Cloud Databases

Team: Cloud9 - Design and Implementation Report

Shankar Mohanbabu Sathyakumari
Department of Informatics
Technische Universität München
shankar.m.s@outlook.com

Han Yang
Department of Informatics
Technische Universität München
yanghancs@gmail.com

ABSTRACT

Since the dawn of the digital age, the challenge involved in the storage and manipulation of data has always been a topic of prime importance and interest for humankind. This is essentially due to the fact that the digital data collected, processed and retrieved acts as the operational lifeline for all the people, businesses and governments that rely upon it. The aforementioned challenge has been addressed in numerous appropriate ways over the years based on the then prevalent technologies and solutions. All such technologies have had their own advantages and disadvantages. One such solution that exists today is the distributed key value storage systems, which tend to be highly scalable, reasonably efficient and offer better availability which renders them highly favourable to store big data. In this project we shall explore this topic further in order to build our own version of a distributed key value storage system, extend its functionality and consequently evaluate its performance. We shall also discuss about its strengths and weaknesses, as well as some possible enhancements that could be implemented in the future.

KEYWORDS

Distributed key-value store, Replication, Consistency, Availability, Fault-tolerant

ACM Reference format:
Shankar Mohanbabu Sathyakumari and Han Yang. 2017. Cloud Databases. In *Proceedings of Lab course: Cloud Data Bases, TUM, Munich, Germany, February 2017,* 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent times, the amount of data that is produced continuously for a single day according to IBM is said to be around 2.5 Quintillion bytes of data which is certainly an enormous amount. This data produced is more data than the total amount of data which was produced for a whole year in the past. In more simpler times, the traditional SQL database, that was proposed over 30 years ago, was still in vogue and was the most widely used technology in this area until quite recently. Traditional SQL databases provide ACID (Atomicity, Consistency, Isolation, Durability) guarantees which are deemed vital for certain use cases, such as bank systems for instance. However these databases are extremely restricted in terms of data models and are not very scalable, which diminishes their value and makes them incapable for handling big data. With the advancement of time and technology, we have seen the conception of several NoSQL databases such as Bigtable, HBase, Cassandra and Dynamo [1]. Contrary to the SQL's ACID guarantees these NoSQL databases provide BASE guarantees which stands for Basically Available, Soft-state and Eventual consistency. The trade-off is between Availability/Efficiency and Consistency. They are indeed more available and efficient, but a certain level of consistency is sacrificed. Most importantly, they are capable of being easily scaled horizontally.

In this report, we shall describe in detail the design and implementation of our distributed database system. Section 2 provides the design architecture and the overview of the whole system. In Section 3 certain basic features are touched upon and Section 4 deals with our chosen use case and specific features that are required for the same. Section 5 shall contain the observations and results, achieved after the performance testing. Section 6 and onwards shall contain our conclusions drawn after the completion of this project, a review of related works and further enhancements for the future that we could implement to our system.

Lab course: Cloud Data Bases, TUM, Munich, Germany
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

2 ARCHITECTURE

2.1 System Architecture

As shown in Figure1, the system is basically comprised of five primary components. A client library, which is used by client applications on the client side, a key-value storage server, that can run on multiple physical servers, an External Configuration Service (ECS) and an ECS client.

The ECS, which is handled by the ECS client through the command line, is responsible for the starting and stopping of servers, assigning of key ranges (metadata) to different servers, dealing with the addition of servers and the removal of the ones that have failed. However this is not responsible for storage service and query.

The KV Storage server is the one that is responsible for handling query requests and storage service. It also provides the following two operations for the client library: get(key) and update(key, value). In case a particular KV server is not responsible for the key it receives in the request by a client, it promptly re-routes the client to the correct one. This is possible as each server on the system possesses metadata and knows the location of each key stored within the distributed KV store.

The client library has get(key) and update(key, value) interface for the client applications. It stores the metadata into its cache, thus enabling itself to know which server should must be contacted after a request is received. Additionally, when existing metadata becomes stale it receives new metadata from the servers.

