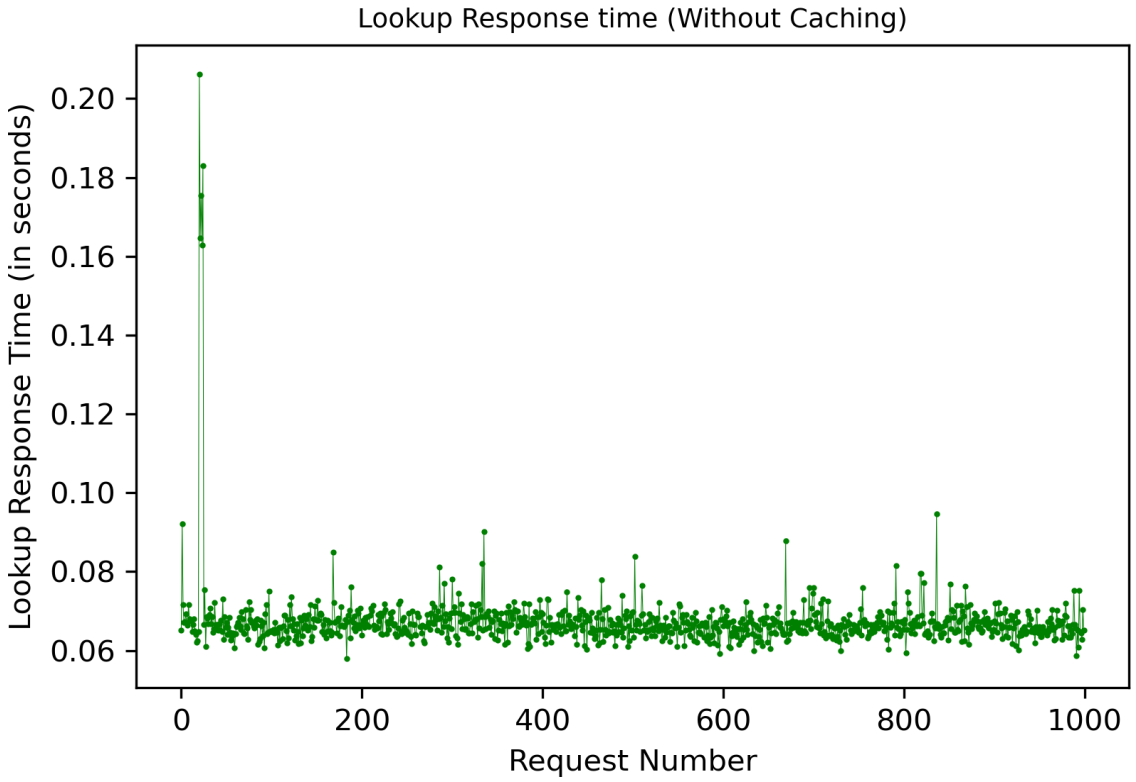
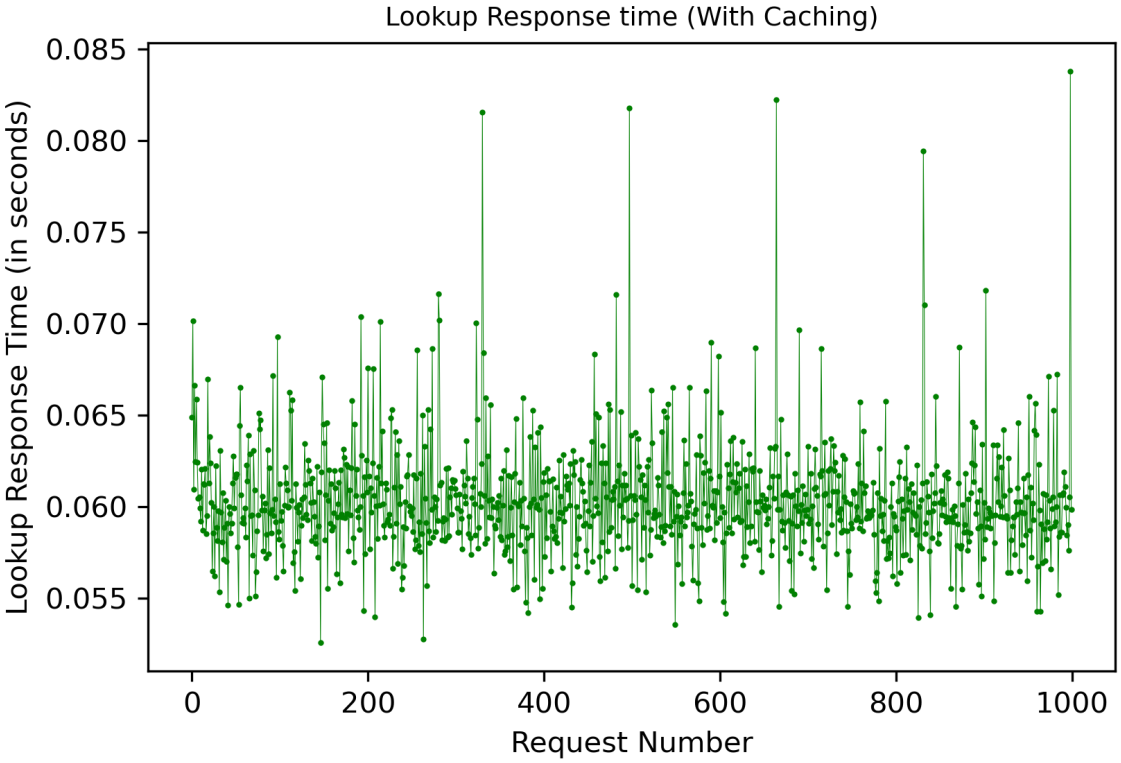
Evaluation and Measurements

