X253

A distributed key-value cache

Abhinand Menon (861131308), Mohit Morarka (861130666) | CS253 | Final Project Report | 12/13/2013

Table of Contents

Executive Summary

This report details the design and implementation of the X253 system. Companies like Facebook and Twitter require efficient access to large amounts of data that are stored in the form of key-value pairs over several machines. Keys may include user ids, tweet ids etc. One method of facilitating fast look-up is by using a distributed hash table to quickly determine the node on which the data corresponding to a particular key is stored. Placing this structure in the main memory further increases the look-up speed of the system by reducing disk I/O.

The X253 system is a distributed in-memory hash table which aims to provide a solution to this problem.

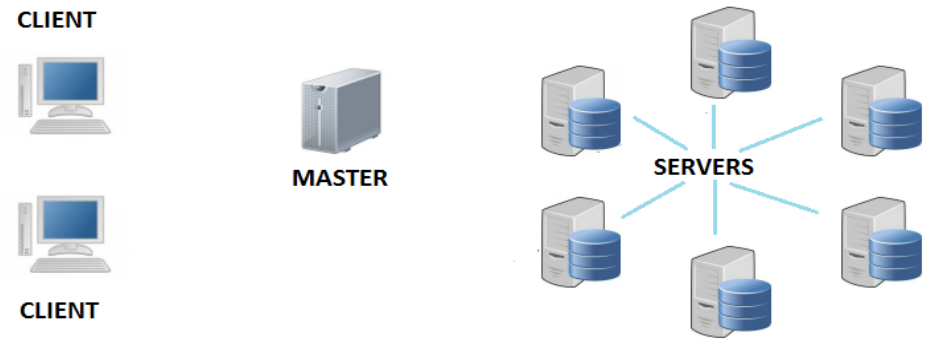
The following section explain the X253 architecture and its implementation. The subsequent section describes the results of preliminary tests performed to measure the performance of the X253 system and plans for final tests.

Team Member Contributions

The overall system architecture and evaluation plan was decided based on numerous discussions between both members. Mohit has been responsible for the design of the X253 Master. Mohit has implemented the Master in Java. Abhinand has been responsible for the design of the X253 Storage Server and the X253 communication protocol. Abhinand has implemented the Storage Server in Python.

System Architecture

Introduction



The X253 System has three type of nodes:

Clients - that query the System,

Master - that distributes and maintains the keyspace among the Storage Servers, and

Storage Servers - that cache and replicate key-value data

The X253 system architecture is most similar to that of Dynamo. However, it differs from Dynamo in two major ways:

1. Data is not versioned, and
2. Data is cached in memory

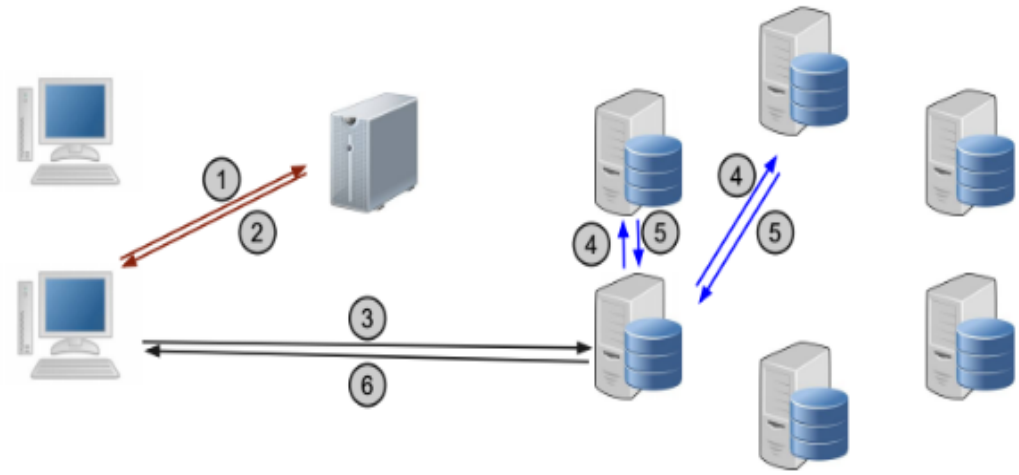
The hash used in the consistent hashing algorithm is MD5, similar to Dynamo, Cassandra etc.

Clients

Our Clients send requests to the Master which informs them about the Storage Servers that are currently serving requests for specified keys. The client then connects to the appropriate server and sends the desired request.

The client talks to the X253 system using a simple protocol with the following request types:

- Data requests:
  - Put
  - Get
  - Delete
- Other requests:
  - Send-Stats



Master

The key features of our master is to:

- Minimize key reassignments when nodes are added or fail.
- Provide good data availability through replication.
- Scalability via minimal inter-server communication.

Consistent Hashing

The algorithm maps the cache to an interval, which will contain a number of object hashes. If the cache is removed then its interval is taken over by the cache with an adjacent interval. All the other caches remain unchanged.

We are making using of the inbuilt MD5 hash function for our consistent hashing code as it provides a large range of integer values which can be spread across the consistent ring and prevents multiple keys to have the same hash value.

The circle is represented as a sorted map of integers, which represent the hash values to caches here. When a hash object is created we add each physical node to a circle map the number of virtual ids.

The location of each virtual id is chosen by hashing a server's IP with with a numerical suffix.

We have included the following methods in our Master code-

- Get
- Put
- Delete

By the means of replication we are ensuring that the data survives single or multiple machine failures. More the number of replicas we have ensures that data survives single or multiple machine failures.

Storage Servers

The Storage Servers cache the key-value pairs in memory and serve them to Clients upon request. They communicate with the Master to learn their keyspace assignments and with other Storage Servers to check failures and to replicate data.