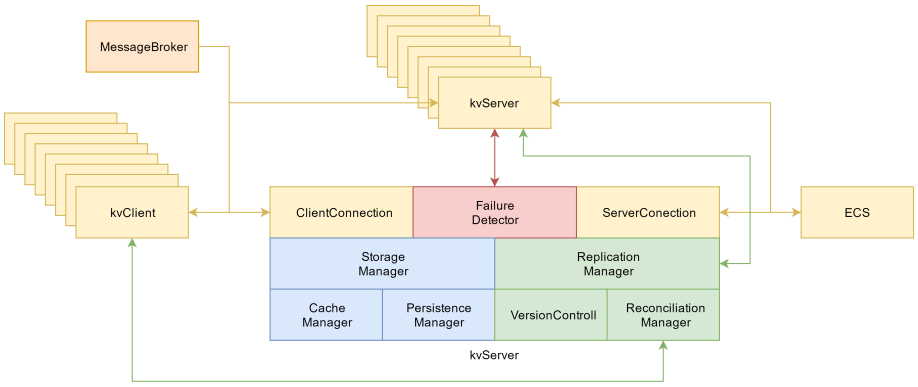

Figure 1: The system overview

2.2 Software Architecture

The following Figure 1 shows all the software components within the distributed KV storage server along with the communication between a client and a KV server, a KV server and other KV servers, a KV server and the ECS, a client, a KV server and the Message Broker.

Some of these components were predefined, but others are unique and indicate our own design decisions, for instance the Gossip-style failure detector, replication of servers based on *Optimistic (Lazy) Replication* mechanism and *Notification mechanism* driven by the Message Broker. The design, implementation and functionality of these extensions shall be explained in further detail in the following sections.

3 FUNDAMENTAL FEATURES

3.1 Persistence Storage

The storage primarily consists of a cache whose size is modifiable and a persistence store which uses native file system. Hash tables are used in the cache in order to make it very efficient for query, when the key is in cache. Otherwise, I/O access will occur, which can be very slow compared to a hit in cache. In order to adapt to various use cases, three displacement strategies FIFO, LRU, LFU can be implemented to store keys between cache and persistence store.

3.2 Scalable Storage

Due to the fact that the system is built for storing big data, it becomes vital for us to make it horizontally scalable, so that adding new servers to obtain more storage becomes simpler. This goal is achieved through Consistent hashing. MD5 algorithm is employed to encode each key and also each server, based on its IP address and port number. Consequently, each key will be mapped to the server with the least MD5 value higher than the keys. Therefore each server will be responsible for a certain key range. This ensures that only one existing server and its key is modified in the event of adding a new server to the system.

3.3 Lazy Replication

Replication is necessary to achieve higher availability, to make the system more fault-tolerant and to provide better query performance. Here we use lazy replication[2] as it can provide us with the most availability. In our system, each key will be stored in one coordinator server which is directly responsible for that key along two replica servers which have the least MD5 values higher than the coordinator server. But sometimes conflicts could occur with the lazy replication due to concurrent writes, which can cause trouble to ensure the eventual consistency. We shall elucidate on this further and provide a solution for the same under the section for extensions.

3.4 Gossip-style Failure Detector

A *gossip style failure detector*[5] is employed in our system. The underlying principle of this mechanism involves each server "regularly" checking for failures among its neighbouring nodes. This is possible as each server contains a list of all other servers in that group along with their identifiers, a counter variable, which is called *heart beat* and another integer variable, which indicates the last time the heart beat of the corresponding member was updated in the list. Essentially, the second variable retains the time at which a heart beat update happened. A certain time called *time of gossip* is defined (T_{gossip}), and in intervals of this time each server increments its own heartbeat and sends its list to one random member. This is called a *gossip message* and after receiving this *gossip message*, each server merges both lists (it's own and the one which is received) and always takes the higher heart beat value for each member. Such heart beat updates are denoted by the second integer variable, mentioned above. Each server checks the list it possesses and evaluates the total time that has passed since the last update of each member's heart beat. That is it subtracts the second integer variable from the value of the current time indicator. If the heart beat of a server was not updated since T_{gossip} seconds, this server is considered to be offline. Logically, the more servers present in the system implies greater the time of failure.