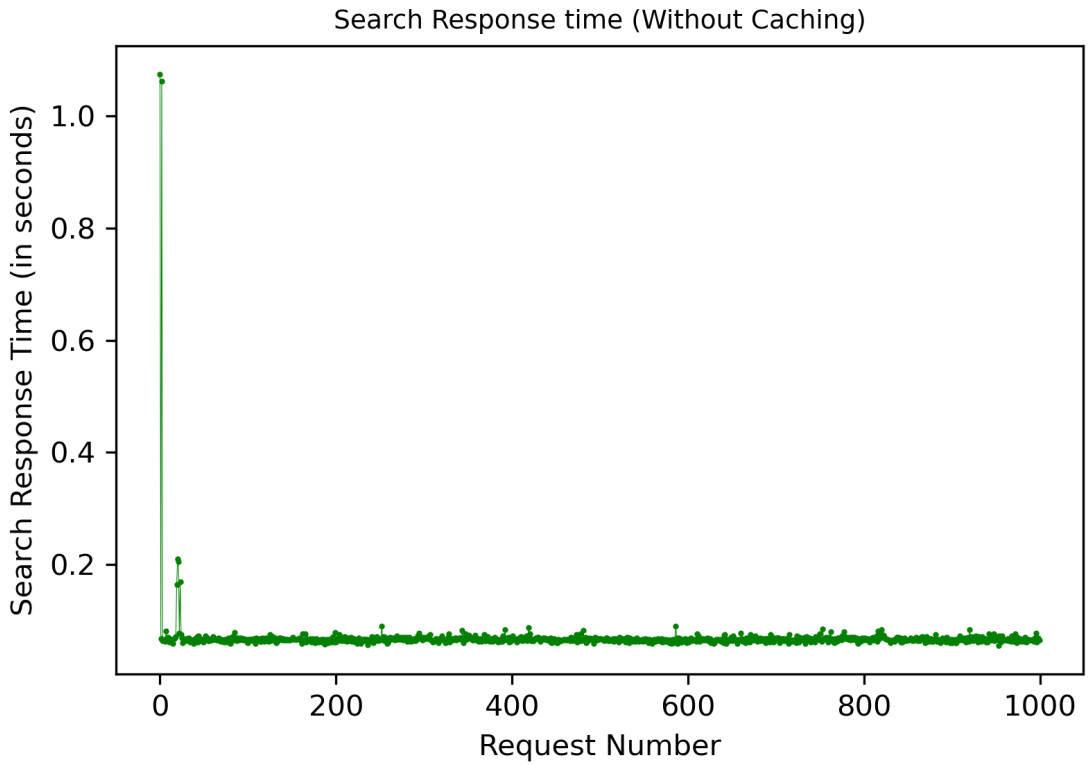
1. **Compute the average response time (query/buy) of your new systems as before. What is the response time with and without caching? How much does caching help?**

