Godzilla: A Cloud Key-Value storage with Replication, Failure Detection and Notification

Mohsen Ahmadvand
mohsen.ahmadvand@tum.de

Ibrahim Alzant
ibrahim.alzant@tum.de

Arash Khatayee
arash.khatayee@tum.de

ABSTRACT

Key value store is at the core of many large scale distributed services. It aims to handle petabytes of data across thousands of servers and network components which distributed over many data centers around the world. At this scale, many components may fail continuously. In order to keep these systems functional, reliability, availability, scalability and persistent state infrastructures are required.

This report presents the design and implementation of a key value store, which is highly available and reliable in terms of persistent state. It also provides the scalability feature. To achieve scalability, a similar approach to chord implementation of DHT is used. On other hand, availability and reliability are provided using a replication strategy along side with a efficient failure detection system. Upon a failure of a node, its data is recovered using its replica servers, so persistent state is guaranteed and the probability of loss of data is very low.

Keywords

Failure detection, Key-Value Storage, Cloud

1. INTRODUCTION

In the past, only one server was enough to serve thousands of clients. This server was separating data layer from application layer by delegating data retrieval to a sql server in order to improve efficiency and transparency. Due to growth of Internet technologies, number of users and amount of data stored and requested, new requirements have been arose. Those requirements could not be provided by typical client server architecture. In addition, these requirements were consisting of highly demanded scalability, reliability, consistency and minimum latency. Moving to Cloud environment was a solution to these requirements. In order to provide these requirements, some problems need to be taken into account such as dealing with failures, keeping data consistent, and preventing the loss of data.Over a decade of research in cloud computing, effective solutions has been achieved which aiming to satisfy mentioned properties using a peer to peer communication concepts between a group of server nodes.

Massive user requests, massive amount of data, and dealing with failures are main motivations to move to the cloud storages. Although relational databases are good solution in dealing with structured data and can respond to complicated queries. By design, they are not easly scalable, so they cannot be used effectively in cloud environment. Furthermore, efficient response time is required and it is not achievable with relational databases in communication with huge number of users. Instead cloud databases deal with data as pairs of key and value, which is known as NoSQL database. In key-value stores there is no consistent structure defined for data. Thus, NoSQL databases are fast, highly scalable, replicable, and can deal with variable structured data. Key value storages are being used by many data exhaustive applications such as Facebook, YouTube and Google Drive

Key value store can be used rigorously by all application developers who are seeking for highly scalable and reliable data storages. Use cases of these kind data stores could be as following: online stores, e-commerces, video sharing, social media applications, electronic votings, hotel booking, travel agency and etc. We insist that, usage of the key value storages is not limited to the mentioned use cases and any massive user targeted solution can be a potential candidate.

In this report, we have implemented and evaluated a highly scalable and reliable key value storage using state of the arts literature and techniques in the field supplied to us during our practical training course.

The rest of this report is structured as following Section 2 Related work, discusses the related literature in the field and their applicability to our implementation. Section 3 Approach presents the design decisions that we have made in our system. In addition, we present the extension component to our key value store in Section 3. Later, we have evaluated our solution and experimented our application aiming to study performance of our key value store. Finally, in the last section Conclusion, we have explained our evaluation results.

2. RELATED WORKS

David Karger et al., 1999 [5] in their research work the presented the idea of web caching with the use of consistent hashing technique. They have shown that using consistent hashing load balance and fault tolerance is handled much efficient. In this work we have used consistent hashing for mapping keys to server nodes, such that adding and removing nodes ensures a balanced load.

A few years later, the need for an efficient way to locate nodes in a distributed peer to peer network motivated Ion Stoica et al., 2002 [6] to came up with the chord protocol. In their protocol, for a given key, it is easily computable to find the responsible server. This has been done with leveraging consistent hashing technique along with a novel finger table per each node. Such that, each node by referring to its own finger table can route the key to mapping server. They have shown average lookup time for N nodes for a given key is (1/2) Log N.

Dong Dai et al., 2012 [3] explained that keeping cloud data storages in the memory rather than disks improves the retrieval and decreases the latency in key value storages. Therefore, we cashing most recent key-value pairs in the RAM to improve the latency in our system.

Beside these literatures which targeted the scalability of the system, in [1] a method for failure detection using heartbeats has been introduced. In their method, failure detection is not based on the timeout but an agreement on failure has been taken into account. Their method is applicable in case of message losses or process failures.