When a server detects a failed server in its vicinity, it promptly contacts the ECS and informs it about the server which has failed. The ECS will take the action as of removing the failed server, and then adds a new node. This allows the system to recover from the

failure and continue to function correctly. It is important to note that after detecting a failed server, the information of that server will not be removed from the list immediately. This is because not all servers will detect a failed member simultaneously. Therefore, if a server receives another gossip message which includes the failed member, which was already deleted, it is going to re-include that in its list. A $T_{cleanup}$ time should be set at a value which is twice than that of $T_{failure}$. In this case, the probability that a server will receive information of a failed server is the same as the probability of failures.

4 USE CASE SCENARIO: AN E-COMMERCE WEBSITE

In today's world, the usage of several e-commerce websites, in order to buy almost everything under the Sun, has risen exponentially over the last few years. This titanic increase of online shopping means an equally massive increase of data to be stored online, a challenge that needs to addressed. Our distributed KV storage attempts to resolve this very same challenge.

With millions of users, browsing through millions of products everyday, the need of it is only growing with its importance by the day. A customer must be provided with the latest information about a product such as its price, availability and number of days required for it to be delivered to him/her. Only then he/she might be compelled to buy the product online. Also when a customer intends to buy a product only of a particular make, model, colour and price range, which is still not available in the inventory he/she could add this to his/her wish-list and subscribe to the latest updates from the seller about the same. Keeping these two aspects of an e-commerce website in mind we have added the following two extensions - a Lazy replication scheme with Reconciliation Manager and a Notification Mechanism, which are going to be described in depth in the following sections.

4.1 Extension 1: Eventual Consistency based on Lazy Replication with Reconciliation Manager

4.1.1 Distributed lazy replication. Since the time we decided that an e-commerce website shall be the main use case scenario of our key-value database, we started to think about what could be the most needed feature for such a website. When people surf the Internet, the speed for rendering a page will directly affect their happiness and browsing experience. Similarly in the case of an e-commerce website, it will consequently change the likelihood of whether or not a customer would buy a product on this website. Therefore we wanted to make high-availability as the main feature of our distributed key-value storage database.

The first decision we made is to enable multiple servers to handle WRITE requests. Compared with the single-coordinator server architecture the system availability is definitely improved through this multi-write mechanism. The workload of WRITE request gets balanced among all servers thus expediting the handling of users' requests.

The second decision we made is to choose the lazy replication[2] as our system's replication protocol. Lazy replication is an asynchronous replication strategy that does not have to wait for all

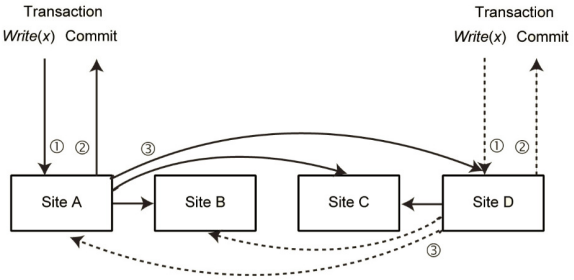

Figure 2: Distributed lazy replication

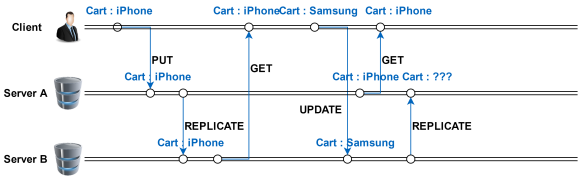

Figure 3: Data conflict with concurrent write

of the copies to be synchronized when updating data, thus helping concurrency and parallelism and improving the overall system availability.

