs according to their probability (assign given popularities to every song)

**Step3:** Forward requests depending upon the probability (more popular song will be played first and more frequent)

**Step4:** Mapping a song request to the designated server (Each song is present in a specific server)

**Step5:** Serve requests in all the servers (All arriving requests in the buffer will be served one server after the other)

**Condition1:** No requests are currently being served (Both buffer and server is idle)

**Sub-condition:** When queue(FIFO) is empty, forward incoming request directly to server

**Sub-condition:** When queue is not empty, move first request from FIFO to Server

**Condition 2:** When requests are being processed (Checking whether there is some unfinished work left)

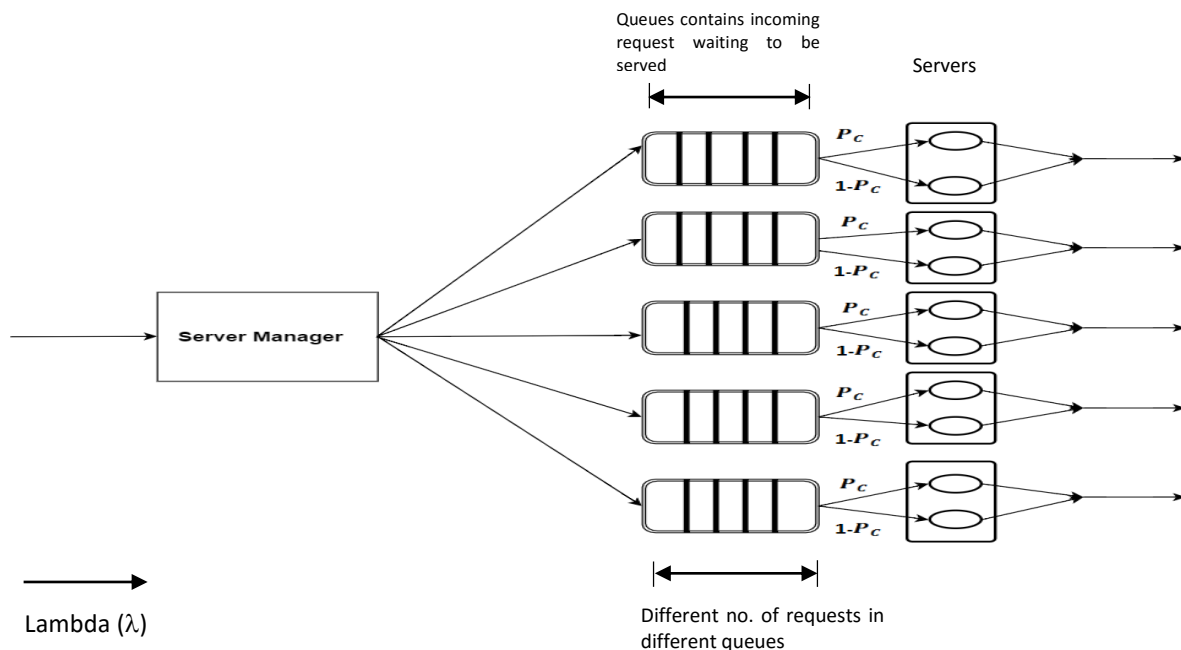
**Sub condition:** New requests arriving after current service finishes will directly go to server

**Sub condition:** New requests arriving before current service finishes, will be added to queue(FIFO)

## Performance Evaluation

**1. Queuing System:** The given model of Spotify back end server is a queuing system in which the requests arrive and wait in the queue and server serves the request in first come first serve manner.

In the given model, there are **5 servers** connected to their own queue (buffer capacity of **100 requests per queue** is given). In parallel to this, the arrival process is a **Poisson process (M)**. Furthermore, the server serves the request in two manners: either the request requires access to cache memory or in hard disk. The former is a **deterministic** (fixed time to serve request) and the latter is **exponential** in nature. This makes the system a **G general system** which is a combination of arrival requests of exponential and deterministic service time. Hence, the Kendall's notation for the proposed system is **M/G/1/K (K=100)**. There will be 5 systems of **M/G/1/K (K= 100 request)**.