3. APPROACH

3.1 Design Decisions

Our key-value storage consists of three subsystems which are Client, Server and External Configuration Service. These subsystems are described in details in the section 4. In this section, we explained our design decisions that effected all subsystems namely **Replication**, **Failure detection** and the final **Extension** that we have added to the system.

3.1.1 Replication

Reliability and consistency are two of the most important factors in designing scalable web services running on cloud environments. With integrating replication into our system we achieved to overcome the loss of data, omitted the problem of single point of failure, and reduced the load on the servers. The strategy was to store each server's data also as replicated data on two other servers. Whenever a server crashes or is not available, the data can be recovered from these replicas, and the probability of the loss of the data is significantly decreased. The replica servers will also respond to clients' GET requests, so the load on each server is distributed between two other replica servers.

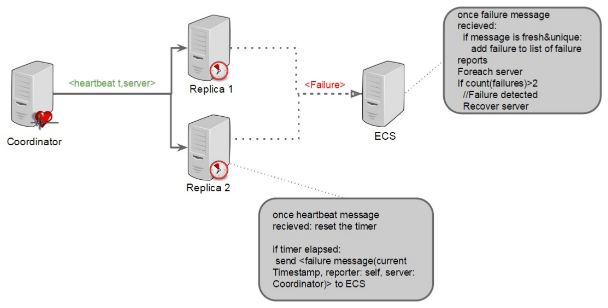
In order to implement replication with gradual consistency we designed our meta-data in a way that each server will recognize its own replicas and coordinators. Any update on the data by the client (PUT, DELETE) will directly go to the main responsible server, called Coordinator. Then the coordinator will send these updates as replication to the two next nodes in the clockwise direction within ring topology, known as replicas. Each Server has 3 persistent storage, one for storing the key value pairs which currently this server is responsible for, and two other storages to store replicated data from its coordinators. During the start up of the system, after the ECS has calculated the meta-data, and all the roles in the system has been assigned, the replication operation starts. During this operation, the ECS will command each sever to send the portion of the data in its main persistent storage that currently (in this topology) is owned by this server for replication to its replicas.

In the scenario of the GET request, each server will also respond to the client, if it has the Key either in is own main storage or as replicated data from of its coordinators. However, to increase consistency if the key is in one of his coordinators' range and no value is mapped to this key, the server will respond with **SERVER NOT RESPONSIBLE** and a meta-data update, so the client would contact the main server. Thus, it is possible for a client to get an old value for a key in the system, but never get a null value if there is already a value mapped to the respected key in the system.

During the add or removal of a node, the ECS will inform the affected servers. Then, some data has to be moved between the servers because of the change in responsibility based on the consistent hashing. In addition, the replication data is also moved. Thus, each new state of the system is also consistent with respect to replication. In case of detection of a failed node, the replication storage of the successor of that node is used to recover the lost data.

3.1.2 Failure Detection

Our failure detection is based on replica failure reports. Coordinatore nodes constantly send heartbeat messages to its replica list. Each replica knows its coordinator node by looking into the metadata. These replicas have been equipped with an internal timeout thread which is currently set to 3 minutes. On the other side, the coordinator in 10 second intervals sends a heartbeat to its replica list. As soon as this message being received by the replica, the recieving replica will reset the timeout thread. In case that the replica does not receive the heartbeat message from the coordinator before the defined timeout, it will trigger the failure detected method on replica side. Thus the replica sends a failure detected message along with his own id and failed coordinator id to the ECS. ECS considers a failure in a server only in case of agreement as described in follows. ECS first ensures the uniqueness and freshness of a failure message. Uniqueness refers to reporter id in which one given replica can only vote once for a failure. Similary, freshness refers to the time that the failure message has been received. ECS ignores the failure reports with old timestamps. In the figure 1 the failure detection sketch has been illustrated.



```
once failure message
recieved:
  if message is fresh&unique:
     add failure to list of failure
reports
Foreach server
If count(failures)>2
  //Failure detected
  Recover server
```

```
once heartbeat message
recieved: reset the timer

if timer elapsed:
 send <failure message(current
Timestamp, reporter: self, server:
Coordinator)> to ECS
```

Figure 1: Failure Detection Sketch

3.2 Extension: Subscription/Notification System

We provided data subscription capability as an extension to our key value store system. By implementing this functionality, the client is able to keep track of interesting data and get the most recent value, whenever a change has been occurred.