When we combine multi-server write and lazy replication together we obtain a system protocol for high-availability. We call this combined protocol as "distributed lazy replication", as shown in the figure 2, which we have defined as follows:

- Every server in the system can handle Write/Read request. The request will be handled locally and returned to the user immediately.
- After a time period (which can be set by the user), a replication message will be sent to other sites.
- When a server receives a replication message, it will then replicate these changes locally.

4.1.2 Vector Clock. As we implemented the distributed lazy replication [3], we were faced by the challenge to ensure eventual consistency even when concurrent writes occur - those that haven't finished executing before a new one is started. 3 shows an example of a concurrent write. Suppose we have a client that communicates with two KV servers: server A and server B. The client first pushes a *WRITE* request to server A, to write the key-value pair cart:iPhone. After some time period, the replication takes place, both server A and B hold the key-value pair cart:iPhone. Later on, the client makes an *UPDATE* request to update the KV pair to cart:Nokia in server B. Now server B holds the KV pair cart:Nokia and server A holds the KV pair cart:iPhone. During the time of replication a conflict occurs, since the system doesn't know the causality of these two different versions of data.

In general, there is no way to tell which of the two writes was issued first without a global clock. But we can use the logic clock to record the logic sequence (causality) of the concurrent writes. Here we implement the vector clock to describe the causality between related or conflicting values. A vector clock is effectively a list of node-counter pairs. One vector clock is associated with every

version of every object. One can determine whether two versions of an object are on parallel branches or have a causal ordering, by examining their vector clocks. If the counters on the first objectfis clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten. After we implement the vector clock, we can see that the conflict is resolved as is shown in figure 4.

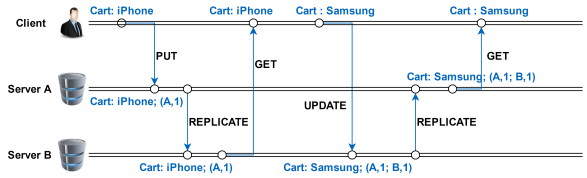

Figure 4: Vector clock to solve data conflict

4.1.3 Reconciliation Manager. Still, there remains a situation for data conflict that cannot be solved in this way. If the vector clock for two objects are considered to be in conflict, the data cannot be merged into one version and may require further reconciliation. An example for this situation is shown in figure 5. If the user provides an *UPDATE* to both servers before the servers complete replication among themselves, conflict can arise. The data in both server A and server B will become cart:Samsung(A1B1);Nokia(A2), which cannot be solved by a vector clock.

Therefore we have introduced a reconciliation manager that a user could use to resolve such a conflict. Upon processing a *READ* request, if our system has access to multiple branches that cannot be syntactically reconciled, it will return all the objects at the leaves, with the corresponding version information in the context. After the user receives the KV pair with conflict values, he can use a new command called *MERGE* to update the KV pair and merge all different data versions into one. An update using *MERGE* command is used to reconcile the divergent versions and the branches are collapsed into a single new version. This process is shown in the figure 5, where the user uses *MERGE* command to merge the values to cart:Nokia(A3,B2). Using the reconciliation mechanism, the conflict is solved and the system can ensure final consistency. Every replica will eventually see every update, and will eventually agree upon a single non-conflicting value.

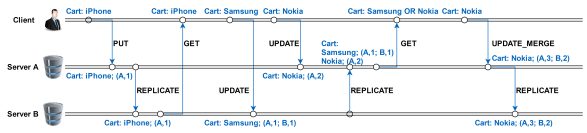

Figure 5: Reconciliation manager with *MERGE* command

4.2 Extension 2: Notification Mechanism[4] based on Publisher-Subscriber pattern with Message Broker

This mechanism ensures that the customer receives updates about any modifications (update/delete) done to the keys to which the customer has subscribed to. In reality, in order to effectively implement this system a few special cases need to be considered. Firstly, in the event of failure of the coordinator node. Secondly, when the key value storage has been rearranged and a new coordinator becomes responsible for a particular key. The notifications have to be generated in both the above cases. Thirdly, one must ponder upon as to how the location of a client can be determined in the system and also how to identify them if their connection gets constantly reset. Additionally, due care must be taken in order to handle these subscriptions efficiently and such that the limitations of the network capacity are also considered.