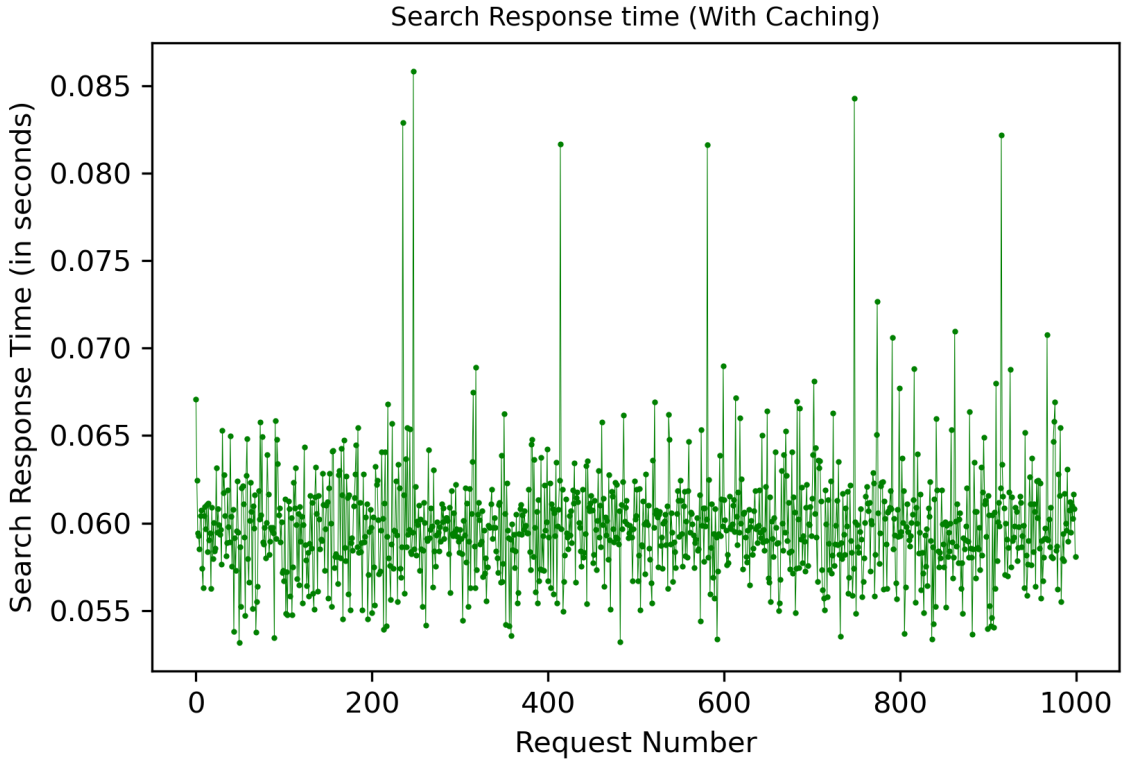
Average Value of Lookup without Caching in milliseconds : 67.34042143821716
Average Value of Lookup with Caching in milliseconds : 60.333189964294434
Average Value of Search without Caching in milliseconds : 68.86626124382019
Average Value of Search with Caching in milliseconds : 59.88228511810303
Average Value of Buy without Caching in milliseconds : 111.4014241695404
Average Value of Buy with Caching in milliseconds : 110.34846591949463

