

## Testing / Evaluation:

Below we describe some of the test cases used in evaluating our key value store. We also provide instructions in running each of the tests.

**All our testcases can be run by running the single file 'python Testhandler.py' . The results of each test case are printed in new file created with the name of the test.**

Preprocessing and important details:

**We consider port no 8081 for all four servers**

1. In the Testing directory you need to enter ip address of each server in the file [servers.sh](#)
2. Copy [servers.sh](#) and into the 'server' directory.
3. Enter the 'server' directory and Run 'bash update.sh'. This copies server code to each VM and then compiles the server on the VM's.

Step 4. and Step.5 are called by TestHandler.py , if you are not using it then you should do Step 4. or Step 5.

5. In the Testing directory there is a file called preprocess.py, it kills any existing servers in the VM's and does cleanup and starts a new server.

6. If you would like to do step 5. manually then before starting the server please remove the "Checkpoint file" called checkpoint.chk in the directory 'p1'.

## Test Cases:

### 1. Throughput

In this test case we perform 100 random read/writes for a certain number of iterations from 10 different processes. We compute the read-only throughput and the write-only throughput. In the case of reads, we have seen throughput in the range of 600 to 1000 reads/second. In case of writes we observed throughput of at least 530 writes/second.

To run this test case :

Run the script: `python Throughput.py #Request File #no of iterations`

Output: It will print the start time and end time taken for each iteration in a file called timer.

To run this test case:

Run this script:

`python ComputeThroughput.py #no of iterations #no of requests in file #no of processes`

Output: It will print the throughput for this run of the test case.

### 2. BaseConsistency

In this case we write 40 unique keys for 10 iterations to different servers and the keep track of the (old value,new value) chain of updates for each key. Our system returns consistent results as the (old value,new value) chain of updates is preserved.

The script for this test case is in the 'BaseConsistency' directory.

To run this test case:

Run the script: `python BaseConsistency.py #no of iterations`

Output: The Output consists the (old value, new value) chain for each key as well as statistics regarding the total number of consistent keys.

### 3. Break Consistency Test Case

In this case we write 10 queries of the same key with different values to different servers. We do this for 40 keys for 10 iterations. In other words for each iteration we perform 400 writes and every 10 consecutive writes are for the same key across different servers. We then check for the preservation of (old value,new value) chain similar to the base consistency test. In this test we observe that our system does not always preserve the (old value,new value) chain of updates as the network latency for message propagation across servers seems to be higher.

The script for this test case is in the 'BreakConsistency' directory.

To run this test case:

Run the script: `python BreakConsistency.py #no of iterations`

Output: The Output consists the (old value, new value) chain for each key as well as statistics regarding the total number of consistent keys.

### 4. ManualPartition

In this case we partition the servers into two partitions. We write 100 keys to partition 1 and 100 keys to partition 2. We then unpartition and then check if the keys get synchronized. Since our keep alive message frequency is 4 seconds we sleep our test script for 3-4 seconds after unpartition before checking for consistency among partitions.

Preprocessing: open `dopartition.sh` and enter the ip address and port no of each server at the top

The script for this test case is in the 'ManualPartition' directory.

To run this test case:

Run the script: `python ManualPartition.py #input for partition1 #input for partition 2 #read input for partition1 #read input for partition2`

Output: The output of this script provides statistics regarding what fraction of keys from partition 1 were consistent and what fraction of keys from partition 2 were consistent.

### 5. Stress Partition

In this case we let the client send write requests to different servers, in between we partition then after sometime we unpartition and then perform reads for the keys that are written and check if the values are consistent across all four servers. Since our keep alive message frequency is 4 seconds it takes a couple read iterations for us to get consistent results from all four servers.

The script for this test case is in the 'StressPartition' directory.

To run this test case:

Run the script: `python StressPartition.py #input for partition1 #input for partition 2 #read input for partition1 #read input for partition2`

Output: The output of this script gives statistics regarding what fraction of keys had consistent values across all four servers after unpartitioning.

## 6. Failure

In this case we check if the service is tolerant to failures. We write some data to server that we plan to kill. We then kill this server and then write data to other working servers. We then read the data of the failed server from other servers and then check if we can get consistent results. We then wake up the failed server and then read data written to other servers during its failed period. Since our heartbeat message frequency is 2 seconds we sleep for about 3 seconds and check if these reads are consistent. We can extend this test for 2 failures or 3 failures.

The script for this test case is in the 'Failure' directory.

To run this test:

Type in the file `failure.txt` whether server is a "failure" or "nonfailure" server. There is an entry in this file for each server.

Run the script: `python Failure.py`

Output: The test outputs the total number of consistent reads from the data of the failed server and total number of consistent reads from the data of the working servers. It also mentions whether the system is tolerant or not for the given number of failures. If there is any inconsistency in the reads the system is considered intolerant to failure.