CS 744 Course Project Phase-II Report

E-Commerce System

Instructor: Mythili Vutukuru

Dhananjay Kumar Sharma
Roll: 173050046

Debanjan Das
Roll: 173050069

Contents

1 Introduction

In phase-I, we have designed a simple prototype of an e-commerce system. By design, it was an open loop system. Because generally the number of users is not fixed for an e-commerce system; any number of requests can come at the server. Hence, first we have done open loop testing on our system and then also performed closed loop testing. Let us describe both of them.

2 Open Loop Testing

2.1 Basic Parameters

In case of open loop system there are several parameters. The parameters are:

1. **Arrival Rate**: Number of requests coming per second to the server. It is denoted by λ.
2. **Response Time/Turn Around Time**: Total time taken to get serviced per request. That is,
$$Response\ Time = \frac{1}{\mu - \lambda}$$
where μ = service rate and λ = arrival rate.
3. **Throughput**: Number of requests successfully completed per second, it is denoted by X. Hence,
$$X = \frac{No.\ of\ requests\ serviced\ successfully}{Total\ time\ taken\ to\ service\ all\ those\ requests}$$
4. **Total Service Time**: Total time taken to service all the requests.

2.2 Load Generator

The load generator is basically the client code from our Phase-I of the project. We just changed it to a multithreaded client where each thread is now a client and these clients are being generated at a certain rate which is fixed by us.

We have used Python 3.5 to write the load generator for our system. For doing the open loop test we have to generate incoming requests at a fixed rate, e.g, 1000 requests/second. We have done this by maintaining a `while` loop within which we are creating threads after a certain amount of time. The `while` loop is repeating for only 1 second and creation of thread is controlled by the `sleep(t)` function call. We can control the arrival rate by changing the value of `t` parameter inside the `sleep()`. The code snippet of thread creation is given below:

```
start = time.time()
i=0
elapsed=0
t = 0.001
while (elapsed<1):
   i=i+1
   thread = Thread(target=doShopping,args=(i,start))
   thread.start()
   time.sleep(t)
   elapsed=time.time()-start
```

Using the above code we can generate 1000 threads at ideal condition. Since, we have initialized $t = 0.001$, hence in 1 second we should have generated 1000 threads. But as there are few other lines also present inside the while loop which will also take some time so, practiaclly 1000 threads will not get created. In our case we were able to create 830 threads on average for sleep time of $t = 0.001$ sec.

Now we have taken readings for the no. of requests that are successfully serviced and the time taken to service them successfully. The readings were also generated by the load generator on the terminal screen.

2.3 Data Readings

	Arrival Rate	No. of Serviced Requests	Throughput	Total Service Time
0	2	2	3.573450	0.559680
1	3	3	4.123220	0.727580
2	4	4	4.918911	0.814982
3	5	5	5.838250	0.856420
4	6	6	6.708900	0.894330
5	7	7	7.632283	0.917156
6	8	8	8.565000	0.934026
7	9	9	9.473050	0.950062
8	10	10	10.388400	0.962605
9	11	11	11.161500	0.985530
10	12	12	12.313900	0.974507
11	13	13	13.228940	0.982693
12	14	14	14.096800	0.993129
13	15	15	14.881160	1.038340
14	16	16	15.197290	1.052819
15	17	17	15.176469	1.120155
16	18	18	15.309864	1.175712
17	19	19	15.150879	1.254052
18	20	20	15.323570	1.341239
19	25	25	15.175000	1.557289
20	30	30	15.243107	1.968103
21	40	40	15.644370	2.556830
22	50	49	15.305884	3.214450
23	75	73	15.185745	4.820310
24	100	96	15.206800	6.286650