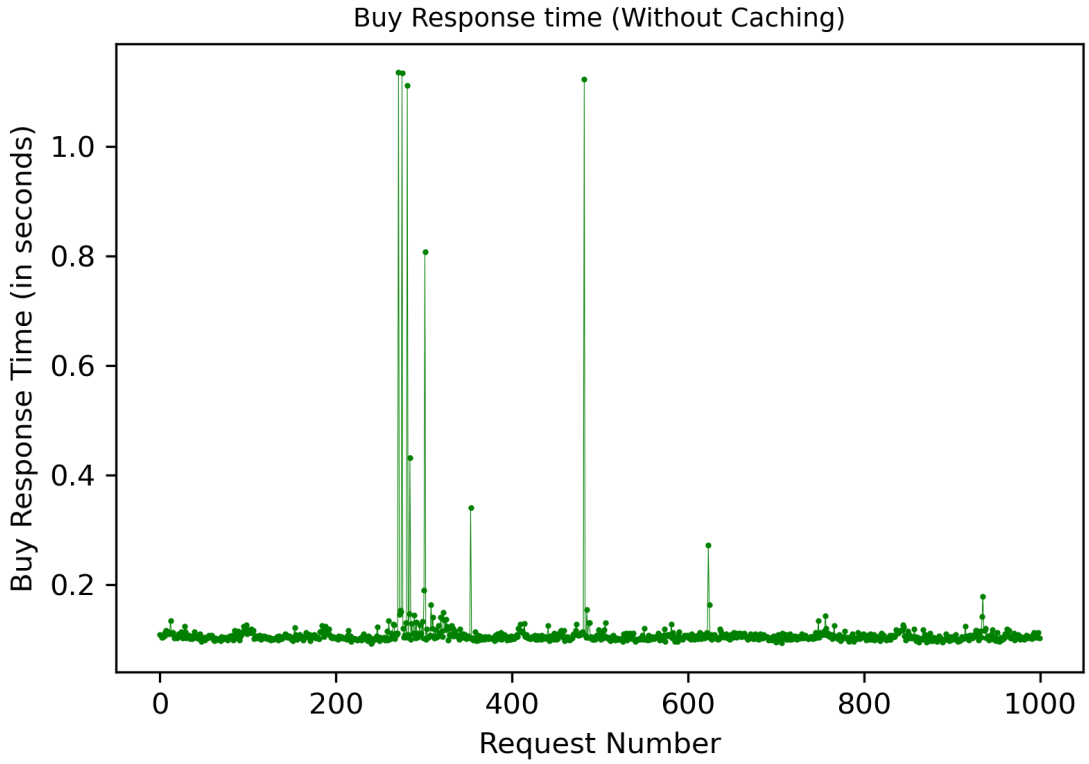
As it can be seen, the caching reduces the average lookup and search time, since both these queries are read requests and are thus optimized via caching. There is no significant improvement in case of buy request since we invalidate the cache with every buy request. Caching provides an improvement of approximately 6-7ms per request. The graphs for 1000 requests for all the cases above can be found below.