The servers use the standard Python dictionary data type to store key-value pairs. This places no restriction on the value format and length. However, keys are restricted to strings, numbers and other immutable objects (like Python tuples, since keys are interpreted as Python objects).

Servers are contacted by the Master on startup and are told their keyspace assignments and the addresses of their four nearest (two prior and two next) neighbors. They may also be contacted by the Master with renewed assignments during failure recovery.

Once their keyspace assignments are known, servers replicate data to their two subsequent neighbors. Servers also periodically ping these neighbor and report to the Master in case a failure is detected in the smart mode.

Network connections are handled using the SocketServer library using the ThreadingMixIn class. Locks are placed on the main storage dictionary and key lists using threading.RLock.

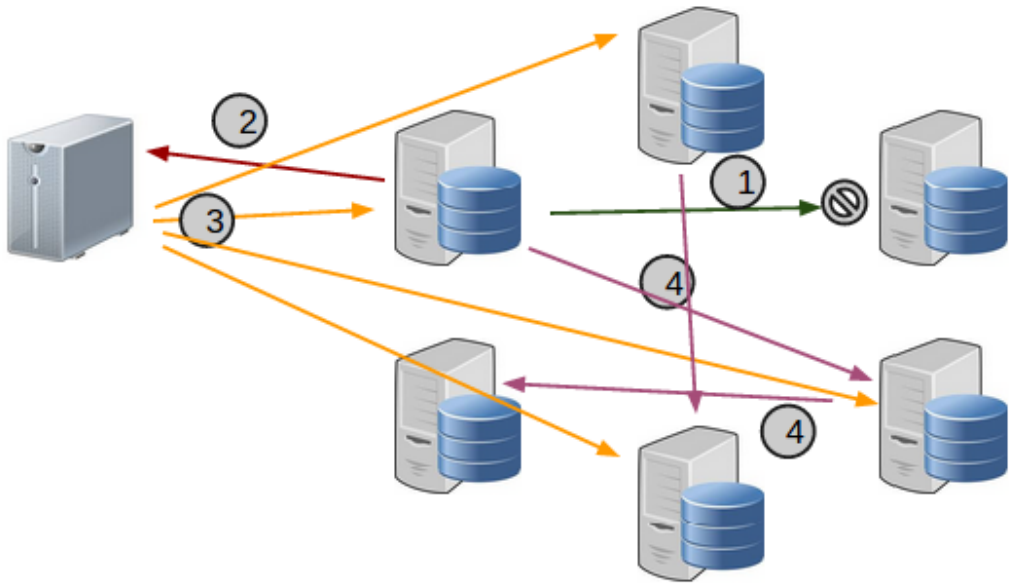
Fault detection and recovery

Each X253 server maintains a dictionary (main_store) that stores all the key-value mappings as well as three lists (keys_n, keys_n_minus_1, keys_n_minus_2) that keep track of the keys that are stored in the main_store by the server itself and those keys that have been replicated by its previous neighbors using Replicated-Put requests.

When a server is running and it receives a new Neighbor-List packet. It could mean two things:

- A neighbor has failed
- The replication factor has been changed at the Master

In either case, the X253 server then merges/deletes keys from the three lists and performs any required replication. The master sends Neighbor-List packets to each server, one by one, and waits for each Acknowledgement. The first server that is contacted by the master during a failure-recovery is the one that is two steps ahead of the failed server.



Communication Protocol

The design of the X253 protocol has been inspired by the Trivial File Transfer Protocol. A non-hostile operating environment is assumed and hence the protocol has no security and authentication features. The protocol limits key and value sizes to 4GB. The preferred transport medium in TCP. UDP may further limit payload sizes.

Operation

1. The server listens for packets on port 1600.
2. All packets consists of a 1 byte opcode that identifies the type of packet and additional fields.
3. Message-lengths fields may contain a zero value.
4. The sender of a request chooses a request-id and the corresponding responses should have the same request-id.
5. Senders should choose a random initial request-id value for their first request and increment that for every additional request.
6. Data and administrative requests result in response types given by the lists below during normal operation or else in Error packets:

- Client - Storage Server requests and replies
    - Put -> Ack
    - Get -> Values
    - Delete -> Ack

- Master - Storage Server requests and replies
    - Stop-Server -> Ack
    - Start-Server -> Ack
    - Send-Key-Range -> Key-Range
    - Neighbor-List -> Ack
    - Dead-Neighbor -> Neighbor-List -> Ack

- Client - Master requests and replies
    - Send-Key-Server-Mapping -> Server-List
    - Send-Stats -> Stats

- Storage Server - Storage Server requests and replies
    - Replicated-Put -> Ack
    - Replicated-Delete -> Ack

Packet Formats

Client - Storage Server Packets

1. Put:

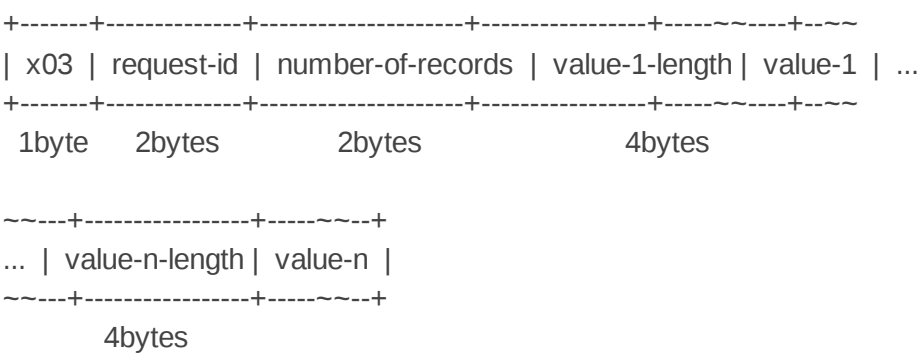
```
+-------+--------------+---------------------+----------------+---~~----+-----------------+------~~----+--~~
| x01 | request-id | number-of-records | key-1-length | key-1 | value-1-length | value-1 |...
+-------+--------------+---------------------+----------------+---~~----+-----------------+------~~----+--~~
  1byte     2bytes            2bytes                4bytes                    4bytes

~~---+----------------+---~~-----+-----------------+------~~-----+
... | key-n-length | key-n | value-n-length | value-n |
~~---+----------------+---~~-----+-----------------+------~~-----+
           4bytes                   4bytes
```