2.4 Necessary Plottings

The Fig 1 is describing the Throughput vs. Arrival Rate of requests. From the readings in Section 2.3 we calculated the throughput as;

$$X = \frac{No.\ of\ Serviced\ Requests}{Total\ Service\ Time}$$

The Fig 2 is the plotting between response time and arrival rate. From Fig 1 we figured out that at λ=15.4367 the curve starts flattening. Hence we concluded that $\mu = \lambda = 15.4367$. Using λ

from the readings and $\mu = 15.4367$ we calculated the response time using the formula;

$$Response\ Time = \frac{1}{\mu - \lambda}$$

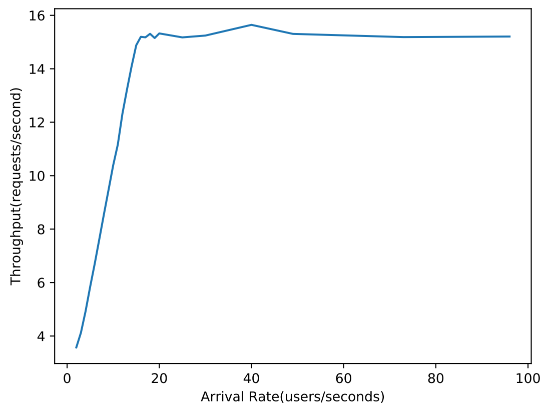

Figure 1: Throughput vs. Arrival Rate

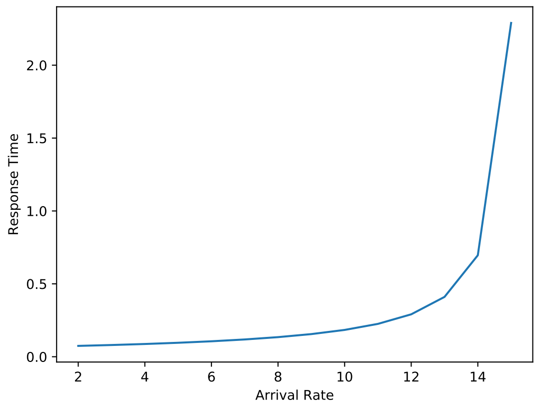

Figure 2: Response Time vs. Arrival Rate

2.5 Bottlenecks

At $\lambda = 100$ we are getting 100% CPU utilization and 97% disk utilization. But our graph is flattening at $\lambda = 15$. At $\lambda = 15$ the disk utilization is 40% and CPU utilization is 38.2%. Hence, there must be some software bottlenecks present in our system. Whenever we bombard our system with higher value of λ then only hardware bottlenecks are coming into picture.

3 Closed Loop Testing

3.1 Introduction

In closed loop testing we fix the number of users and run the load generator for a fix amount of time. Let this fixed number of users is N. And response time is $E[R]$, also here think time i.e, $E[Z] = 0$. Each of the N users will be issuing one request, waiting for it to complete, and issuing the next request immediately afterwards without any think time.

Little's Law for Closed System: Throughput increases almost linearly with N initially. But as queueing delays start to build up, throughput flattens out. Response time is almost constant initially, but increases as N increases and queues build up. Hence for closed system, according to Little's law;

$$N = X * E[T]$$

3.2 Load Generator

In closed loop load generator we have fixed the number of users where each user is a thread and we are also setting a time frame within which each user will keep on sending multiple requests. Suppose the time frame is 30 seconds then each user will send requests to the front-end server for 30 seconds.

Let, $time\ frame = 30\ seconds$. Then after creation of a thread, it goes into `while` loop which will continue for 30 seconds for that thread and it will send requests only after it gets back response for it's previous request. The code for this part is given below.

```
elapsed = 0
j = 0
start = time.time()
while elapsed<tc:
   req_time = time.time()
   order_description = 'Book 1'
   order = s.send(order_description.encode('ascii'))
   try:
      order_status = s.recv(1024)
      order_status = order_status.decode('utf-8')
      delta_t = time.time()-req_time
   except:
      print("ERROR: Thread ",i," could not placed the order.\n")
   elapsed = time.time()-start
   j = j+1
flag = s.send("$".encode('ascii'))
```

To accept multiple requests from clients the front-end server is also maintaining a `while` loop. Inside this `while` loop it will keep on accepting orders from clients and also sending response to the corresponding clients. The code for this part is given below.

```
int flag=1;
if(flag_authentication){
   while(flag){
      flag = get_order(sock,back_end_sock);
   }
   close(sock);
}
int get_order(int new_socket,int back_end_sock){
   int n_order,tc;
   char order[BUFFER_SIZE],buying_status[BUFFER_SIZE],temp[BUFFER_SIZE];
   bzero(order,BUFFER_SIZE);
   n_order = read(new_socket,order,BUFFER_SIZE-1);
   if(n_order<0) error("ERROR reading from socket");
   if(strcmp(order,"$")==0) return 0;
   strcpy(buying_status,_process_order(order,back_end_sock));
   n_order = write(new_socket,buying_status,strlen(buying_status));
   if(n_order<0) error("ERROR writing to socket");
   return 1;
}
```

After completion of the time frame the client will send a $flag = \$$. Whenever the server gets this flag it will exit from the `while` loop and close the socket for the corresponding client.