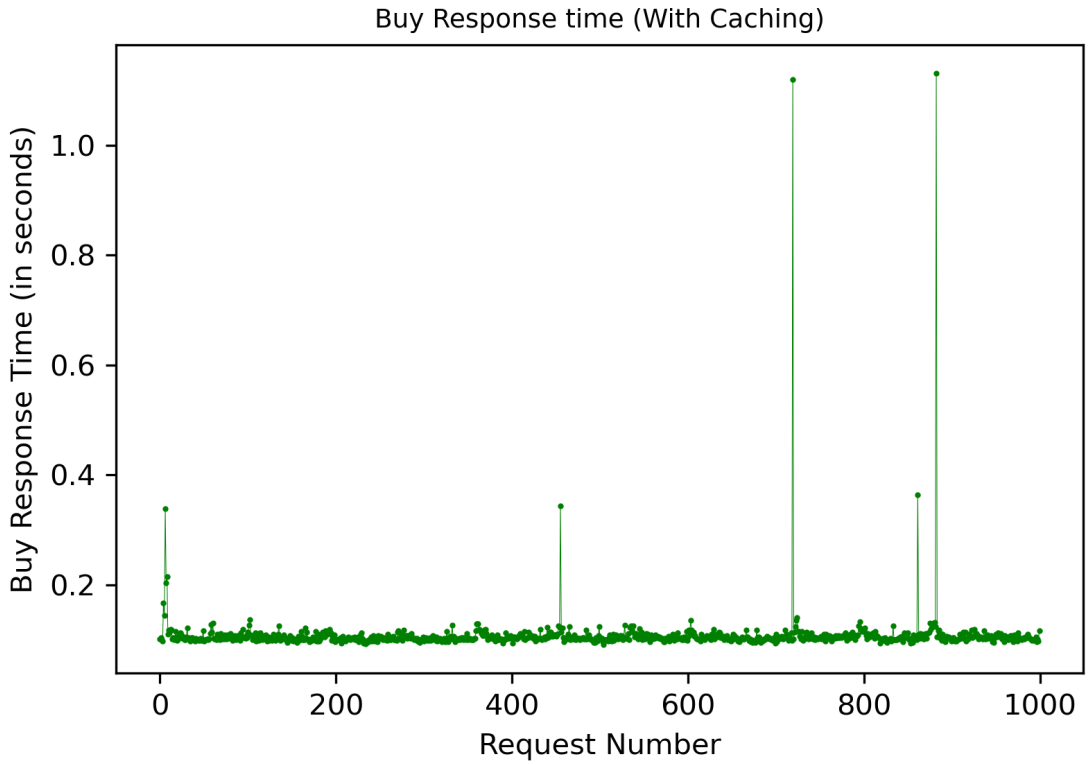



Lookup Response time (With Caching)
Lookup Response Time (in seconds)
0.085
0.080
0.075
0.070
0.065
0.060
0.055
0
200
400
600
800
1000
Request Number


Search Response time (Without Caching)
Search Response Time (in seconds)
1.0
0.8
0.6
0.4
0.2
0
200
400
600
800
1000
Request Number


Search Response time (With Caching)
Search Response Time (in seconds)
0.085
0.080
0.075
0.070
0.065
0.060
0.055
0
200
400
600
800
1000
Request Number


Buy Response time (Without Caching)
Buy Response Time (in seconds)
1.0
0.8
0.6
0.4
0.2
0
200
400
600
800
1000
Request Number



2. **Construct a simple experiment that issues orders or catalog updates (i.e., database writes) to invalidate the cache and maintain cache consistency. What is the overhead of cache consistency operations? What is the latency of a subsequent request if it sees a cache miss?**

We have set up the following experiment for checking the cache invalidation which is done at every buy request (catalog/order updates).

1. We call lookup for a particular id (Cache is empty before this step).
2. We call search for a particular topic (Cache only contains the id lookup before this step, from step 1).
3. We call the above lookup and search again, with the same topic and id, to make sure that we are hitting the entry in cache.
4. We call the buy method.
5. We call the above lookup and search again. Expectation is that the cache is currently empty because of the cache invalidation which was done in step 4.
6. We note the response time in step 3 and step 5 and compare their average values over 1000 requests.

Following results are obtained for the above experiment -

Average Value of Lookup with Caching in milliseconds : 60.042853355407715
Average Value of Lookup without Caching in milliseconds : 66.2558376789093
Average Value of Buy with Caching in milliseconds : 60.1681444644928
Average Value of Buy without Caching in milliseconds : 67.28768968582153

As it can be seen, there is a significant increase in the lookup and search response time which aligns with our expectation of cache invalidation in step 3. We also observe that the time difference observed due to Caching is approximately similar to our observations in Evaluation 1, which conclusively proves that the cache is getting invalidated correctly.
The overhead of the consistency operations can be seen in the increase of request response time (approximately 6ms). The latency of a subsequent request if it sees a cache miss is approximately 66.7ms (as opposed to approximately 60ms when there is a cache hit).

3. Construct an experiment to show your fault tolerance does the work as you expect. You should start your system with no failures and introduce a crash fault and show that the fault is detected and the other replica can mask this fault. Also be sure to show the process of recovery and resynchronization.

We start our experiment by running the script “runme.sh”. The script deploys the servers containers and runs the client.py that in turn starts the traffic by invoking search, lookup and buy API of frontend for 2 clients parallely. After running the script, we check the client.log file in the main folder, and see that every request has been served and the items are bought successfully. Please find the snippet from the client.log below:

```
2021-04-28 22:28:28,860 Buying the item with id '7' from front server.
2021-04-28 22:28:28,861 Starting new HTTP connection (1): 52.205.243.54:8010
2021-04-28 22:28:28,862 http://52.205.243.54:8010 "GET /lookup?id=3 HTTP/1.1" 200 177
2021-04-28 22:28:28,863 Lookup of item '3' successfull.
2021-04-28 22:28:28,863 Buying the item with id '3' from front server.
2021-04-28 22:28:28,866 Starting new HTTP connection (1): 52.205.243.54:8010
2021-04-28 22:28:58,996 http://52.205.243.54:8010 "GET /buy?id=7 HTTP/1.1" 200 105
2021-04-28 22:28:58,997 Succesfully bought the item '7'
2021-04-28 22:28:59,004 http://52.205.243.54:8010 "GET /buy?id=3 HTTP/1.1" 200 105
2021-04-28 22:28:59,005 Succesfully bought the item '3'
```