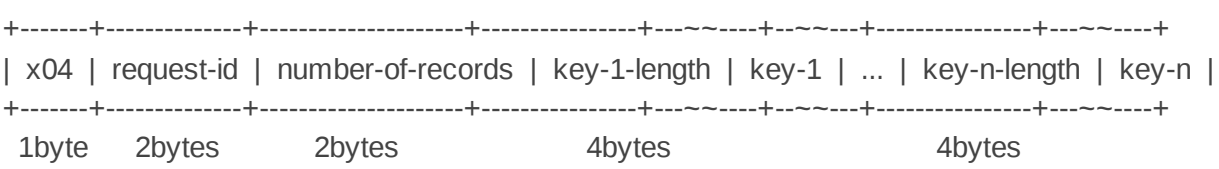
2. Get:

```
+-------+--------------+---------------------+----------------+---~~----+--~~---+----------------+---~~----+
| x02 | request-id | number-of-records | key-1-length | key-1 | ... | key-n-length | key-n |
+-------+--------------+---------------------+----------------+---~~----+--~~---+----------------+---~~----+
  1byte     2bytes            2bytes                4bytes                          4bytes
```

3. Values:



4. Delete:



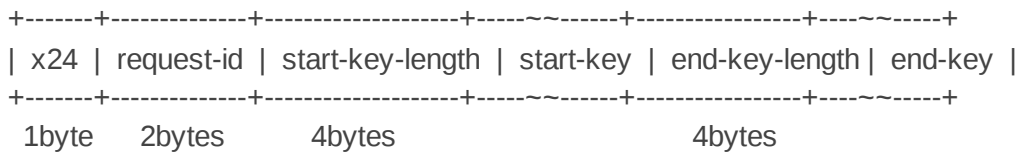
Master - Storage Server Packets

5. Stop-Server:

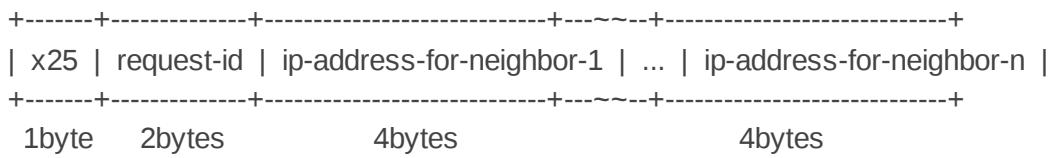
6. Start-Server:

7. Send-Key-Range:

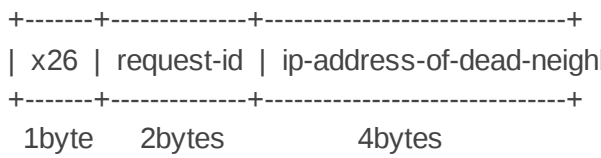
8. Key-Range:



9. Neighbor-List:

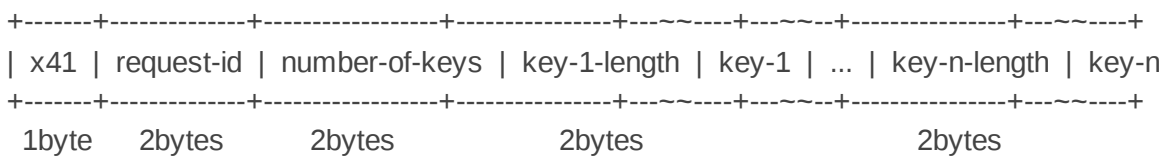


10. Dead-Neighbor:

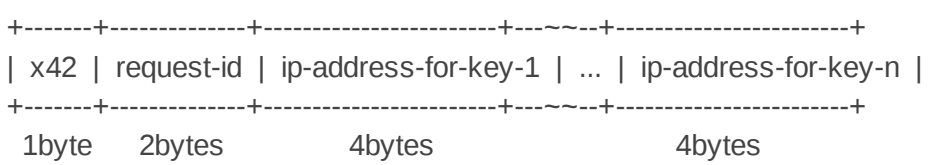


Client - Master Packets

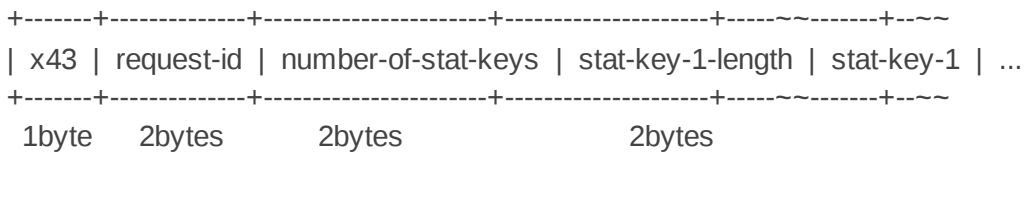
11. Send-Key-Server-Mapping:



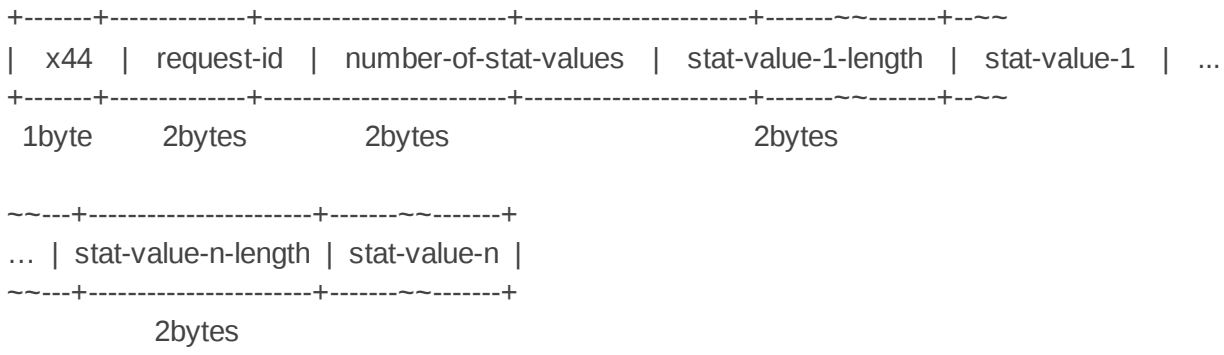
12. Server-List:



13. Send-Stats:

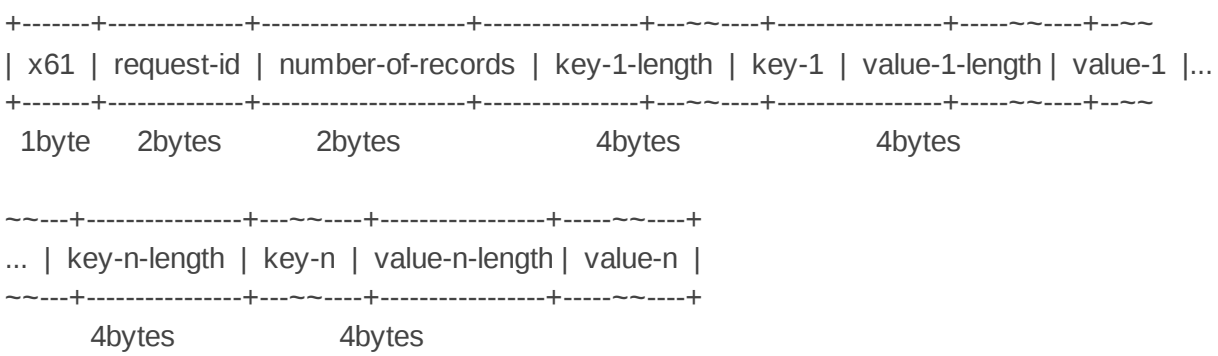


14. Stats:

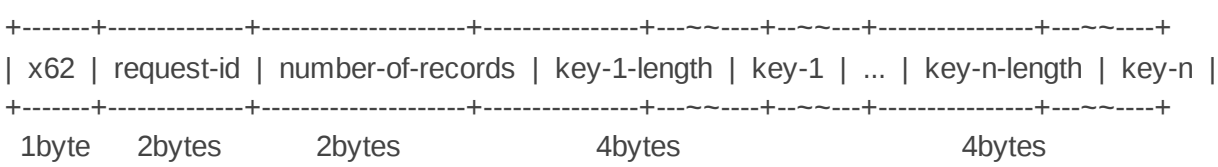


Storage Server - Storage Server Packets

15. Replicated-Put:

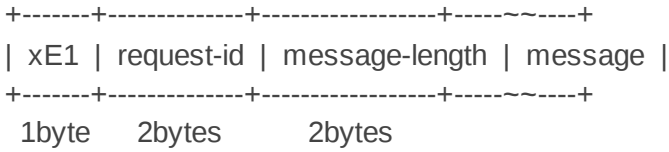


16. Replicated-Delete:

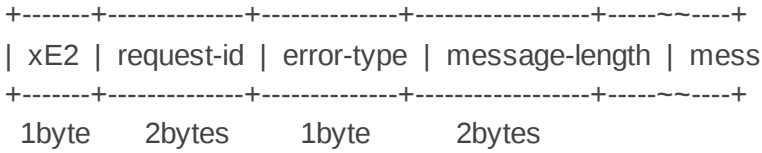


General Purpose Packets

17. Ack:



18. Error:



System Evaluation

Preliminary Test Results

For the tests described in this section, we deployed one test client, one X253 master and ten X253 servers deployed on 12 Amazon EC2 t1.micro instances. Each EC2 instance provides compute power equivalent to approximately twice the the power of a 1.0 to 1.2 GHz Opteron 2007 CPU when run for short durations and has about 600 MB of memory.