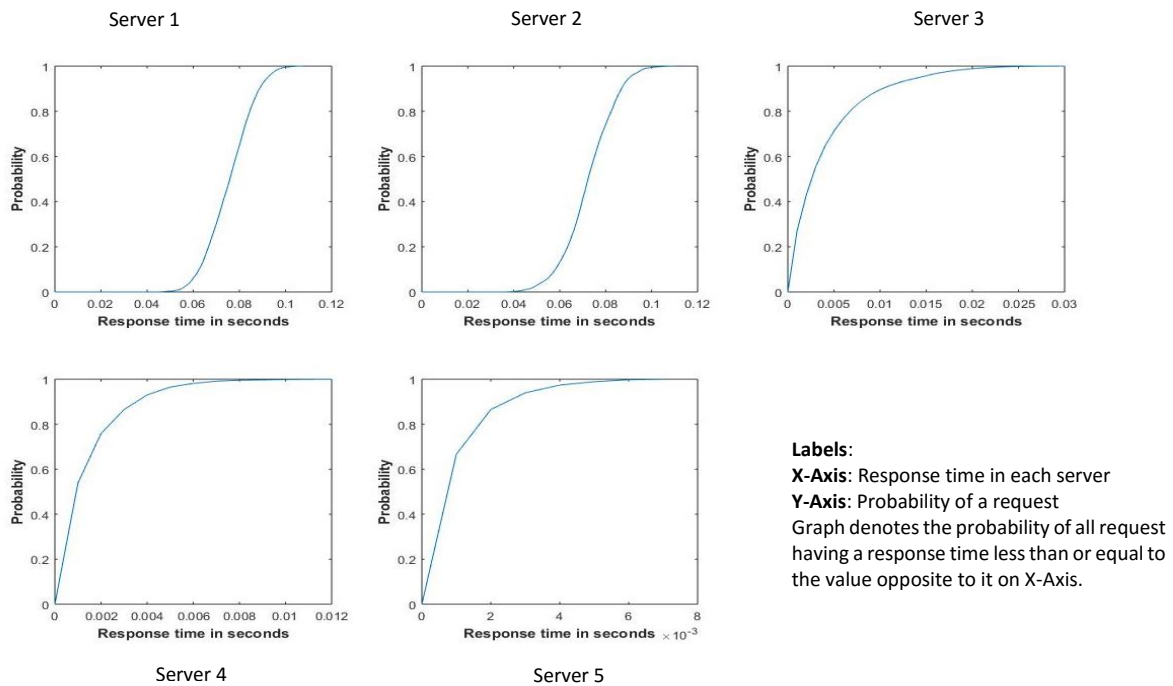
Above figure represent the block diagram of 5 different system of **M/G/1/K (K=100 request)**. The request arrives at Spotify **server manager**. The server manager directs the request of a song to the associated server. If the associated server is busy, the request goes to the **queue (FIFO)** of the designated server. However, the queue has a limited capacity of holding maximum of 100 request at one moment. Once the last request is served, another request moves from the

queue to server (FIFO manner). The request is either served in fixed time if the song is present in cache memory or the request is served in exponential service time (accessing the hard disk).

The arrival process is a Poisson process denoted by **lambda ( $\lambda$ )**. It is defined as the number of requests made per unit time ( **$\lambda=5000$  requests per second**). The arrived requests are sent immediately to their dedicated server where the request will be served.

In our model, the service time distribution depends on the service cache time ( $T_c^c$ ) which has a value of  $10^{-4}$  seconds and service hard-disk time ( $T_c^H$ ) which has a value of  $10^{-3}$  seconds. Thus, the service time distribution will be the combination of two different service times. In addition to this, the probability that the request will be served by cache memory is given by  $P_c$  which has a value of **0.25** while the probability that the request will be served by hard disk is given by  $1-P_c$  which has a value of **0.75**.