The subscribed key value pairs are stored in a cache on the client side alongside a hash list of subscribed servers and keys. This cache is used to respond to GET requests of subscribed keys. As a result, the load on the network and the servers will be reduced. Furthermore, this cache is guaranteed to provide the recent value of a key until the responsible server of that key has been changed, due to an addition or removal of a node. When the new meta data has reached to the client, the client will resubscribe to all the previously subscribed keys, which their responsible server have been changed, and the cache is fully reliable again. By moving the responsibilty of keeping the subscriptions consistent to the client, there is no overhead on the server side regarding this issue.

On other hand, the server is keeping a list of interested client for each key. In case of any changes to the value, all the subscribed clients will get notified with the recent change. These information is kept in servers memory in order to have a fast access to the data and more efficiency. Thus, in case of removal or failure of a node these information will be lost on the server side, but as previously mentioned. The client will always keep its subscription list updated whenever it recognizes any change in the system structure.

4. IMPLEMENTATION

Key-value storage consists of three subsystems namely KVClient, KVServer, KVECS. In this section we describe each subsystem in detail. However, full source codes are available on Github accessible at [4].

4.1 Client

KVClient is the user intermediate layer in which the end-user with knowing only one server location can start using the cloud storage. KVClient application has following main features:

CONNECT which allows the user to establish a connection to a server.

PUT allow user to store a key value pair in a server. In case that server is not responsible for demanded key, switching to the responsible server is done automatically to ensure better user experience.

GET user can query a key and like put command the responsibility of the servers has been encapsulated.

GETS it retrieves the queried key by the user exactly like the previous get command. In addition to that, the client will be notified about the queried key changes. In other words, by using this command, user will be subscribed to the corresponding server. Therefore, any value changes of the key by any client will lead to a push notification to the subscribed client.

PUTS It stores a key-value pair and subscribe to the key changes just like the getS.

4.1.1 *Subscription on client side*

Regarding the subscription part, we have implemented a separated thread to keep listening to the notification messages from the responsible servers. Whenever a client wants to subscribe for a key, the port which this client is listening to get notifications is also send to the servers, so servers can identify the clients (Using Ip and port) and contact them in case of sending notification.

4.2 Server

KVServer is the main component in the server side. The KVServer is starting a new thread for each new client request. This new thread will handle any request coming from clients including all client available operations. In the same context, the server component includes a database manager component. This manager is responsible of storing and retrieving data in pairs from and into the server database. The database manager is a thread safe, supporting multiple threads accessing the data and perform different operation simultaneously.

On other hand, the server side has multiple statuses. The reply from the server to client includes this status. The statuses of the server side are :

UNDER INITIALIZATION: This status represents that the server has not started to serve client yet and its under initialization operation from the ECS component.

STOPPED: In case of the server is stopped and not serving clients temporarily.

SERVER WRITE LOCK: This status means that the server is currently moving or receiving data from or to other servers.

SERVER NOT RESPONSIBLE: This status indicates that the requested data is not located on this server and also not replicated on this server. In case of this status, the server will send a copy of metadata to the client so it can redirect the request to the responsible server.

GET ERROR, GET SUCCESS, GETS SUCCESS: Indicates the status of the requested GET operation.

PUT SUCCESS, PUTS SUCCESS, PUT UPDATE, PUT ERROR: Indicates the status of the requested PUT operation.

DELETE SUCCESS, DELETE ERROR: Indicates the status of the requested PUT with null value - Delete- operation.

4.3 External Configuration Service

ECS is the central component in our system, which coordinates KVServers, knows how the data is distributed in the system, and modifies this distribution whenever it is needed, such as when a node is added, removed, or failed. The ECS calculates the meta data and assigns key ranges to the servers. ECS's role in this system is comparable to master's role in Google's Big Table key value store.[2]. Furthermore, ECS has a User Interface to interact with the admin. Adding and removing a node(**add, remove**), starting and stopping the system(**start, stop**), and shutting down the system (**shutDown**) are the commands available for the admin of the system.

ECS will be started from a configuration file consisting of the list (IP,Port) of the servers called server's repository. Before the beginning of the cloud system or adding a new node to the system, ECS will try to run and start the required number of the servers from this repository list with remote procedure calls made through SSH to the remote machines hosting the servers.

During the add and remove operations, ECS will coordinate the affected servers to move their data, either for reassigning them (in case of add a new node) or for saving them as replication to another servers. ECS will keep track of data movement operations, so whenever one of these move data operations failed and caused the system to lose some portion of data or put the system in an unstable state, the ECS will undo the operation and will return the system back to the previous state before the add or removal operations. However, in case of no problems happened, ECS will send the new meta-data to the other servers, which were not affected during these operations, so did not know about the current changes of the system till now.