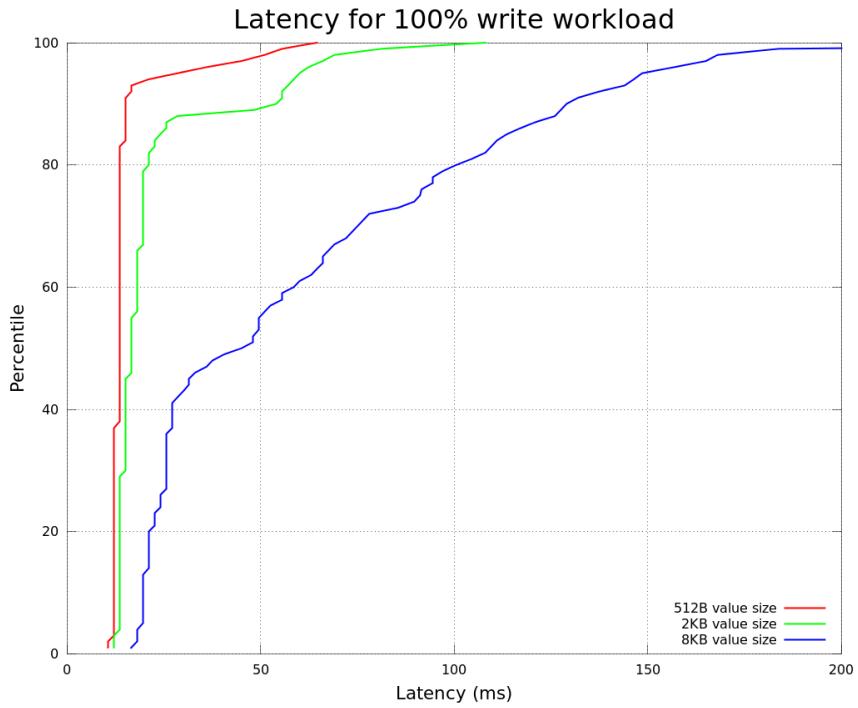
The test client asked for one key per get/put request. Hence each request comprised of two stages: 1) get the server IP address from the master and 2) get/put request to the server. Each machine locally logged all activity.

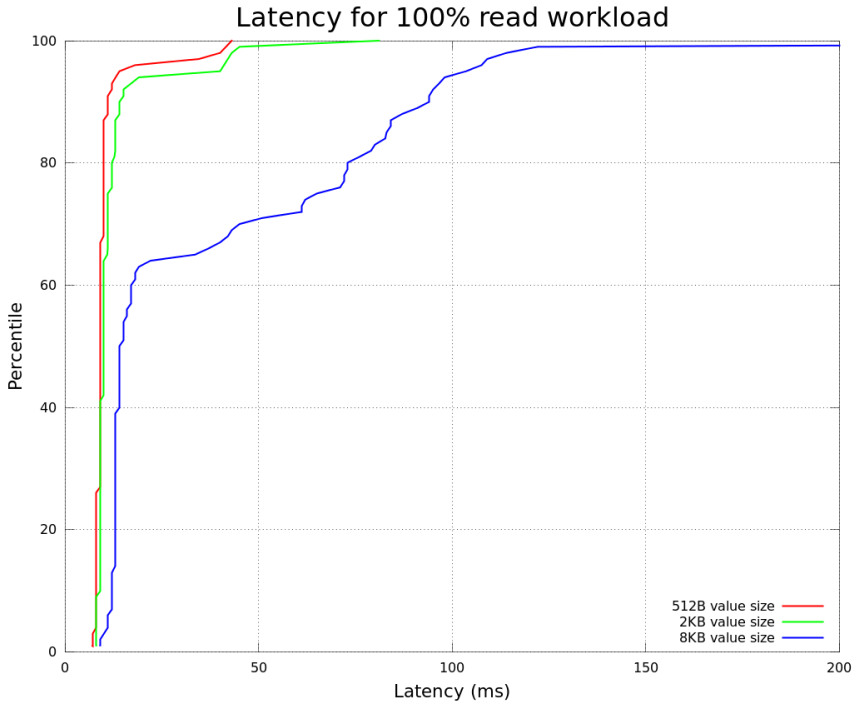
One set of tests was performed for each of the following workloads:

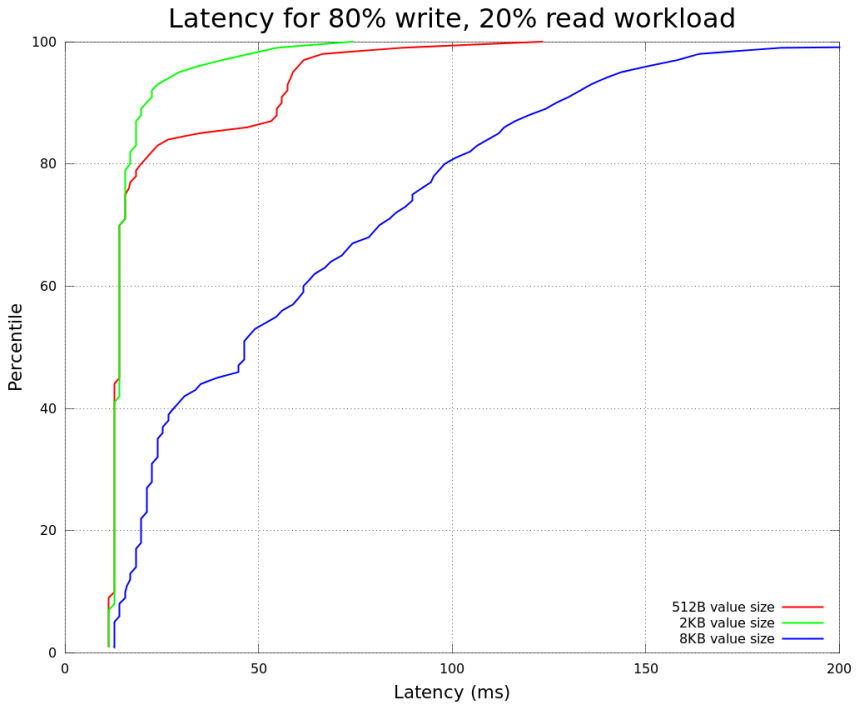
- 100% write requests
- 80% write and 20% read requests
- 20% write and 80% read requests
- 100% read requests