**2. Response Time:** After simulating the *pseudo code* on Matlab, various interesting results were witnessed. Response time in each server was calculated and following are the graphs of CDF for response time in each server in the original system:



In our *pseudo-code*, the average response time list is denoted by responseAverage. The list contains the response time of all the request made to a particular server. The average of all the response time to one server is calculated over this list.

Average response time in Server 1: **0.0762 seconds**

Average response time in Server 2: **0.0728 seconds**

Average response time in Server 3: **0.0036 seconds**





Average response time in Server 4: **0.0014 seconds**





Average response time in Server 5: **0.0011 seconds**

Average response time is gradually decreasing from server 1 to server 5

It is evident from the responseAverage list that the average response time in Server 1 is the most and the average response time in Server 5 is the least, i.e., Server 1 must be having more number of songs (or perhaps more number of popular songs) as compared to other servers.

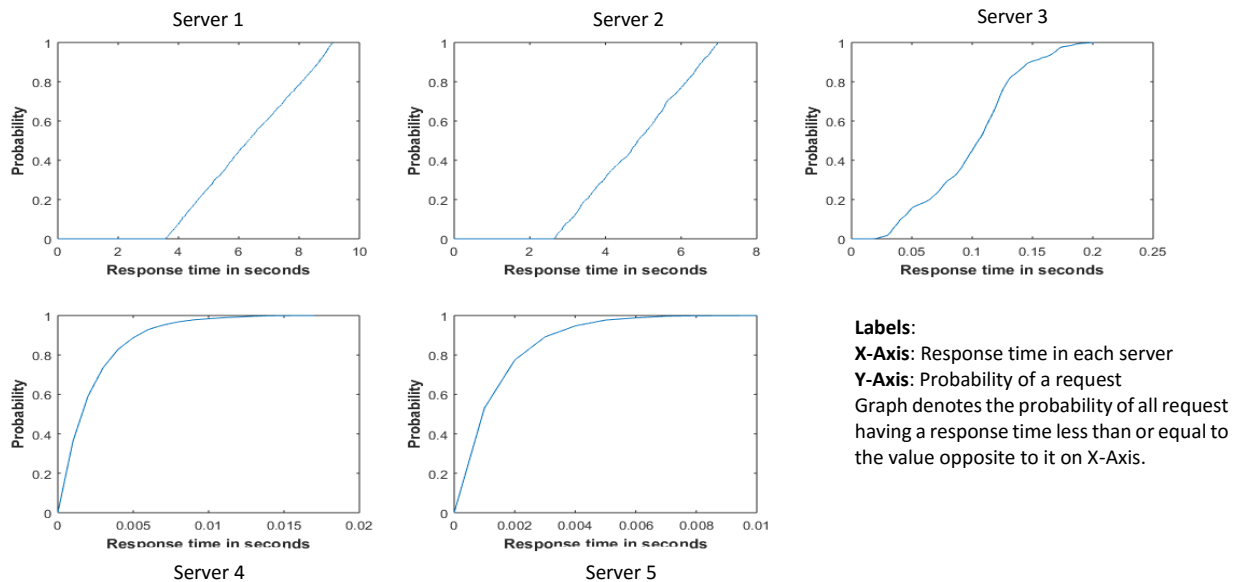
**3. Effect of Caching:** On gradually decreasing the value of cache probability  $P_c$  until 0, the probability of **request made to hard disk will increase**. Hence, the exponential service time will take place more prominently as compared to deterministic service time. And at  $P_c$  equals 0, there will be no requests that will go to cache memory, in turn, there will be no requests that will be served with fixed service time. On the other hand, all the requests will be served in hard disk hence all the requests will have exponential service time. Following table shows the effect of reducing  $P_c$  gradually to 0 in each server on response time. (**Note:** List shows Server 1 (Left most) to Server 5 (Right most).)