In order to get the failure reports from the servers, the ECS has a thread which is listening to a specific port. This port is send to the servers during their initializations by ECS. Whenever a failure of a node has been dicovered (see section 3.1.2) ECS will start the recovery operation. In recovery operation a new meta-data is calculated, and ECS will command the successor of the failed server to recover the data of the failed server from its replication storage. Later on, the affected servers will send replication data to their new replicas, and get replicated data from their new coordinators. Finally, ECS will try to add a new node to the system.

5. EVALUATION

5.1 Dataset

Data for our experiment, key value pairs, were gathered from the UNIX dictionary of British-English words. A large data set consisting of approximately one hundred thousand words.

5.2 Test prepration

In order to simulate a real environment, only 80 percent of data is populated to the servers. So the client have the chance of quiering a key which is not stored in the system.

5.3 Test parameters

The test plan was designed using Jmeter to measure the system performance under different parameters. The common setting for all test cases were:

1. Number of users that has been set to 1000

2. Ramp-up time that has been set to 20, which means 1/20 of total number of users, in this case 50, per second are being added to the system.

3. The main testing variable in each test was the number of launched KV servers.

5.4 Test Steps

1- Setup test environment by launching ECS with requested number of servers by the test definition.
2- Executing parallel write operations simulating different users. The key-value pair are randomly retrieved from the test dataset as stated before.
3- Executing concurrent read operations of a random key retrieved from the test dataset. As only 80% of the keys are available in the cloud storage, there is 20% chance of requesting a key which does not exist on the servers and this will result to error(GET ERROR).
4- Shutting down ECS.

5.4.1 Test Environment

The test was conducted deploying all the components on the same machine, the test machine capabilities: Processor: Intel Core i5 CPU @ 2,4 GHz (4 CPUs) , Memory: 8 GB 1600 MHz DDR3

5.5 Tests and Results

5.5.1 Test case1: (1000 clients, 1 server, 20 s ramp-up time)

Write operation: Table 1 presents the performance results and figure 2 illustrates the response time graph for this test case.

avg	med	min	max	error	Throughput/sec
97661	106326	309	366	0	4.77

Table 1: Write operation measures(1000 clients, 1 server)

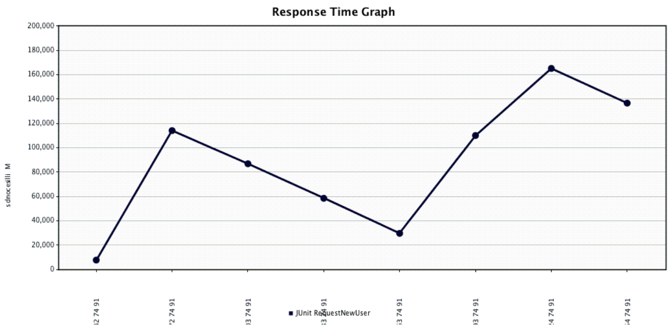


Figure 2: Write operation (1000 clients, 1 server)

Read operation: Measures results shown in table 2. The response time graph for the get requests has been shown in figure 3.

avg	med	min	max	error	Throughput/sec
13	7	2	381	0.002	43.36

Table 2: Read operation measures (1000 clients, 1 server)

5.5.2 Test case2: (1000 clients, 5 servers, 20 s ramp-up time)

Write operation: performance results shown in figure 4 and table 3.

Read opearation: expermient results illustrated in figure 5 and table 4.

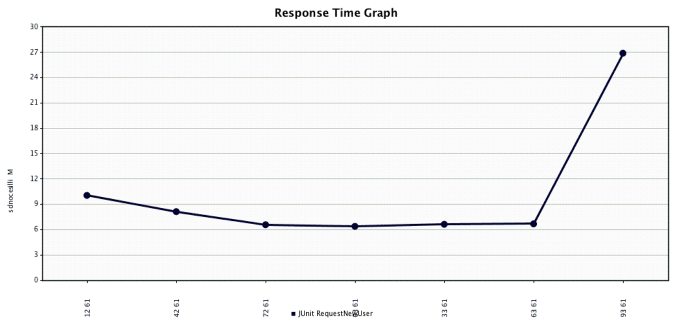


Figure 3: Read operation (1000 clients, 1 server)

avg	med	min	max	error	Throughput/sec
20639	17827	40	56853	0	14.37029373

Table 3: Write operation (1000 clients, 5 servers)