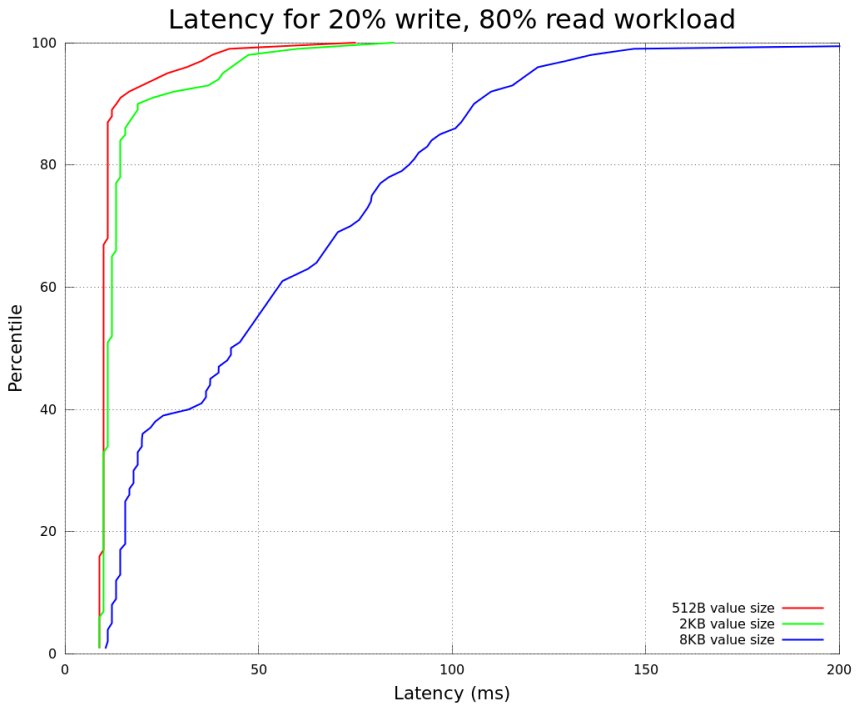
Each set consisted of 3 runs. Key lengths were set to 6-byte for every run in every set. Each run consisted of 1000 iterations. The key for a particular iteration was randomly generated and followed a Uniform distribution. The value sizes used for the three runs were 512 Bytes, 2 KBytes and 8 Kbytes respectively.

The following graphs show the variation in latency with increasing value length for each of the four sets:









The graphs above show a latency of about 7-8 ms for read requests and 12-13 ms for write requests for about 90% of requests where the value sizes are 512 Bytes and 2 KBytes.

Final Test Results

This section gives details about a more comprehensive set of tests that were conducted to study the throughput, resilience to faults and scalability of the X253 system. A final subsection describes the increase in throughput of the system by using a simple optimization technique.

Latency with ZIpf distribution of keys

For this test, we deployed one test client, one X253 master and ten X253 servers deployed on 12 Amazon EC2 t1.micro instances.

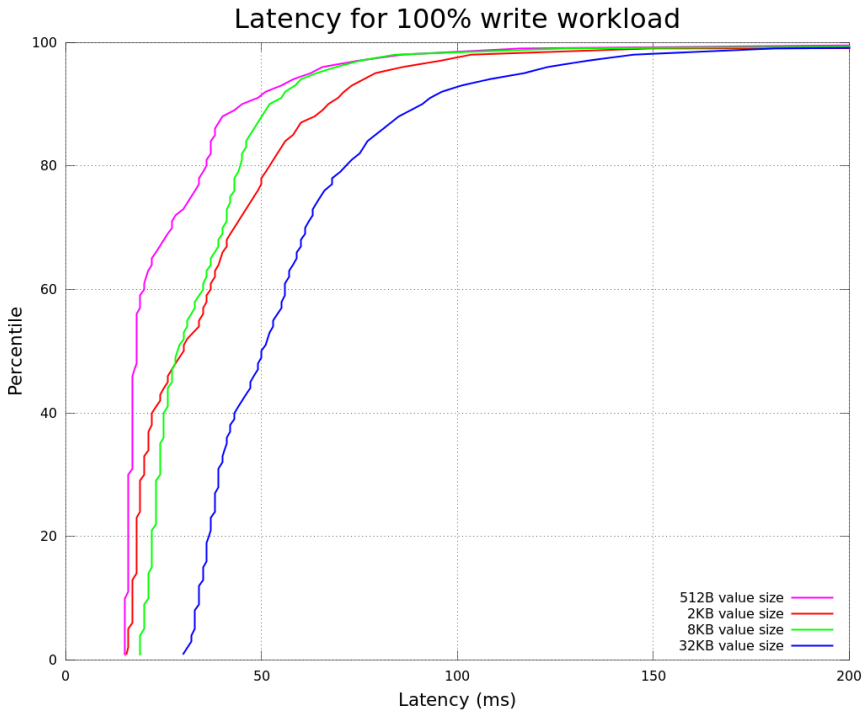
The test client asked for one key per get/put request. Hence each request comprised of two stages: 1) get the server IP address from the master and 2) get/put request to the server. Each machine locally logged all activity.