 responseAverage	[0.0762,0.0728,0.0036,0.0014,0.0011]	When $P_C = 0.25$
 responseAverage	[0.0813 0.0782 0.0055 0.0015 0.0011]	When $P_C = 0.20$
 responseAverage	[0.0891 0.0890 0.0140 0.0017 0.0012]	When $P_C = 0.10$
 responseAverage	[0.0997,0.0981,0.0593,0.0021,0.0013]	When $P_C = 0$

 droprequest	[5232,2340,0,0,0]	When $P_C = 0.25$
 droprequest	[6930,3274,0,0,0]	When $P_C = 0.20$
 droprequest	[7781,5054,0,0,0]	When $P_C = 0.10$
 droprequest	[9408,6061,361,0,0]	When $P_C = 0$

From the above tables, we can infer that the number of dropped packets increases as the service time taken to serve one request (also the response time increases) increases on reducing the value of  $P_C$ . As the value of  $P_C$  decreases, the efficiency of system as a whole decreases. Other noticeable fact is that majority of the requests goes to Server 1 and Server 2 while moderate number of request go to Server 3 and least number requests go to Server 4 and Server 5. This results in high request drop in queues of Server 1 and 2 but very few or less request drop cases in other servers as the queue never reaches the full capacity of 100 requests at one point.

When  $P_C$  equals 0 and there is no buffer constraint in each queue, i.e., infinite buffer capacity, there will be no packet drop in any of the server. Below graph is representing the CDF at  $P_C = 0$  and infinite buffer.



 responseAverage	[6.0904,4.6749,0.0933,0.0022,0.0014]
 droprequest	[0,0,0,0,0]

From the list above, it is visible that when  $P_C$  equals 0 along with infinite buffer, then the average response time is going high for each server as there are no packet drops so all the requests present in each queue are being served.

**4 Service Denial:** Performance of Back-end server (Original system) in terms of service denial= Percentage of client requests dropped by each server because buffer was full (B=100).

$$\% \text{ of dropped request} = \frac{\text{Number of dropped requests in each server}}{\text{Number of dropped requests} + \text{Number of served requests}} \times 100$$

Probability of service denial in Server 1: **27.46%**

Probability of service denial in Server 2: **14.49%**

Probability of service denial in Server 3: **0%**

Probability of service denial in Server 4: **0%**

Probability of service denial in Server 5: **0%**

**No request dropped**

As maximum number of requests flow to Server 1 and Server 2, which results in too many requests waiting in queue for first two servers. If any requests arrive when the buffer is full, then the request is dropped because buffer capacity

is limited. The incoming request will only be added to the last of the buffer if and only if the server serves one or more request from the queue (FIFO) and creates space for the incoming requests. On the other hand, in Server 3, 4 and 5 the request arrives in a manner that buffer (queue) never reaches its limit, as a result no requests are dropped and there is no service denial. Hence, it can be said that the performance of Server3, 4 and 5 are good but performance of Server 1 and 2 can be improved by either increasing the buffer capacity for these two servers or uniformly distributing the popular songs over all 5 servers.

**5. Load Balancing:** Variance means how far each member of the set is from the mean of all the members of the set. Variance at each server are as follow:

Variance at Server 1	Variance at Server 2	Variance at Server 3	Variance at Server 4	Variance at Server 5
<b>8.6286e-05</b>	<b>1.3256e-04</b>	<b>1.4268e-05</b>	<b>2.4672e-06</b>	<b>1.5884e-06</b>