As shown in the Figure 6, the Publisher-Subscriber pattern employs a separate channel, wherein the servers publish their changes and the clients register for updates about certain products, which are essentially represented by a key in key-value paradigm. Consequently, the core of the notification system's functionality is to filter all the information received and propagate key changes to the appropriate clients who have subscribed to that specific key.

This is managed by the crux of the notification mechanism called the *MessageBroker* which handles all subscriptions. This component provides a server socket and is started on a remote server by the ECS. All machines connected to the *MessageBroker* can act as publishers or subscribers. The communication occurs in the message format *KVBrokerMessage* providing four general types of messages. The first type allows the user to subscribe or unsubscribe to products (keys). The second type is confirmation of such actions. The third type of message is used to inform the client about any possible errors, such as a non-existent key, an already existent subscription etc., The last one provides servers with the opportunity to publish their updates.

The connection that exists between the *MessageBroker* and an individual publisher or subscriber is enabled by a *BrokerConnection*. This is very similar to the *ClientConnection* of the *KVServer*, In order to locate the clients, the individual connections are stored in an instance of *SubscriberData*. It handles a list of subscribers, which are represented by their individual instances of *BrokerConnection* and the keys that these subscribers have subscribed to. Naturally, it's possible for a client to subscribe to many keys, in the same way, a key can have multiple subscribers, thus there exists an n to m relation between keys and subscribers. A subscriber can add more

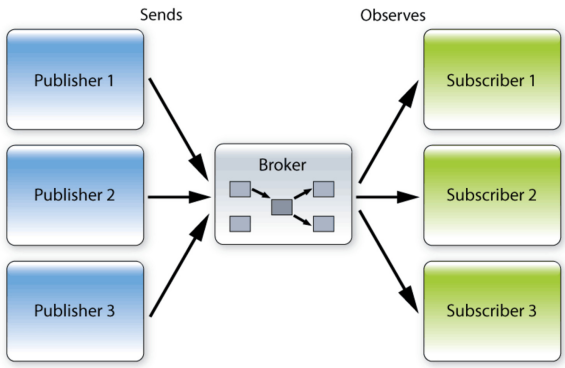

Figure 6: Notification Mechanism

keys in its instance of *SubscriberData* when it wants to subscribe to updates for a new product, or delete subscribed keys whenever a client decides he/she is not interested in a particular product anymore. Furthermore, the *MessageBroker* must be capable of removing subscribers from the list when they go offline. Also it must be able to retrieve key-server mappings in order to distribute published data. There are a few steps involved in the process of publishing. The publisher maintains a connection to the *MessageBroker* and sends any updates of its data to it. Then the *MessageBroker* obtains the list of subscribers of that data from its *SubscriberData* and send the updates to the appropriate clients. On the subscriber's side the client opens up a connection to the *MessageBroker*, whenever they want to subscribe or unsubscribe. A subscription request is then sent to the *MessageBroker* which in turn answers with a confirmation message or an error message if need be. Following which the *MessageBroker* transfers the received data updates to the appropriate subscribed client.

4.2.1 Advantages of the Design. The most advantageous aspect of this implementation is that the *MessageBroker* can easily locate subscribers in the system, without having to expend any time looking for them. As pointed out earlier, the published notifications must still be delivered even in the event of the coordinator's failure. Also the problem associated with possible rearrangements of the Storage Service is also handled gracefully, since new servers only need to connect to the *MessageBroker*. Finally, the purpose of the project was to implement a distributed storage system, therefore another advantage is that the handling of key subscriptions need not be handled by the *KVServer*.

