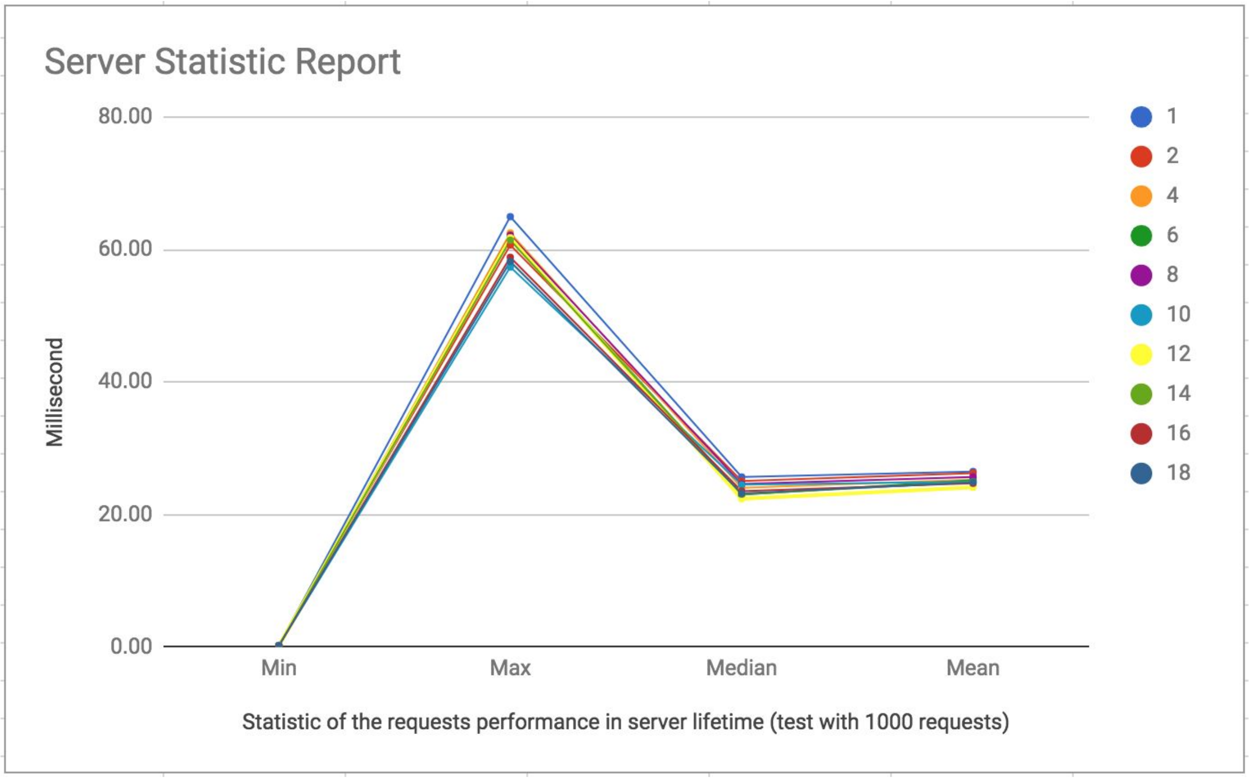
For testing the server performance, I used httperf tools running 1000 requests with 10 sessions. I found that once the thread number exceeds the session number, the per-request performance is generally better than the one from smaller thread number. In addition, once the thread number has exceeded the session number, the performance will reach a threshold and won't be improved anymore as we increase the thread number. In this test, having 12 threads would make the server perform the best.

The result distribution of each thread number shown below all has a positive skew. The means are all larger than the median. It means we have more number of short request time than long request time.

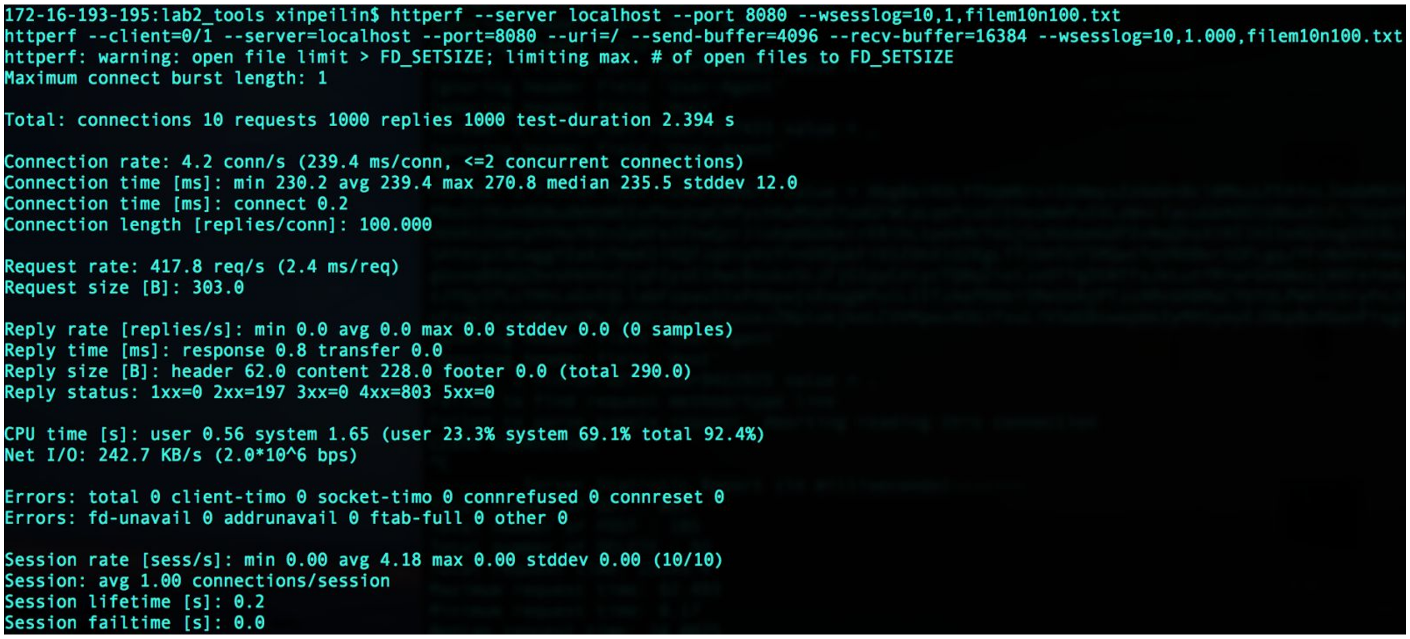
Total Request	Thread#	Min	Max	Median	Mean	Total Time
1000	1	0.23	64.95	25.66	26.48	26,483.40
Session	2	0.22	60.68	25.04	26.24	26,243.10
10	4	0.17	62.49	24.01	25.25	25,249.40
	6	0.16	61.80	22.98	25.18	25,175.40
	8	0.21	62.15	24.59	25.66	25,657.10
	10	0.20	57.34	24.48	24.93	24,926.50
	12	0.17	61.76	22.36	24.09	24,092.70
	14	0.17	61.39	23.04	24.90	24,902.80
	16	0.17	58.82	23.52	24.70	24,695.70
	18	0.20	58.17	23.17	24.85	24,852.10