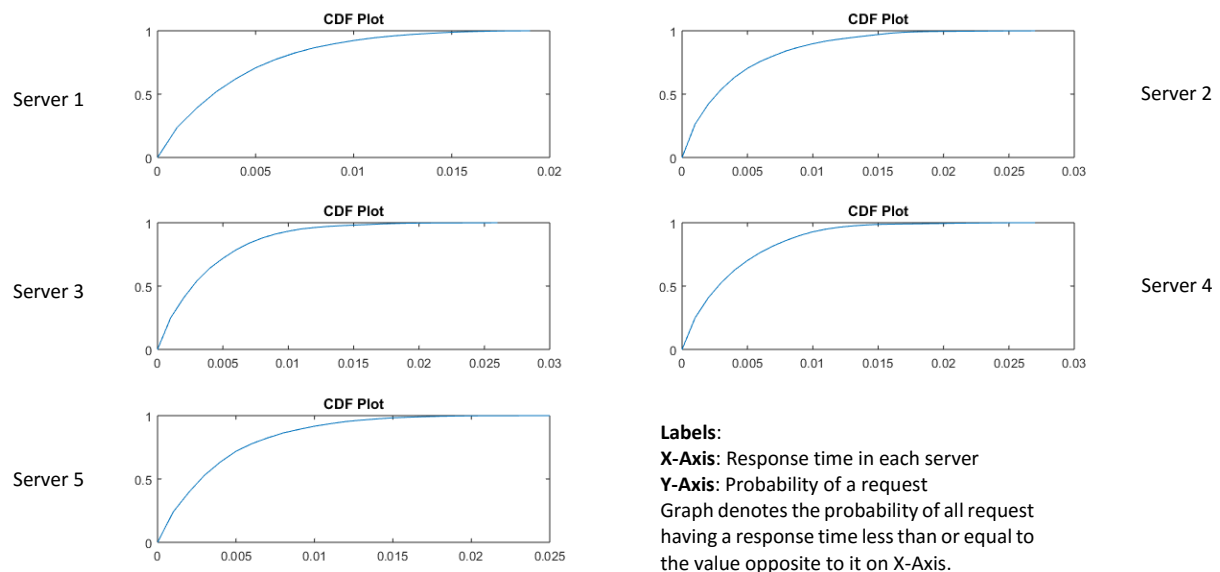
Sample variance in the system depends upon the summation of square of difference between the variance at each server and the mean variance of all the server. The division of the Summation by the total number of values give sample variance.

$(Mean_{var})$	$\sum (Mean_{var} - Var(i))$	$[\sum Mean_{var} - Var(i)] / \text{No\_of\_Server}$
<b>4.7434e-05</b>	<b>139.7972e-05</b>	<b>27.9594e-05</b>

Sample variance is equal to **27.9594e-05**. All the servers have different response time because of variation in number of requests going to each server. It might be possible that more popular songs are going to Server 1 and Server 2 and less popular songs are going to remaining servers.

Both of these cases can be rectified if every server serves songs that constitute to 20 percent of total probability of all the songs popularities. In simple words, Server 1 should have songs whose sum of probabilities of popularity is approximately 0.2. Similarly, Server 2, 3, 4 and 5 should have songs whose sum of probabilities of popularity is 0.2 for each server.

In this way, each server will serve approximately equal number of songs if probabilities of popularity of songs are uniformly distributed in each servers. On simulating the proposed allocation, the following CDF graph was observed.



droprequest	[0,0,0,0,0]
responseAverage	[0.0039,0.0041,0.0038,0.0039,0.0039]
variances	[1.2874e-05,1.7787e-05,1.3686e-05,1.4168e-05,1.4294e-05]
serverrequest	[12680,12677,12730,12670,12718]

From the above table and graph, it is clear that **requests drop reduced to 0** on changing the songs allocation. Secondly, average response time and variances are approximately same in each server. And lastly, the number of requests served by each server are also approximately equal. In the original system, there were request drops, different average response time for each server, different variances and varied number of requests were served by each server creating burden on few servers and wasting the resources of other servers by keeping it free for a period of time. All these improvement in results as compared to original system shows that the **new proposed system is better than the original one**.