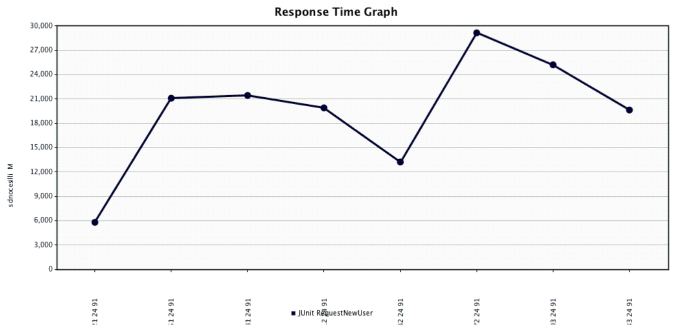


Figure 4: Write operation (1000 clients, 5 servers)

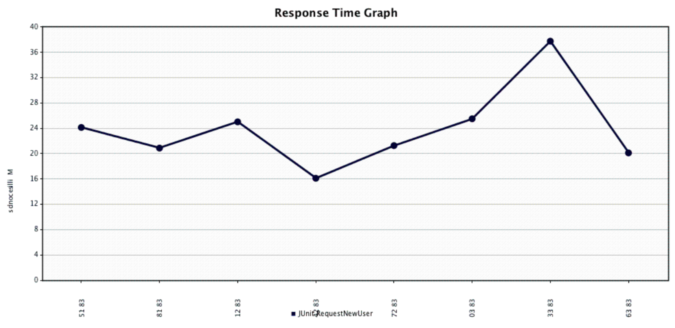


Figure 5: Read operation (1000 clients, 5 servers)

5.5.3 *Test case3: (1000 clients, 10 servers, 20 s ramp-up time)*

Write operation: operation experiment for this test case is shown in figure 6 and table 5.

Read operation: requests performance results for this test case is presented in figure 7 and table 6.

5.5.4 *Test case4: (1000 clients, 10 servers, 20 s ramp-up time) without saving after each put request*

avg	med	min	max	error	Throughput/sec
55	12	2	1662	0.26	43.08487721

Table 4: Read operation measures (1000 clients, 5 servers)

avg	med	min	max	error	Throughput/sec
9995	2600	9	50262	0	18.85298443

Table 5: Write operation measures (1000 clients, 10 servers)

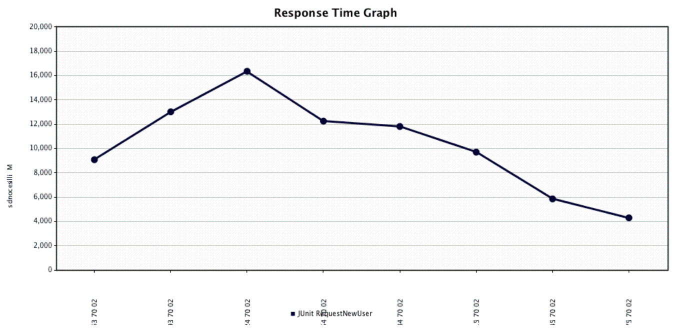


Figure 6: Write operation (1000 clients, 10 servers)

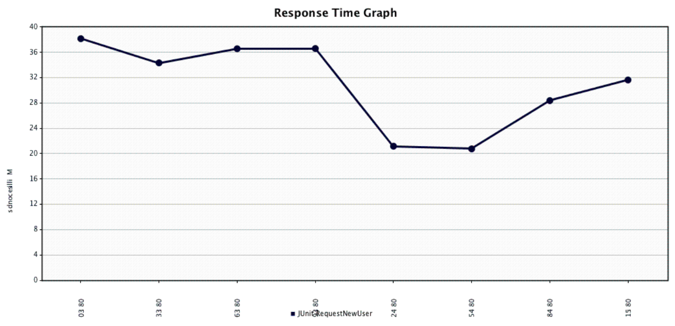


Figure 7: Read operation (1000 clients, 10 servers)

Write operation: operations performance results without constantly saving to persistance storage after each request is presented in figure 8 and table 7.

Read operation: Not mentioned as it is the same as test case 3 read results.

6. DISCUSSION

This section is about discussing the results of the previous performance measurements. In the beginning, it is important to mention that the number of users (running threads) and ramp-up are fixed in all previous test cases. The only change is with the number of the servers, since those test cases aim to test the load balance between servers and its influence on the overall performance.