3.3 Data Readings

We have performed each experiment for 30 seconds i.e, $time\ frame = 30$. The data generated from these experiments is given below.

	Avg. Response Time	No. of Users	Throughput
0	0.093631	1	10.561400
1	4.314900	50	10.765900
2	8.154600	100	10.809500
3	31.996700	500	11.022500
4	41.992900	700	11.010244
5	52.050920	900	10.745080
6	58.115119	1000	10.535900
7	63.508729	1100	10.439150
8	68.547659	1200	10.431140

3.4 Necessary Plottings

The Fig 3 is describing the Response Time vs. No. of Users. From the readings in Section 3.3 we calculated the average response time as;

$$R = \frac{Total\ time\ taken\ to\ complete\ all\ the\ requests}{Total\ number\ of\ requests}$$

The Fig 4 is the plotting between Throughput and No. of Users.

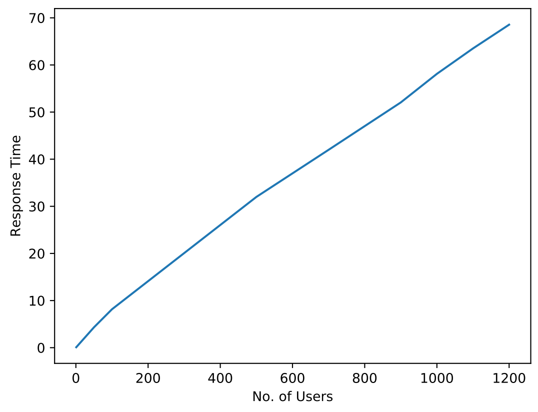

Figure 3: Response Time vs. No. of Users

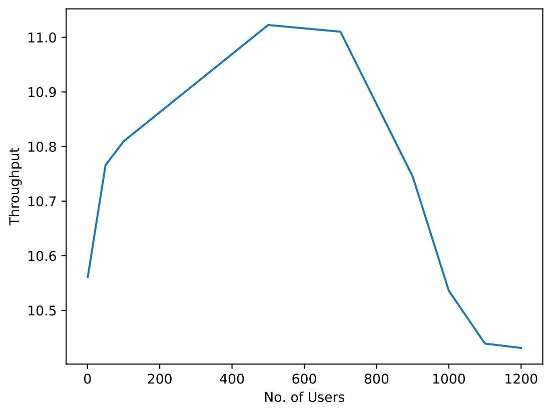

Figure 4: Throughput vs. No. of Users

3.5 Bottlenecks

When number of user is 500, from that point of time the graph of throughput is gradually decreasing. Also the disk utilisation is nearly 100% in this case. Hence in case of closed loop testing the system saturates at $N = 500$.

4 Conclusion

- After performing both open loop and closed loop testing we found out that there is some anomaly between the results of both these tests. If time frame is 30 seconds then in case of open loop system we are getting $\mu = 15$. So, number of requests that will be handled successfully will be $15 * 30 = 450\ requests$.

  But according to our measurements we found out that in case of closed loop system for $N = 500$ and $time\ frame = 30\ sec$, it is generating 830 requests.

  Later we figured out that in case of open loop testing we were creating one thread for each requests but in case of closed loop testing we were generating the requests with a `while` loop for each user. And we know that there is a overhead of creating a thread compared to just only generating requests through `while` loop. This explains the anomaly in both of the results.

- We executed the front-end server, back-end server and the load generator on different cores of the same machine. But still the readings we got is machine dependent. When we executed the same code on different machine we were getting different results.

- There must be some kind of software bottlenecks present in our system. In phase-III we would like to remove these bottlenecks and optimize our system.