## APPENDIX

### Programming Language: Matlab

- Initialize Lambda=5000, Cache\_time=  $10^{-4}$ , Service\_time= $10^3$ , No\_server=5, Buffer\_capacity=100, No\_of\_songs=1000, Probability\_cache=0.25, Maximum\_time=23, Warmup\_time=8, Songs\_alloc: Read 'song\_allocation.txt', Songs\_popularities: Read 'files\_popularities.txt', Time=0, Arrivals list, Assigned\_songs list, Request\_serving:0, Last\_request\_arrived:0, Finish\_servicetime:0, Temp\_time: 0, Response\_time list, Temp\_popular list, Servers list, End\_time list, Service\_times list, Dropped\_requests list, Served\_requests list
- While Time<Maximum\_time **// Step1: Create Poisson arrivals till maximum time**
- Time+=expnd(1/lambda)
- Add Time to Arrivals list
- For i: 2 to No\_of\_songs **//Step2: Map songs according to their probability**
- Temp\_popular+=Song\_popularities
- For i: 1 to length(Arrivals) **// Step3: Forward requests depending upon the probability**
- Temporary\_list: Copy all the request> rand
- Assigned\_songs(1)=Temporary\_list(1)
- For i: 1 to length (arrivals) **//Step4: Mapping a song request to the designated server**
- Servers(i)=Songs\_alloc(Assigned\_songs(i));
- For i:1 to No\_of\_servers **//Step5: to serve requests in all the servers**
- Arrivals\_in\_Server: find(Servers==i)
- Fifo list, Request\_serving:0, Last\_request\_arrived:0, Finish\_servicetime:0, Temp\_time:0, Response\_time list
- While Temp\_time< 0.9\*Maximum\_time
- If Temp\_time< Warmup\_time     Rollback End\_time, Service\_times, Dropped\_request, Served\_request to default values
- If Request\_serving==NULL **//Condition1: No requests are currently being served**
- If length(Fifo)==0 **//Sub-condition inside Condition 1 When queue(Fifo) is empty, forward incoming request directly to server**
- Temp\_time=Arrivals(Arrivals\_in\_Server(Last\_request\_arrived+1))
- If rand<Probability\_cache     Set Service\_times: Cache\_time
- Else     Set Service\_times: expnd(1/Service\_time)
- Finish\_servicetime=Temp\_time+Service\_times(Arrivals\_in\_Server(Last\_request\_arrived +1))
- Request\_serving= Arrivals\_in\_Server(Last\_request\_arrived+1)
- Increment Last\_request\_arrived
- Else **//Sub-condition inside Condition 1 When queue is not empty, move first request from Fifo to Server**
- If Rand<Probability\_cache     Set Service\_times(Request\_serving)=Cache\_time
- Else Set Service\_times(Request\_serving)=expnd(1/Service\_time)
- Finish\_servicetime=Temp\_time+Service\_time
- Else **//Condition 2 when requests are being processed**
- Next\_request\_arrival=Arrivals(Arrivals\_in\_Server(Last\_request\_arrived+1))
- If Finish\_servicetime<Next\_request\_arrival **//Sub condition inside Condition 2 When new request arrives after the current service finishes, new requests directly go to server**
- End\_time(Request\_serving)=Finish\_servicetime
- Add Finish\_servicetime-Arrivals(Request\_serving) to Response\_time
- Update Request\_serving, Temp\_time, Request\_serving(i)
- Else **//Sub condition inside condition 2 when a new request arrives before current service finishes, new requests are added to queue(Fifo)**
- If length(queue)<B     Add Arrivals\_in\_Server(Last\_request\_arrived+1) to Fifo
- Else     drop\_request
- Update Last\_request\_arrived, Temp\_time
- Variance(i)=var(Response\_time) **//Step6 Calculate Variance**
- Step\_var:0 to Response\_time in step of 0.0001
- For i=1 to length(Step\_var) **//Step7 Calculate CDF for each server**
- CDF: length(find(Response\_time<Step\_var(i)))/length(Response\_time)
- Plot(Step\_var,CDF)
- End of For loop at Step 5
- End