In the first test case, with only one server running, the graph for the read operation shows that the performance started with a good response times. By the end of the test

avg	med	min	max	error	Throughput/sec
30	12	2	397	0.21	42.86877867

Table 6: Read operation measures (1000 clients, 10 servers)

avg	Med	Min	max	error	Throughput/sec
14	14	3	97	0.21	43.07188698

Table 7: Write operation measures (1000 clients, 10 servers) no save after each operation

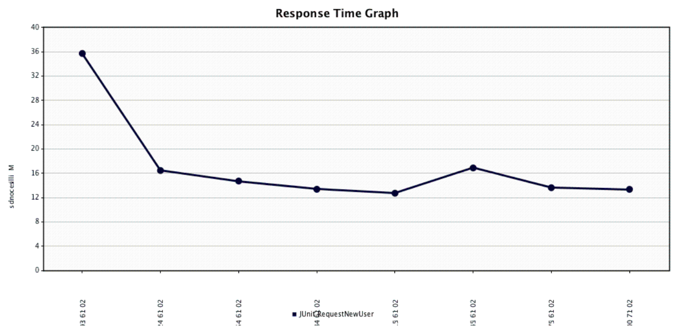


Figure 8: Write operation (1000 clients, 10 servers)

case, the performance is getting slower and the response time increased. The reason behind this increasing in the response time is that the number of user threads is increasing 50 users per second since the ramp up time is 20.

On other hand, the graph of the write operations shows that the performance of write operations has some overhead. This overhead is due to the synchronization between threads in the writing operation, since the system is moving the ”in-memory” cache storage into the persistence storage after each write operation.

By comparing the results from the second test case, which has five servers instead of one server, the read operations graph shows that the performance of the read operation is getting slower after increasing the number of servers. This leak of performance is due to switching the connection from one server to other by the client threads. The client threads are switching the connection based on the hash value for the inserted key. On other hand, the interesting point is that the performance of the write operation is improved five times comparing to the previous test case. The reason of this improvement is that the load has been balanced between five servers instead of only one server. In that case, the synchronization between threads for the write operation and moving the ”in-memory” cache into persistence storage is distributed between servers which has the impact the performance of the write operations.

In the third test case, the number of the servers has increased up to 10 servers. Again, the performance of the read operations is getting slower since the users threads need to switch between ten servers instead of five. Although, comparing the results with the second test case results, the performance of the write operations has been improved twice. This improvement is a result of balancing the load into ten servers.

Eventually, the fourth test case is exactly the same as the third one, but without moving the ”in-memory” cache into persistence storage after each write operation. Instead of that, moving the ”in-memory” cache is moved into persistence storage only at the shutdown time. In that case, all the operations are done in the memory without performing any I/O operations. As a result of this change, the performance has been improved dramatically, since the user threads do not need to wait for persistence storage operations. In conclusion for the previous results, the more servers the system has, the slower performance for the read operations. This leak of performance is due to the switching

7. CONCLUSION

We have provided a key value storage which is highly reliable in terms of data persistance state using replication mechanism. Beside that, a heartbeat system with minimum overheads while being able to detect actual failures has been introduced. Furthermore, we designed and implemented a publish-subscribe extention for our KV-Store which can be really handy for cloud based applications such as chatting and data storages.

According to our evaluation results, the more servers the system has, the slower performance for the read operations. This leak of performance is due to the switching between servers based on the hash value of the key. On other hand, the performance of the write operations is getting better since the synchronization between user threads is distributed among servers. In the same context, avoiding the persistence storage after each write operation and just performing that once at the shutdown time will improve the performance dramatically as seen in the last test case.

As future work of this project we propose, a mechanism to periodically persist the running cache and discover the trade off between increasing the cache size (to load key-value pairs into the memory) and latency of the servers.

8. REFERENCES

[1] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. *Distributed Algorithms*, pages 126–140, 1997.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, 2006.

[3] D. Dai, X. Li, C. Wang, M. Sun, and X. Zhou. Sedna: A memory based key-value storage system for realtime processing in cloud. In *Cluster Computing Workshops (CLUSTER WORKSHOPS) IEEE International Conference on*, 2012.

[4] GodzillaTeam. Key-value storage database, 2015.

[5] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, and R. Dhanidina. Web caching with consistent hashing. *Computer Networks*, 31.11:1203–1213, 1999.

[6] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2002.