Checking the working of the frontend server, we can see that the requests of two clients are being load balanced by the server replicas efficiently. The same can be seen in the log snippet below:

```
Thread-5 INFO Search method called with the topic name 'distributed systems' in catalog server.
Thread-5 DEBUG Catalog A is up, Calling Catalog Server A
Thread-6 INFO Search method called with the topic name 'graduate school' in catalog server.
Thread-6 DEBUG Catalog B is up, Calling Catalog Server B
Thread-6 INFO Searching of items with topic 'graduate school' successful.
Thread-5 INFO Searching of items with topic 'distributed systems' successful.
Thread-7 INFO Lookup method called with the id '3' in catalog server.
Thread-7 DEBUG Catalog A is up, Calling Catalog Server A
Thread-8 INFO Lookup method called with the id '1' in catalog server.
Thread-7 INFO Searching of items with id '3' successful.
Thread-8 DEBUG Catalog B is up, Calling Catalog Server B
Thread-8 INFO Searching of items with id '1' successful.
Thread-9 INFO Buy method called with the item with id '3' in order server.
Thread-9 DEBUG Order A is up, Calling Order Server A
Thread-10 INFO Buy method called with the item with id '1' in order server.
Thread-10 DEBUG Order B is up, Calling Order Server B
Thread-9 INFO Purchase of item '3' successfull.
Thread-10 INFO Purchase of item '1' successfull.
```

Now we will run the script “simulate_fault_tolerance.sh” which shuts down catalogA and then invokes the client.py which as before starts the traffic by invoking search, lookup and buy API of frontend for 2 clients parallely. We again observe the client.log to see if the requests have been served successfully despite the catalogA server being down.

```
2021-04-28 22:40:53,433 Buying the item with id '2' from front server.
2021-04-28 22:40:53,434 Starting new HTTP connection (1): 52.205.243.54:8010
2021-04-28 22:40:53,437 http://52.205.243.54:8010 "GET /lookup?id=4 HTTP/1.1" 200 175
2021-04-28 22:40:53,438 Lookup of item '4' successfull.
2021-04-28 22:40:53,438 Buying the item with id '4' from front server.
2021-04-28 22:40:53,439 Starting new HTTP connection (1): 52.205.243.54:8010
2021-04-28 22:40:53,492 http://52.205.243.54:8010 "GET /buy?id=2 HTTP/1.1" 200 105
2021-04-28 22:40:53,493 Succesfully bought the item '2'
2021-04-28 22:40:53,515 http://52.205.243.54:8010 "GET /buy?id=4 HTTP/1.1" 200 105
2021-04-28 22:40:53,516 Succesfully bought the item '4'
```

We see that the client does not get to know that one of the servers is down, and is easily able to do all the operations. The fault tolerance of the system is transparent to the client and it does not get to know the heuristic of the system. Now checking on the frontend level, we can see that after detecting that heartbeat of catalogA server is down, request is being sent to the other catalog server. You can check the logs snippet on the next page to see things at work:

```
INFO Search method called with the topic name 'graduate school' in catalog server.
DEBUG Catalog A is down, Calling Catalog Server B
INFO Searching of items with topic 'graduate school' successful.
INFO Search method called with the topic name 'distributed systems' in catalog server.
DEBUG Catalog B is up, Calling Catalog Server B
INFO Lookup method called with the id '4' in catalog server.
INFO Searching of items with topic 'distributed systems' successful.
DEBUG Catalog A is down, Calling Catalog Server B
INFO Lookup method called with the id '2' in catalog server.
INFO Searching of items with id '4' successful.
DEBUG Catalog B is up, Calling Catalog Server B
INFO Buy method called with the item with id '4' in order server.
INFO Searching of items with id '2' successful.
DEBUG Order A is down, Calling Order Server B
INFO Buy method called with the item with id '2' in order server.
DEBUG Order B is up, Calling Order Server B
INFO Purchase of item '4' successfull.
INFO Purchase of item '2' successfull.
```

Now to ensure that there is a successful recovery of the servers, the script “simulate_fault_tolerance.sh” starts the catalogA server again and invokes the “test_server_recovery.py” to check if both the servers are in sync with the latest transactions. We see that the crashed server syncs its db with the already up server after the re-deployment to ensure consistency. You can check the logs in ‘test_server_recovery.log’ to see the same.