4.2.2 Disadvantages of the Design. There are a few drawbacks in this kind of implementation which must be listed out. Firstly, a very large number of connections are handled on one server. A possible, but yet not so simple solution to this would be the implementation of a distributed system of its own. Secondly, there is only one instance of the *MessageBroker* server running, just like the ECS, if this instance fails, all the information will be lost. Therefore it has a single point of failure and thus it's not a robust system. Thirdly, there is no permanent subscription storage, because in order to store such data, some kind of client identification would be necessary. Consequently, if a client goes offline, all the information about their subscriptions is duly lost and cannot be retrieved once again. In order to resolve this a client identification could be made possible with help of a username secured by a password, which actually means creating an account for every client. Then all this data must be stored somewhere. All of these could be considered as enhancements that the system might require in the future for better operation.

5 PERFORMANCE

In a distributed system where there is a possibility of several servers providing services to multiple clients at the same time, propagating a huge stream of data, it becomes a matter of prime importance to evaluate how the overall system behaves when these individual components function in concurrency retaining the data consistency and its associated performance measure. Also there is a need for us to evaluate the performance of this distributed architecture with respect to the conventional client-server architecture. Several test cases were performed varying the number of servers (5, 10, 15 and 20), cache sizes (5, 10, 15 and 20) and displacement strategies (FIFO,

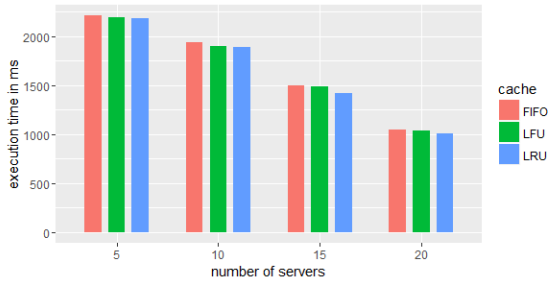

Figure 7: Performance test for *PUT*

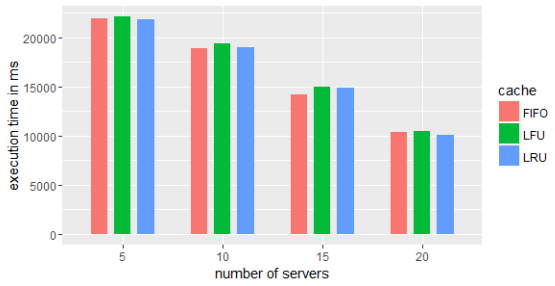

Figure 8: Performance test for *GET*

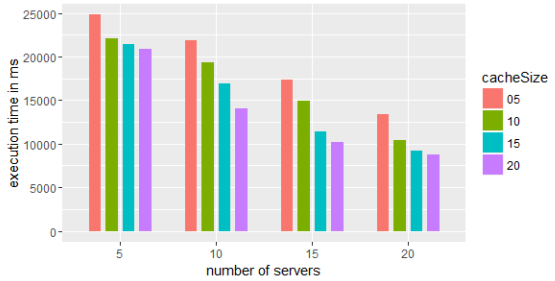

Figure 9: Performance test for *CacheSize*

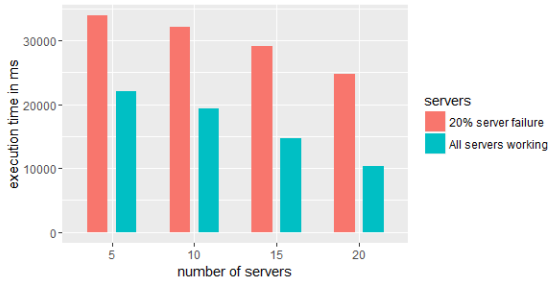

Figure 10: Performance test for *Failure*

LFU and LRU) on our distributed environment and the observations were noted down.

In our test cases a constant number of 10 clients were connected to the service to simulate a moderate amount of users. The service was then populated with 1000 key-value pairs from different clients so as to simulate reasonable data volume. The figure 7 illustrates the variation of execution time when these key-values were entered from multiple clients using PUT. It was observed that the time taken for completion of these operations reduced as number of KV servers increased. There were no significant variations among values obtained for different displacement strategies, as expected. The plot in 8 illustrates the same variation of factors as the previous figure, but only now for the GET operations. It was noted that the latency time gradually decreased as the number of KV severs were increased and reached an optimal constant value after a certain threshold for the number of KV servers was reached.