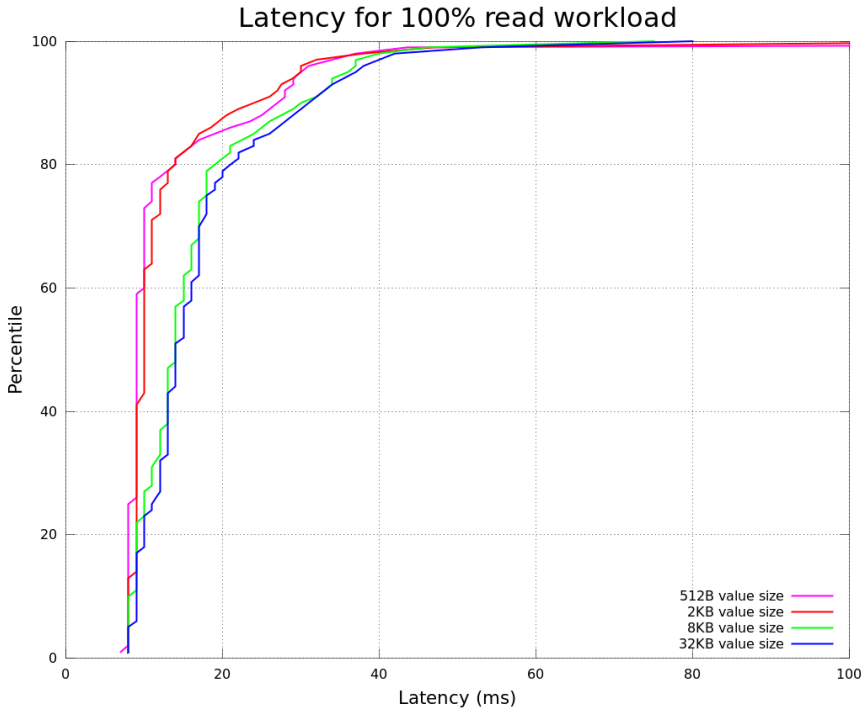
One set of tests was performed for each of the following workloads:

- 100% write requests
- 100% read requests

Each set consisted of 4 runs. Key lengths were set to 6-byte for every run in every set. Each run consisted of 1000 iterations. The key for a particular iteration was randomly generated and followed a Zipf distribution. The value sizes used for the three runs were 512 Bytes, 2 KBytes, 8 Kbytes and 32 Kbytes respectively.

The following graphs show the variation in latency with increasing value length for the two sets:





These graphs show a latency of about 7-8 ms for read requests and 12-13 ms for write requests for about 90% of requests where the value sizes are 512 Bytes and 2 KBytes. The latency values for runs using larger value lengths are higher as expected.

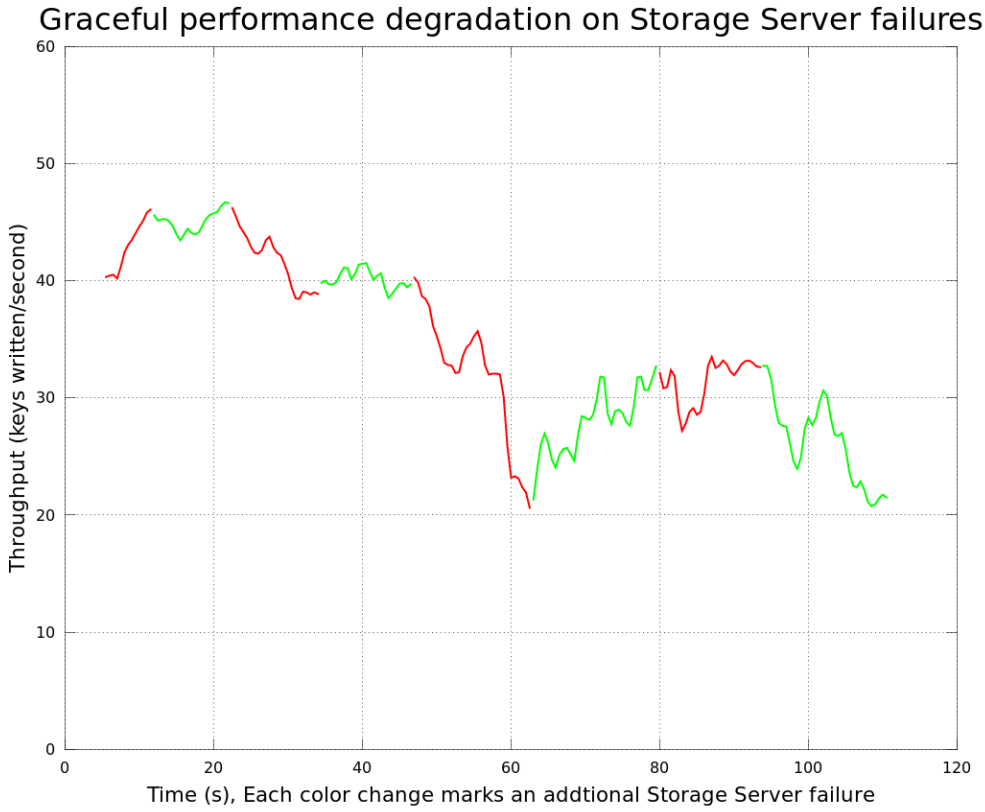
This values observed in this test are similar to the latency values from the test using a Uniform distribution of keys for the lower percentiles. However, the curves are much smoother compared the previous test.

This may be because the Zipf distribution results in much smaller number of keys (from the set of possible 100 unique keys) actually being stored on the servers. This likely prevents the dynamic resizing of the Python dictionaries during runtime on the X253 Storage Servers; this resizing may have caused the sudden jump in latency values in the previous test for the run with value length of 8 KBytes.

Demonstration of fault tolerance

For this test, we initially deployed one test client, one X253 master and ten X253 servers deployed on 12 Amazon EC2 t1.micro instances. The test client asked for one key per get request. The key for a particular iteration was randomly generated and followed a Zipf distribution. Each Put request wrote a 2 KBytes value on a server.

During the test one X253 server was killed after every 1000 Key-IP-mapping requests received by the X253 master by sending a Stop-Server request to a randomly chosen X253 server. Seven out of 10 servers were killed, one at a time, resulting in a total of 8000 Put requests for the entire test.