Another fact is that the per-request work time from client report is way longer than the one provided from server side. This can be because of the different start and end time definition from client and server side. From the viewpoint of client, the request starts from the time of sending and ends with receiving the response at client side. On the other hand, from the viewpoint of server, the request starts from receiving at server side and ends with sending back the response to clients. The gap can be from the communication and connection setup.

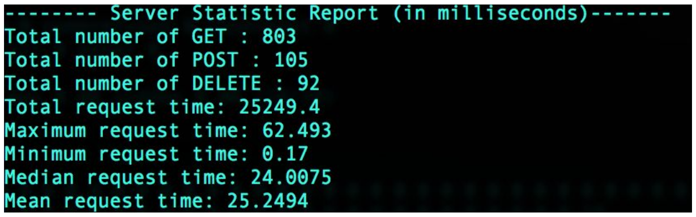
Thread#4

Client Report



```
172-16-193-195:lab2_tools xinpeilin$ httperf --server localhost --port 8080 --wsesslog=10,1,filem10n100.txt
httperf --client=0/1 --server=localhost --port=8080 --uri=/ --send-buffer=4096 --recv-buffer=16384 --wsesslog=10,1.000,filem10n100.txt
httperf: warning: open file limit > FD_SETSIZE; limiting max. # of open files to FD_SETSIZE
Maximum connect burst length: 1

Total: connections 10 requests 1000 replies 1000 test-duration 2.394 s

Connection rate: 4.2 conn/s (239.4 ms/conn, <=2 concurrent connections)
Connection time [ms]: min 230.2 avg 239.4 max 270.8 median 235.5 stddev 12.0
Connection time [ms]: connect 0.2
Connection length [replies/conn]: 100.000

Request rate: 417.8 req/s (2.4 ms/req)
Request size [B]: 303.0

Reply rate [replies/s]: min 0.0 avg 0.0 max 0.0 stddev 0.0 (0 samples)
Reply time [ms]: response 0.8 transfer 0.0
Reply size [B]: header 62.0 content 228.0 footer 0.0 (total 290.0)
Reply status: 1xx=0 2xx=197 3xx=0 4xx=803 5xx=0

CPU time [s]: user 0.56 system 1.65 (user 23.3% system 69.1% total 92.4%)
Net I/O: 242.7 KB/s (2.0*10^6 bps)

Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0

Session rate [sess/s]: min 0.00 avg 4.18 max 0.00 stddev 0.00 (10/10)
Session: avg 1.00 connections/session
Session lifetime [s]: 0.2
Session failtime [s]: 0.0
```

Server Report



```
-------- Server Statistic Report (in milliseconds)-------
Total number of GET : 803
Total number of POST : 105
Total number of DELETE : 92
Total request time: 25249.4
Maximum request time: 62.493
Minimum request time: 0.17
Median request time: 24.0075
Mean request time: 25.2494
```

To improve the performance, I implemented the server with a thread pool manager. By creating multiple threads, we can process the requests concurrently to save some work time when there are available resources on the server. I also made each thread process all the request from a connection until closing the connection. There are pros and cons with this design. The advantage is that we can save time from pushing the thread and connection information back to the queue and pull it out for each of the request. The disadvantage is that we are sacrificing the availability for efficiency. In this case, a thread is occupied by one connection until it process all the requests. It might keep other connection waiting for an amount of time unfairly.

Comparing the result from the code with a single global state lock and the implementation I made for Lab 2, the later solution performs slightly better but surprisingly not much better. I think it might be because the payload is not very large and there is not much independent computation for each request. There are only constant time lookup, delete and insert and they all need to lock the data storage while doing the operation. Thus, there is little time we can save from running concurrently.