Number of Clients	Latency of single-tier server	Latency of 2-tier server
5	0.41	0.55
8	0.40	0.59
10	0.41	0.0.56

Table 1: Time taken to process requests by clients for several number of concurrent clients

1 Performance Evaluation

We perform two experiments on our program. All experiments were performed on a 4-Core Intel Core i7-3610QM CPU with hyperthreading enabled and each core running at maximum frequency 3.30GHz. Our system was running Linux Kernel 4.11.12-100.fc24.x86_64. Since, hyperthreading is enabled, our system has 8 logical cores. In all our experiments, 100 get requests are sent by each client, with 50% incrementMedalTally and 50% getScore requests.

Latency of 2-tier server over Single tier server : In this experiment we measure the latency of every pull request performed by the server for several number of clients. Experiments are performed for both 2-tier front end and database server, and a single-tier server written in Lab1. In both cases there is only one server. Results are taken for 5,8,10 number of clients. Table 1 describes results for this experiment. We can see that the latency of 2-tier server is about 30% more than the latency of single tier server.

This experiment can be performed by first starting the server by executing `python server.py --n_servers 1`, then executing command `python evaluation_latency.py 5` and varying first command line argument of script from 5, 8, 10.

Load Balancing Test In this experiment, we evaluate the load balancing for our servers. We execute 20 clients on 1, 2, 3, and 4 servers. Table **??** shows our results for this experiment. As we can see in our results, there is some slight improvement in the response time, with increase in the number of servers. However, this is minimal because (i) the bottleneck now becomes the database server, of which there is only a single instance, and (ii) there is not much work which is done by the front end server, and hence, the cost of creating a new thread for every request does not really pays of because this thread only creates a new request to be sent to the database.

This experiment can be performed by first starting the server by changing the n_servers parameter of command `python server.py --n_servers 1` and then executing command `python evaluation.py 10`.

1.1 Clock Synchronization

For clock synchronization, our experiment is similar to the test case we used. Here, we set the clock offset for each front end server and database to a random

No. of Servers	Latency of each request(ms)
1	0.55
2	0.53
4	0.52
8	0.52
16	0.53

Table 2: Time taken to process requests by clients for 10 concurrent clients

value less than δ. We started 2 front end servers and one database server, all of which takes part in the berkely clock synchronization algorithm. In our case offset of servers are 2, 1, 7. After first iteration of synchronization started by server with offset 2, we get new times as 0, -1, -6, i.e. changed by offset 1. In this way we continue with clock synchronization. Clock synchronization can be executed by running dispatcher (and front end servers) by `python server.py --is_leader_election True --is_clock_sync True --is_raffle False` and database by `python server.py --is_leader_election True --is_clock_sync True`.

1.2 Totally Ordered Multicast

This experiment is similar to the test case of totally ordered multicast. Here, we create 2 front end servers and the databse server. 2 more clients are created, which are registered on different servers. These clients continuously send several requests to their respective servers. Every 100th requests is stored in raffle, and after some time raffle winner is randomly selected.