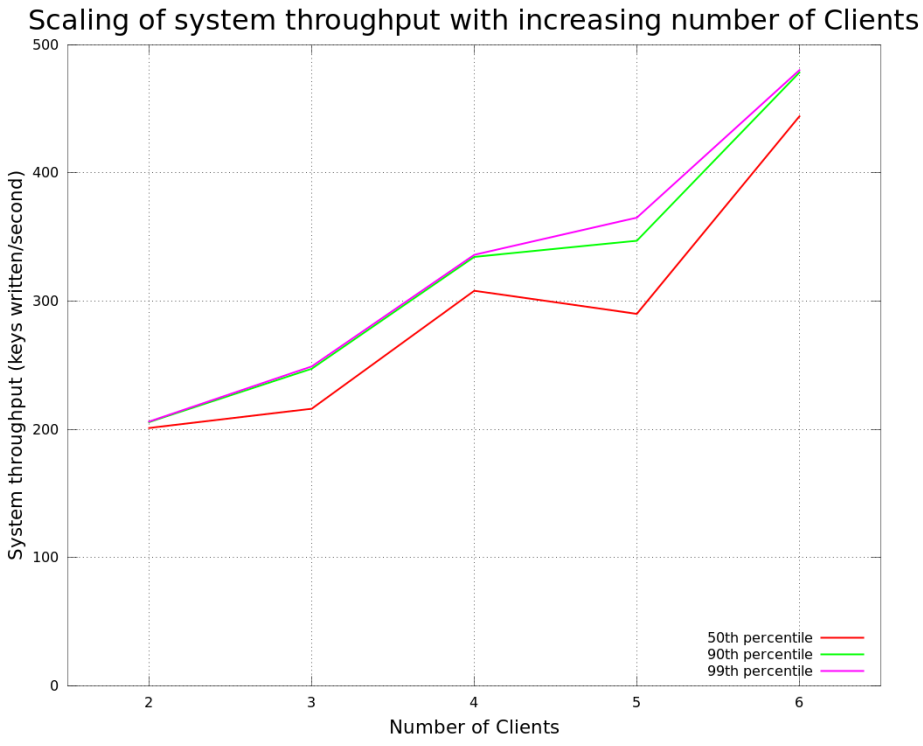
The graph above shows a gradual reduction throughput as the X253 servers are servers are killed.

Scaling X253 system throughput

For this test, we initially deployed **two** test client, one X253 master and **four** X253 servers deployed on 12 Amazon EC2 t1.micro instances.

The test consisted on 5 runs. During each run, the throughput was measured for 1000 Put requests. The test client asked for one key per get request. The key for a particular iteration was randomly generated and followed a Zipf distribution. Each Put request wrote a 2 KBytes value on a server.

The number of test clients was increased by one during every subsequent run. The number of X253 clients in the fifth and final run was six.



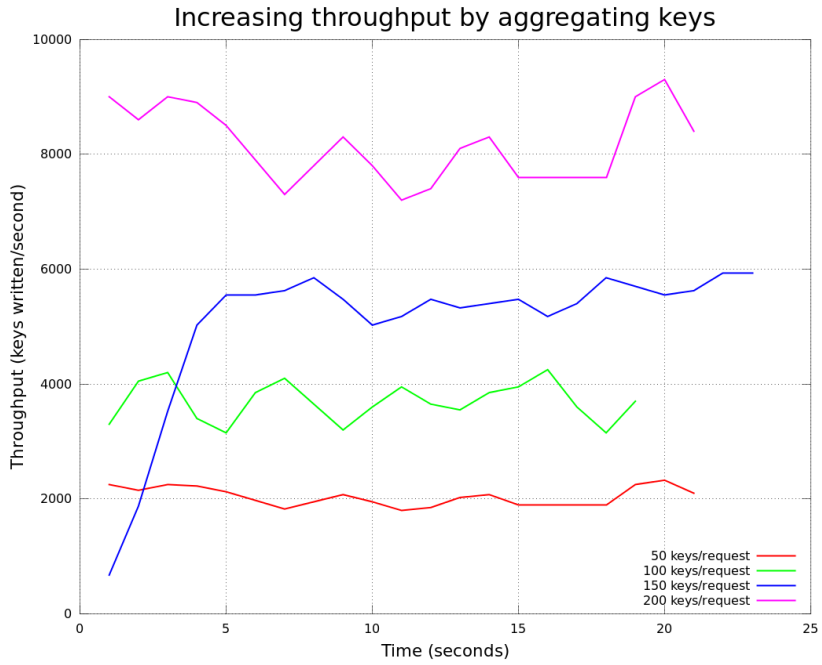
The graph above shows a nearly linear increase in throughput as the number of clients are increased. At seven clients the four X253 servers were unable to handle to the load.

Optimizing X253 system usage: Aggregating keys

For this test, we initially deployed one test client, one X253 master and ten X253 servers deployed on 12 Amazon EC2 t1.micro instances.

This test consisted of four runs. Each run consisted of 1000 requests. The number of keys for which the test client asked the X253 master for addresses in one request (say n) was increased in every run. 'n' was varied from 50 to 100, 150 and 200 keys per Key-IP-mapping request. The key for a particular iteration was randomly generated and followed a Zipf distribution. Each Put request wrote one 2 KBytes value per key-value pair on a server.

Once the master responded with the addresses for the keys, the client aggregated each of the keys that are mapped to a particular X253 storage server into a single Put request meant for that server. The client then sent the final set of Put requests to the appropriate servers. This usually resulted in 6 to 7 Put requests per run (likely due to the Zipf distribution).



The graph above demonstrates the substantial increase in throughput achieved by aggregating multiple keys in each Put request. The numbers in the legend correspond to the number of keys in the initial Key-IP mapping request sent to the X253 master.