Consequently we tried to verify if the performance really improves with increased cache and our observations proved it accurately. As you can see in the 9, the average execution time considerably decreases with the increase in cache size irrespective of the number of KV servers active in the system. In a real world scenario, as there is a high possibility of server failures to occur anytime, we ventured to simulate this by disconnecting twenty percent of total active servers and compared it with the performance when there are no failures in the system. 10 illustrates how significantly these failures affected the execution times for GET. This is due to the fact that the ECS must re-distribute the key-values from the failed nodes among the remaining active nodes. As the load among the remaining servers increase, the performance evidently gets deteriorated. Over all, considering the limitations of time, only a test sample of scenarios were simulated. However with the observations made in our test cases one can successfully understand the salient behavioural aspects of a distributed key-value storage system.

6 CONCLUSIONS

In conclusion, the distributed system that was designed and implemented offers a highly available, reliable and scalable KV Storage system which evidently satisfies all the basic requirements of our use case of an e-commerce website. The notification mechanism ensures customers get their updates on time, and the Lazy replication mechanism backed by a *Reconciliation Manager* ensures that the replica servers are eventually consistent and possible data loss is averted. Furthermore, performance of the developed system can compete well with simpler architectures and also scales easily. Effectively, the system meets all the requirements of an up to date distributed storage system. From this experience, we realized that there are no definite set of "good" design decisions for such a distributed storage system. All design decisions must be made in accordance to the use case at hand. For instance, there always exists a trade-off between availability and consistency and one must be chosen over the other only after a thorough evaluation with respect to the use case and requirement of the system.

7 RELATED WORK

There is always a trade-off that exists between Availability and Consistency. Most systems sacrifice consistency for better availability, such as Dynamo and Amazon S3 which only provide eventual consistency. Our key-value storage also is quite similar to these systems as we have retained focus on availability rather than consistency. Similar to how tremendously Amazon has benefited by using these systems, we have also envisaged that our database could be used in the same lines for a successful e-commerce website. Although conceptually similar, Cassandra focuses mainly on consistency and provides several configurable consistency levels including quorum-based consistency and eventual consistency. Other databases such as Oracle NoSQL Database, Project Voldemort and Riak also provide a key-value storage with high scalability.

8 FUTURE ENHANCEMENTS

In this course, we have only managed to design and develop a simple key-value based distributed system where in the type of keys implemented are simple strings which does not allow us to use them efficiently in a real-life e-commerce website. This is because each product and its associated information is just practically a lot of information to be stored as a simple string. Also we need to take into consideration the fact that any product, although identified by a unique identification number such as Universal Product Code (UPC) or Amazon Product Code (APC), might have a different make attribute such as colour or manufacture date which also needs to be stored and therefore a simple key string containing UPC will not be sufficient. To overcome this, the key can be extended into two parts (key, column) and can be stored either as (UPC, colour) $\rightarrow stringor(UPC, ManufactureDate) \rightarrow string$.

Further improvements could be made with the ECS and MessageBroker. In the current implementation, both these components are a single point of failure which is not desirable in any robust system. In order to make it more reliable, we could replicate these crucial components and maintain consistency among them with the use of algorithms such as Paxos. Furthermore modifications can be made in the notification mechanism to identify clients with a username. This would allow the MessageBroker to identify clients even in the times of frequent client reconnects.

REFERENCES

[1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[2] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)* 10, 4 (1992), 360–391.

[3] Michel Raynal and France IRISA. 2002. Fundamentals of distributed computing: A practical tour of vector clock systems. (2002).

[4] Stephen James Paul Todd. 2003. Message broker apparatus, method and computer program product. (Jan. 21 2003). US Patent 6,510,429.

[5] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. 1998. A gossip-style failure detection service. In *Middlewarefi98*. Springer, 55–70.